Assignment 2 - Report

Jose Zavala, *Student, University of Essex*

***Abstract*— Reinforcement Learning algorithms used for sequential decision-making problems have been one of the main subjects of study in the latest years, however, the proposed solutions rely on Deep Learning algorithms that are usually computationally expensive. This report proposes a lighter implementation of the state-of-the-art Reinforcement Learning algorithm *Expert Iteration* (ExIt), which has been demonstrated to train agents capable of playing complex games such as Hex and Go. Our implementation aims to use a Decision Tree classifier to learn the apprentice policy with the purpose of training an agent capable of playing the simpler game of Tic-Tac-Toe. We use an online version of ExIt that, contrary to suggestions by previous work, creates a new 'small' dataset for each iteration of the algorithm, avoiding the creation of an ever-expanding move-set. The proposed implementation was able to successfully learn to play the target game, as well as creating an agent with a win-rate 58% and a no-loss-rate of 95% against the original *Expert* with only 100 iterations of the algorithm.**

I. Introduction

THIS project aims to implement a light version of the Expert Iteration (ExIt) algorithm originally by [1] and [2] to create an agent that can successfully play the Full Information of Tic-Tac-Toe. We explore the definition and implementation of the Expert Iteration algorithm in further sections.

ExIt is based on Reinforcement Learning algorithm methodology, with the aim of creating a model that learns from self-play even trained tabula rasa [1] [2] [3]. Motivation on this problem comes from the need of creating algorithms that can emulate human-thinking based on dual-process thinking for solving task as 2-player Full Information games [1].

Following the approach by [1] and [2], we will use a MCTS as our decision making *Expert* while using a Decision Tree classifier to learn the *apprentice policy*. This aims to demonstrate that the ExIt algorithm is robust enough to work with 'simpler' *apprentices* than the state-of-the-art Deep Neural Network solutions.

The motivation behind using Decision Trees is that they have been demonstrated to imitate the used dataset for simple classification problems with categorical variables [4].

This report is structured to first show the theoretical background of the problem to be solved and the proposed solutions. Then, it describes our implementation and contrasts it to the state-of-the-art ExIt solutions. Lastly, we define the experiments to be conducted, analysis of the obtained results, and considerations for future expansion of this report.

II. Background

The famous Tic-Tac-Toe, or OXO, is a "Full Information" game, which refers to the fact that all players have complete knowledge of the state of the match at any given point in time, unlike games like "Poker", in which certain information (i.e. the hand of a given player) is unavailable [5] [6]. Other examples of this type of games are Chess, Go, and Checkers.

A. Tic-Tac-Toe Knowledge

The size of the board for a game of Tic-Tac-Toe is 3x3 and each slot can have any of three symbols ('X', 'O', *empty*) at any given time, so intuition indicates that there are 3^9 or 19,683 possible game states [6]. However, these are limited by both conditions that restrict the feasibility of a game state of occurring (Figure 1) and the fact that by rotating the board some game states are equivalent (Figure 2) [6].

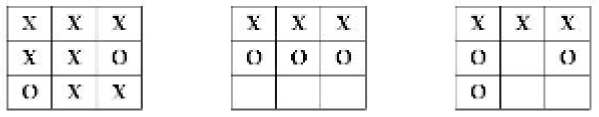

Figure 1 - Infeasible game-states [6]

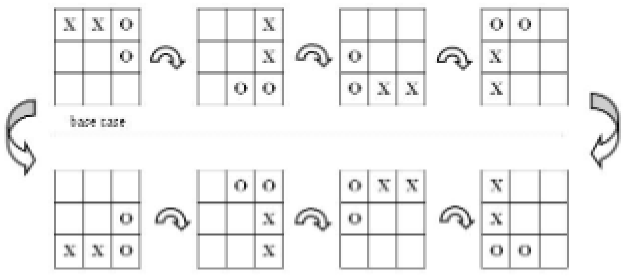

Figure 2 - Equivalent game-states [6]

These conditions reduce the previous calculated number to 765 unique game-states [6].

B. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a commonly used technique for the creation of agents capable of playing these types of games [2] [7].

MCTS uses a variation of the Upper Confidence Bounds (UBC) algorithm, called UCB applied to trees (UTC) [8]. This multi-armed bandit approach takes the action that takes to a node which maximizes Equation 1 for the current state [8]. Where w_i is the number of wins after visiting the ith node, n_i is the number of simulations after the ith node and N_i is the total number of simulations ever considered [8].

$$\frac{w_i}{n_i} + C\sqrt{\frac{\ln N_i}{n_i}}$$

Equation 1 - UCT Node Selection [8]

This allows the algorithm to explore the possible scenarios of different games just by correctly encoding its mechanics into the algorithm, which demonstrates the effectiveness of MCTS for tracking future game states and predicting the optimal move based on that information [2] [7]. However, its implementation is inherently time and resource expensive since it needs to recreate a full tree in each turn since the game state changes with every move. Also, the random nature of the algorithm allows to overlook specific game paths with low representation that may result on a loss, since these may be the consequence of "weird" plays.

MCTS has been used for agents capable of playing turn-based board games, such as Hex [9] and Go [1] [2], due to its capability of lookout for multiple future scenarios using the default random policy or enhanced policies based on successful strategies [9]. MCTS is also frequently used since it is capable of playing its target game successfully immediately after deployment without the need of any training based on expert play given that it was implemented with knowledge of the game's mechanics [1] [2] [7] [9].

The success of MCTSs in more complex games such as Go can be observed in a recent model, referred to as *Alpha Go Lee*, that uses two networks: A Policy Network that assigns probabilities to moves trained first by supervised expert play and subsequently refined by policy-gradient reinforcement learning, and a Value Network that outputs a position evaluation trained by reinforcement learning to predict the winner of games of the Policy Network against itself [2]. These two networks were combined with a MCTS to provide a lookahead search, using the Policy Network to narrow down the search of high-probability moves and the Value Network to evaluate positions on the tree [2]. This model went as far as defeating the world champion of Go, Lee Sedol, back in 2016.

C. Expert Iteration

Expert Iteration (ExIt) is an algorithm first introduced in [1]. It is a Reinforcement Learning algorithm based on the dual-process theory that explains that humans have two systems of thought: System 1 focus on automatic and unconscious thought, also known as intuition; while System 2 is a slow, conscious, explicit and rule-based mode of *reasoning* [1]. ExIt makes use of MCTS as an analogue of System 2, assisting the training of a neural network that works more similarly to System 1 [1].

To understand ExIt, we first need to introduce Markov Decision Processes and Imitation Learning:

Markov Decision Processes (MDPs) aim to generate *policies* for optimal play in a two-player, Full Information, zero-sum games that can maximize the rewards for the respective player [1]. A *policy* $\pi(a|s)$ is formed by taking an action a in a given state s. It uses a Value Function $V^{\pi}(s)$, that denotes the reward obtained after following policy π in a given state s. In a Reinforcement Learning problem, in which we do not know all rewards, the function $Q^{\pi}(s,a)$ is introduced, that defines the value of taking action a in a state s and then following the optimal policy afterwards [1].

Imitation Learning attempts to solve MDP by mimicking an existing *expert* policy π^*, which is provided by an expert [1]. The policy learnt through imitation is called the *apprentice* policy, and it needs a dataset of states of expert play with the targets provided by the expert in order to attempt to predict it [1]. The two solutions are: Direct action prediction by imitating optimal move provided by the expert $\pi^*(a|s)$, imitating the expert; Estimate the action-value function $Q^{\pi^*}(s,a)$ and act in a greedy selection algorithm for the optimal move by a trade-off of prediction errors and their cost [1].

ExIt is built upon Imitation Learning by implementing an expert improvement step. The following is a description of the algorithm.

```
$\hat{\pi}_0 = \text{initial\_policy}(\ldots)$
$\pi_0^* = \text{build\_expert}(\hat{\pi}_0)$
$\boldsymbol{for}\ i = 1; i \leq \max\_iterations; i++\ \boldsymbol{do}$
  $S_i = sample\_self\_play(\pi_{i-1}^*)$
  $D_i = \{(s, imitation\_learning\_target(\pi_{i-1}^*(s))) | s \in S_i\}$
  $\hat{\pi}_i = train\_policy(D_i)$
  $\pi_i^* = build\_expert(\hat{\pi}_i)$
$\boldsymbol{end\ for}$
```

Figure 3 - Expert Iteration Algorithm [1]

At each iteration i*, the algorithm proceeds as follows: we create a set* S_i *of game states by self-play of the apprentice* $\hat{\pi}_{i-1}$*. In each of these states, we use our expert to calculate an Imitation Learning target at* s *(e.g. the expert's action* $\pi_{i-1}^*(a|s)$*); the state-target pairs (e.g.* $(s, \pi_{i-1}^*(a|s))$*) form our dataset* D_i*. We train a new apprentice* $\hat{\pi}_i$ *on* D_i *(Imitation Learning). Then, we use our new apprentice to update our expert* $\pi_i^* = \pi^*(a|s; \hat{\pi}_i)$ *(expert improvement).* [1]

This algorithm was implemented by [1] using a Neural Network as the *apprentice* to play the game of Hex with successful results. A similar implementation of this algorithm was used by the *AlphaGo Zero* team in [2], which used a Deep Neural Network for the *apprentice* to learn to play the game of Go. This version of the *AlphaGO* agent surpassed the previously mentioned *AlphaGo Lee,* winning 100-0 matches against it.

D. Decision Trees

In Machine Learning, a Decision Tree (DT) is a hierarchical tree model for supervised learning, in which recursive splits determined by the value of a certain *attribute* of the provided example conduct to a leaf node that contains a classification [4].

Different implementations of the Decision Tree algorithm exist in the literature, namely Classification and Regression Tree (CART), Iterative Dichotomiser 3 (ID3), C4.5, and C5.0 [4] [10]. All of these algorithms have the same basis for tree construction, which is to recursively split a node based on the values of a feature and their relation to the classified class in the training examples [4]. When no further splits can be performed, the most represented class in the remaining examples is set as the classification in the leaf node [4].

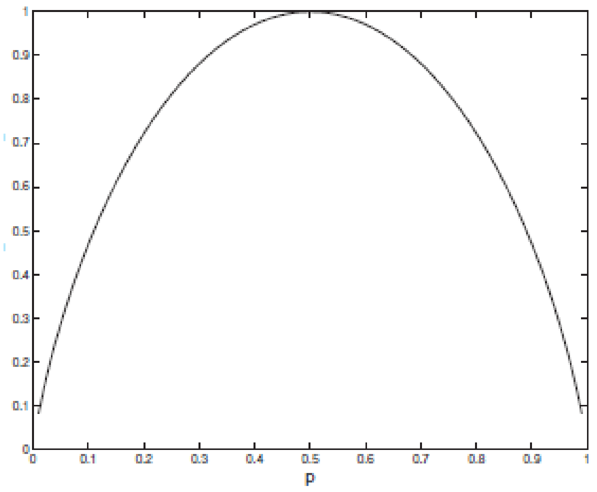

Figure 4 - Entropy Function for a two-class problem [4]

$$I_m = -\sum_{i=1}^{K} p_m^i \, log_2 \, p_m^i$$

Equation 2 - Entropy Function [4]

$$I'_m = -\sum_{j=1}^{n} \frac{N_{mj}}{N_m} \sum_{i=1}^{K} p_{mj}^i \, log_2 \, p_{mj}^i$$

Equation 3 - Entropy Function for Attribute [4]

$$Inf_{gain} = I_{class} - I_m$$

Equation 4 - Information Gain [4]

To determine the splits of the tree, the entropy function will be used to determine Information Gain [4]. Other metrics for node splitting exist, such as Gini Index, but Information Gain was selected for its simplicity of use and because it favors unique values in a split [4]. The Entropy Function (Equation 2) allows us to know the homogeneity of the dataset for a given class, where p^i_m is the probability of classifying for class i. For a binary classification problem, this equation is graphed in Figure 4, where while probability of one given class increases, the probability of the second decreases; this causes a peak of entropy when both classes are equally represented. The more homogeneous it is, the more relevant it becomes to know which action to take [4]. To calculate the entropy of the dataset when a given feature (a cell in this case) is selected, Equation 3 is used, where n represents the number of attributes for a given features and K the number of classes. This equation let us get a weighted mean for the entropy of the dataset if it was discriminated by a given feature. The Information Gain takes the Entropy of the dataset (Equation 2) and the Entropy of the dataset if a split was to be determined by a given feature (i.e. cell in the board) (Equation 3). The feature that gives the most Information Gain is selected to split the node at the given game state.

This algorithm permits DTs to create a set of recursive rules that distinguish which features are more relevant in higher stages of the decision process. Also, they provide an efficient model for predicting decisions based on the whole provided context. The build methodology also synergizes well with the MCTS algorithm, since the leaf nodes predict for the actions that are most popularly represented by the available examples [4].

III. Methodology

The goal of this project is to implement the Expert Iteration algorithm using a Decision Tree classifier as the *apprentice* policy and a Monte Carlo Tree Search as the improved *expert*, to then evaluate the performance for learning to play the game of Tic-Tac-Toe.

A. Dataset Collection

To gather data to train the *apprentice policy*, the MCTS implementation provided by the Monte Carlo Tree Search Research Hub (http://mcts.ai/code/python.html) was used as the target *expert* [1]. The original code provides an implementation of the UCT algorithm for three games: Nim's Game, OXO (Tic-Tac-Toe), and Othello. The Code was modified to save the game states during a match from the perspective of the player whose current turn is being played.

Previous research mentioned in II.A indicates that there are 765 unique game states in the board at any possible time [6]. However, since neither MCTS nor DT can determine when a state is a rotation of another game state, we undo this knowledge to have a number of 6,120 game states. To be sure to capture all this, an arbitrary number of 10,000 games will be stored as to also account for the validation set. No repeated game states will be removed since they reflect the frequency the MCTS makes those choices.

To allow our *apprentice policy* to play either as "X" or "O", we encoded the board of the game depending on the current player, and not as "X"'s and "O"'s, , being "X" if it indicates a piece placed by the current player, "Y" if it was placed by the rival, and "_" represents an empty slot (Figure 7).

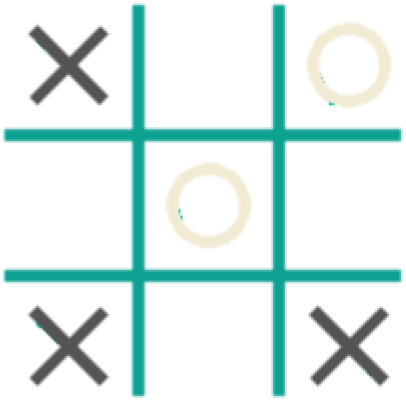

Figure 5 - Game State

1	0	2	0	2	0	1	0	1

Figure 6 - Original Encoding by MCTS Research Hub

Y	_	X	_	X	_	Y	_	Y

Figure 7 – Our Encoding for Second Player's (O) Perspective

Upon inspection of data, it was decided that every move should be stored, even from the perspective of a losing agent. This was determined since MCTS, our *expert*, will always perform the best possible move available for a given scenario [8]. In a naïve and optimistic approach, even if the agent loses, it means that no better move was found by the *expert* for that scenario [8]. Also, it is assumed that in the long run, the MCTS algorithm will chose the best moves more frequently over the non-optimal ones [2][8], which the DT implementation should capture. A representation of a possible resultant DT trained on this *expert* is given in Figure 8.

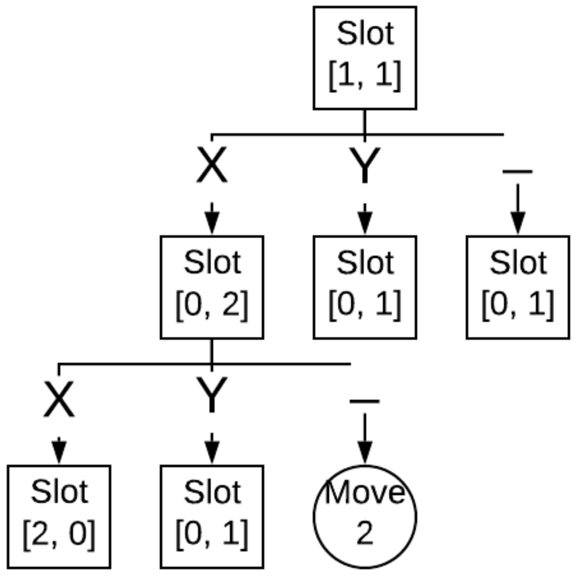

Figure 8 - Decision Tree Representation for State in Figure 5 from Second Player's (O) Perspective with the new encoding

B. *Apprentice Policy Model*

To achieve the goal, a CART with Information Gain will be used as the *apprentice policy* [1] [4] [10]. We will use the implementation provided by Scikit-Learn (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html) [11]. This implementation was selected due to its ease of use and its integration of fast data-structure libraries in Python such as NumPy.

Originally, it was set that the model would receive the state of the board as an input of 9 features (Figure 7), each one symbolizing a position, and target those results to the cell that was selected by the MCTS algorithm. However, since Scikit-Learn's DecisionTreeClassifier class does not support categorical (i.e. our initially proposed encoding) and only numerical, we used One-Hot Encoding in runtime to feed the DT an understandable feature vector, transforming it from a 9 feature vector to 27; a significant but necessary feature space dimensionality increase. While the collected dataset will remain the same, the encoding in runtime will look as in Figure 9 for each feature (i.e. slot in the board):

$$X \rightarrow [1,0,0]$$
$$Y \rightarrow [0,1,0]$$
$$\_ \rightarrow [0,0,1]$$

Figure 9 - One-Hot Encoding of the Proposed Feature Vector

This issue also occurs with the target value (i.e. player's move). Even though it is a numerical value and it's accepted by Scikit-Learn's DT implementation, the distribution of values with numerical ranges may cause problems while training as it could assign an inexistent numerical priority of some slots over others. To solve this, One-Hot Encoding was also used for the target column in runtime.

The DT was selected for our *apprentice policy* for two major reasons: First, it will be able to identify the most commonly selected plays by the *expert* for a given game state. Since the game has a very limited number of states [6], the DT should be able to determine which choices were most played overall, removing the *random policy* factor of the MCTS algorithm [8]. The second reason is that, since the DT is going to be trained to emulate the *expert* with Imitation Learning [1], and not merely to learn "how to play", the resulting tree will provide more insight about the relevance of certain cells over others at given game states.

This allows the DT to understand which cells are more important for the MCTS algorithm in a given situation having enough training examples to choose from. To "understand" the importance of a cell is to learn which cell's states possess less entropy as to correctly discriminate the next action taken. At a leaf node, the action taken is the one with the most representation available, which statistically should be the most effective move performed by the target MCTS [8]. This is useful if we further need to manually analyze the *apprentice policy* to understand its decisions.

As a last note, it is important to note that Decision Tree algorithms **work poorly on unseen examples** [4]. Since our *apprentice* is trained using only the *expert* play, moves generated by the *Expert* Tree algorithm may not always be optimal, which may cause that the model predicts invalid moves. This is solved by observing the prediction during play and selecting a random available move if an invalid move was predicted.

C. *Apprentice Training*

To ensure that the Decision Tree implemented is a successful *apprentice policy* for our model, we need to train and evaluate its performance on imitating the *expert*. To ensure we are training the best model for our *apprentice*, we are going to evaluate for different metrics using 10-Fold Cross Validation, with the GridSearchCV method provided by

[0:0]	[0:1]	[0:2]	[1:0]	[1:1]	[1:2]	[2:0]	[2:1]	[2:2]	Move
_	_	_	_	_	_	_	_	_	4
_	_	_	_	Y	_	_	_	_	0
Y	_	_	_	X	_	_	_	_	2
X	_	Y	_	Y	_	_	_	_	6
Y	_	X	_	X	_	Y	_	_	3
X	_	Y	Y	Y	_	X	_	_	5
Y	_	X	X	X	Y	Y	_	_	1
X	Y	Y	Y	Y	X	X	_	_	7
Y	X	X	X	X	Y	Y	Y	_	8

Table 1 - Capture of a single MCTS game using the proposed encoding

Scikit-Learn (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) [4] [11].

The metrics to be evaluated are Split Criterion and Maximum Tree Depth. Split Criterion will compare between Gini and Information Gain, in order to solidify our selection of Information Gain to split the nodes in the DT; Maximum Tree Depth will ensure that overfitting is not an issue for our classifier to imitate the *expert policy*, this is done by pre-pruning the tree when a maximum depth is reached [4].

To determine the best parameters, we will only evaluate our Cross-Validation process using the accuracy score, since the only purpose of the *apprentice policy* is to faithfully imitate the original MCTS, a.k.a. our *expert* [1].

D. Expert Model

As stated in [1], the role of the *expert* is to perform exploration and accurately determine strong move sequences from a single position. This makes tree search algorithms the canonical choice for the *expert*. We decided to use MCTS as our *expert* for this project based on the work done by [1] and [2], which had successful results in the past.

Since MCTS uses repeated game simulations to estimate the value of a states, expanding the tree further in more promising lines and then selected the **most repeated** play, we are confident that it will synergize well with our selected *apprentice* policy, described in the previous section.

The implementation of the MCTS algorithm is heavily based on the code provided by the Monte Carlo Tree Search Research Hub (http://mcts.ai/code/python.html), with minor modifications to add our *apprentice* policy.

Reinforcement Learn will be performed following the original ExIt algorithm, with a small variation. The first iteration of the *apprentice* will be trained using only a small sample of the original dataset generated by *expert* play, rather than the whole as the literature suggests [1] [2]. Further iterations will be trained by creating a move set of N number of games between our *Expert* with the selected *apprentice policy* (DT) against the original MCTS with the default policy (random play). This was decided to prevent the new dataset of only focusing on previous optimal moves, opening the possibility of new unseen strategies to emerge.

Literature on the ExIt suggest that an online version of the algorithm aggregates all datasets generated so far at each iteration on the Imitation Learning step [1]. We chose against this since our selected *apprentice policy* will only classify the most prominent move in the dataset. Small aggregations to the *expert* play dataset on each iteration, as suggested by [1], may cause that some optimal plays are ignored since their incidence is small compared to the previous dataset. By creating a new dataset on each iteration i, we make sure that each *apprentice policy* is trained with maximum representations of the moves selected by the $i-1$ version.

To ensure that our *apprentice policy* is learning and becoming better than its previous iterations, each 10 versions of the *Expert* we will make the latest version compete with an iteration 10 steps *younger*. We will store these results for evaluation.

All models and move datasets are stored for later evaluation and to further continue training if needed.

IV. EXPERIMENTS

In this section we perform the experiments detailed in Section III to measure the performance of our ExIt performance. First, we find the correct parameters for our DT classifier (*apprentice policy*). After finding the correct parameters, we execute the ExIt algorithm and evaluate the performance of the models as new datasets are created.

Analysis will be performed by observing the accuracy of the *apprentice policy* on the selected dataset with the best set of parameters, and the win rate of the new *Expert* models over previous iterations. The former will be performed with a similar approach to the experiments by [1] and [2], however, as explained in the previous section, adversarial play will be between the ExIt model and the original *expert* MCTS (random policy) in order to not enclose the future learning sets to only previously seen moves.

A. Apprentice Parameter Tuning

The parameters to be tuned on the training of the Decision Tree are Split Criterion and Maximum Tree Depth. The two split criteria to be compared are 'Gini' and 'Information Gain'; The different tree depth values are $TreeDepth \in \{11, 13, 15, 17, 19, 21, 23, 25, 27, 29\}$.

As mentioned, this was evaluated using 10-Fold Cross-Validation on the whole initial dataset generated by the *expert* (MCTS vs MCTS).

The best mean accuracy scores across all folds were: 76.2035446% for training and 75.8903012% for testing, obtained with the combination of values **'Information Gain' (Entropy) for Split Criterion** and **13 for Maximum Tree Depth** (see Appendix VIII.A). However, as it can be seen in the table of results, this combination outperforms the rest by a negligible amount, which suggests that the resulting trees are very similar.

For the rest of the experiments we will stick to the best performing values, even if no significant advantage is visible.

B. *Expert Iteration Performance*

Expert Iteration was executed as mentioned in Section III.D. Iteration $i = 0$ of the Model was trained with a **100 moves** sample from the original dataset generated by the *expert* described in Section III.A. For $\mathbf{1 \leq i \leq 100}$, a new dataset will be generated from **1000 matches** between the most recent iteration of our *MCTS expert with apprentice policy* versus the original MCTS with random policy. The resulting dataset will be the most recent version of the *expert* policy to train the new *apprentices*. This means that a total of **100 versions** of the *apprentice policy* were generated, each one better than its predecessors.

To test the performance increase of the *apprentice policy*, each 10 iterations the latest version (i) will play **10 matches** against a version **10 iterations** younger ($i - 10$). The win rate score of the latest iteration against its predecessor will be used as the success metric for our model.

Note: The MCTS with random policy generates **1,000 iterations** per move while the *Expert's* MCTS being trained only generates **100 iterations**. This was performed this way because searching over a Decision Tree run in O(n) time (being 'n' the size of the tree) while random policy is O(1). Having a large number of iterations for our *Expert* was notoriously computationally expensive, and with the available hardware and time constraints, results would not have been achieved in a timely manner.

The results are shown in Table 2:

Iteration i	Wins	Losses	Draws
10	6	2	2
20	5	0	5
30	3	2	5
40	2	0	8
50	5	0	5
60	7	0	3
70	3	1	6
80	4	1	5
90	2	0	8
100	5	0	5

Table 2 - Match Results for Expert Against i-10 Iteration

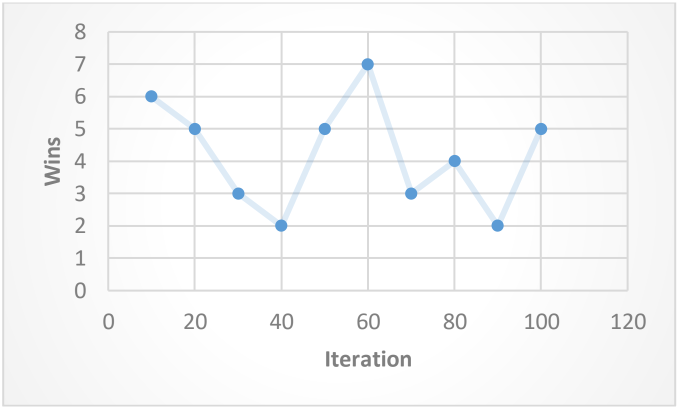

Figure 10 - Wins of the Expert over its predecessor

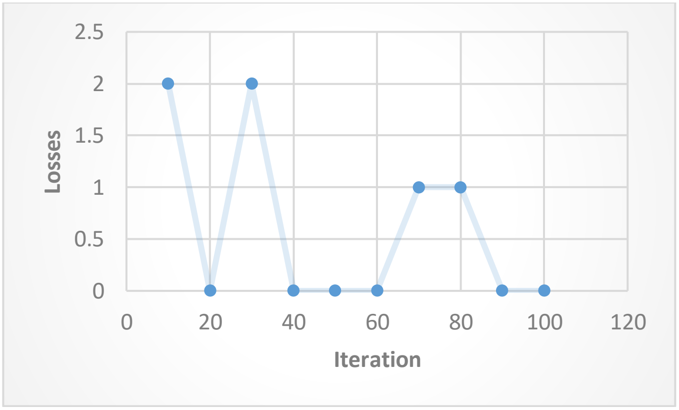

Figure 11 - Losses of the Expert against its predecessor

Tests were carried against only the most recent predecessor and not all previous iterations because under the assumption that the transitive property holds for model performance (i.e. If A wins against B, and B wins against C, A wins against C).

The number of **10 iterations younger** was arbitrarily decided since we considered it to be a good enough increase in performance to see a noticeable change in strategy.

C. *Performance Against Random Policy*

Following tests by [1] and [2], we now compare the performance of our latest model, *Model_100*, against the original random policy in which we initially trained our *apprentice policy*.

For this, we ran 100 games between our expert with the 100th iteration of our *apprentice* against the original MCTS *expert* with random policy. The results are shown in Table 3:

Wins	Losses	Draws
58	5	37

Table 3 - 100th Apprentice Against MCTS Random Policy

This means that our *Expert* using our latest *apprentice policy* has achieved a **win rate of 58%** over the original random policy, with an overall **95% no-loss rate**.

V. Discussion

A. 10-Fold Grid-Search Results

Results of parameter tuning, shown in Appendix VIII.A show that all trained DTs have an accuracy score of ~75.8% over the testing dataset, which is an expected value. A Decision Tree, as any other classification model, will always predict the same class for a given example, and since the *Expert* generated dataset provides different moves for the same game-state, our model will classify for the move that was most commonly played by the *Expert policy*.

The similarity in performance across all different selected parameters suggests that there is a threshold in the ~75% accuracy mark that will inhibit performance from going any further with the given dataset. This is satisfactory, as our *apprentice policy* can now successfully imitate the target *expert* using the most common plays for the board states.

B. Expert Iteration Results

The win rate of the latest models against its predecessors show a successful increase in the performance of the *apprentice policy* when trained on a refined number of games. Figure 10 shows a consistent number of wins between 2 and 7 across all plays, while Figure 11 shows that in 4 different iterations, the new *apprentice policy* lost a maximum of 2 matches.

Overall, the model has a no-loss rate of 94%, against previous versions, and a win rate of 72%. This is satisfactory, given the restricted number of iterations in the MCTS algorithm and the small number of games available for each version. It must be noted that our *apprentice policy* is not 100% followed, and that there exists a random factor in play. First, our implementation states that in a 10% of cases, a random play will be selected by the *apprentice*. Also, in the other 90% of the times in which a predicted move is performed, since we cannot capture all possible game-states in our 'small' dataset, the DT may suggest moves that are not available in the current state; Since this was solved by detecting these scenarios and making random moves when this occurs, we have to appoint that there may be a large random factor in our *apprentice*. Nevertheless, the high win-rate of newer versions against old ones, and the 58% win-rate of our best *Expert* with DT fueled *apprentice* against the random policy MCTS shows that our Reinforcement Learning approach was satisfactory.

VI. Conclusion

We have implemented a light version of the Reinforcement Learning **Expert Iteration** algorithm using a **Decision Tree classifier** to learn the *apprentice policy* with a MCTS as our *Expert* that was successful to train an agent capable of playing Tic-Tac-Toe. Our latest version has a successful win-rate and non-loss rate score against the original *Expert*, even after only 100 iterations of the algorithm.

The algorithm also shows a substantial increase in performance against previous versions in only 10 iterations even if only 1,000 games are stored in each dataset. This shows that an Expert Iteration using a MCTS as *expert* with a DT as an *apprentice policy* is a strong enough algorithm to learn to play Tic-Tac-Toe successfully, even being trained tabula rasa.

However, there were some hardware limitations that inhibited a more powerful model to be developed. First, the iterations performed by the MCTS are hard limited to 1,000 for the random policy and just 100 for the DT *apprentice policy*, while the literature suggest a number of 10,000 iterations for the MCTS implementation [1] [2]. Even with these parameters, it took each version (*Expert* dataset) roughly ~40 minutes, which deterred more in-depth experiments.

For future work of our solution, we plan on increasing the number of iterations in the MCTS algorithm to 10,000 for both *Experts* with random and *apprentice* polices and generate 10,000 move datasets. We can also experiment with an approach similar to [1], in which new moves are stored over the pre-existing datasets to see if this technique proves both to be more efficient in runtime and memory, and increase performance of our *Expert*.

Overall, our lite implementation of ExIt shows to be enough to train an *Expert* that is capable of playing the game of Tic-Tac-Toe to a successful degree of mastery.

VII. References

[1] T. Anthony, Z. Tian and D. Barber, "Thinking Fast and Slow with Deep Learning and Tree Search," in *31st Conference on Neural Information Processing Systems*, Long Beach, 2017.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature,* vol. 550, pp. 354-359, 2018.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv,* p. 19, 2017.

[4] E. Alpaydin, Introduction to Machine Learning, Massachusetts: Massachusetts Institute of Technology, 2014.

[5] S. Sriram, R. Vijayarangan, S. Raghuraman and X. Yuan, "Implementing a no-loss state in the game of Tic-Tac-Toe using a customized Decision Tree Algorithm," in *International Conference on Information and Automation Information and Automation*, Zhuhai/Macau, 2009.

[6] A. Bhatt, P. Varshney and K. Deb, "In Search of No-loss Strategies for the Game of Tic-Tac-Toe using a Customized Genetic Algorithm," Indian Institute of Technology, Kanpur, 2008.

[7] S. S. Sista, *Adversarial Game Playing Using Monte Carlo Tree Search,* Cincinnati: University of Cincinnati, 2016.

[8] L. Kocsis and C. Szepesv ́ari, *Bandit based Monte-Carlo Planning,* Budapest: Computer and Automation Research

Institute of theHungarian Academy of Sciences, 1999.

[9] B. Arneson, R. B. Hayward and P. Henderson, "Monte Carlo Tree Search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251-258, 2010.

[10] L. Breiman, J. Friedman, C. J. Stone and R. A. Olshen, Classification and Regression Trees, Wadsworth : Chapman & Hall, 1984.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and É. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825-2830, 2011.

VIII. Appendices

A. 10-Fold Cross Validation Grid Search Results

Criterion	Max Depth	Mean Test Score	Standard Deviation Test Score	Test Score Rank	Mean Train Score	Standard Deviation Train Score
gini	**11**	0.758204512	0.002848817	20	0.76169701	0.000264978
gini	**13**	0.758845757	0.002546861	9	0.762036718	0.000238391
gini	**15**	0.758822856	0.002553542	18	0.76203799	0.000237207
gini	**17**	0.758845757	0.002530212	9	0.76203799	0.000237207
gini	**19**	0.758834307	0.002532222	14	0.76203799	0.000237207
gini	**21**	0.758845757	0.002512764	9	0.76203799	0.000237207
gini	**23**	0.758845757	0.002550188	9	0.76203799	0.000237207
gini	**25**	0.758834307	0.002540227	14	0.76203799	0.000237207
gini	**27**	0.758868659	0.002529318	4	0.76203799	0.000237207
gini	**29**	0.758834307	0.00253756	14	0.76203799	0.000237207
entropy	**11**	0.758215962	0.002828559	19	0.761761899	0.00023678
entropy	**13**	0.758903012	0.002526869	1	0.762035446	0.000232081
entropy	**15**	0.75888011	0.002538356	3	0.76203799	0.000237207
entropy	**17**	0.758891561	0.002576225	2	0.76203799	0.000237207
entropy	**19**	0.758868659	0.002557987	4	0.76203799	0.000237207
entropy	**21**	0.758868659	0.002583134	4	0.76203799	0.000237207
entropy	**23**	0.758845757	0.002568001	9	0.76203799	0.000237207
entropy	**25**	0.758834307	0.002550511	14	0.76203799	0.000237207
entropy	**27**	0.758857208	0.002572332	8	0.76203799	0.000237207
entropy	**29**	0.758868659	0.002555207	4	0.76203799	0.000237207