An AlphaZero application for simple perfect-knowledge games

Lorenzo Mambretti, Justin Wang, Chandni Nagda ,
Daniel Loran, Jagveer Singh, Manpreet Kang, Sophie Quynn,
Tengteng Zhou, Yilan Liu, Zihan Xiao

27 November 2018

Abstract

One of the most fundamental problems in the field of reinforcement learning is the search for strong general purpose algorithms that can be used, either as is or with few modifications, in a variety of different contexts. The algorithms need to be scalable and easily applied to a variety of problems to be useful in the field. In the context of perfect-knowledge games, a Google DeepMind team has developed AlphaZero [7], a general purpose reinforcement learning algorithm which achieved superhuman levels in the games of Shogi, Chess, and Go. We recreated the same structure as AlphaZero and applied our algorithm to three simple perfect-knowledge games to test performance and efficacy of the technique, as well as implementing a new heuristic to the Monte Carlo tree search to improve agent performance. Our results indicate that our implementation of the heuristic did in fact lead to the creation of a stronger agent by incentivizing faster termination and reduced computation time. These results might indicate a potential efficiency and performance upgrade that could be applied to DeepMind's AlphaZero algorithm.

1 Introduction

Games have served as a classic testing ground for reinforcement learning algorithms. In particular, board games provide interesting results as they have an extremely sparse reward and require an agent to be capable of both exploring the environment in an effective way and generalizing what it has already learned.
One of the most popular exploration techniques used in reinforcement learning environments with sparse reward is the Monte Carlo tree search (MCTS). This technique is known to explore an environment in a consistent way and it can be structured so that it will improve its confidence over time. Browne et al. in "A Survey of Monte Carlo Tree" examined the different variations of the algorithm and found that specific update rules can be utilized to directly identify a confidence measure [3]. The Upper Bound Confidence equation, in particular, became widely used in MCTS applications due to its consistency in different games. However, the exploration technique provided by the MCTS does not generalize well with games that have a large number of states. This is due to challenges traversing a large tree in a timely fashion.
To solve this problem, David Silver et al. in "Mastering the game of Go without human knowledge" [8] proved that it is possible to use a deep residual neural network to create a generalized representation of the exploration results obtained by the MCTS. In particular, the neural network is used in tandem with the MCTS and trained with the policy distribution gained directly from the MCTS. Therefore, the trained neural network can be used to predict a

policy distribution from a board representation. Instead of using a random roll-out, the trained neural network can be applied to enhance the MCTS where no explorations have been done. In a following work Davis Silver et al. proved that their solutions also generalize in other games such as chess and shogi [7]. This generalized version of the algorithm is called AlphaZero.
One of the downsides of the algorithms provided by previous works [8], [7], is the enormous amount of computation that is necessary. Another limitation of this method is that it relies on both MCTS and neural networks; therefore, any possible heuristics that we can apply to the MCTS or hyperparameter tuning that we can apply to the neural network can and should be seen as an improvement.
With regards to MCTS heuristics, a number of improvements were explored by Cazenave et al [4]. These include discounting - a process that rewards fast completion of the game - and pruning - the elimination of unpromising branches of the tree early on. These heuristics primarily improve performance by reducing computational time, but do not necessarily guarantee superior game performance. A potential pitfall of these heuristics that occurs when specific types of pruning are used in conjunction with discounting is the premature elimination of otherwise promising branches. It is for this reason that these heuristics are generally considered unsafe. However, when implemented carefully, this risk is far outweighed by the benefits of faster game completion and reduced computational requirements.

2 Methods

2.1 Main Loop

In our main loop, we run an arbitrary number of games where the agent plays itself. In our main loop, we train the model several times. At each step, the agent selects the best move by running 1600 subgames with the Monte Carlo Tree Search. After n episodes (in our case we are setting n to 200), we train the neural network with the data from the games. After the network is trained, we will run the Monte Carlo Tree Search with the neural network against the previous version of itself. Then, we update the model in the next session only if performance improves.

2.2 Game Implemention

The simple perfect-knowledge games we implemented are Tic-Tac-Toe, Connect 4, and Checkers. We are in the process of implementing Chess. These games are all considered perfect-knowledge games as the player has full knowledge of the result of their actions and the action space of their opponent. We decided to create our own implementation of each game so that they have a uniform structure and a standard naming for all the functions used. This has allowed us to create highly optimized implementations customized for the code. At the same time, we are keeping the number of functions as limited as possible, allowing for the possibility to easily test other games in the future.

2.2.1 Tic-Tac-Toe

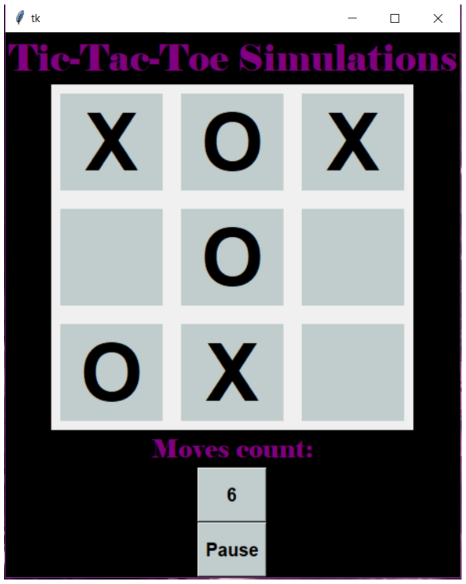
Tic-Tac-Toe was the first and simplest game that we implemented. It has a board size of 3×3 with 3 possible values for each tile: a piece placed by player 1, a piece placed by player 2, and an empty tile [1]. Therefore, it has an observation space of size 27 and an action space of 9 possible actions. The game is won once one of the two players has 3 pieces in a row either vertically, horizontally, or diagonally. If the board is full and neither players are declared a winner the game is considered a draw. There are approximately 3×10^5 possible games.

2.2.2 Connect 4

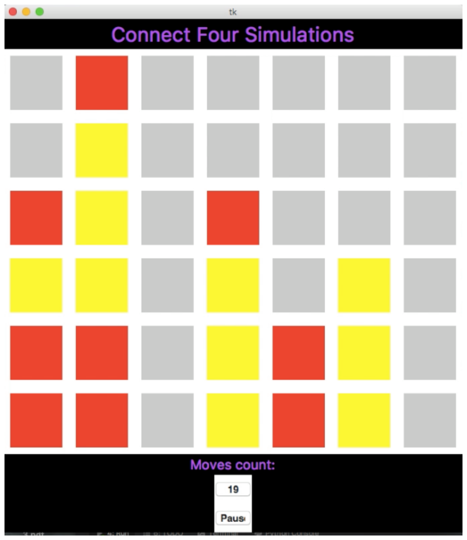
We implemented Connect Four on a board of 6 rows $\times$ 7 columns. Similarly to Tic-Tac-Toe, Connect 4 has 3 possible values for each tile: a piece placed by player 1, a piece placed by player 2, and an empty tile [5]. When a player makes a move, they must drop their piece in a column that has at least one empty slot and the piece proceeds to fall until it hits the first available slot. Therefore, Connect 4 has an observation space of size 126 and an action space of 7 possible columns. The game is won once one of the two players has 4 pieces in a row either vertically, horizontally, or diagonally. If the board is full and neither players are declared a winner the game is considered a draw. There are approximately 4×10^9 possible games.

2.2.3 Checkers

Our implementation of checkers uses standard American Checkers rules [2]. Although a checkers board is visualized as 8×8, with 64 possible positions, the pieces move diagonally so there really are only 32 possible positions on the red spaces. Thus, we represented the board as 4×8 valid positions. In addition to improving space efficiency, removing the unused black squares make it easier to check diagonals in the implementation. One of the rules of checkers is promotion. Once a player moves their piece to the opposing side of the board, it becomes promoted and then the player is allowed to move the piece backwards as well as forwards. Thus, there are 5 possible values for each tile: player 1 regular, player 1 promoted, player 2 regular, player 2 promoted, and an empty tile. Thus, the game has an observation space of 160. The action space includes all of the moves from any given tile to any other given tile, or $32 \cdot 32 = 1024$ total actions. There are approximately 10^{20} possible games.



(a) Tic-Tac-Toe



(b) Connect Four

Figure 1: UI interfaces of the different games

2.3 User Interface Design

In order to provide a clearer view into the actual gaming procedure generated by computer, a User Interface (UI) enables people to interact with the machine. Moreover, a good visualization of user interface should prove that the gaming procedure corresponds to the actual gaming in real life. With each game, there is one UI, enable people to visualize those games step by step . Different users are presented with different marks or different colors. For example, in Tic-Tac-Toe, X and O represented two distinct users, while in Connect Four, red and yellow tells the difference. Start and Pause bottoms were provided. As simple as it sounds, Start initiate gaming procedure, Pause interrupt while second Pause will re-initiate the procedure. Also, Moves count were provided for visualization purpose. When the game reaches to its end, UI will start a new circle, and display steps for new round.

2.4 Monte Carlo Tree Search

The Monte Carlo Tree search is one of the main components of the algorithm, and it is used extensively for searching the optimal policy. To choose the correct move at each board position, the Monte Carlo Tree search completes an arbitrary number of games and assigns a value to each valid move that can be performed at a specific board position. This search is divided in two parts: the roll-out, and the back-propagation.

2.4.1 Roll-out

During the Roll-out a series of operations is performed recursively. Looking at the current board position, the algorithm examines all the legal moves. There are then three cases:

1. If no legal moves are available, and hence the game is terminated, perform back-propagation.
2. If only one legal move is available, perform that move
3. If multiple legal moves are available, then evaluate them.

The most important and common case is the case where multiple legal moves are possible. In this case, if the algorithm never explored this board position, then it would choose a move using the neural network prediction. Notice there is one exception to this, and it is at the first iteration of the algorithm, before the neural network is trained. At that first iteration, the algorithm will choose a random move instead.

If the algorithm already explored the board position, then it will choose the move which has the highest value. We define value using the Upper-Bound limit equation

$$v = \frac{s}{n} + c^* \cdot \sqrt{\frac{ln(n)}{N}}$$

where v is the value, s is the sum of the scores obtained passing through that specific board position, n is the number of times it passed through that board position, c^* is an exploration parameter, and N is the number of times it passed through the previous board position. We set the exploration parameter to $c^* = \sqrt{2}$ for all our testing, but it can be reduced for more "greedy" performances.

Once the move is selected, the algorithm performs the move and looks at the next board position, but with opponent's perspective. We switch to the opponent's view by switching piece values and rotating the board in the case of Checker. The exploration happens uniquely through self-play. This step is repeated until the termination of the game through a win, loss, draw. Once the game is terminated, the algorithm will perform back-propagation.

2.4.2 Back-propagation

At the termination of the game, the agent will receive a score $\bar{s}$, which will be -1, 0, or 1. This score will update the value of all the board positions from the termination of the game, up until the board position where the algorithm started the exploration. To update the value, the score $\bar{s}$ is summed to the current score s which is associated to each node. However, at each step back, we multiply $\bar{s}$ to account for the fact that the players are alternating turns. In other words, a victory for one player is counted as defeat for the other, and vice versa. Moreover, the count of how many times we visited each board (n in the equation) is updated.

2.4.3 Heuristics

An attempt to improve the performance of the MCTS was made using the Discounting Heuristic introduced by Cazenava et al [4]. This heuristic replaces the standard evaluation function of the MCTS - a payoff function evaluated at a terminal state - with the the same function divided by the time required to reach the given terminal state. The effect of this heuristic is that it favors rapid completion of the game by maximizing the payoff function at lower values of time. A table summarizing the difference in performance achieved between the standard MCTS and the modified MCTS over an increasing number of simulations is presented below.

Discounting	Episodes	Tic-Tac-Toe	Connect Four
Yes	64	393	400
No	64	355	398

Table 1: MCTS with Heuristics

Further attempts to improve performance using the pruning on depth heuristic[4] were unsuccessful due to challenges in implementation. However, we hypothesize that in conjunction with the discounting heuristic, pruning on depth would further improve performance by favoring terminal states that occur at shallower depths over those at greater depths.

2.4.4 Implementation constraints

The number of roll-outs is completely arbitrary, but we decided to establish a upper bound or 1600 roll-outs or 10 seconds of time for each board position that is evaluated. We observed that with Tic-Tac-Toe the 1600 roll-outs are always performed in less than 6s, so it is never limited by time. However, most of the positions evaluated in Connect 4 and Checkers will reach 10s before being able to perform 1600 roll-outs. This effect is caused by the increased average length of the game. Checker games last an average of 40 moves.

2.5 Neural Network Architecture

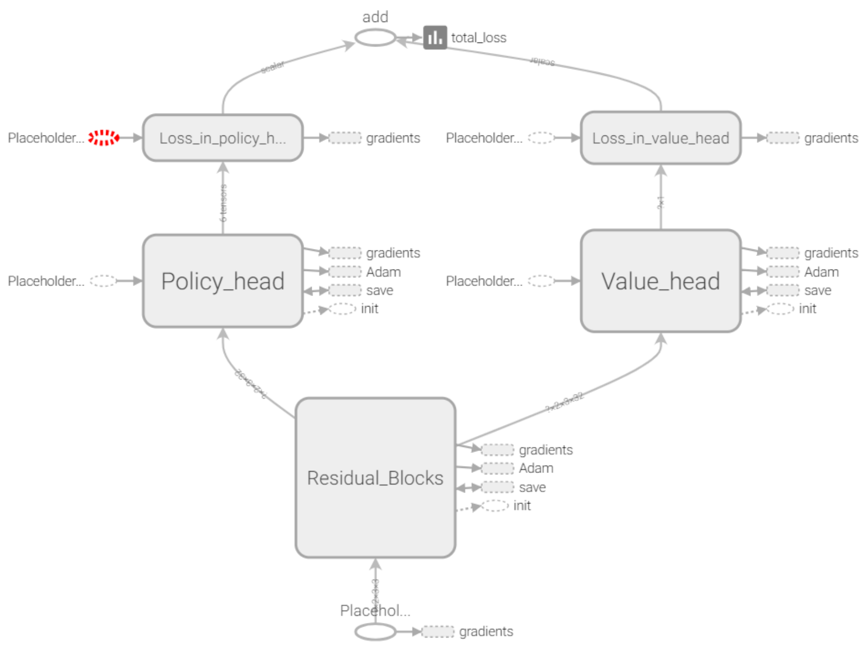


Figure 2: Neural network structure as visualized on the TensorBoard

Input Layer
The game state is represented as one-hot encoded layers. For example, a Tic-Tac-Toe board would have two 3x3 layers, one for each player. One layer maps a 1 to wherever an X is, and the other layer maps a 1 to wherever an O is. Every other cell is a 0, representing an empty space in the board. The advantage of using one-hot encoding over a whole number encoding (say, 0, 1, 2...) is that the network sees the pieces as pieces and what they do, instead of viewing the pieces with respect to the value that they are encoded as. This differentiation becomes more important in complex games such as checkers or chess, where pieces have different values associated with them.

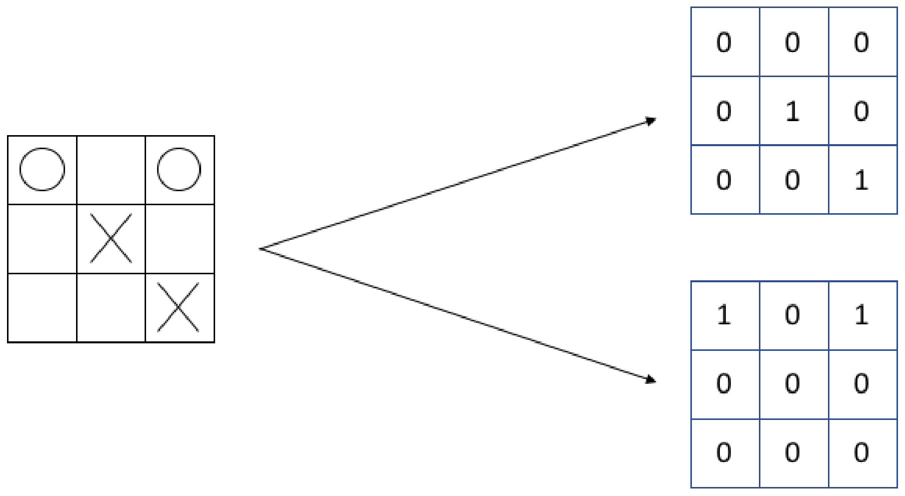


Figure 3: OneHotEncoding of inputs

Hidden Layers
Each hidden layer is a residual block, which is a series of convolutional layers added to the input. More about motivation and implementation is explained below.

Outputs
The value head predicts the winner, which is continuous from -1 to 1. A 1 means player one wins and a -1 means player 2 wins.
The policy head is trained to predict the move that the MCTS decided on, effectively condensing all the expensive MCTS searching into one forward pass in our network. Eventually the network is tasked to predict the value of children that have not yet been explored by the MCTS [9].

2.5.1 Motivation

Avoiding Accuracy Degradation
We want to use a deep network (10 convolutional layers), as deeper networks are, in general, more accurate for complex problems. However, Kaiming He identified a problem: when networks become deeper than a certain nondeterministic depth, the training accuracy degrades. Note that this degradation is different from the degradation of the evaluation accuracy after overfitting, as it is the training accuracy that decreases, not the evaluation. To avoid this degradation but retain the improved performance with increased depth, residual networks add identity mappings to the convolutional layers.

Increasing Efficiency

For each input x, the residual layer returns F(x) + x, in which F(x) is the result of two convolutions. The residual is F(x) = H(x) - x. F(x) is the residual, H(x) is the output of the layer, and x is the input.

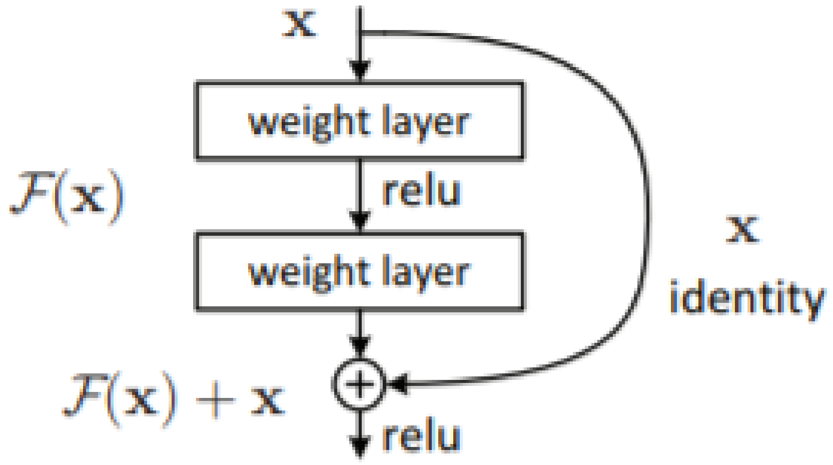

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 4: Residual Layer Architecture

If the residual F(x) is small compared to the input x, the result will be close to the input. This allows us to effectively "skip" convolutional layers that are not needed.

This approach assumes that the ideal solution is closer to an identity mapping than a zero mapping. Consider a shallow network with some convolutional hidden layers, and then create a deeper network by adding an identity mapping to each layer. Experiments performed on ImageNet proved that the deeper networks did not produce a higher training error than the shallow network [6].

2.5.2 Loss Function

From the neural network structure, we have two different outputs: policy head and value head, where policy head is a vector of next moving probabilities and value head is a scalar value estimating the expected outcome from previous position. Here we denote move probabilities $\mathbf{p}$ with components $p_a = Pr(a|s)$ and scalar value $v = \mathbb{E}[z|s]$, where a denotes as each action in playing chess and z is the expected outcome for next move from position s. In neural network, parameters θ are updated in order to minimize loss function by using Adam Optimizing Algorithm or Gradient Descent Algorithm. Loss function in this neural network is defined as the sum of losses from both policy head and value head. As policy head outputs a vector and value head outputs a scalar, we apply cross-entropy losses as loss function in policy head and mean-squared error in value head. Loss function l can be denoted as following[7]:

$$(\mathbf{p}, v) = f_\theta(s), \qquad l = (z - v)^2 - \pi^T log(p) + c||\theta||^2$$

2.5.3 Training

In practice, it is not practical to train the whole dataset in one iteration in each epoch, since the matrix operations are very expensive for a large set. Instead, we trained a mini-batch each iteration which rapidly reduces the time spent per iteration. We set the mini-batch size as 64

and trained for 100 epochs. Moreover, in order to visualize model changes, we write a summary each 20 iterations to show the real-time value of loss function and output value in policy head and value head. Also, we save the newest updated model every 1000 iteration in case we lose the model if program corrupts. The total loss decreased and converged towards 0.2, so our network achieved some degree of functionality.
After training and saving the latest model, we can achieve prediction function in neural network. After loading trained model, we use trained parameters to make prediction on next playing step and final winner.

2.6 Evaluation

The evaluation is a central component of the training loop as it allows to measure the performances of the algorithm and update the networks. While collecting the episodes that are used for the training of the Neural Network, the final result of each game is collected. The score s_ij of the agent i at episode j is $s_ij \in \{-1, 0, 1\}$. We perform the measurement for n episodes to obtain the following rating equation using the ELO-400:

$$ELO(i) = ELO(opponent) + \frac{\sum_{j=1}^{n} s_ij}{n} \cdot 400$$

The network is updated only if the ELO is greater than the opponent ELO, and it becomes in this way the opponent in the following game.

This method easily scales for all level of skill of the opponent, but it needs an initial condition. We define therefore the ELO rating of a random agent as $ELO = 0$. The first ELO evaluation is then executed against a random agent to understand the starting point of the agent.

We chose this method because it easily generalize for all games, and the implementation of a random agent is trivial. However, we recognize there are two possible drawbacks:

1. A good player, which wins all the games against the random agent, will score the maximum possible ELO rating of 400. In the same way, the best possible agent will score 400 against the random agent. Therefore, the first evaluation is prone to error and may not be accurate. All the successive evaluation, however, are accurate.
2. In very simple games where the observation and action space are limited, the random agent may perform better than expected, and may often tie against a suboptimal or even optimal agent. This is true in particular for Tic-Tac-Toe, because of its limited action space and observation space.

3 Results

The Algorithm has been evaluated in two steps for each game. First an evaluation that uses the base model, using uniquely Monte Carlo Tree Search. Afterwards, an evaluation using Monte Carlo Tree search and Neural network.

For the first evaluation, with only Monte Carlo Tree search, we run the model for 800 episodes (100 episodes, 8 parallel processes). The results are reported in the table below:

Name	Tic-Tac-Toe	Connect Four	Checkers
ELO rating (800 episodes)	354.5	398.0	not run yet

Table 2: Only Monte Carlo Tree Search

With the collected 800 episodes from each game, we trained the neural network to predict the policy and the value. In this process, we are measuring the loss and observing that the model improves through time. We are here reporting the loss after 1000 epochs, batch size of 100. From Figure 3, we are able to see that total loss of the model converges as expected.

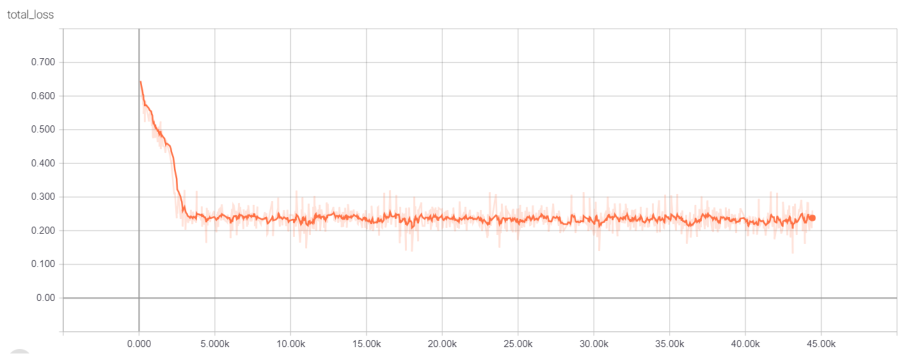


Figure 5: Loss of the model

Discounting	Episodes	Tic-Tac-Toe	Connect Four
Yes	64	393	400
No	64	355	398

Table 3: MCTS with Heuristics

This table was shown earlier and represents the ELO reached by our agent in the games of Tic-Tac-Toe and Connect Four with the implemented heuristics.

4 Discussion

From the initial results of the Monte Carlo Tree search evaluation it is possible to notice that the results achieved are greatly superior to a random agent. The difference in the ELO rating between the game of Tic-Tac-Toe and the game of ConnectFour, 354.5 and 398.0 respectively, are explained by the higher probability of a random draw in a game with the smaller observation space of Tic-Tac-Toe. These results indicate that our implementation of the base AlphaZero algorithm was somewhat successful since it improved upon a random agent.
After implementing the discounting heuristic we see the expected results. The ELO of Tic-Tac-Toe increases, as seen in the table above. We see another small increase in the ELO for ConnectFour. We attribute this to our modification of the payoff function which had higher values at lower times; thus, it made the games terminate faster. Although this doesn't necessarily always mean better performance, in our simplistic games, short winning is advantageous. The

higher ELO that we see in our results might simply be a factor of the simplistic games that we chose to teach our agent. Our results indicate that in games with a relatively small state space - Tic Tac Toe ($3x10^5$) and Connect Four ($4x10^9$) - AlphaZero with the discounting heuristic results in a higher performing agent that learns faster than the base AlphaZero algorithm. We suspect that our heuristic gives us faster results than the base AlphaZero algorithm.
We acknowledge that the number of episodes we trained over are low, so we suggest further training to confirm that the discounting heuristic is truly a performance and speed upgrade.

References

[1] Anderson, E. (2018, October 31). All About Tic-Tac-Toe. *The Spruce Crafts.* Retrieved from *https://www.thesprucecrafts.com/tic-tac-toe-game-rules-412170*

[2] Anderson, E. (2018, November 5). Learn to Play Checkers. *The Spruce Crafts.* Retrieved from *https://www.thesprucecrafts.com/play-checkers-using-standard-rules-409287*

[3] Browne C. et al. (2012, March). A Survey of Monte Carlo Tree, *IEEE Transactions on Computational Intelligence and AI in Games, Vol. 4, No. 1* Search Methods

[4] Cazenave T., Saffidine A., Schofield M., Thielscher M. (2016) Nested Monte Carlo Search for Two-Player Games

[5] Connect 4. (2018). *Ludoteka.com.* Retrieved from *http://www.ludoteka.com/connect-4.html*

[6] He, K. et al. (2015, December 10). Deep Residual Learning for Image Recognition. Retrieved from *https://arxiv.org/pdf/1512.03385.pdf*

[7] Silver D. et. al. (2017) Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm `arXiv:1712.01815 [cs.AI]`

[8] Silver D. et al. (19 October 2017) Mastering the game of Go without human knowledge *Nature* volume 550, pages 354359

[9] [The Artificial Intelligence Channel]. (2018, Jan 18). *Deepmind AlphaZero - Mastering Games Without Human Knowledge* [Video File]. Retrieved from *http://y2u.be/Wujy7OzvdJk*

5 Author Contribution

In alphabetical order:

1. **Chandni Nagda** developed the game implementation for Checkers and worked on the game implementation for other games.
2. **Daniel Loran** Assisted in data normalization, paper writing, and in the game implementation for Checkers.
3. **Jagveer Singh** developed the Monte Carlo Tree search structure, the search algorithm and the roll-out of the games. Moreover, he developed the methods to connect the game implementations with the MCTS itself.
4. **Justin Wang** designed the neural network structure, implemented residual layers, the policy head, and value head. He explained the neural network architecture and motivation behind it in the paper.

5. **Lorenzo Mambretti** (Project Manager) implemented the main training loop, the parallel processing management, and the ELO rating algorithm. Moreover he implemented the MCTS backpropagation and developed the game implementation for Tic-Tac-Toe. He also contributed to the structure of the neural network. Finally, he integrated the neural network prediction into the MCTS.
6. **Manpreet Kang** developed the generalized UI display for Tic-Tac-Toe and also assisted in developing the Monte Carlo Tree search structure.
7. **Sophie Quynn** developed the game implementation for Connect 4 and worked on the game implementation for Tic-Tac-Toe and other games.
8. **Tengteng Zhou** modified the UI for Connect Four. Visualized input layers and helped with the format and content.
9. **Yilan Liu** modified the UI for Checkers. Edited User Interface Design and Method part.
10. **Zihan Xiao** developed the methods to train the neural network, to calculate the loss function and to predict a value from a new imput. Moreover, she implemented methods to select a new batch, to save and load a previously computed model. Finally, she cured the visualization of graph and network variables in the Tensorboard. She describe the loss and training loop of neural network in paper.