Report

Pages 1–9
2023

An AlphaZero Implementation of Ultimate Tic-Tac-Toe

Hoang Tran Nhat Minh

Project 1

Hanoi University of Science and Technology
School of Information and Communication Technology

March 2023

Keywords: Graphical user interface, AlphaZero, traditional board game, ultimate tic-tac-toe, artificial intelligence, reinforcement learning, convolutional neural network

ABSTRACT

This project implements a graphical user interface (GUI) game of ultimate tic-tac-toe with many modes. The algorithmic play engine is powered by a modified version of AlphaZero, which was trained to only make use of local features by a convolutional neural network without human-extracted features and in resource-cheap prediction mode.

CONTENTS

1 INTRODUCTION

Recent advances in reinforcement learning The field of reinforcement learning has advanced rapidly in recent years. Now, at the start of 2023, the hottest topic in technology is ChatGPT [1], a state-of-the-art chatbot that can interact with us like a normal human being, debug our code, write code based on our instructions, write poems, tell stories, or even write a book in hours (which usually would take months for human authors to write) [2]. Reinforcement learning and supervised learning are used by the OpenAI team to fine-tune ChatGPT [3]. In 2022, we can mention AlphaTensor, which solved a 50-year-old mathematical question by finding the new fastest method to multiply matrices using reinforcement learning [4]. Or, back in 2020, the Google research team introduced a new machine learning framework named JAX to achieve higher performance for machine learning research [5], which DeepMind mainly used to accelerate theirs [6], especially for reinforcement learning.

Environments of traditional board games Reinforcement learning aims to learn an optimal policy that maximizes a predefined reward function [7]. There are various types of environments in reinforcement learning [8], and the simplest to work on is the discrete, deterministic, single-agent, episodic, and fully observable environment. Most traditional board games, from easiest to hardest, have three out of the five properties listed above, and the differences are that their environments are multi-agent and sequential instead of single-agent and episodic. Those differences are a game changer since they can increase the difficulty of a game to any degree. Popular board games that have discrete, deterministic, multi-agent, sequential, and fully observable environments are tic-tac-toe, checkers, chess, connect four, reversi, and Go, the most difficult board game in history [9].

Reinforcement learning in traditional board games As traditional board games have fairly simple rules but are difficult to play well, researchers have developed many systems that try to play them in a nearly perfect manner. We know that the bots are nowhere near perfect [10]; they have a huge room for improvement, but already all of the best bots play a lot better than any professional player.

Garry Kasparov, regarded as the best chess player of all time [11], lost 3½-2½ to Deep Blue, a chess computer developed by IBM; and this match in 1997 showed a sign that artificial intelligence was starting to overtake human intelligence in specific fields [12]. After that event, human players stand no chance against artificial intelligence in chess.

Another symbolic match of professional humans versus artificial intelligence occurred in the game of Go in 2016, in which the 18-time international champion Lee Sedol was defeated 4-1 by the DeepMind-developed computer AlphaGo. Interestingly, it is believed that AlphaGo lost in the fourth game of the match due to a known flaw in the Monte Carlo tree search algorithm, as the algorithm pruned moves that it considered less relevant. [13]

Clearly, as computers become more powerful over time due to the fact that big companies are still investing in artificial intelligence, human beings will never claim their best place in those games again.

Introduction to ultimate tic-tac-toe Tic-tac-toe is one of the simplest two-player board games, in which the players take turns marking X and O, respectively, in a three-by-three grid. The winner is the first to align three of their marks in any row of three verticals, three horizontals, or two diagonals. [14]

Some of the winning positions are shown in Figure 1.

A position is considered a draw if all nine cells are filled without a three-in-a-row for any player.

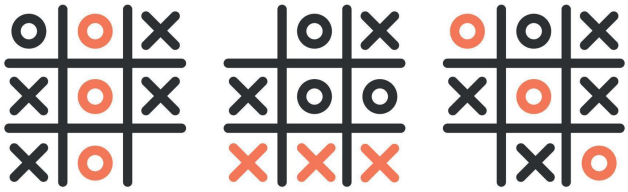

Figure 1. Three (classic tic-tac-toe) positions were won by player 2, player 1, and player 2, respectively. The three orange marks in each position form a row for the winning player. [15]

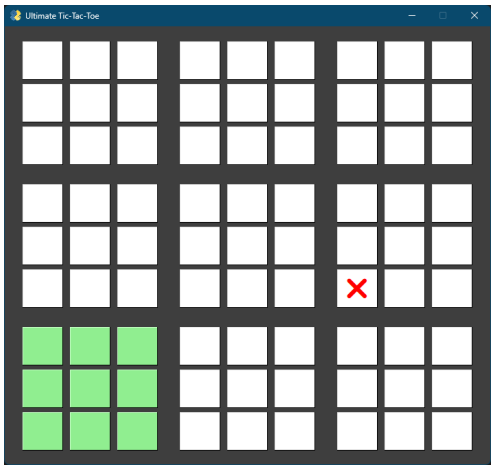

Figure 2. Player 1 placed their X mark in the bottom-left corner of a local board, so player 2 must place their O in the bottom-left local board, which has a light-green background.

Since tic-tac-toe only has 26,830 different possible games, including rotation and reflection of the board, it is simple and cheap to develop a naive program that uses brute force, that is, searches the whole search tree to find the best move, not to mention a hardcoded playstyle that always plays optimally, which Newell and Simon introduced in 1972 [14].

To preserve the elementarily simple rules of this game but make it a lot more difficult to play, ultimate tic-tac-toe, a variation, is introduced. It is regarded as one of the most complex variations of tic-tac-toe, with no known clear strategy for playing. [16]

Rules of ultimate tic-tac-toe This project worked on the original version of ultimate tic-tac-toe.

Nine three-by-three tic-tac-toe boards, referred to as local boards, are arranged on a three-by-three grid as the global board. The first player, who plays as X, can play anywhere on the 81 cells. In the following move, the second player, who plays as O, must play on the local board that has the same relative location (compared to the global board) as the cell that the first player placed (compared to the local board). This rule of sending the opponent to the relative local board is then applied repeatedly through out the game. [16]

Figure 2 illustrated the rule.

A local board is determined if it is won (by any player) or drawn using the original tic-tac-toe rule. A determined local board cannot be played on for the rest of the game, so if a player is sent to a determined local board, they can play on any other board that is not yet determined. Figure 3 shows an example.

The winner of the game (or the global board) is the first to win three local boards in a row, with the same rule as the original tic-tac-toe. The game is drawn if all nine local boards are determined but no one wins.

An example of the end game is shown in Figure 4.

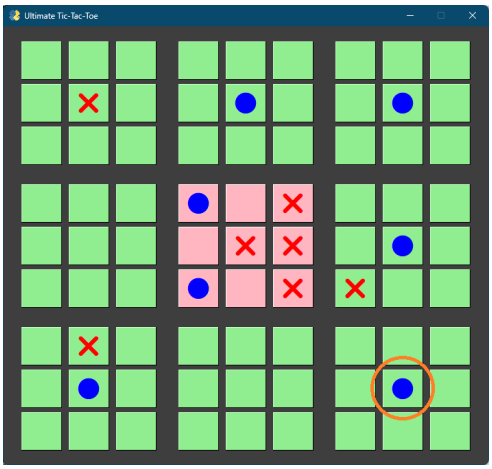

Figure 3. The first player won the local board which has a light-red background (and the background color would be light-blue if the second player won). Afterwards, the second player sent their opponent to the middle board (the move is indicated by the orange circle), so the first player could play anywhere else in their next move.

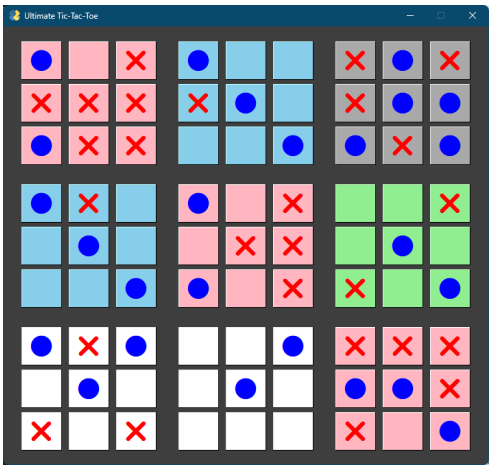

Figure 4. Player 1 won the game. Note that the light-grey local board indicates that it is drawn.

This project This project worked with the AlphaZero algorithm [17], to create a bot to play ultimate tic-tac-toe.

2 OBJECTIVES

The main objective of this project is to develop a GUI-friendly desktop application that plays ultimate tic-tac-toe with the player, powered by a powerful artificial intelligence (AI) system, while keeping the program resource-cheap to evaluate the moves. The player can choose to play as X (play first) or as O (play second). Some of the other non-default features include the modes of human versus human and AI versus AI.

The road to finding those objectives and whether or not they are achieved will be discussed in the following sections.

3 LITERATURE REVIEW OF ALPHAZERO AND ITS MAJOR MODIFICATIONS IN THE PROJECT

Introduction to AlphaZero There are many approaches to reinforcement learning algorithms, which can be categorized into three categories:

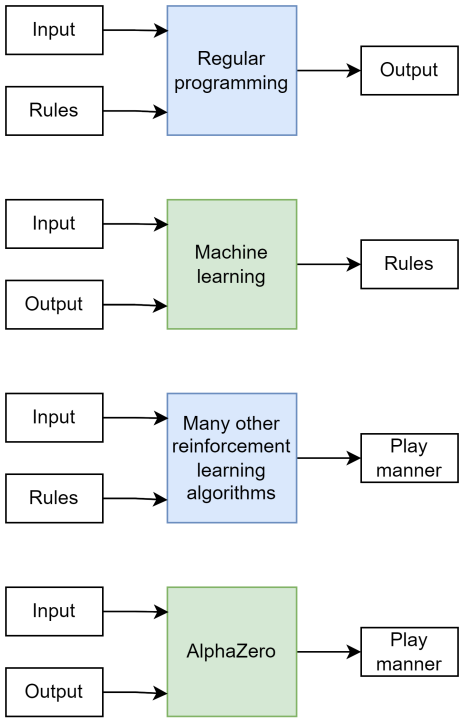

Figure 5. A comparison of similarities and differences between regular programming and machine learning, and between many other approaches and AlphaZero.

value-based, policy-based, and model-based algorithms [18]. Model-based algorithms seem infeasible in most games other than the most basic ones, since they try to make use of a game that is modelled as a Markov chain [19].

AlphaZero was proposed in 2017 by the DeepMind team [17], and this method can obviously be classified as a policy-based and value-based algorithm, since their neural networks output p and v, with respect to policy and value, for each board (along with human-extracted features from the board). Simply put, in the implementation of AlphaZero in this project, the policy is the probability distribution of actions (whose sum of its elements is 1), and the value is the evaluation of the board, or, in other words, which side is currently stronger, represented by a real number in the open interval (-1, 1), in which if the value is larger, then the board favors player 1 and vice versa.

The idea behind AlphaZero The idea behind AlphaZero is that it tries to maximize the potential of the algorithm purely by itself, without human knowledge. The way it achieves that is by improving through self-play and making use of deep learning to learn the game optimally instead of using human-extracted features or previously-played human games. However, the game search tree is still necessary to lead the moves, which is why it uses the Monte Carlo tree search.

Surprisingly, the algorithm does not even know the rules of the game that it is playing. The other human-extracted features might include heuristics and predefined and evaluated rules. This way, the algorithm is no longer limited by human features, games, or evaluation and has the potential to surpass the human expert level in such game play.

This approach can be seen as the original idea of machine learning in general at its inception, which was to make a program from a lot of data instead of from human rules. And how reinforcement learning, a subfield of machine learning, is starting to return to this idea with powerful mathematical computation in hand, is impressive (Figure 5).

The AlphaZero training loop In the original AlphaZero implementation, there are three stages that are executed in parallel [17]: self-play, retrain, and evaluate. However, in this implementation, those stages in a training loop are executed sequentially in that respective order for simplification. This project also only uses one neural network to output both policy and value instead of two different ones like the original AlphaZero.

In the self-playing step, a Monte Carlo tree search implementation, guided by the current best neural network, plays against itself. (It is a weight-randomized network initially.) For each move of each player, the three following data are stored: the board (along with its metadata and human-extracted features), the search probability π (outputted by the tree search), and the winner (1, -1, 0, or a ε small number with respect to player 1 won, player 2 won, not determined, or drawn). It plays against itself for a certain number of games to append those data to the dataset. The oldest data is discarded if storage is needed and to improve performance by removing old bad data from the training data.

For the retraining step, a copy of the neural network is kept (referred to as the old network). It is then retrained on the whole dataset (the original implementation sampled many mini-batches instead) so that its outputs p and v converge to π and winner, respectively (this model is referred to as the new network).

The evaluation step simply pits the new and the old networks against each other (using Monte Carlo tree search to guide their moves). If the new network wins more than 55% of games (disregarding drawn games), it is chosen as the new best network; otherwise, discard it.

Figure 6 illustrates the training loop in this project.

4 METHODS

Introduction to implementation The implementation of AlphaZero and its GUI game in this project is implemented in pure Python. To develop the project seamlessly, GitHub and Git Large File Storage (LFS) are used for version control (the GitHub repository for the project is in the Result section), and Anaconda is used for package management.

Requirements The required Python packages for the project are exported to the file ”environment.yml” using Anaconda and are as follows (they should be installed in a Python virtual environment to ensure complete reproducibility and machine safety, but most of the time, just running the project and installing the necessary packages as raised import errors should do the trick):

```
- blas=1.0=mkl
- ca-certificates=2022.10.11=haa95532_0
- certifi=2022.12.7=py38haa95532_0
- intel-openmp=2021.4.0=haa95532_3556
- libffi=3.4.2=hd77b12b_6
- mkl=2021.4.0=haa95532_640
- mkl-service=2.4.0=py38h2bbff1b_0
- mkl_fft=1.3.1=py38h277e83a_0
- mkl_random=1.2.2=py38hf11a4ad_0
- numpy=1.23.5=py38h3b20f71_0
- numpy-base=1.23.5=py38h4da318b_0
- openssl=1.1.1s=h2bbff1b_0
- pip=22.3.1=py38haa95532_0
- python=3.8.15=h6244533_2
- setuptools=65.5.0=py38haa95532_0
- six=1.16.0=pyhd3eb1b0_1
- sqlite=3.40.0=h2bbff1b_0
- vc=14.2=h21ff451_1
- vs2015_runtime=14.27.29016=h5e58377_2
- wheel=0.37.1=pyhd3eb1b0_0
- wincertstore=0.2=py38haa95532_2
```

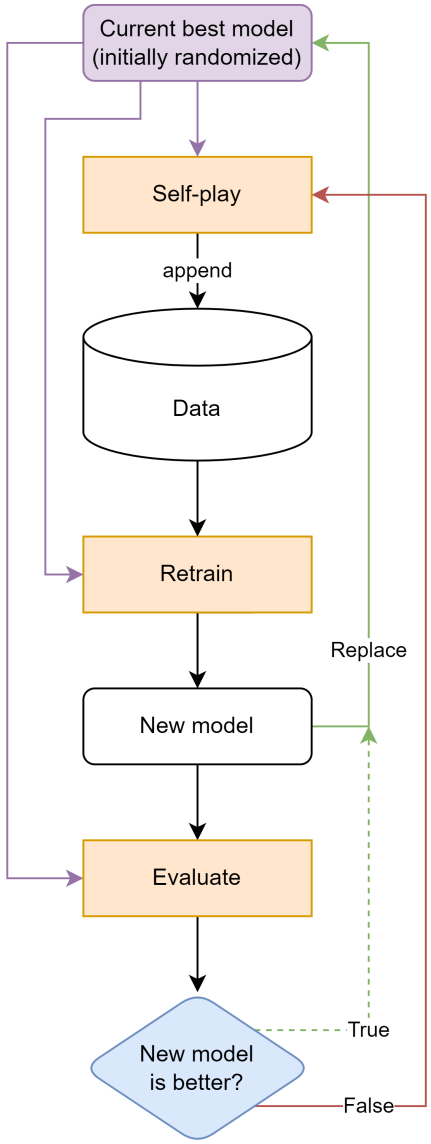

Figure 6. The training loop.

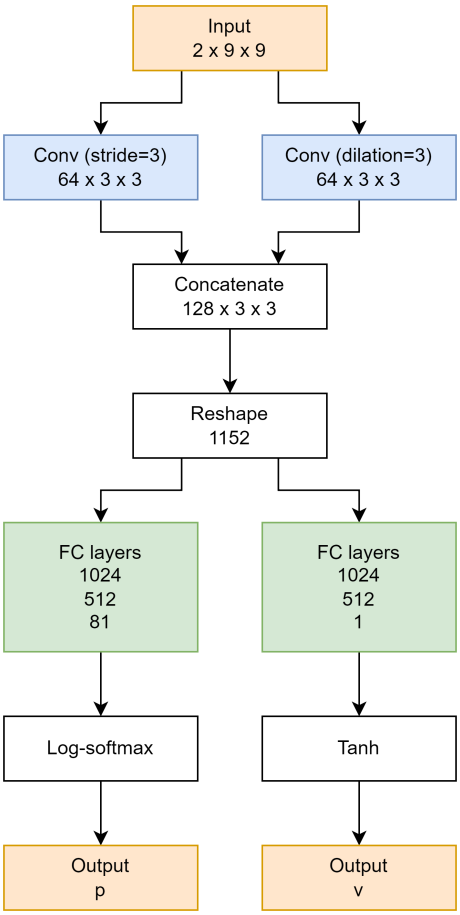

Figure 7. The network architecture.

```
- pip:
    - charset-normalizer==3.0.1
    - click==8.1.3
    - colorama==0.4.6
    - coloredlogs==15.0.1
    - humanfriendly==10.0
    - idna==3.4
    - importlib-metadata==6.0.0
    - itsdangerous==2.1.2
    - jinja2==3.1.2
    - markupsafe==2.1.2
    - pyreadline3==3.4.1
    - pysimplegui==4.60.4
    - python-graphviz==0.20.1
    - remi==2022.7.27
    - requests==2.28.2
    - torch==1.13.1
    - torch-summary==1.4.5
    - torchview==0.2.6
    - tqdm==4.64.1
    - typing-extensions==4.4.0
    - urllib3==1.26.14
    - werkzeug==2.2.2
    - ya-dotdict==1.0.2
    - zipp==3.12.0
```

The network architecture In general, the neural network is composed of some initial convolutional layers for effective feature extraction, followed by fully connected layers. The illustration in Figure 7 briefly describes it.

Each input has two channels of size 9x9. The first channel is the current state of the board, with values of 1, -1, or 0, with respect to X-played, O-played, and not-played cells. The second channel is the valid-move mask of the board, with values of 1 or 0 (true or false), with respect to the playability of the respective cells. Figure 8 shows an example.

This input is forwarded to two different convolutional layers with the same kernel size 3x3. The first convolutional layer uses a stride parameter of 3 (Figure 9). The second convolutional layer uses a dilation parameter of 3 (the space between kernel points) (with stride of 1), as illustrated in Figure 10. Both produce 3x3 images. The two outputs are then concatenated along the channel axis. This output is flattened to feed to two different sequences of fully connected layers to get the final outputs of p and v.

The core network described above is written in PyTorch, as are other neural networks, inheriting the ”nn.Module” class. In the ”nn” package, ”Conv2d”, ”BatchNorm2d”, ”Linear”, and ”BatchNorm1d” are used in the architecture, along with some utility functions in ”nn.functional” that

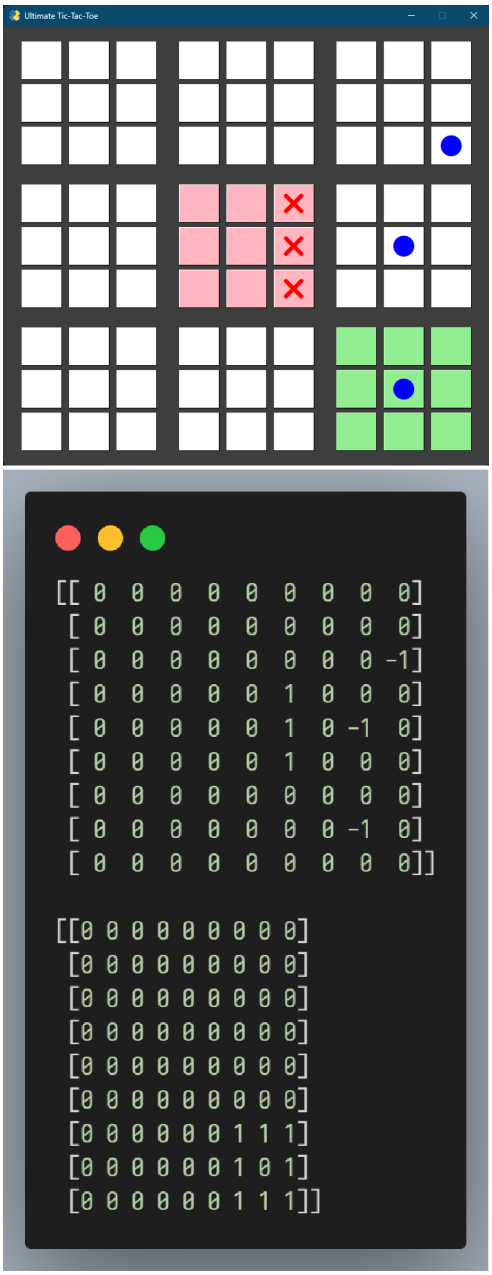

Figure 8. The board position and the corresponding two input channels.

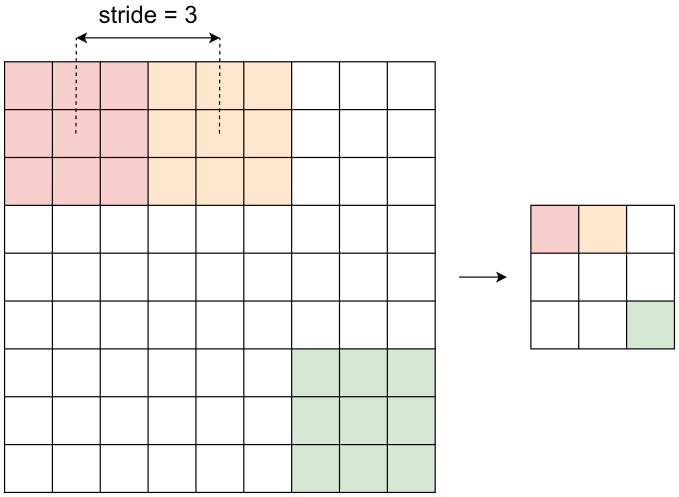

Figure 9. The input and output of the first convolutional layer with stride of 3 (dilation=1 as default).

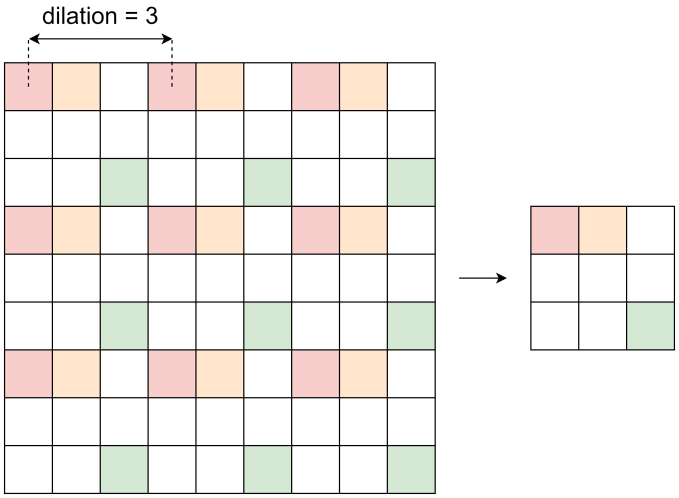

Figure 10. The input and output of the second convolutional layer with dilation of 3 (stride=1 as default).

are ”relu”, ”dropout2d”, ”cat”, ”view”, ”dropout”, ”log_softmax”, and ”tanh”.

All those implementations are written in the file ”UTTTNet.py”. Running this script will do nothing but draw a graph named ”graph.png” with its full architecture and the shapes of input and output in each layer (this picture is too high to include here). Note that the ”graphviz” executable library must be included in the repository, which was removed at the later stage of the development due to its large size, but one copy can be downloaded here: `https://graphviz.org/download/`.

The neural network wrapper The ”UTTTNet.py” script, as mentioned, only defines the network architecture. The functions of training, predicting, saving, and loading checkpoints of the neural network are implemented in ”implemented_NeuralNet.py”.

The methods for predicting, saving, and loading checkpoints are straightforward. The training method initializes the optimizer as Adam with the parameters of the neural network. For each epoch, it samples the training dataset, predicts (or calculates) its output, calculates the two losses (each for p and v), and then backpropagates the sum of them to update the weights. All of these phases should work the same as the training step of a regular PyTorch model.

The original game The core game is implemented in ”original_game.py” using NumPy only. The ”OriginalGame” class stores the current state of the game, executes a move, raises an exception if the move is invalid, and tries its best to provide the required information for its wrapper, which is proposed later.

The information that the class stores as the current state of the game are the ”cell_state” (a 3x3x3x3 NumPy array, whose entries are the same as the input of the neural network mentioned above), the won local boards, the won global board (a 3x3 NumPy array and a scalar, respectively, whose entries are the same as the winner of the board previously described, in order to prevent redundant computations), the ”curr_area” (the index of the local board as a tuple of two entries that the player must play next, or ”None” if the player can play anywhere), and

the current player (1 or -1). All the information is updated after every valid move.

The game wrapper The class in "implemented_Game.py" wraps the game to be used by the AlphaZero algorithm (but not the GUI of the game), which is the same purpose as the wrapper of the neural network. This game wrapper implements the following main methods with the information given by the core game: get the initial board, get the next state (given a state and a move), get the valid moves (return a mask given a state), get the current winner (given the state), get the canonical form (swap the current state's entries of X and O (or 1 and -1) if the current player is -1, else return as is, given a state and the current player), get symmetries (return all symmetries of the board, including all distinct possible rotations and reflections, given a state), and get string representation (return a hash string of the board, required by Monte Carlo tree search, given a state).

The following nested "for" loops implement how all possible symmetries can be taken from a board:

```
for flip_big in [False, True]:
    for flip_small in [False, True]:
        for k_big in range(4):
            for k_small in range(4):
                # ...
```

The "flip_big" and "flip_small" boolean variables keep track of whether or not to flip the global board and all 9 local boards, respectively. The "k_big" and "k_small" integer variables in $\{0, 1, 2, 3\}$ are used to determine the number of 90-degree rotations applied to the global board and all 9 local boards, respectively. This is exhaustively tested in "_test_rotate_flip.py" to ensure all $64 = 2 \times 2 \times 4 \times 4$ derived boards are correct and distinct.

The Monte Carlo tree search In general, the Monte Carlo tree search outputs a probability distribution of moves based on the neural network, given the current state of the game. However, instead of the pure state of the game, the canonical state is needed as the input here, so the search can be simplified (since the board is reversed, player 2 can now search like player 1, maximizing the reward instead of minimizing it, and also simplifying many other steps during search).

The state of the Monte Carlo tree search includes: "Qsa" - the Q values of all visited (state, action) pairs; "Nsa" - the numbers of times the search tree has visited all visited (state, action) pairs; "Ns" - the numbers of times the search tree has visited states; "Ps" - the initial policies returned by the neural network of states; "Es" - the winner of states; and "Vs" - the valid move masks of states. The string representation of a state is used in order to store all the above information effectively in Python dictionaries.

The search function will return the negative value of the input board. Each completed, non-recursive call to it counts as one iteration of the simulation. Simply, it works as follows: check if it is the terminal node (the game has ended), if yes, return the negative value of its winner; check if the state is in "Ps", if no, return the negative of the neural network predicted v; calculate and choose the action with the highest upper confidence bound (UCB); execute that move; and recursively call the search function itself with the new state. During searching, update the state of the tree where necessary, typically at the beginning and end of the function.

The UCB is controlled by a parameter named "cpuct" that balances exploration and exploitation. [20]

The "getActionProb" function is used to get the action probabilities of a state, in which a number of simulations are performed on that state. It will then calculate the action probabilities based on the number of visits from the state to each action, and in general, the more the action is visited, the higher the probability of the move.

There is a "temp" parameter to "smooth" the probabilities. In particular, if $temp = 0$, the probability of the best move is 1 and the rest are 0, or if $temp = 1$, it will not smooth the probabilities. Theoretically, if $temp = +\infty$, the probabilities of each move are equal, which is $1/81$, but the only values used for temp in this project are 0 or 1.

For detailed implementation, see the "MCTS.py" script.

The evaluation implementation The "Arena.py" script pits two functions against each other, in which each function takes a state as input and outputs the action. Half of the games are played with the first function being the first player, and the other half are played with the second function being the first player. The number of won games for each and the number of draws are returned.

The training loop implementation The training loop in "Coach.py" is composed of several functions that are described below. This implementation uses most of the scripts modularized above.

The "executeEpisode" function executes one self-play episode to generate the training data, since for each move, all 64 symmetries are added to the training examples, along with their corresponding policies and values (the values are equal, but the policies also need to be rotated or flipped correspondingly). The moves are guided by the Monte Carlo tree search.

During the self-play step mentioned above, the parameter "tempThreshold" is used to control the "stability" of the algorithm. In particular, if the number of moves performed is smaller than tempThreshold, it uses $temp = 1$, otherwise, $temp = 0$.

The "learn" function runs a number of iterations of "executeEpisode", appends the generated data to the training set, discards the old data if necessary, saves the training set, saves and loads the neural network, trains the network using the dataset to get a new one, pits the new model against the old one, and if the new model is better (that is, it wins more than 55% of the games of the total games won), then saves it; otherwise, loads back the old model. (Figure 6)

Training details The training phase of the algorithm used the standard NVIDIA Tesla P100 GPU-accelerated environment on Kaggle without any modification.

The neural network training settings are as follows: learning rate = 0.001; dropout rate = 0.3; number of epochs = 15; batch size = 64.

The general AlphaZero training loop settings are set as follows: number of episodes = 20; tempThreshold = 25; number of Monte Carlo tree search simulations = 15; number of evaluation games (in pitting) = 24; cpuct = 1.0; and some other parameters.

In total, it took about 36 hours to run the training loop for 29 iterations.

The GUI application The GUI application in "gui.py" used a Python package named PySimpleGUI, to which I personally contributed by finding some bugs while working on it earlier and posting some issues on their GitHub.

The application implemented the modes of "Human vs Human", "AI vs AI", "Human vs AI" and "AI vs Human" (the former is who plays first). There are some frames that are the project information, the turn information (which shows whose turn it is currently), and the error information (as a failsafe in the case where the player plays an illegal move or in the case when the game is won or drawn). There is also a button to open the GitHub repository of this project. The Figure 11 is an overview of the GUI application.

The player can play simply by clicking on the button corresponding to the move they want to play. If versus an AI, it will "think" for some moment and then play it on the board in return.

After each move by the player or AI (when the function "play" is called), the GUI is updated correspondingly to show the new state of the game (involving a mix of many GUI update functions). For the event

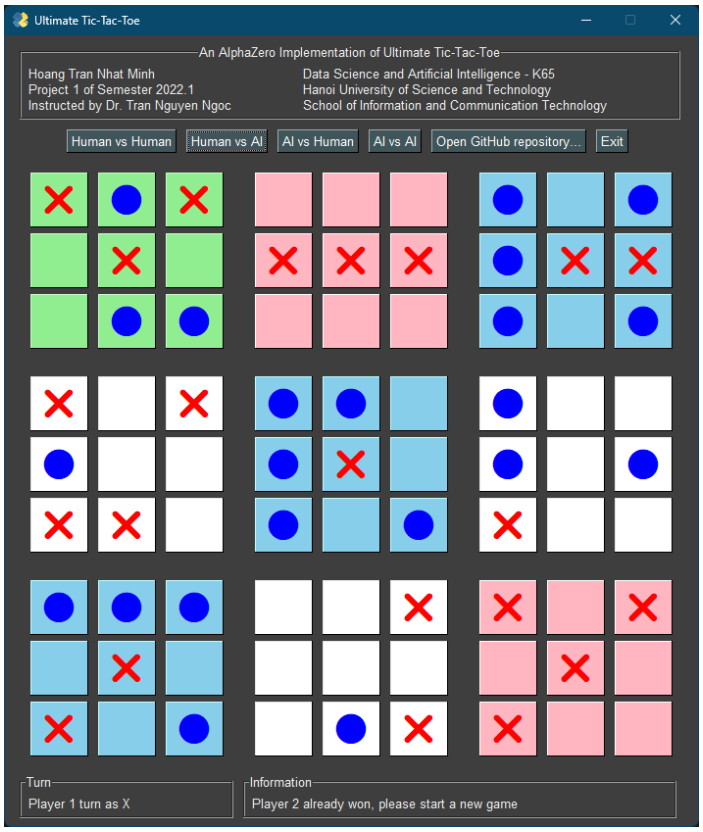

Figure 11. The application.

Iter.	Win	Loss	Draw	Win rate
1	10	10	4	50.0%
2	12	8	4	60.0%
3	5	8	11	38.5%
4	3	8	13	27.3%
5	5	7	12	41.7%
6	6	4	14	60.0%
7	3	4	17	42.9%
8	7	1	16	87.5%
9	4	3	17	57.1%
10	4	3	17	57.1%
11	12	11	1	52.2%
12	10	9	5	52.6%
13	10	8	6	55.6%
14	9	9	6	50.0%
15	10	9	5	52.6%
16	8	9	7	47.1%
17	11	11	2	50.0%
18	10	11	3	47.6%
19	8	8	8	50.0%
20	12	8	4	60.0%
21	11	8	5	57.9%
22	8	11	5	42.1%
23	8	9	7	47.1%
24	9	7	8	56.3%
25	9	10	5	47.4%
26	10	9	5	52.6%
27	9	7	8	56.3%
28	11	8	5	57.9%
29	10	8	6	55.6%

Figure 12. The result table from the evaluation phase of each iteration.

loop, it handles when the player clicks on the move button of the game, the open repository button, the change mode buttons, or the ”Exit” or ”X” button on the window.

There are also some parameters and functionalities for easy development of the application. The boolean ”full_gui” parameter, if set to false, will show a simpler GUI. The boolean parameter ”debug_printing”, if true, will print some debug information to the terminal without showing it in the GUI. The theme can also be changed easily via the ”theme” parameter, and its default value is ”DarkGrey6”.

5 RESULTS

The GitHub repository All the work is shown in the GitHub repository `https://github.com/htnminh/AlphaZero-Ultimate-TicTacToe`. This includes the code, the coding process as commits in the repository, the AlphaZero algorithm, the evaluation, the GUI game, the images, and the analysis of the results.

The GUI application The application is aesthetic and fully functional, with many modes for a player to experience, as mentioned above. The user can also find out more about the project by reading the source code and contributing to it.

The time consumption of the evaluation mode of the algorithm (the ”thinking” time of AI) is relatively short on a regular desktop computer nowadays without a dedicated GPU. It is usually less than one second.

The AlphaZero algorithm It is relatively difficult to benchmark this type of reinforcement learning model like what is usually done with other types of machine learning problems like supervised or unsupervised learning. However, this AlphaZero algorithm really tested itself through the training loop, and one can easily see if the model really improves by keeping track of it during the training phase simply by observing if the algorithm is constantly discarding old models. As shown in Figure 12, the algorithm kept discarding its previous model, which is a good sign of improvement over time.

One can also benchmark the performance by playing against it (as mentioned, some of the human versus AI games became part of history). By playing against the trained AI, one can see that it is capable of playing at the level of a beginner. In particular, it does try to win on some local boards; it is able to block moves that create a three-in-a-row for the opponent; and it can win the game if the player is also a beginner. However, most of the time, it will lose to an experienced player, given that it cannot really infer about a few moves ahead and occasionally sends the opponent to a local board where they have the advantages, and just a few of those moves pretty much decide the game.

6 DISCUSSION

The design principle of the project The principle of the project is to make use of object-oriented programming (OOP) as much as possible to simplify the coding, testing, and evaluating processes. It also helps the project in terms of scalability and reusability in future work.

The general implementation The use of Git LFS is deprecated since it is really limited in practical use without a proper paid subscription plan. Instead, the data is hosted on Kaggle, and work along with the training process.

The GUI application Unlike the AlphaZero training process, the GUI software can run smoothly on any modern, regular computer with a display.

The network architecture It is obvious that the first convolutional layer is used to extract the features of the local boards. For the second convolutional layer, it is used to extract the features of the cells that have the same relative position; for example, all nine top-left corners of all local boards; in an effort to help the network to learn about the ”sending opponent” rule of the game.

The benefits of batch normalization layers are the decrease in training time, the easier weight initialization, the effective ReLU activation, the better regularization itself (hence less need for dropout), and the better generalization. [21].

The wrappers The two wrappers, which wrap the neural network and the game with extensive functions that can conveniently be used by the algorithm, show an important property of OOP: encapsulation.

The original game The game is really hard to solve in terms of complexity, which has been proven as mentioned in the first section. An interesting human strategy is that giving up a local board to the enemy may lead to a win globally. The game has many factors of surprise due to the ”sending” rule. Also, if both players are really good, the game will be really long in terms of the number of moves, which increases the complexity of the game by a lot. And just like in chess and many other traditional board games, only one seriously mistaken move will eventually lead to an unpreventable loss.

The evaluation implementation The evaluation step will be more reliable if more games are played, due to the fact that if the new model is really better, it will win games more certainly. Less games played means random factors may affect the outcome of the series of games more severely.

The Monte Carlo tree search implementation More tree search simulations also improve the performance for the same reason.

The training loop implementation One of the hardest implementations for the training loop is concurrent execution, which was later deprecated due to its complexity. The difficulty is that the data cannot be shared between threads or processes that easily due to the modularization of the project, and this is also one of the weaknesses of the implementation so far.

However, the script for the main training loop is pretty short and straightforward compared to other scripts due to the fact that it uses those scripts as fundamental blocks.

Results Given enough computational power and storage, one could theoretically train the algorithm indefinitely to get better and better results until the model architecture and tree search limitations do not allow it to learn more patterns. The training phase, or even the inference phase, will take a lot longer with a bigger network or a more exhaustive search using the Monte Carlo tree search. As any of those improvements requires a lot of the mentioned resources, in the scope of this particular project, the trained model is limited to its current state. However, it is remarkably proveable that if more resources are provided, simply expanding the search and the network will make the system more powerful.

7 CONCLUSION

To conclude, this project successfully implemented a convenient GUI for the game ultimate tic-tac-toe, one of the most difficult variations of tic-tac-toe, with various modes. The artificial intelligence of the game is a modified version of the AlphaZero algorithm, which was trained with a new convolutional neural network model architecture for ultimate tic-tac-toe and has a low evaluation time on a normal computer. Future works may include parallel computation, expanding the model architecture, experimenting with a greater variety of Monte Carlo tree search and its parameters, fine-tuning with more training algorithms and their hyperparameters, lowering the evaluation cost, improving human-extracted features, improving the GUI, mode additions like online player versus player, and implementing gaming features like the Elo rating system with logged-in accounts.

REFERENCES

[1] ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/. (Accessed on 01/17/2023).

[2] ChatGPT, Midjourney Helped a Man Write a Children's Book in 72 Hours. https://www.businessinsider.com/chatgpt-midjourney-ai-write-illustrate-childrens-book-one-weekend-alice-2023-1. (Accessed on 01/17/2023).

[3] ChatGPT: Understanding the ChatGPT AI Chatbot — eWEEK. https://www.eweek.com/big-data-and-analytics/chatgpt/. (Accessed on 01/17/2023).

[4] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, Oct 2022.

[5] NeurIPS 2020 : JAX MD: A Framework for Differentiable Physics. https://neurips.cc/virtual/2020/public/poster_83d3d4b6c9579515e1679aca8cbc8033.html. (Accessed on 01/17/2023).

[6] Using JAX to accelerate our research. https://www.deepmind.com/blog/using-jax-to-accelerate-our-research. (Accessed on 01/17/2023).

[7] Reinforcement learning - Wikipedia. https://en.wikipedia.org/wiki/Reinforcement_learning#cite_note-3. (Accessed on 01/18/2023).

[8] Basic Understanding of Environment and its Types in Reinforcement Learning - MLK - Machine Learning Knowledge. https://machinelearningknowledge.ai/basic-understanding-of-environment-and-its-types-in-reinforcement-learning/. (Accessed on 01/18/2023).

[9] Google AI Takes Down Human Champ of World's Most Complex Board Game — The Takeaway — WNYC Studios. https://www.wnycstudios.org/podcasts/takeaway/segments/google-i-takes-champion-most-complicated-board-game-world. (Accessed on 01/18/2023).

[10] The Rising Tide of AI – Chess and Computers — Delancey UK Schools' Chess Challenge. https://www.delanceyukschoolschesschallenge.com/the-rising-tide-of-ai-chess-and-computers/. (Accessed on 01/18/2023).

[11] 12 Greatest Chess Players of All Time (2023) — Amphy Blog. https://blog.amphy.com/greatest-chess-players-of-all-time/. (Accessed on 01/18/2023).

[12] Deep Blue versus Garry Kasparov - Wikipedia. https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov#Impact_and_symbolic_significance. (Accessed on 01/18/2023).

[13] AlphaGo versus Lee Sedol - Wikipedia. https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol. (Accessed on 01/18/2023).

[14] Tic-tac-toe - Wikipedia. https://en.wikipedia.org/wiki/Tic-tac-toe. (Accessed on 01/18/2023).

[15] Tic tac toe Game. Business Strategy line art concept. Vector illustration 10251915 Vector Art at Vecteezy. https://www.vecteezy.com/vector-art/10251915-tic-tac-toe-game-business-strategy-line-art-concept-vector-illustration. (Accessed on 01/18/2023).

[16] Ultimate tic-tac-toe - Wikipedia. https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe. (Accessed on 01/18/2023).

[17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, 2017.

[18] Reinforcement Learning: What is, Algorithms, Types & Examples. https://www.guru99.com/reinforcement-learning-tutorial.html. (Accessed on 03/23/2023).

[19] Model-free (reinforcement learning) - Wikipedia. https://en.wikipedia.org/wiki/Model-free_(reinforcement_learning). (Accessed on 03/23/2023).

[20] AlphaZero paper, and Lc0 v0.19.1 - Leela Chess Zero. https://lczero.org/blog/2018/12/alphazero-paper-and-lc0-v0191/. (Accessed on 03/23/2023).

[21] Batch Normalization. The idea is that, instead of just... — by Manish Chablani — Towards Data Science. https://towardsdatascience.com/batch-normalization-8a2e585775c9#:~:text=Using%20batch%20normalization%20allows%20us,difficult%20when%20creating%20deeper%20networks. (Accessed on 03/30/2023).