Self-play Reinforcement Learning for Simple Games: Implementing AlphaZero from Scratch

JUAN CAMILO OSPINA VILLA[1]

[1] *Applied Optics Research Group, Universidad EAFIT.*

Compiled November 12, 2024

The following report presents a implementation of the Self-Play Based Reinforcement Learning algorithm AlphaZero on the games of ConnectFour and TicTacToe. An interface for adapting other turn-based two player games to the algorithm is provided. Additionally, evaluations are done to show the effectiveness of self-play and neural networks to learn dominant strategies of games without previous explicit domain knowledge.

Implementation Github Repository.

http://dx.doi.org/10.1364/ao.XX.XXXXXX

1. INTRODUCTION

The study of game playing agents dates back to the beginnings of computer science [1]. Widely popular, traditional board games, such as chess and checkers have been heavily studied. Several algorithms, specialized software, and theory has been developed through the years tailored specifically to improve the ability machines have to play such games [1]. The interest of Artificial Intelligence researchers in chess led to significant advances, DeepBlue [2] is perhaps the most recognized example, being the first agent capable of beating the human world champion of chess as early as 2001. However, this and other similar systems rely heavily on highly tuned domain techniques, and as a result cannot be easily generalized to other problems without significant human effort [1]. In 2016, researchers at DeepMind presented the AlphaGo algorithm [3], an agent that leverages Monte Carlo Tree Search (MCTS) and Deep Neural Networks to learn the game of Go. Their work introduced the concept of self-play, which involves generating training data by playing simulated games against itself, as part of the training process. This concept was revolutionary in the field of Reinforcement Learning, as it exhibited better performance compared to previous methodologies, while using only knowledge about the precise rules of the game. In 2017, they introduced AlphaZero, a more general approach [1] to implement self-play based learning to any turn-based two player game. In this article, we introduce an implementation of the later, using AlphaMCTS [4] and ResNet [5] as the core components of the AlphaZero algorithm. Implementation details are provided, as well as a common interface for training on other games that comply to it. General performance, training, and interpretability results are also presented.

2. METHODOLOGY

The algorithm was implemented in Python, using *BLAS* based *Numpy* and popular machine learning framework *Pytorch*.

As reference, the AlphaZero algorithm has two main components, a Monte Carlo Tree Search (MCTS or AlphaMCTS) for simulating possible future outcomes of the game, and a Deep Neural Network that predicts both a policy and a value for a given game state [3]. During training, both are used to generate a data set of game states and their respective fit (which depends on how positive the outcome of the game is for a given player in that position). The dataset can be used to train the network to predict both the policy, in the form of a probability distribution of the next moves, and a value, which is the general fit of the game position [3]. This newly trained network can be used to further generate more synthetic data, by playing against itself in a process known as self-play [1]. This cycle can be repeated several times, until the neural network has a sufficiently good approximation of a function $V_\theta(S)$ that outputs the best moves based on the current state of the game S [3]. During inference, the approximated function $f_\theta(S)$ is used to expand the tree search to all possible next moves, without the need of roll-out techniques (random simulation) to estimate the fit of each position [1]. As a result, the algorithm can often find the best moves given a certain game state much faster than other tree expansion based algorithms [1]. This section provides details on the implementation of the MCTS, the ResNet used as driving neural network, and both the training and inference processes.

A. Monte Carlo Tree Search

MCTS is a powerful approach for designing game-playing bots and solving sequential decision problems [4]. MCTS provides a method to explore a tree in such a way that balances exploration and exploitation. It consists for four main phases: tree traversal, node expansion, rollout, and back-propagation.

During the tree traversal phase, a $UCB1(S)$ function is used to select which node to expand. The child node that maximizes $UCB1(S)$ is selected, and expanded with all the possible valid actions given the current state S_i. A diagram of this process is shown:

The general form of the function to select which node to expand is:

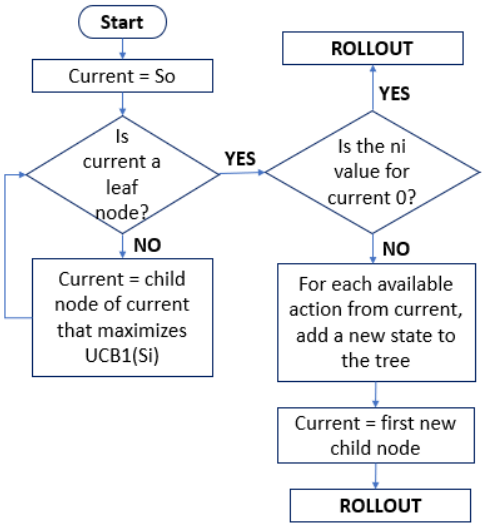


Fig. 1. Tree traversal and exploration flowchart. S_0 is the initial state of the game, $UCB1(S_i)$ the exploration/exploitation function, and n_i the number of visits the i-th node has received.

$$UCB1(S) = \hat{Q}(s,a) + C\sqrt{\frac{\ln(n_s)}{n_{s,a}}} \quad \textbf{(1)}$$

Where $\hat{Q}(s,a)$ is defined as the win-rate of the child node and is referred as the exploitation term because it's maximum when the node's game state has promising results [4]. On the other hand, n_s is the visit count of the current node, $n_{s,a}$ is the visit count of the child node, and C is the exploration constant. The term accompanied by C is generally regarded as the exploration term, since it is maximum in nodes that have not been visited much compared to their parent node [4].

The actual $UCB1(S)$ function implemented for our MCTS has a slightingly different definition. This is because in two-player turn based games, the child nodes and their parent are generally a different player. Therefore, the formula should be written from the perspective of the parent player [3]. In general, this means that poor performance of the child node should amount to a higher $UCB1(S)$ from the parent perspective. The reformulated function, used in the implementation is as follows:

$$UCB1(S) = \left[1 - \frac{1}{2}\left(\frac{v_{s,a}}{n_{s,a}} + 1\right)\right] + CP_s\frac{\sqrt{n_s}}{n_{s,a}} \quad \textbf{(2)}$$

Where $v_{s,a}$ is the value or fit of the child node, and $P_s = \pm 1$ is the player value of the parent, which is one for the starting player, and negative one for the second player. Using 2, we select the child node that leads to a better outcome for the parent node, taking exploration into account, instead of the node that's best fit from its own perspective.

In 1, it's shown that if the initial value or fit v_i of a certain node i hasn't been detected we do something called a rollout. Rollout essentially means to play the game randomly until a stop condition is met [4]. For most games, the rollout ends when there's either a winner or a tie between both players. The fit of the expanded child node is determined by the outcome of the game in respect to itself. For example, if the child node corresponds to player one, and it won the rollout, then the fit of the roll out would be 1. One can chose, for example, 1 to be a win, -1 to be a lose, and 0 to be a tie. The value is summed into the value $v_{s,i}$ of the node, and then back propagated, by adding it to the value total of all its predecessors. Note that the perspective of the node should also be taken into account during backpropagation [4].

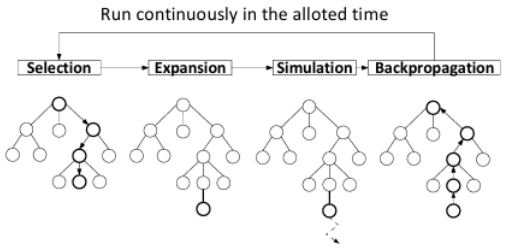


Fig. 2. Enter Caption

However, AlphaZero uses a different approach, where a neural network is used to expand, and rollout all the children nodes, of the parent selected for expansion. The function approximated by the neural network is defined as:

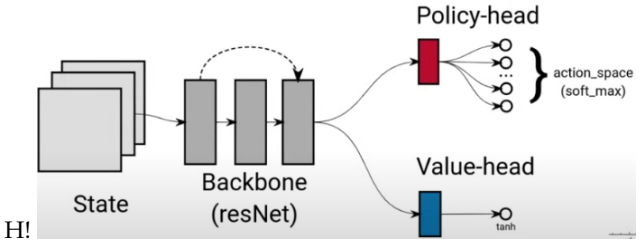
$$f_\theta(S) = (P, v_{nn}) \quad \textbf{(3)}$$

Where θ are the weights of the network, S is the input state, P is the policy output, and v_{nn} is the general fit of the current state. By having estimated v_{nn}, one can directly assign this value to the expanded node and back propagate it without needing to simulate the game until an end condition is met. Additionally, the policy computes the fit of each of the possible child node states, effectively expanding to all possible plays, and estimating their effectiveness, on a single run of the network [1]. Self-play acts based on the policy distribution, until a leaf node is reached, before relying again on $UCB1(s)$ for tree traversal. In practice, the algorithm is allowed to expand the tree for a certain amount of time or iterations in order to find the best move given each state S_i.

B. ResNet Deep Neural Network

A residual convolutional deep neural network [5] is used to approximate the $V_\theta(S)$, that outputs the best move based on any arbitrary game state S. Networks that are commonly used in image processing domains are generally a great option for this task, since the game is represented as a series of plains, with the dimensions of the board [1]. The plains should represent the complete state of the game. In games like chess, it's common to use a plain for each piece of each player, where it's position in the board would be a 1, and the rest of the plain matrix would be filled with zeros [1] [3]. Ïn the games of TicTacToe and ConnectFour, all the pieces are equal except for the player that they belong to. Consequently, we used 3 three plains for representing the state of the game. The starting player plain contains a 1 in each position in which there's a piece belonging to him. The second player plain is defined exactly the same way. The third plane has a one in each position neither player has yet played. The general dimensions of each plane are the same as the size of the board (ej. 3x3 in the case of TicTacToe). A general diagram of the network is provided in 3:

As observed in [5], the networks contains two heads. The policy head, outputs a probability distribution of the best next move from the current player perspective after passing through a *softmax* function. The skip connections in the ResBlock improve convergence stability [5]. The input layer implements batch



H!

Fig. 3. Overview of the resnet architecture used [5]

normalization to preserve the distribution of the inputs during training [6]. For a detailed implementation, see the *resnet.py* file in the github repo. The value head outputs the general fit of the current state S. The internal layers are as follows for each game 1:

	TicTacToe	ConnectFour
ResBlocks	4	9
Hidden Layers	64	128

Table 1. Comparison of ResBlocks and Hidden Layers for TicTacToe and ConnectFour

The approximated function to be trained is as described in 3. Each sample of the training-set contains:

$$s, \pi, z = Sample \tag{4}$$

Where s is the state of the samle, π is the policy distribution of the possible next moves, and z is the fit of the game state s. Therefore, the target loss function to minimize, in respect to a given sample is:

$$l = (z - v)^2 - \pi^\top \log \mathbf{p} + c||\theta||^2 \tag{5}$$

The first term of 5 represents the $L2$ distance between the predicted fit value v and the actual fit value z. The second term represents the cross-entropy loss between the actual policy distribution π and the predicted distribution $\mathbf{p}$. Finally, the last term is the regularization term that penalizes weights θ from being too high, and c is the weight decay constant.

C. Game Interface

The code was implemented so that it works with any game that implements the interface described bellow. A class that complies with it, will work with the rest of the implementation without additional modifications:

- **__init__**: Initializes general parameters, mainly board dimensions (rows and columns), and size of the action space (space of valid moves).
- **__repr__**: Returns string with name of the game.
- **get_initial_state**: Returns initial board state as *numpy* array.
- **get_next_state**: Returns board state after a specific players does a specific move.
- **check_win**: Returns True if current player has won, otherwise returns False.
- **get_value_and_terminated**: Returns (True, 1) if the game ended and current player won, (True, 0) if game ended in a tie because there's no more valid moves, (False, 0) if the game hasn't ended.
- **get_opponent**: Returns $P_s * (-1)$.
- **get_opponent_value**: Returns $v_{s,a} * (-1)$
- **change_perspective**: Returns current state from the perspective of the opponent. (Multiply board by (-1))
- **get_encoded_state**: Returns a given state in its plains/image representation.

D. Training

Training was performed using a AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz CPU and 12GB of RAM. The model can be trained with both CPU or GPU. MCTS was implemented both in series and parallel. The parallel version runs much faster, since it simulated multiple games (trees) at the same time, thus generating more synthetic data in a single training loop, and reducing unnecessary calls to the Neural Network. Implementation details are beyond the scope of the report, but the detailed implementation can be found in *alphaMCTS.py* file in the github repo. Inference always uses serial MCTS, since a human opponent only plays one game at a time.

Parameter	ConnectFour	TicTacToe
`C`	2	2
`num_searches`	600	60
`num_iterations`	8	3
`num_selfPlay_iterations`	500	500
`num_epochs`	4	4
`num_parallel_games`	100	-
`batch_size`	128	64
`temperature`	1.25	1.25
`dirichlet_epsilon`	0.25	0.25
`dirichlet_alpha`	0.30	0.30
`LR`	0.001	0.001
`weight_decay`	0.0001	0.0001
`training_time`	302.5 min	29.8 min
`MCTS`	Parallel	Serial

Table 2. Comparison of Parameters for ConnectFour and TicTacToe

The parameter `temperature` is applied to output policies during training:

$$P' = P^{\frac{1}{T}} \tag{6}$$

Leveling out the output distribution P' and therefore increasing exploration during training. $T < 1$ will favor exploitation.

The `dirichlet_epsilon` and `dirichlet_alpha` parameters are used to introduce dirichlet distributed noise to output policies. This adds randomness to the search iterations, allowing the algorithm to find unconventional plays.

E. Inference

Two inference scripts are provided, *eval_TicTacToe.py* and *eval_ConnectFour*. Both are essentially the same, and provide a command line interface for a player to be able to play against the model. The player may chose to go second or first, and is provided with a list of valid moves in each iteration. The script works by generating an instance of AlphaMCTS using both the game class, and the trained ResNet. The human player and the machine take turns. The agent decision is computed by performing a MCTS search on a given state and computing the argmax of the output distribution.

$$D_i = \textbf{argmax}(f_\theta^P(S_i)) \tag{7}$$

The parameter `num_searches` can be modified to specify how much time the machine has to think. The default is set to 60 for rapid inference, but the agent performs better the higher the amount of searches allowed.

3. RESULTS

A. TicTacToe

The agent was set to play 30 games an expert human player. The human player always played first. The agent was given 60 MCTS searches. Not surprisingly, due to the simplicity of the game of TicTacToe, all games ended in draws. Both the human player and the agent, even with a small number of searches, are capable of determining an optimal strategy.

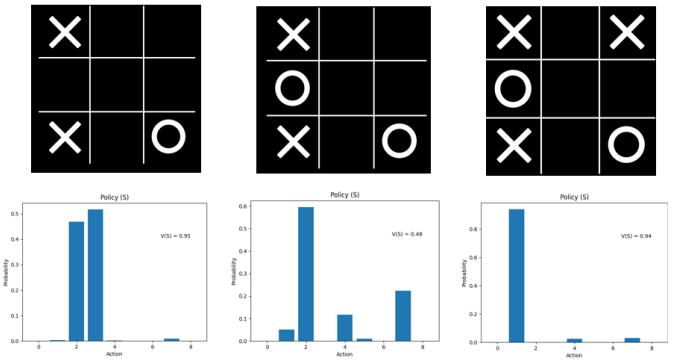


Fig. 4. Example policy and value results from the network on several input board positions.

The figure 4, several output policies P and values v are shown. In each position, the best move for the current player is also the move with highest probability in the policy distribution. Additionally, the v value is always computed in respect to player one. In the last game state to the right in figure 4, $v = 0.94$, indicating the game state is almost completely in favor of player one. This implies that the model is working properly, since there's no valid move that will make player two win in such situation. Also note that the policy automatically learns to mask out invalid moves depending on the game state, only valid moves have probability values greater that are greater than 0.

Inference time was also measured in respect to the number of searches:

B. ConnectFour

The agent was set to play 20 games against a casual player. The human player always played first. Out of the 20 games, 16 ended in a tie and four were won by the agent player. In the four

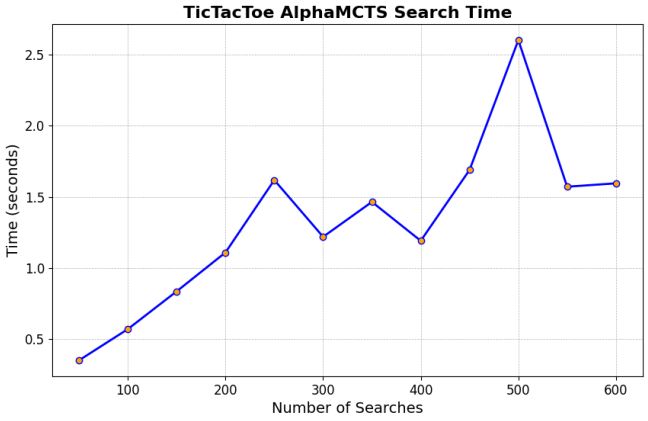


Fig. 5. MCTS best move search computing time in respect to the number of searches

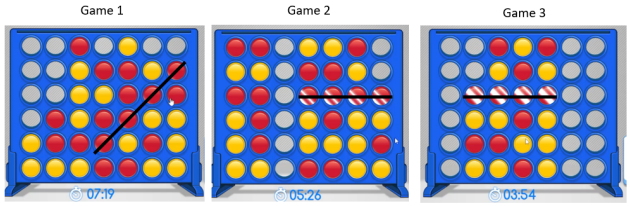


Fig. 6. Terminal states for the 3 games played agains CBCt Canada bot. AlphaZero plays red.

games won by the agent, the board state was in an unrecoverable position for the human player.

An additional 3 games were played againt the CBC Canada connect four bot [7]. Our agent always played second and was provided with 100 MCTS searches. It won the 3 games. The final game states are provided (AlphaZero plays red):

The move sequence for the games are reported. We refer to 'R' as AlphaZero, and 'Y' to CBC. The 7 columns are indexed starting from 0 in the left to 6 in the right.

- **Game 1**: [(Y, 6), (R, 3), (Y, 6), (R, 3), (Y, 6), (R, 6), (Y, 0), (R, 4), (Y, 2), (R, 2), (Y, 4), (R, 3), (Y, 3), (R, 4), (Y, 1), (R, 4), (Y, 2), (R, 1), (Y, 0), (R, 4), (Y, 4), (R, 1), (Y, 2), (R, 3), (Y, 2), (R, 2), (Y, 5), (R, 5), (Y, 5), (R, 5), (Y, 5), (R, 6)]
- **Game 2**: [(Y, 5), (R, 4), (Y, 4), (R, 4), (Y, 5), (R, 4), (Y, 5), (R, 5), (Y, 6), (R, 3), (Y, 3), (R, 6), (Y, 3), (R, 3), (Y, 1), (R, 4), (Y, 4), (R, 3), (Y, 5), (R, 1), (Y, 3), (R, 5), (Y, 0), (R, 1), (Y, 0), (R, 1), (Y, 1), (R, 1), (Y, 0), (R, 0), (Y, 0), (R, 0), (Y, 6), (R, 6)]
- **Game 3**: [(Y, 2), (R, 3), (Y, 3), (R, 3), (Y, 4), (R, 3), (Y, 4), (R, 3), (Y, 3), (R, 2), (Y, 4), (R, 4), (Y, 1), (R, 1), (Y, 2), (R, 2), (Y, 2), (R, 2), (Y, 4), (R, 4), (Y, 1), (R, 1)]

Even though it's clear that AlphaZero learnt how to play the very well, 3 sample games against an unknown algorithm is not a conclusive benchmark. Future work may include comparison against other state of the art techniques.

We computed the time it took AlphaMCTS to search the best play as a function of the turn for the third game:

Note that we refer to a turn as both players playing their move. In 7 we observe that, as the turns pass, the amount of valid moves decreases, and therefore, search time decreases rapidly.

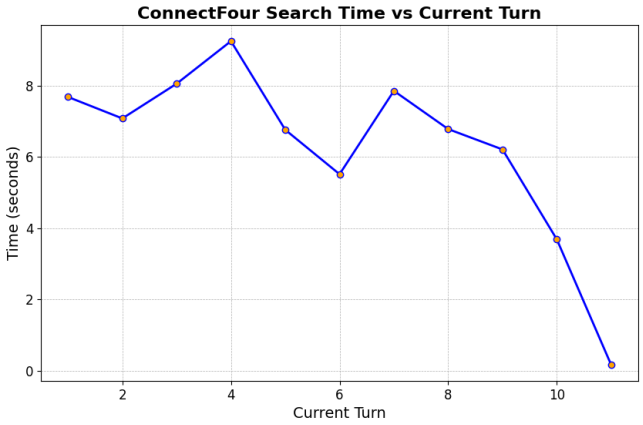


Fig. 7. AlphaMCTS search time as a function of the current turn in a game of ConnectFour with `num_searches` = 100.

4. CONCLUSIONS

- A functional and flexible implementation of AlphaZero was achieved. The model performance on both games is super-human, for a sufficient amount of search iterations in reasonable time. The modularity of the implementation allows for effective training, prototyping, and understanding of the model.

- A common interface was developed to allow adaptation of new games to the model. A class that complies with such an interface and implements the target game can be used to train a model to play such game without additional changes to the rest of the implementation.

REFERENCES

1. D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, CoRR **abs/1712.01815** (2017).
2. S. Russell and P. Norvig, Artif. Intell. **113**, 110 (2001).
3. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, Nature **529**, 484 (2016).
4. M. Swiechowski, K. Godlewski, B. Sawicki, and J. Mandziuk, CoRR **abs/2103.04931** (2021).
5. K. He, X. Zhang, S. Ren, and J. Sun, CoRR **abs/1512.03385** (2015).
6. S. Ioffe and C. Szegedy, CoRR **abs/1502.03167** (2015).
7. C. Kids, "Connect 4 game," (2024). Accessed: 2024-11-11.