CE888 Assignment 2: Decision Trees for Expert Iteration (Reinforcement learning)

Damian Machlanski[1]

***Abstract*— Recently, remarkably strong agents have emerged through a combination of machine learning techniques and Monte Carlo Tree Search (MCTS). However, modern reinforcement learning is almost exclusively focused on neural networks, leaving other directions unexplored. One of these unvisited territories are decision trees. Although they might be less powerful than deep learning approaches, they are highly transparent, which might be further used in knowledge extraction. This work shows that decision trees combined together with MCTS through expert iteration algorithm can also yield interesting results. Moreover, it appears that pure decision tree models are able to beat vanilla MCTS agents quite regularly. Both of these results make decision trees in the context of reinforcement learning a promising research direction for the future.**

I. INTRODUCTION

General purpose agents able to interact with various environments is one of the grand challenges of the field of artificial intelligence (AI). Such agents, ideally, are fully autonomous that learn optimal behaviours and improve over time through trial and error. This is primarily what reinforcement learning (RL) is concerned about, a thorough survey of which is presented in [1], [2].

As pointed out in [2], despite previous RL successes, they lacked scalability, stability and often required careful hand-crafted feature engineering. This has changed dramatically, due to recent advances in deep learning (DL), which enabled automatic high-level feature extraction (see [3], [4] for a survey). Some of the most prominent breakthroughs in DL were reported in computer vision [5], [6] and speech recognition [7]–[9]. It was therefore natural to try to incorporate these techniques into the RL realm, which was the case in [10]. More specifically, they developed a Deep Q-Learning (DQN) algorithm that utilises convolutional neural network (CNN) for function approximation, does not require manual-engineered features and is able to play in real time. However, despite this huge success, the average game score obtained was nowhere close to planning-based approaches, such as monte carlo tree search (MCTS), which also have their disadvantages, like substantial computational resources required and unfeasibility to use them for real-time play.

This situation presented another challenge: to combine planning and model-free approaches in such a way as to retain DL advantages of not needing hand-crafted features and low computational resources and also exploit the data generated by MCTS agents. Several papers explored this idea, two of which are particularly interesting [11], [12] and constitute a starting point of this project. The concept is to generate expert moves by slow MCTS and then train a neural network (NN) on them. Next, more data is generated but this time the expert is guided by the previously-trained NN, which results in choosing more promising directions throughout the game. This whole cycle is repeated as many times as possible and gives astonishing outcomes. Not only it produces very strong game agents able to beat human experts, but also trains the agents tabula rasa, i.e. via pure self-play with no prior knowledge.

The goal of this project is to reproduce such expert iteration technique but with some modifications. First and foremost, instead of using a CNN as the apprentice model, a Decision Tree (DT) is going to be utilised to guide MCTS. As a consequence, the solution proposed cannot accept a raw game play image but rather requires an artificial representation of game states and actions. On the other hand, DTs are easier to set up, due to having less hyperparameters, require less data and are generally quicker to train and run. As a bonus, they are also more interpretable as they can be visualised or even transformed into IF-THEN rules. Secondly, the learning environment being explored here is the OXO game, also known as Tic-Tac-Toe.

Therefore, the main motivation of this work is to explore the possibility of using DTs instead of NNs in the expert iteration setting. Although there have been some signs of utilising DTs in RL in general [13], [14], there is little to no evidence of using them in a combination with MCTS techniques, making this work even more worthwhile doing.

This document is structured as follows: section II reviews some of the similar efforts done in the past and discusses their findings. Section III explains all the methods and analysis used to achieve desired goals. Then, section IV outlines performed experiments and presents obtained results followed by a discussion and interpretation in section V. The paper finally lands on conclusions in section VI and suggests possible future work.

II. BACKGROUND

Games can be considered a sequential decision making problem, where at each timestep, an agent observes a state and chooses to take an action. Once a terminal state is reached, an episodic reward is provided, which is intended to be maximised. This is basically how a Markov Decision Process (MDP) is defined. Furthermore, a set of available actions in a given state is called a policy. The main problem of MDPs is to discover such a policy, more specifically the optimal one, which maximises the discounted cumulative

[1]ID: 1802490, dm18389@essex.ac.uk
GitHub: https://github.com/dmachlanski

reward. In MDPs it is assumed that the state of the game is accessible, however, this is not always the case. For instance, humans playing video games are usually provided with a screen capturing some part of the game only, whereas the rest of the information remains hidden in the game's memory. This situation is defined as a Partially-Observable Markov Decision Process (POMDP).

The project at hand aims to solve MDPs from imitation learning perspective. In other words, an apprentice (DT in this case) tries to mimick the expert policy provided, or more specifically, find a function approximation between states and optimal actions. An expert can be a human performing a task or a planning-based agent, such as MCTS. Through expert play a set of game states is generated together with actions taken, which the apprentice learns to predict. The Expert Iteration (ExIt) approach goes one step further by improving the expert every iteration once the apprentice learned from previously generated dataset. The details of ExIt are separately discussed in section III.

The related work can be divided into three categories. Model-free agents are those tackling POMDP problems as they usually cannot access state information directly. Planning-based methods take advantage of accessible states and simulate any possible move to choose the best one. Finally, the last approach attempts to combine the two above with the hope of retaining strengths of both methods. This is the closest approach to the one incorporated as part of this project.

A. Model-free agents

Due to partial observability issue, these agents cannot access the state of the game. One way of dealing with that is to provide an agent with hand-engineered features of past frames. Model-free RL methods are then used to learn policies as a function of those features. This was done in [15], where SARSA algorithm was provided with a set of hand-crafted features. HyperNEAT-GGP [16], on the other hand, utilised evolutionary algorithms to build a policy search based agent. Ultimately, due to deep learning advancements, [10] developed DQN, an extended version of Q-Learning incorporating CNN for function approximation. It was considered a huge success as it delivered an end-to-end solution not requiring manual feature engineering. It is also worth noting that they used four last frames as input to compensate partial observability issue. Despite this victory, model-free methods were still lagging behind with regards to game scores obtained when compared to planning-based agents.

B. Planning-based agents

These methods, unlike model-free ones, are able to access full state information and hence simulate internally future moves and plan accordingly. One of the most popular algorithm, perhaps due to remarkable AlphaGo success [17], is Monte Carlo Tree Search. It builds a tree of consecutive states starting from the current one and chooses the most promising action. MCTS applied to the game of Hex has been presented in [18], where the MoHex algorithm won a gold medal at the 2009 Computer Olympiad. These techniques however never build an explicit policy, forcing them to build the state tree every time a decision has to be made. As this operation is computationally expensive, tree search methods are usually unusable for real-time playing.

C. Combining the two

As both aforementioned approaches have their disadvantages, such as not the best game scores of model-free agents and long execution time of planning-based ones, another idea for solving game players has been proposed. More precisely, to combine them in such a way as to retain their strengths. Therefore, the goal is to retain low computational needs of model-free solutions and exploit the data generated by planning-based ones. In addition, when utilising deep learning, there is no need to manually prepare input features, which makes this combination a powerful end-to-end algorithm. The project at hand, although not incorporating DL architectures, falls into this category as it aims to combine fast decision trees with accurate monte carlo tree search algorithm.

One of early works in this area is described in [19], where a general agent for Atari games was built. Motivated by promising DQN results [10], the authors noticed the opportunity of taking this one step further by utilising offline MCTS. They outlined three different methods in terms of combining UCT, an MCTS algorithm, with a neural network. First two, UCTtoRegression and UCTtoClassification, trained a regressor and a classifier respectively on a whole dataset of UCT-agent runs, which unfortunately did not perform very well. In contrast, the third method, called UCTtoClassification-Interleaved, trained the CNN classifier only on initial portion of the data. Then, the pre-trained network was used to suggest new actions during next games, results of which were fed again into the CNN. This expert improvement step turned out to be crucial and enabled method 3 to outperform the other two. In their other work [20], the authors also explored the idea of improving UCT agents by adapting the policy-gradient for reward-design, which was motivated by computational limitations of MCTS methods.

Later on, a different paper [11] explicitly introduces Expert Iteration (ExIt) algorithm, which is proposed as an extension to Imitation Learning (IL) by adding the expert improvement step. To some extent, it resembles the aforementioned UCTtoClassification-Interleaved approach. However, they pointed out that learning simply the moves of MCTS, referred to as chosen-action targets (CAT), is sub-optimal and hence proposed a tree-policy target (TPT), which, due to its cost-sensitivity, enables the agent to trade off its accuracy on less important cases.

In the meantime, similar ideas were explored by Deep Mind. They undoubtedly reached an important milestone in AI history by beating the best human player in the game of Go [17]. The winning agent, however, apart from incorporating MCTS algorithm, was also exposed to expert moves provided by professional players. This was done only

in the early stages of the training but certainly had a major effect on final performance. On the other hand, its successor, AlphaGo Zero, was finally trained tabula rasa via pure self-play massively outperforming AlpahGo by winning 100:0 [12]. Not only the result is impressive but also the fact that AlphaGo Zero required far less computational resources when compared to AlphaGo and reached superhuman performance just after 36 hours of training, instead of several months, like it was the case with AlphaGo. The technique used in AlphaGo Zero is almost identical to the one from the previous paragraph. It utilises a neural network to guide MCTS in order to build even stronger experts.

III. METHODOLOGY

This section describes the methods incorporated as part of this work, some of which are separately discussed in the hope of clearer explanation. It starts with a brief description of OXO game, which is the chosen game environment. MCTS techniques are also discussed as it is the core element of Expert Iteration approach. Due to the fact that this work, unlike any other related one, uses decision trees instead of neural networks for apprentice agent, a brief presentation of DTs is also included. Then, it is explained how data were generated for the purpose of DT training and how game states are represented. Finally, the ExIt algorithm is thoroughly discussed including a proposition of combining it together with DTs.

A. OXO

OXO, also known as Tic-Tac-Toe, is a two-player, perfect information, zero-sum, strategic game. The board consists of nine cells, arranged in three rows and three columns. Each cell can be either empty or contain 'X' or 'O' symbol, denoting each player's move. First player that manages to place three consecutive symbols in a full row, column or diagonal wins the game. If, however, all the nine cells have been already filled and no winning condition was fulfilled, the game ends with a draw.

O		**X**
	O	**O**
X	**X**	**X**

TABLE I
AN EXAMPLE STATE OF OXO GAME. PLAYER 'X' WINS THE GAME.

This game is particularly suitable for this work as its states can be easily represented, for instance, as a table of numbers, which is something a decision tree algorithm can consume. Moreover, as the board constitutes of just nine cells, a search tree will not grow to unreasonable sizes and therefore the evaluation time of the planning agent will be kept under a sensible time-frame. Also, the rules of OXO are simple, making it a good starting point for the proposed solution in general.

B. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a tree search algorithm that finds optimal decisions in a given scope of problems. A comprehensive survey of these methods is presented in [22]. The general MCTS approach is divided into four main stages: 1) selection, 2) expansion, 3) simulation and 4) backpropagation. The first step, a *tree phase*, is where the tree is being traversed according to a *tree policy*, which is then followed by, usually random, expansion. The next step, a *rollout phase*, consists of following some *default policy* until a terminal state is reached. Lastly, the reward obtained in the terminal node bubbles up to the root node, updating every node on its way back.

The most popular variation of MCTS is Upper Confidence Bounds for Trees (UCT), originally introduced by [23]. It strong aspect is a good balance between exploration and exploitation by utilising a solution known from multiarmed bandit problems (UCB1), which is used as tree policy (selection). The formula is presented in the equation 1, where r(s,a) and n(s,a) relate to all rewards accumulated by an edge and the number of times it was visited respectively, while n(s) relates to the number of times a node was visited. The constant c_b is to control the amount of exploration versus exploitation, which in this project has been set to 1 for simplicity.

$$UCT(s,a) = \frac{r(s,a)}{n(s,a)} + c_b\sqrt{\frac{2\log n(s)}{n(s,a)}} \tag{1}$$

After reaching a predefined number of iterations (referred to as 'itermax' later in project), the algorithm returns the best move found, corresponding to the most visited one.

C. Decision Trees

Decision Trees are one of Machine Learning (ML) algorithms able to capture relationships within provided data. Its structure is being created during the process called tree induction. DTs can be used for both classification and regression problems, though this work uses the former in order to classify the most promising move in the game. As there are nine cells in the game board, there are nine classes to predict from. Data structures are discussed in detail in the following subsection.

Neural networks are extensively used in RL, especially the "deep" ones, hence it is worth comparing them to DTs and point out some of the key differences. First and foremost, DTs, unlike NNs, are highly interpretable as they can be visualised or transformed into IF-THEN rules. These techniques can be further used to extract knowledge from them, which in the case of this project could mean discovering new game strategies. DTs are also easier to use and set up due to them having less hyperparameters to tune and not requiring normalised data. Moreover, because of their simpler nature, they usually require less amount of data to deliver good enough accuracy. Perhaps the major drawback of DTs, when compared to deep networks, is their inability to be used in end-to-end systems. CNNs on the other hand, can

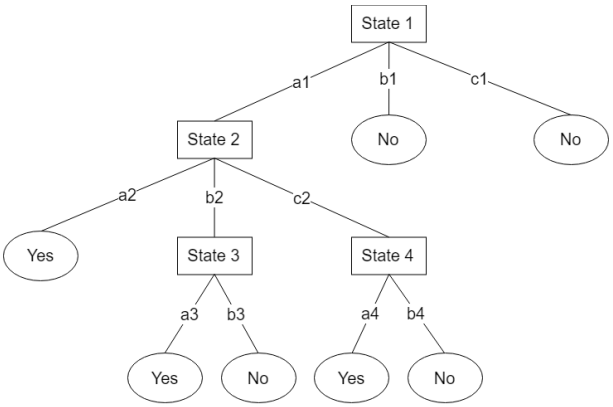

Fig. 1. An example of a simple decision tree

directly consume a screen of the game and thus not require hand-engineered features. Fortunately, the OXO game is very straightforward and hence coming up with informative representation of it is not a difficult task. Finally, according to [21], DTs tend to perform better when dealing with discrete/categorical data, which is the case here, whereas NNs rather prefer continuous ones. Overall, decision trees seem to be a promising choice for the problem at hand. The fact that recent works in RL almost exclusively focus on NNs makes this direction even more exciting.

D. Data generation and collection

The approach taken here (ExIt) aims to generate and collect new data constantly every new iteration of the algorithm, which is discussed in detail in the next subsection. Every OXO game consists of a maximum of nine states/moves (nine cells available). It has been decided to represent each state of the game as: the current state of the board, player number taking the move and the move taken, where each board cell can contain one of three integers: zero (empty), one (player 1), two (player 2). A move taken is produced by MCTS algorithm and is denoted by the cell number. The table II presents board numbering chosen. Thus, a move of "4" means a player placed its symbol at the centre of the board.

0	1	2
3	4	5
6	7	8

TABLE II
THE GAME BOARD PRESENTING EACH CELL'S INDEX

In order to produce an initial set of states, 10,000 games were played, where each player used MCTS agent with equal amount of iterations (1,000). As pointed out in [11], correlations between states might be detrimental to the learning process and effectively reduce dataset size. To counteract such situation, it is advised to collect only one state from each game. Thus, after playing 10,000 games, a set of 10,000 states was collected and stored as CSV file, a few records of which is presented by table III.

#	C0	C1	C2	C3	C4	C5	C6	C7	C8	P	M
0	2	0	1	1	1	0	2	0	0	2	5
1	2	1	2	2	1	1	1	2	0	1	8
2	1	0	0	0	0	0	0	0	0	2	4
3	1	0	0	0	0	0	0	0	0	2	4
4	0	0	0	0	0	0	0	0	0	1	4

TABLE III
EXAMPLE OF DATA COLLECTED FROM A FEW GAMES (CX - CELL NUMBER X, P - PLAYER, M - MOVE)

In addition, it is also possible to perform a simple data augmentation, where each cell's state and player's value can be changed from 1 to 2 and vice versa. As a bonus, this would also ensure the decision tree can learn both player's perspectives equally good.

E. Expert Iteration

Expert iteration approach utilises many clever ideas, but the combination of Imitation Learning (IL) and expert improvement step is perhaps the crucial one. IL essentially introduces the concept of an expert presenting the best moves and an apprentice that tries to mimic its teacher. This simple idea is very powerful, but only when an actual, highly performing, expert is available to learn from. The expert improvement step is intended to solve this problem by using the apprentice to guide the search process and eventually establish even stronger policies. As demonstrated by AlphaGo Zero [12], this technique can produce agents performing at superhuman level without any human intervention.

In this particular setting here, the goal is to use MCTS agent as the expert, whereas a decision tree serves the purpose of the apprentice. DT model is helpless without first training it, thus for the first iteration a pure MCTS approach is used to generate some samples, which can be then fed into the learning model. Once trained, the apprentice can suggest promising moves so the expert explores these directions more likely. In particular, DT model is used in the rollout phase of UCT so it builds simulations in favourable directions instead of randomly. To keep exploration to some extent, these moves are still chosen at random but only 10% of the time. Then, more games are played using the combination of MCTS and DT and producing another set of data, which are fed into DT model again. This cycle continues as long as possible until reaching satisfactory point. As decision trees cannot learn incrementally, they start over from scratch every iteration. To help them accumulate knowledge overtime, every new data set generated is appended to the previous one. The algorithm 1 presents general steps at a high level.

It is also worth noticing that [11] suggests learning to predict simply the moves taken by MCTS agent might be not enough. Instead, they propose a cost-sensitive approach, where the apprentice is less severely penalised for misclassifications when the expert is also less certain about the next move. This is definitely something to keep in mind if the standard approach fails.

1) Generate data using pure MCTS;
2) Train DT model on updated data;
3) Generate new data with MCTS + DT;
4) Augment data by swapping values (optional);
5) Append new data to the original dataset;
6) Repeat steps 2-5 for as many iterations as possible;

Algorithm 1: DTs for ExIt

IV. EXPERIMENTS

There are two types of experiments that were performed to see whether ExIt algorithm works as expected in combination with DTs: a) playing against previously created agents and b) playing pure DT agent vs pure MCTS one. Regardless of the experiment, however, there were some common settings shared among all of them to make the results consistent across the project and hence overall analysis easier to perform.

The algorithm as a whole was run for 200 iterations, generating 1,000 game states per each iteration, which means 200,000 data records plus 20,000 (10,000 doubled by data augmentation) entries of initial data produced separately via pure MCTS play. Every algorithm iteration also produced new decision tree model that was further trained on current dataset and used as the apprentice in the ExIt setting. All the models produced over the course of running the program were kept in memory to use them later for experiments purposes. Furthermore, every 10 iterations of entire algorithm, a set of experiments was performed, results of which were again stored in memory to finally export them to external files for further analysis before program termination. To explore the problem more thoroughly, the program as a whole was run 4 times with slightly different setting. More precisely, the variable changing across those runs was the number of iterations the MCTS agent was allowed to use for internal simulations before selecting the best move, further referred to as 'itermax'. The reason for this was to explore how the number of those iterations affects MCTS behaviour and the quality of data generated, which in turn directly affects decision tree models that were trained on them. The following values of *itermax* were tried: 100, 200, 400 and 800.

In terms of the experiments that were run every 10 iterations, each of them shared three common aspects among each other. Firstly, each set of games was summarised by three numbers: wins, loses and draws. These statistics were then combined to a score value to make the collected results easier to interpret and present. The score formula simply gave one or half of the point for each game won or drawn respectively. Secondly, it matters which player starts a game, because the one making the first move is in the advantageous situation of picking the best cell and hence has a higher chance to win. To eliminate such a bias from experiments, it was imperative to provide both players with equal chances (50%) of starting a game. Thus, the starting player was always picked randomly before every match. Lastly, unlike data generation games, the ones played for experiments purposes ended as soon as a player won a game and so there was no need to wait for the game board to be entirely filled and hence finish the game faster.

From the hardware point of view, all tests were run on AWS EC2 c5.large machines with Linux on-board. Time required varied from 2.5 to 20 hours depending on *itermax* value currently tested.

Apart from running experiments, it was also possible to analyse decision tree models, once all the data were generated. Some of the interpretable aspects of them are usually feature importance and tree graph, both of which may help to understand what the model has learnt from the data.

A. *Latest vs past*

This type of experiment was concerned about the latest expert performance compared to previously generated ones, where an expert is a combination of a decision tree model, trained on data produced so far, and MCTS algorithm. Every test run, a set of 10 past experts was sampled from a collection of all created ones to that point, playing 10 games with each of them against the latest expert and giving 100 matches per each experiment iteration. Therefore, the maximum amount of points an expert was able to get in each iteration was 100.

As aforementioned, the entire algorithm was run four times with different *itermax* values: 100, 200, 400 and 800. This means that not only the data were generated by agents using currently tested *itermax* value, but also the experts competing against each other as part of this trial were using the same number of iterations internally. In other words, if agents generating data were provided with *itermax=100*, the experts competing in this trial also used *itermax=100*.

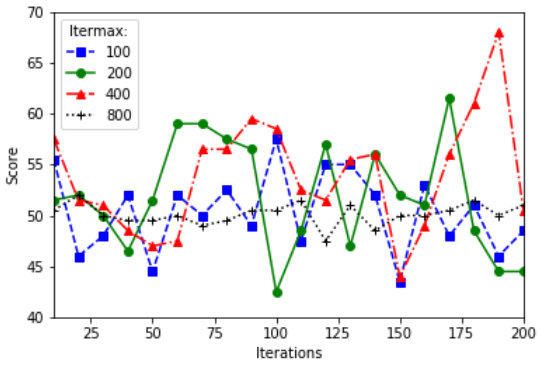

Fig. 2. The performance of the latest expert agent when competing against its past self and the influence of itermax value. Score=100 is the maximum. The chart is zoomed-in a bit for convenience.

As presented by figure 2, changing *itermax* does affect agents behaviour and their potential score. It is also worth noticing that getting a score of 50 out of a 100 games likely means that most of the games ended with a tie. Thus, *itermax(800)* agents hardly improved over time as their

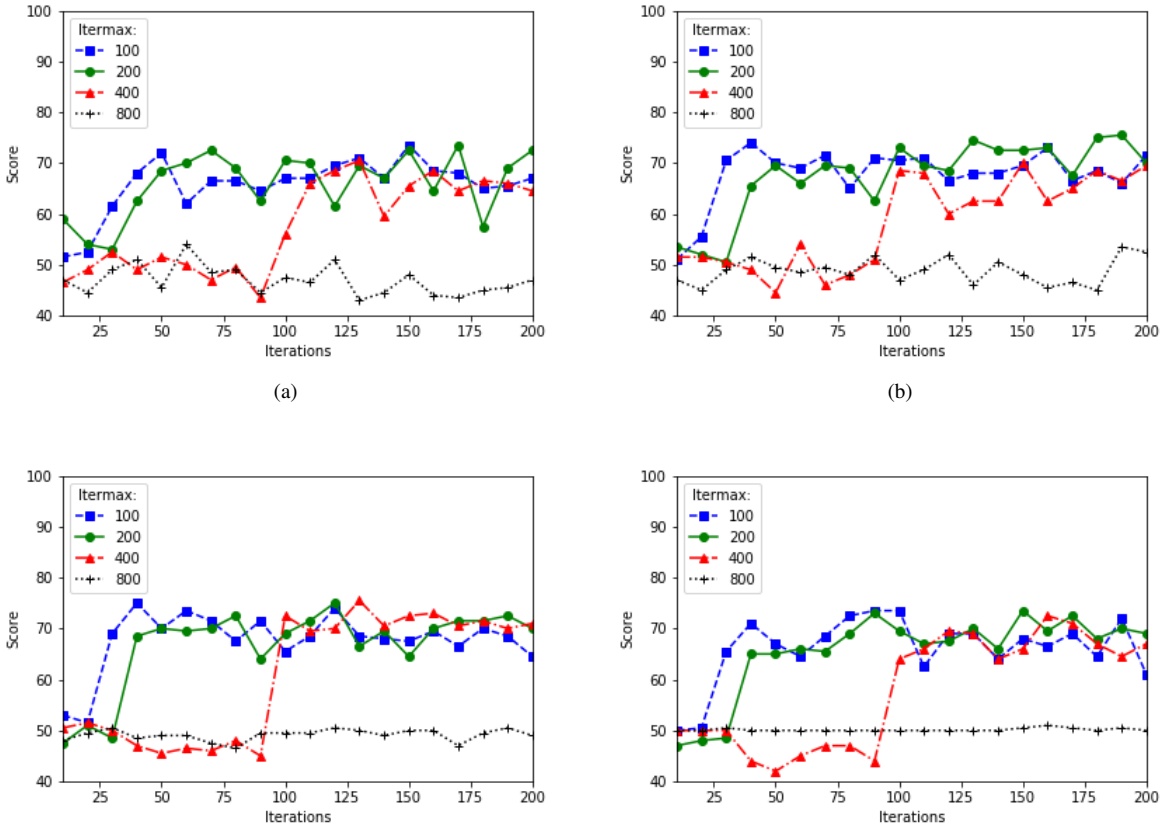

Fig. 3. Decision tree models trained on data generated by experts with various itermax values and competing against MCTS agent with itermax set to: (a) 100 (b) 200 (c) 400 and (d) 800.

score was always somewhere near 50. It appears that other cases delivered more interesting results, especially values 200 and 400 as they were able to reach a score of 60 and more, which clearly indicates some improvement in agents development. It might be not so apparent initially why score values fluctuate so much, making the analysis quite difficult certainly, but there is an explanation for that. Firstly, it can be observed that almost every peak in score is followed by a sharp decrease. Secondly, assuming agents were improving over time, both the number of difficult opponents and their capabilities also grew in time, making it more and more difficult for the latest expert to win games. As a consequence, the most recent expert was able to outperform its past self only for a while, only to compete with its almost-best-self in the next iteration.

Another outcome of this experiment is that MCTS agents are sensitive to the allowed number of internal simulations (itermax) and hence picking the right one is a crucial task. There are two possible interpretations of how exactly *itermax* affects agent's score. First of all, it could be the case that either too high or too low values can be detrimental to agent's performance, where low values (100) are not enough to exploit game's environment and high ones providing an agent with too much freedom, ending up with too much exploration. Therefore, values 200 and 400 seem to be just right and strike a good balance. On the other hand, the opposite interpretation could be that the higher *itermax* value, the better for agent's performance. In this case, the results of *itermax(800)* trial could be explained that those agents were already very good players and hence it was difficult for any side to win and resulting in ties. Then, agents with lower *itermax* could be perceived as less sophisticated players making more mistakes, which could explain aforementioned fluctuations in the score obtained. More on *itermax* and its influence in the next subsection.

B. DT vs MCTS

The second type of experiments involved a pure DT model and a pure MCTS agent. To keep the overall score in the same range as the previous experiment type, a hundred games were played in this setting, giving the maximum score of 100. The decision tree used here was a model trained on the data available at that moment, which were generated by expert agents with a given *itermax* value. Furthermore, each set of 100 games was played against four different MCTS agents, each of them using different *itermax* value: 100, 200, 400 and 800. As a consequence, this experiment yielded 16 sets of results, which are presented on the figure 3.

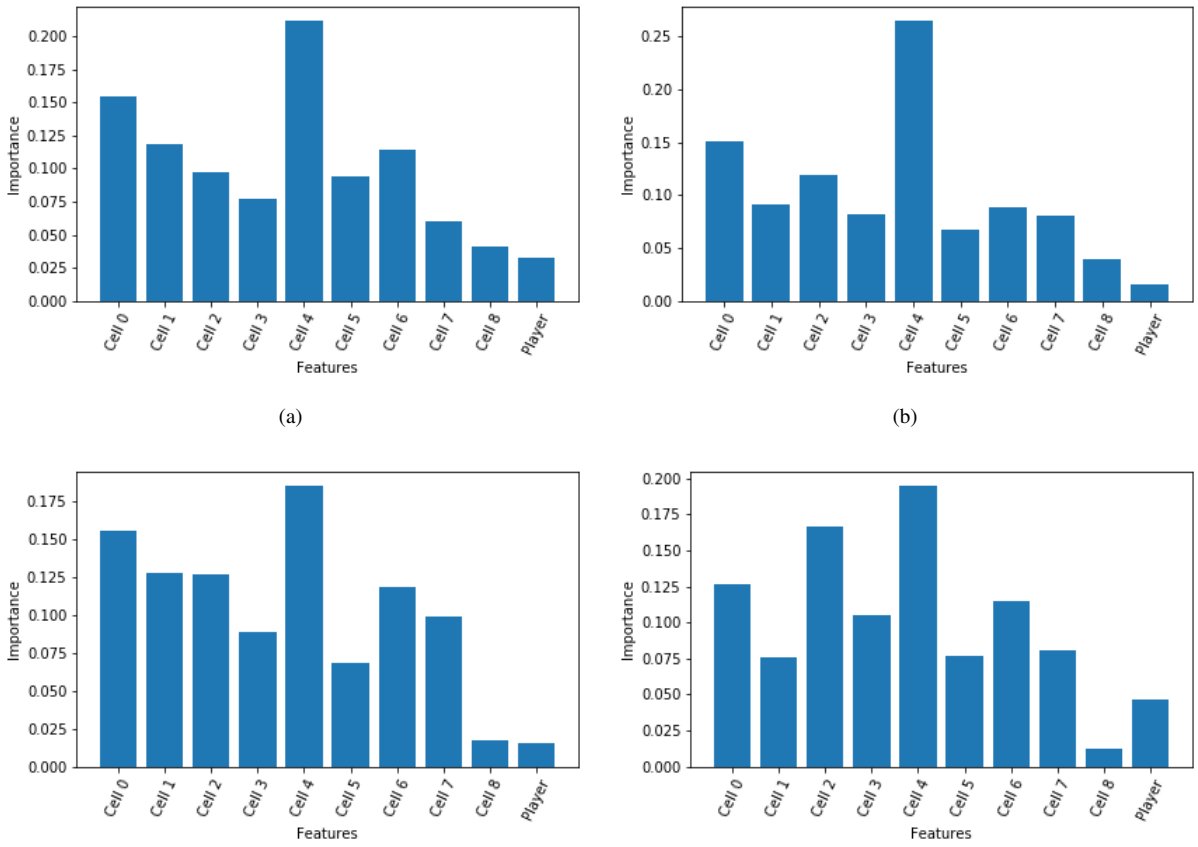

Fig. 4. Feature importance of a decision tree model trained on expert moves generated with itermax set to: (a) 100 (b) 200 (c) 400 and (d) 800.

The reason behind this kind of experiment was to see how decision tree models alone perform when compared to vanilla MCTS agents, which are considered strong players on their own. In addition, by trying different *itermax* values, it was also possible to observe whether that variable affects agent's quality, its decisions and overall difficulty as an opponent.

First of all, as presented on figure 3, some of the decision trees were able to beat MCTS agent quite often. The models trained on *itermax(800)* data performed the worst as they tied most of the time (score=50). The *itermax(100)* and *itermax(200)* cases clearly delivered the best outcomes, whereas *itermax(400)* was somewhere in the middle because of poor results until approximately 100 iterations and promising scores after that point. Another interesting observation is that MCTS agents with *itermax* set to 100 and 200, figure 3 (a) and (b) respectively, appear to be better players than the ones with 400 and 800 values as the scores obtained against the former ones tend to fluctuate more. Likewise, the scores obtained against MCTS agents with *itermax* 400 and 800, subplots (c) and (d), seem to be more stable, implying they were easier to defeat.

Another interesting outcome is the fact that a DT model trained on *itermax=100* data can easily defeat MCTS agent with *itermax=800* (figure 3 (d)), whereas a model fed with *itermax=800* examples merely ties with MCTS using *itermax=100* (figure 3 (a)). The implication is that too high *itermax* values can be detrimental to agent's performance and the quality of expert moves it generates as the DT model trained on such data clearly struggles to win.

C. Knowledge extraction

As decision trees always learn from scratch, it was possible to take all the data generated after running the entire program and train a DT on them to obtain the latest model again and see what it has learnt. This technique is sometimes useful in order to discover new game strategies or understand why a model made certain decision.

One way of extracting knowledge from a decision tree model is to examine its feature importance collection, where the higher feature value, the more important to the model it is. The results are presented on the figure 4, where each subplot refers to a model trained on different data, depending on *itermax* value. It appears that across all the models the most important feature is cell number 4, which is situated at the centre of the game board. This indeed makes sense as that cell usually puts a player in an advantageous position if they manage to occupy it. Furthermore, it is surprising that cell 8 (bottom-right corner) is almost useless as it should be no different than other three corner cells.

V. DISCUSSION

Overall, the ExIt approach appears to work well in combination with decision trees as presented by experimental results. It is fair to say that the OXO game is pretty simple and limited so even strong players have a good chance of not winning a match. Thus, it is not surprising that no player managed to win all 100 games at any time, which could also explain why DT models in second experiment were not able to surpass 70 points threshold. Experimenting with more complex games could answer the question whether this is the model's or game's limitation.

It is also very interesting to see pure decision trees able to outperform vanilla MCTS agents, meaning that DTs are capable of capturing such complex relationships within data, even in two-player games. The fact that only a standard DT model was used here makes this approach even more promising as there are more flexible tree-like solutions available, such as forests and boosted trees.

Finally, the *itermax* value of MCTS agent appears to affect its performance markedly and, as a consequence, DT results as well. Although the first experiment might not be enough to say how exactly it relates to the scores, the second one combined with feature importance information make it clear that too high *itermax* values can harm agent's performance. As presented on figure 4, it appears that models trained on data generated with *itermax* set to 100 or 200 (subplots (a) and (b)) are more concerned about exploitation and winning a game, whereas the other two seem to be more interested in exploration. This hypothesis seems to be supported by the second experiment and figure 3, where aforementioned trees (100 and 200) tend to perform better than the other two.

VI. CONCLUSIONS

In summary, it appears that decision trees can be effectively combined with MCTS via expert iteration approach to create strong players. On top of that, decision tree models proved to be capable of defeating pure MCTS on its own. Both these results make DTs an interesting alternative to neural networks in terms of reinforcement learning.

This opens up a promising direction for a future work, which is certainly needed. Firstly, trying more complex games could uncover true capabilities of DTs or eventual shortcomings. Secondly, it would be useful to see how other tree variations, like random forest, behave in this setting, though it might be computationally expensive. Lastly, formalising test outcomes could help make produced results official by incorporating, for instance, Elo rating.

APPENDIX

Full project resources, containing the code, data and experimental results, can be found here: LINK.

REFERENCES

[1] Li, Yuxi. "Deep reinforcement learning." arXiv preprint arXiv:1810.06339 (2018).

[2] Arulkumaran, Kai, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. "A brief survey of deep reinforcement learning." arXiv preprint arXiv:1708.05866 (2017).

[3] Y. Bengio. Learning deep architectures for AI. Foundations and trends in Machine Learning, 2(1):1127, 2009.

[4] J. Schmidhuber. Deep learning in neural networks: An overview. Neural Networks, 2014.

[5] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2012, pages 36423649. IEEE, 2012.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems, 2012.

[7] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2013, pages 66456649. IEEE, 2013.

[8] H. Lee, P. Pham, Y. Largman, and A. Y. Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In Advances in Neural Information Processing Systems, pages 10961104, 2009.

[9] A. Mohamed, G. E. Dahl, and G. Hinton. Acoustic modeling using deep belief networks. IEEE Transactions on Audio, Speech, and Language Processing, 20(1):1422, 2012.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with deep reinforcement learning. In Deep Learning, Neural Information Processing Systems Workshop, 2013.

[11] Anthony, Thomas, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In Advances in Neural Information Processing Systems, pp. 5360-5370. 2017.

[12] Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert et al. Mastering the game of Go without human knowledge. Nature 550, no. 7676 (2017): 354.

[13] Pyeatt, Larry D., and Adele E. Howe. "Decision tree function approximation in reinforcement learning." In Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models, vol. 2, no. 1/2, pp. 70-77. 2001.

[14] Uther, William TB, and Manuela M. Veloso. "Tree based discretization for continuous state space reinforcement learning." In Aaai/iaai, pp. 769-774. 1998.

[15] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: an evaluation platform for general agents. Journal of Artificial Intelligence Research, 47(1):253 279, 2013.

[16] M. Hausknecht, P. Khandelwal, R. Miikkulainen, and P. Stone. HyperNEAT-GGP: A hyperNEAT-based Atari general game player. In Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference, pages 217224. ACM, 2012.

[17] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016a). Mastering the game of go with deep neural networks and tree search. Nature, 529(7587):484489.

[18] B. Arneson, R. Hayward, and P. Hednerson. Monte Carlo Tree Search in Hex. In IEEE Transactions on Computational Intelligence and AI in Games, pages 251258. IEEE, 2010.

[19] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Rree Search Planning. In Advances in Neural Information Processing Systems, pages 33383346, 2014.

[20] Guo, Xiaoxiao, Satinder Singh, Richard Lewis, and Honglak Lee. "Deep learning for reward design to improve monte carlo tree search in atari games." arXiv preprint arXiv:1604.07095 (2016).

[21] Kotsiantis, Sotiris B., I. Zaharakis, and P. Pintelas. "Supervised machine learning: A review of classification techniques." Emerging artificial intelligence applications in computer engineering 160 (2007): 3-24.

[22] Browne, Cameron B., Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. "A survey of monte carlo tree search methods." IEEE Transactions on Computational Intelligence and AI in games 4, no. 1 (2012): 1-43.

[23] L. Kocsis and C. Szepesvri. Bandit Based Monte-Carlo Planning. In European Conference on Machine Learning, pages 282293. Springer, 2006.