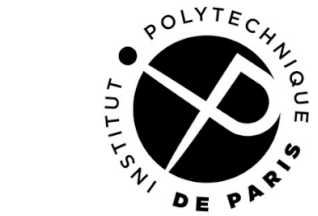

INSTITUT
POLYTECHNIQUE
DE PARIS

Reinforcement Learning

Research Paper

M2 Data Science 2023-2024

Article : Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm[1]

Victor Gertner, Romain Amédée, Maïssane Merrheim

Contents

1 Introduction

Ever since the premises of artificial intelligence, there has always been an interest about solving games. In particular, the game of chess is the most studied case : finding the best combinations of moves (there are approximately $10^{1}20$ possible games of chess) is a very complex problem that has been tackled by many, in all fields of computer science. Developing efficient models and algorithms for this task has required the input of many domains combining the expertise of professional chess players, mathematicians, and computer science professionals. Other chess-like games (strategic combinatorial games) such as Go or Shogi have undergone the same interest. Some very efficient algorithms and models that were able to compete with world-class champions were developed (in 1997, Deep Blue defeated Garry Kasparov which was the chess world champion at the time).

The goal of this paper is to offer a generalization of these many domain-specific algorithms. This new approach, Alpha Zero, allows *tabula rasa* (the idea of starting with a completely clean slate, devoid of any pre-existing knowledge or heuristics about the task at hand) performances in many domains and generalize on games without prior domain knowledge on the task except for the rules of the game. It will learn only based on the rules and games of self-play (an agent learns by playing against itself rather than against human opponents or predefined data).

2 Games considered

This research and the experiments attached to it were mainly done on three famous stategic combinatorial games : chess, shogi and go. Here, we propose to briefly present them.

2.1 Chess

Figure 1: Chess board

Chess is a strategy game played on an 8x8 grid board. Each player controls sixteen pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The objective is to checkmate the opponent's king, putting it in a position where it is threatened with capture and has no legal moves to escape. Players take turns moving their pieces across the board, employing tactics such as pins, forks, and sacrifices to gain an advantage.

2.2 Shogi

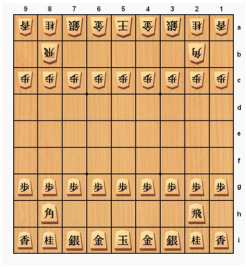

Figure 2: Shogiban (board)

Shogi, often referred to as Japanese chess, is a board game deeply rooted in Japanese culture. Played on a 9x9 grid board, each player commands twenty pieces: one king, one rook, one bishop, two gold generals, two silver generals, two knights, two lances, and nine pawns. The objective mirrors that of chess: to checkmate the opponent's king. However, in Shogi, captured pieces can be reintroduced onto the board under the player's control, adding a dynamic layer of strategy.

2.3 Go

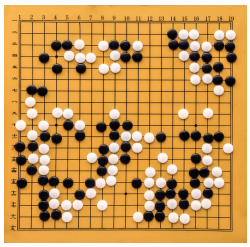

Figure 3: Goban (board)

Go is an ancient and strategic board game played by two players. The game is typically played on a 19x19 grid board, although smaller boards can be used for shorter games or beginners. Players take turns placing black and white stones on the intersections of the grid, aiming to surround territory and capture their opponent's stones. The objective of Go is to control more territory than the opponent by strategically placing stones to build and maintain influence across the board.

3 Prior Work on Computer Chess

Throughout the years, there have been many attempts at cracking those games and developing the best algorithm that would yield super-human performances. Here, we acknowledge some of the important previous reinforcement learning works in computer chess.

3.1 NeuroChess (1995)

NeuroChess is a program developped by Sebastian Thrun in 1995 that proposes an approach to computer chess that consists in learning to play based on the final outcome of games.[2] Its two main components are the use of explanation-based neural network (EBNN) as the central learning mechanism and an evaluation function constructed using temporal differences (TD).

TD learning allows to do predictions in the case an event that we want to predict might be delayed by some (unknown) number of steps. The goal is to recursively learn an evaluation function V that is able to rank chess boards with respect to their chance of winning.

The EBNN allows to use less training example while still being able to generalize more accuratly compared to a traditional neural network. This is possible because EBNN rely on both training examples and domain knowledge (it will know if a certain position for the queen is relevant for instance). In this case, is is another neural network that is used only to represent the domain-specific knowledge.

	GNU d=2, NeuroChess d=2		**GNU d=4, NeuroChess d=2**	
# of games	Back-propagation	EBNN	Back-propagation	EBNN
100	1	0	0	0
200	6	2	0	0
500	35	13	1	0
1000	73	85	2	1
1500	130	135	3	3
2000	190	215	3	8
2400	239	316	3	11

Table 1: Performances of NeuroChess vs. GNU-chess during training (d=depth) – Source : [2]

3.2 KnightCap (2000)

KnightCap was developped in 2000 and proposes an approach that uses a variation of temporal differences : $TD_{LEAF}(\lambda)$ which allows to learn an evaluation function for use in deep minimax search.[3] Though KnightCap allows the computation of complex positional features, it is also much slower than other chess programs at that time.

$TD_{LEAF}(\lambda)$ is a reinforcement learning algorithm combining temporal difference learning ($TD(\lambda)$) with game-tree search. Instead of updating values for all states in the tree (like $TD(\lambda)$), it focuses on the leaf nodes (end states) reached through the principal variation (most likely sequence of moves). This reduces computational cost while still incorporating information from past experiences through λ.

KnightCap was able to achieve master level by mixing human and computer opponents.

3.3 Stockfish (2008-2023)

Stockfish is a computer chess program, its first version was developed in 2008 and it was continuously updated up until 2023 (last release to this day). For many years, Stockfish has been considered one of the state-of-the-art chess program and achieved super-human performances (in 2021 its Elo ranking was estimated at 3548 points). In 2016, it won the TCEC computer chess competition.

Let us briefly descripe how Stockfish works :

- a sparse vector $\phi(s)$ is used in order to represent a position of the board s. The features represented by ϕ were handcrafted and include a lot of information and

metrics about the state of the game (imbalance between Black and Whites, central control, king safety, and any other evaluation metrics). We get the features ϕ_i by mixing handcrafted metrics and automatic tuning. This vector is combined wy a weight vector in order to get the position $v(s, w) = \phi(s)^T w$.

- A minimax search is used in order to evaluate the position s with alpha-beta pruning to eliminate branches that are completely dominated. The search extends the depths of promising variations while decreasing the depth of unwanted variations.

3.4 Meep (2009)

Meep is an algorithm developped in 2009 and that introduces a new method for updating the parameters of a heuristic evaluation function.[4] During the tree search, all nodes are updated contrary to what was done before. Furthermore, the training signal for the evaluation function is the outcome of a deep search. It implements an alternative to standard temporal differences called TreeStrap that updates search values towards deeper search values.

Meep was able to defeat human master players 13 times (out of 15) after it was trained through self-play with random initial weights.

3.5 DeepChess (2016)

DeepChess is a computer chess program that leverages deep neural networks in order to learn an evaluation function from scratch that doesn't need domain-specific knowledge (rules) and that doesn't require to handpick features.[5] During training, the model is fed with pair of positions : one drawn from a game that was won by Whites and one from a game that was lost by Whites. The assumption is made that, generally, a position taken from a game won by Whites must be more preferable than one taken from a game lost by Whites (from the White's perspective).

The total architecture is actually made of two parts :

- Pos2Vec : Deep autoencoder that has a role of feature extractor and that converts chess board positions into a vector of high level features.
- DeepChess : Siamese network that leverages the previously trained Pos2Vec in order to train on pairs of positive/negative examples (advantageous vs. disadvantageous positions for Whites).

DeepChess achieved grandmaster level, and especially was able to learn some grandmaster preferences that were often lacking in previous chess computer programs (for instance, preferring a situation where there are attacking opportunities even if it has less material).

4 From Alpha GO

The game of go as long been considered as one of the most complicated strategic combinatorial games to deal with artificial intelligence, due to the huge number of possible variations and the subtleties of the game's strategy. AlphaGo is a computer program developed by DeepMind Technologies (Google's company) in 2016 and that defeated the world champion Lee Sedol.[6]

Taking the inspiration from deep convolutional neural networks, AlphaGo takes an input as the 19x19 board image and construct a representation of the position. The network is trained following a pipeline of several components that we propose to describe briefly :

- Supervised learning of policy networks p_σ: this is done in order to predict expert Go player moves using supervised learning.
- Training of fast policy p_π to quickly sample actions during rollouts.
- Training of reinforcement learning policy network p_ρ to optimize the final outcome of games of self-play.
- Training of value network ν_θ to predict the winner of the game played by the reinforcement learning policy against itself

The policy and value networks are combined in a Monte Carlo Tree Search (MCTS) in order to select the actins by a lookahead search.

AlphaGo implements efficient move selection and evaluates the positions by neural networks that combine supervised and reinforcement learning.

5 To The idea of AlphaZero

Alphazero tries to generalize the approach of AlphaGo Zero by leveraging what was done and trying to apply it to any game. This algorithm can achieve, tabula rasa, superhuman performance in many challenging domains and this paper especially looks at its performance in the domain of chess and shogi. Moreover, this algorithm doesn't need any additional domain knowledge except the rules of the game and only uses a general-purpose reinforcement learning algorithm.

5.1 The algorithm

The way AlphaZero differs from classical game playing programs is that it doesn't use handcrafted knowledge and domain specific augmentation, but uses deep neural networks and a *tabula rasa* reinforcement learning algorithm.

5.1.1 Neural Network

Indeed, instead of a handcrafted evaluation function and move ordering heuristics, AlphaZero relies on a deep neural network to learn both a policy function and a value function defined as :

$$(p, v) = f_\theta(s) \text{ where } \theta \text{ parameterize the neural network.} \tag{1}$$

This neural network takes as input the board position s and it outputs a vector of move probabilities p for each action a, and also a scalar value value v which represents the expected outcome of the game. This output is very specific to the game considered as different games can have different way to finish a game.

In order to select the move for both players, it doesn't use the classical Alpha-Beta Search, but uses Monte-Carlo Tree Search.

5.1.2 Monte-Carlo Tree Search :

The Monte Carlo Tree search is a way to have a trade-off between exploration and exploitation which are really important concepts of reinforcement learning. It does simultaneously tree growing, rollout and backup by DP. The root is the initial configuration of the game. Each node is a configuration and its children are the subsequent configurations. MCTS keeps a tree in memory corresponding to the nodes already explored in the possibility tree. A leaf in this tree (a node with no children) is either a final configuration (i.e. we know whether one of the players has won, or whether there is a draw), or a node whose children have not yet been explored. In each node, we store two numbers: the number of winning simulations, and the total number of simulations.

It has 4 steps : **Selection, Expansion, Simulation and Backup.**

Selection :

Indeed, in order to select a node in the tree, it uses the upper bound formula.

$$UCB(S_i) = \bar{V_i} + C\sqrt{\frac{ln(N)}{n_i}} \tag{2}$$

having :

- $\bar{V_i}$ is the average value of the state S_i
- C is the exploration constant.
- N is the number of time the parent was visited
- n_i is the number of time this node was visited

In our case, $\bar{V_i}$ represents the winrate of a move. Hence, this formula both takes in account the winrate and by the second term, it puts forward moves that have not been selected

many times before. So we understand that the role of C in this equation is to adjust the trade-off we do between exploration and exploitation as the algorithm selects the node of the tree with the highest UCB.

Expansion :

Expension is the next step if the selected node here is a leaf. Then the algorithm creates child nodes of the previous node for each possible move the game's rules allow.

Simulation :

Then it selects a node from the child nodes it just created by using the UCB criterion. And it either randomly simulates a game or it uses an evaluation heuristic from this node until it reaches a final configuration : in this case, an end game situation. If the previously selected node was in a final configuration, this step is not done.

Backpropagation :

Once the simualation step is finished, MCTS goes through the backpropagation process. It uses the result of the random game and update the branch information from the child node to the root. Thus, each winrate V_i and number of selection n_i is updated.

Those steps are repeated a certain number of times, and then it can select the best possible move.

Practical exemple (taken from wikipedia) :

First steps :

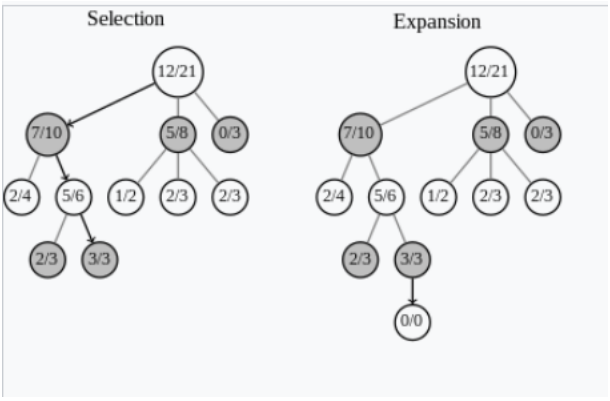

Figure 4: Selection and Expansion steps

So here we can see that the selected node is the 3/3 on the left, annotated by bold arrows. Then the expansion step adds the 0/0 node to it.

Then, we can see that there is a simulation from this 0/0 node, as we can see it's a lose for the white side of the board. We therefore only increment the number of total simulations on the branch for the white nodes and we increment the number of simulations won and the number of total simulations for the black nodes on the branch.

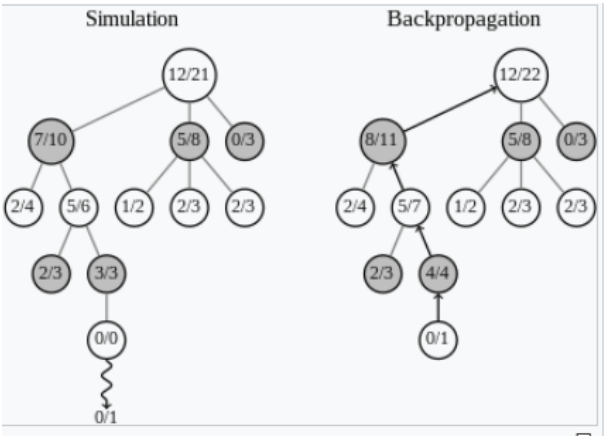

Figure 5: Simulation and Backpropagation step

5.1.3 Training :

So now that we understand how games are simulated, we need to comprehend how to link this and the neural network.

The purpose of MCTS in AlphaZero is to guide the search for promising moves and to improve the quality of the neural network's policy and value estimates by using the results of simulated games. MCTS helps AlphaZero efficiently explore the large search space of possible moves in complex games like Go, chess, or shogi. And this is shown in the loss function used :

$$l = (z - v)^2 - \pi \cdot \log p + c \|\theta\|^2 \tag{3}$$

Having :

- z is the real outcome of the game
- v is the output value of the neural network
- π is search probabilites computed by the MCTS
- p is the policy vector computed by the neural network
- c is a regularisation parameter

The parameter θ is updated through a Gradient Descent.

This loss combines both a mean square error of the value function and a cross entropy loss between policies, and a L_2 regularization parameter is added. This is done in order to update the parameter θ so that it minimizes the error between the predicted outcome and the real outcome while at the same time having a policy vector close to the search probabilities.

Another advantage of having those two component is that AlphaZero assesses positions through non-linear function approximation utilizing a deep neural network, departing from the linear function approximation employed in conventional chess programs. While

this is interesting, it can lead to the introduction of inaccurate approximation errors. Monte Carlo Tree Search (MCTS) mitigates these approximation errors by averaging them out, which consequently diminishes their impact when assessing a substantial subtree. Leveraging MCTS could enable AlphaZero to effectively integrate its neural network representations with a robust, domain-independent search capability.

5.2 Main differences from AlphaGo Zero

5.2.1 Goal :

While AlphaZero goal is to estimate and optimise the expected outcome, taking into account draws or potential other outcomes, AlphaGo Zero was only made to estimate and optimise the probability of winning, assuming binary win/loss outcomes.

5.2.2 Rules

Because AlphaGo Zero only tackles Go, it can leverage the fact that its rules are invariant to rotation and reflection. This permitted data augmentation by using symmetries. Moreover, during the Monte Carlo Tree Search (MCTS) phase, board positions underwent transformation through random selection of rotations or reflections before being assessed by the neural network. This process ensured that the Monte Carlo evaluation was diversified across various biases. This was exploited by both the algorithms.

However, the rules of shogi and chess are asymmetric, so the two operations that were used before cannot be used for AlphaZero on those games.

5.2.3 Self-Play

In AlphaGo zero the games are generated through self-play by the best player of the previous iteration of the algorithm. They select a new best player if its performances with respect to the actual best player make it win by a margin of 55%, and so on until the end of training. On the other hand, AlphaZero just updates its neural network. And the self-play games are generated by the latest update of the neural network. There is no notion of best player.

6 AlphaZero in experimentation

6.1 Model Setup

AlphaZero uses a neural network to represent the inputs (board state) and outputs (moves) of the game. The inputs are a stack of N x N x (MT + L) binary feature planes, repre-

senting the current player's pieces and the opponent's pieces, as well as information about the player's color, total move count, and special rules. The outputs are probability distributions over possible moves, represented by a stack of 8x8x73 (chess), 9x9x139 (shogi), or 19 x 19 + 1 (Go) planes. The policy for each game is represented differently, with illegal moves masked out and probabilities re-normalized. The representation used by AlphaZero was found to be robust in experiments, although other representations could have also been used.

6.2 Training setup

The model was trained for Chess, Shogi and Go (a model for each). There was 700,000 steps of training using 5,000 TPUs in order to do self play and 64 TPU in order to train the neural network.

Throughout the training process, each MCTS conducted 800 simulations. The quantity of games, positions, and contemplation time varied from one game to another, primarily influenced by diverse board dimensions and game durations.

The learning rate for each game was initially set to 0.2 and subsequently decreased three times during the training process (to 0.02, 0.002, and 0.0002). Moves were chosen based on the root visit count proportionally. Dirichlet noise Dir(α) was incorporated into the prior probabilities at the root node; this noise was adjusted inversely according to the estimated number of legal moves in a standard position, resulting in values of $\alpha = 0.3$, 0.15, 0.03 for chess, shogi, and Go, respectively. AlphaZero selected moves based on the root visit count. Each MCTS was run on a single machine with 4 TPUs.

6.3 Results

Elo :

In order to evaluate the performance we use what is called the Elo Rating. We estimate the probability that a player a will defeat player b by a logistic function p(a defeats b) = $\frac{1}{1+\exp(c_{elo}(e(b)-e(a)))}$ and estimate the ratings e($\cdot$) by Bayesian logistic regression, computed by the BayesElo program using the standard constant $c_{elo} = 1/400$.

6.3.1 Performance

In order to compare the performance, Alphazero was compared to Stockfish, Elmo for Shogi and AlphaGo Lee for Go. And each time, the model had one minute per move.

As we can see in 400,000 steps it had a better performance than StockFish in chess. For Shogi, it took AlphaZero only 150,000 steps to overperform Elmo. And lastly for Go, it takes 180,000 steps to Beat AlphaGo Lee and 500,000 steps in order to beat AlphaGO Zero. So we can see that this model can adapt to games in order to give state of the art performances.

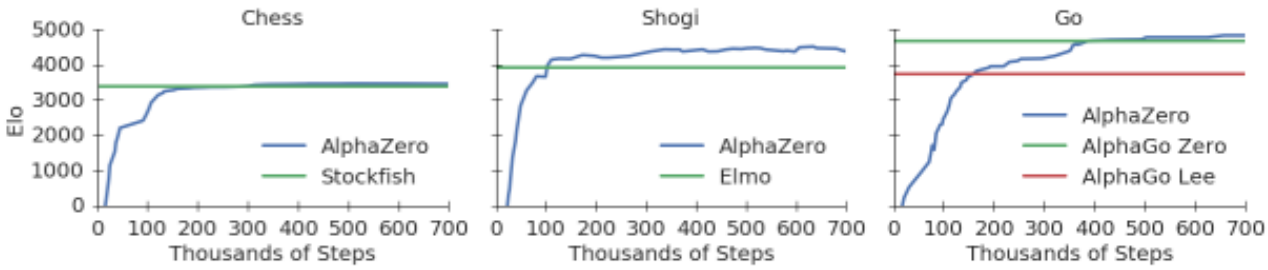

Figure 6: AlphaZero performance compared to state of the art models

Game	White	Black	Win	Draw	Loss
Chess	*AlphaZero*	*Stockfish*	25	25	0
	Stockfish	*AlphaZero*	3	47	0
Shogi	*AlphaZero*	*Elmo*	43	2	5
	Elmo	*AlphaZero*	47	0	3
Go	*AlphaZero*	*AG0 3-day*	31	–	19
	AG0 3-day	*AlphaZero*	29	–	21

Figure 7: Tournament evaluation of AlphaZero in chess, shogi, and Go

An interesting thing to do now is to confront a fully trained AlphaZero to state of the art model in real games. In this setup it faces StockFish (Chess), Elmo (Shogi) and AG0 3-days (Go). And each program has a 1 minute thinking time per move. Moreover, we can see that Alphazero never lost a game of chess against StockFish even with the black pieces, which is impressive. Concerning Shogi, for both sides, the winrate is higly in favor of AlphaZero only losing 8 games in total. Yet, the result is a bit more contrasted for Go as it has an above 50% winrate but does not outperform as it did with other games.

6.3.2 Scalability

Now another thing that can be tested is the scalability of Alpha Zero : how well does it improve its performances with an increase of thinking time.

It is intriguing because MCTS explores only 80,000 positions per second in chess and 40,000 in shogi, which is little in comparison to Stockfish's 70,000,000 and Elmo's 35,000,000. And one could think that if we give more time to think to the model that search quickly they will perform better over time. But in fact, this trend is not real : AlphaZero compensates for the lower number of evaluations by using its deep neural network to focus much more selectively on the most promising variations. Hence we can see that MCTS are better at scaling than the previous state of the art search algorithms for Computer Chess Program that was alpha-beta.

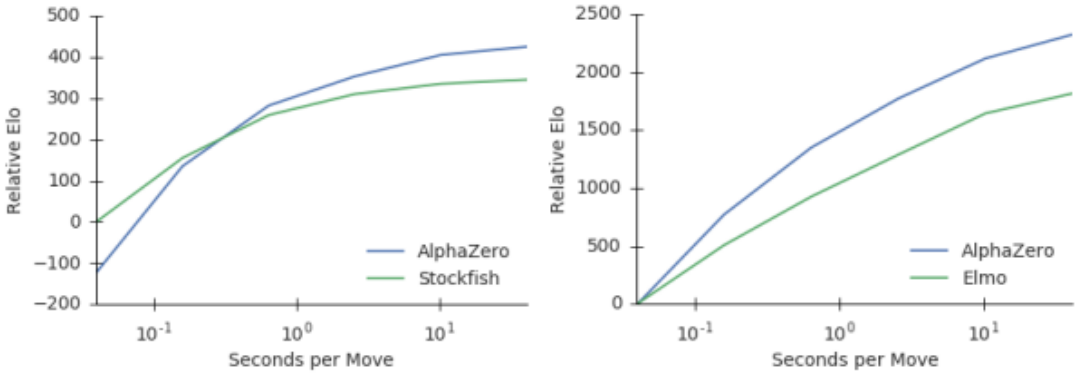

Figure 8: Scalability of AlphaZero with thinking time, measured on an Elo scale

7 Learning capacities

One interesting thing to do when considering Computer chess or any Computer action on game is to compare it to human behavior. In our case, we are trying to make the best possible model, so we need to compare it to the behavior of what is called the GrandMasters of Chess in order to see if it is having any kind of human behavior.

In chess, the title of "Grandmaster" (GM) is the highest title awarded by the Fédération Internationale des Échecs (FIDE), the international chess federation. It is bestowed upon players who have achieved a certain level of skill and proficiency in the game.

In order to do so, we use the tool provided by https://www.chesstree.net/ that permits us to have statistics on games played by Masters from 2005 to 2015.

7.1 A10: English Opening

According to the data, in real life this opening is done 7% of the time which makes it the 4th most popular opening for masters. On the other hand, AlphaZero really takes it as a good opening as it performs it 12% of the time fully trained. Looking at the Masters performances, this opening has a 35% winrate and AlphaZero when it plays it has a 40% winrate. On the other hand, master achieve to win over this opening 24% of the time, on the other hand, AlphaZero only achieves to win over this 16% of the time, it prefers to draw.

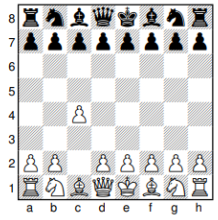

Figure 9: English Opening

7.2 D06: Queens Gambit

According to the data, in real life this opening is done 8% of the time. On the other hand, AlphaZero really takes it as a good opening as it performs it 12% of the time fully trained. So we can see that again it does this opening more than a master player. A master manages to win 35% of the time with this, and AlphaZero matches this performance with 32%. Like before it is underperforming when defending this opening.

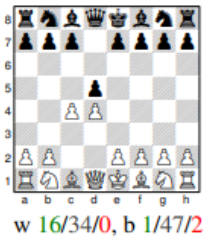

Figure 10: D06: Queens Gambit

7.3 A46: Queens Pawn Game

According to the data, in real life this opening is done 4% of the time. On the other hand, AlphaZero sees a viable opening as it has 6% of the time fully trained. So we see that Alphazero is still overdoing this opening, yet the margin is lower than the other openings. As for Masters, this opening mainly leads to draw either defending or attacking.

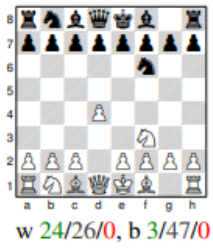

Figure 11: A46: Queens Pawn Game

7.4 E00: Queens Pawn Game

According to the data, in real life this opening is done 7.5% of the time. On the other hand, AlphaZero only uses it 4% of the time fully trained. Likewise, Masters and Alphazero mainly have a draw after this opening.

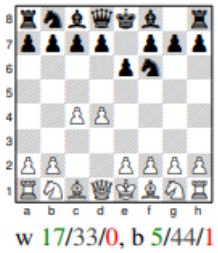

Figure 12: E00: Queens Pawn Game

7.5 E61: Kings Indian Defence

According to the data, in real life this opening is done 4% of the time. On the other hand, AlphaZero really takes it as a good opening as it performs it 0% of the time fully trained. This time, not only is this opening underplayed, but it's not played at all. It can be understandable as it mainly draws, and has the worst winning rate for attack and defence that we saw.

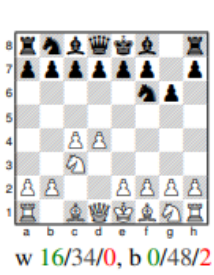

Figure 13: E61: Kings Indian Defence

7.6 C00: French Defence

According to the data, in real life this opening is done 5% of the time. On the other hand, AlphaZero never plays it when fully trained. So it is performing a lot better than Master player, which seems logic, as AlphaZero has a 78% winrate with this opening whereas Master only have a 34% winrate with it. Yet alphazero doesn't play it.

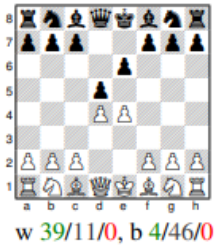

Figure 14: C00: French Defence

7.7 The rest of the openings :

For the rest of the openings (B50: Sicilian Defence, B30: Sicilian Defence, B40: Sicilian Defence, C60: Ruy Lopez, B10: Caro-Kann Defence, A05: Reti Opening) we have a similar behavior for all of them : AlphaZero never plays them even if for some, during the training at a time it plays them a lot (C60: Ruy Lopez). All of those openings lead to quite bad statistics in terms of winrate (either in defence or attack). Moreover, it achieves to lose a game while starting with some of those openings, which was never the case before.

7.8 Conclusion about the learning capacities

Looking at what we saw right above, we see that AlphaZero achieves to learn some of the main openings played in chess games. Yet, we see that for most of the openings, even if at one time in the training it learned about them, AlphaZero took the decision not to play them anymore while fully trained. More over, we see that often it prefers to draw than to lose, we can easely see that when alphaZero is starting as Black, it rarely loses (way less than any master) but it wins a few games on black. This is logical since we say that

a draw is neutral whereas a lose is negative. It also tells us that AlphaZero is playing in a less risky way than masters : in some case in a tournament a win is needed and they take more risk.

8 Beyond Alphazero

We are now going to first explore the possibilities that AlphaZero opened and then we will view other architecture that were created after to outperform AlphaZero.

8.1 MuZero

The researchers that came up with AlphaZero decided to go even further in their work by, two years after the release of AlphaZero, putting out a new model : MuZero [7]. This was seen as a significant step forward in the pursuit of general-purpose algorithms. MuZero masters Go, chess, shogi and Atari without needing to be told the rules, thanks to its ability to plan winning strategies in unknown environments, for instance in a chess game, the algorithm is unaware of the legal moves or what constitutes as win, draw or a loss.

And as shown in the next figure, we see that having even less information doesn't lead to worse performance, indeed, we see that it matches the performance of AlphaZero in both Shogi and Chess, while it outperforms it in Go. We also see that by not asking for rules, MuZero can be used on many games and not only board game as we can see it was tested on Atari's games. The MuZero algorithm, the successor to AlphaZero,

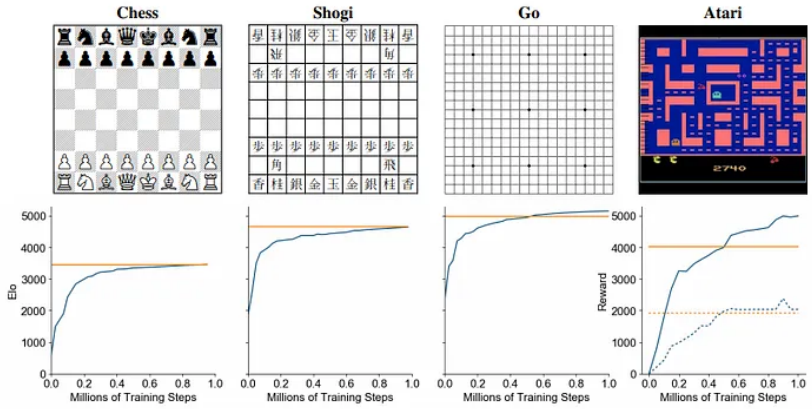

Figure 15: Evaluation of MuZero throughout training in chess, shogi, Go and Atari. The x-axis shows millions of training steps. For chess, shogi and Go, the y-axis shows Elo rating, established by playing games against AlphaZero using 800 simulations per move for both players. MuZero's Elo is indicated by the blue line, AlphaZero's Elo by the horizontal orange line. For Atari, mean (full line) and median (dashed line) human normalized scores across all 57 games are shown on the y-axis and the orange is the performance of R2D2

follows much of the similar approaches of AlphaZero, however a key difference is the use

of a learned model to improve its training system. Additionally, it is able to achieve this ability to plan and predict the future accurately with the help of its three neural networks, unlike its predecessor AlphaZero which makes use of only one. The objective of the MuZero algorithm is to effectively forecast key aspects and specifics of future events essential for strategic planning. Initially, the algorithm takes input, such as an image depicting a chess board, and transforms it into a concealed state. This concealed state then evolves through iterations based on the preceding state and a proposed succeeding course of action. With each update of the concealed state, the model forecasts three factors: policy, value function, and immediate reward. The policy determines the subsequent move, the value function predicts the eventual winner, and the immediate reward gauges the move's effectiveness in enhancing the player's position. Subsequently, the model undergoes training to precisely anticipate the values of these three variables.

8.2 Stockfish NNUE

As we saw during this paper's first parts and also during each of the comparisons, Stockfish was at first the baseline for chess. It was beaten by AlphaZero, but its creators didn't choose to stop enhancing the qualities of Stockfish and decided to create Stockfish NNUE.

Stockfish NNUE (Efficiently Updatable Neural Network) is an advanced version of the Stockfish chess engine, one of the strongest open-source chess engines available. NNUE is a significant enhancement in Stockfish's ability to evaluate positions and make moves during a chess game.

Traditionally, chess engines like Stockfish have relied on handcrafted evaluation functions, which involve assigning values to different features of a chess position, such as piece mobility, pawn structure, king safety, and so on. While these methods have been very effective, they are limited by human understanding and the complexity of chess.

NNUE, on the other hand, incorporates a neural network-based evaluation function. It employs a neural network to directly evaluate a chess position based on the board state rather than predefined features. This neural network is trained using vast amounts of human-played games, allowing it to learn and generalize patterns and evaluations more effectively than handcrafted functions.

The "Efficiently Updatable" part of NNUE refers to the ability to update the neural network efficiently with new data or improved algorithms, allowing Stockfish to continually improve its evaluation capabilities.

Yet, even if this new version of stockfish took lessons from AlphaZero by implementing a neural network into its architecture, it still uses the traditional AB-minimax pruning search technique. Moreover since it uses a tiny network, it's able to perfrom really efficient searches (better than alphazero). Also, it uses an efficent-updating strategy : Indeed, when AlphaZero wants to evaluate a position, it calculates the entire network from scratch even if the changes on the board are not relevant, whereas StockFish NNUE is only recalculating important parts of the network, and assumes the parts that aren't important, can just reuse the values from the previous position. So it has a small network to begin with, and only has to calculate part of it for each move.

All of this permits the use of a CPU in order to train StockFish NNUE.

Hence this model is more performant than AlphaZero because it combines the strenght of AlphaZero (the neural networks) and the strenght of the previous architecture (quickly evaluate huge numbers of positions).

9 Connect 4 game

9.1 Implementation of Connect 4 game

As part of this project, we implemented a Connect 4 game that operates with a simple version of the AlphaZero algorithm. To do this we followed freeCodeCamp tutorial [8], originally intented for the TicTacToe game, and which we adapted for Connect 4 game.

Firstly, we first implemented a class called `Connect4`, where we established the basic rules of the game, including functions like `get_legal_actions()` and `check_win()` functions. To detect if 4 pawns were aligned consecutively, whether it is in diagonally or horizontally or vertically, we used a kernel of size 4 and convolve2d layers.

```python
class Connect4:

    def __init__(self):
        self.row_count = 6
        self.column_count = 7
        self.action_size = self.column_count

    def get_initial_state(self):
        return np.zeros((self.row_count, self.column_count))

    def get_next_state(self, state, action, player):
        #action integer : the chosen column
        column = action
        for row in range(self.row_count-1, -1, -1):
            if state[row, column] == 0:
                state[row, column] = player
                break
        return state

    def get_legal_actions(self, state):
        return np.array([state[0, col] == 0 for col in range(self.column_count)])


    def check_win(self, state, action):
        if action == None :
            return False

        kernel = np.ones((1, 4), dtype=int)

        # Horizontal and vertical checks
        horizontal_check = convolve2d(state, kernel, mode='valid')
        vertical_check = convolve2d(state, kernel.T, mode='valid')

        # Diagonal checks
        diagonal_kernel = np.eye(4, dtype=int)
        main_diagonal_check = convolve2d(state, diagonal_kernel, mode='valid')
        anti_diagonal_check = convolve2d(state, np.fliplr(diagonal_kernel), mode='valid')

        # Check for winner
        if any(cond.any() for cond in [horizontal_check == 4, vertical_check == 4, main_diagonal_check == 4,
anti_diagonal_check == 4]):
            return 1
        elif any(cond.any() for cond in [horizontal_check == -4, vertical_check == -4, main_diagonal_check == -4,
anti_diagonal_check == -4]):
            return -1

        # No winner
        return 0

    def get_value_and_terminated(self, state, action):
        if self.check_win(state, action) :
            # value = 1 , terminated = True
            return 1, True
        if np.sum(self.get_legal_actions(state)) == 0:
            #pas de gagnant value = 1, terminated = True
            return 0, True
        return 0, False

    def get_opponent(self, player):
        return -player

    def get_opponent_value(self, value):
        return -value

    def change_perspective(self, state, player):
        return state * player

    def get_encoded_state(self, state):
        encoded_state = np.stack(
            (state == -1, state == 0, state == 1)
        ).astype(np.float32)
        return encoded_state
```

Figure 16: Class `Connect4`

We then implemented two neural networks : `Resblock` and the `Resnet` classes. These neural networks architectures enable to train deep neural networks and often used for image recognition. In our implementation of the AlphaZero network, Resnet takes in input a matrix that represents the board state and return to outputs :

- the value of the position of the pawn : either 1 if player 1 wins or -1 if player 1 looses.
- a policy,that represents a vector a prior probabilities for each next legal move.

```python
class ResBlock(nn.Module):
    def __init__(self, num_hidden):
        super().__init__()
        self.conv1 = nn.Conv2d(num_hidden, num_hidden, kernel_size = 3, padding = 1)
        self.bn1 = nn.BatchNorm2d(num_hidden)
        self.conv2 = nn.Conv2d(num_hidden, num_hidden, kernel_size = 3, padding = 1)
        self.bn2 = nn.BatchNorm2d(num_hidden)

    def forward(self, x):
        residual = x
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.bn2(self.conv2(x))
        x += residual
        x + F.relu(x)
        return x
```

Figure 17: Class `Resblock`

```python
class ResNet(nn.Module):
    def __init__(self, game, num_resBlocks, num_hidden):
        super().__init__()
        self.startBlock = nn.Sequential(
            nn.Conv2d(3, num_hidden, kernel_size=3, padding=1),
            nn.BatchNorm2d(num_hidden),
            nn.ReLU()
            )
        self.backBone = nn.ModuleList(
            [ResBlock(num_hidden) for i in range(num_resBlocks)]
        )
        self.policyHead = nn.Sequential(
            nn.Conv2d(num_hidden, 32, kernel_size = 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(32 * game.row_count * game.column_count, game.action_size)
        )
        self.valueHead = nn.Sequential(
            nn.Conv2d(num_hidden, 3,  kernel_size=3, padding=1),
            nn.BatchNorm2d(3),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(3 * game.row_count * game.column_count, 1),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.startBlock(x)
        for resBlock in self.backBone:
            x = resBlock(x)
        policy = self.policyHead(x)
        value = self.valueHead(x)
        return policy, value
```

Figure 18: Class `Resnet`

In order to run a MCTS algorithm (Monte-Carlo Tree Search) algorithm, we first needed to implement a class called `Node`, that implement the 3 main steps of the MCTS algorithm for the AlphaZero model :

- Selection of a node : `select()`

- Expansion of a node : `expand()`
- Backpropagation : `backpropagate()`

The function `get_ucb()` enables to compute the ucb score of a node following this formula :

$$v_i + C \times \frac{\sqrt{\ln N}}{n_i}$$

with v_i the estimated value of the node i, n_i is the number of the times the node has been visited and N is the total number of times that its parent has been visited.

```python
class ResNet(nn.Module):
    def __init__(self, game, num_resBlocks, num_hidden):
        super().__init__()
        self.startBlock = nn.Sequential(
            nn.Conv2d(3, num_hidden, kernel_size=3, padding=1),
            nn.BatchNorm2d(num_hidden),
            nn.ReLU()
            )
        self.backBone = nn.ModuleList(
            [ResBlock(num_hidden) for i in range(num_resBlocks)]
        )
        self.policyHead = nn.Sequential(
            nn.Conv2d(num_hidden, 32, kernel_size = 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(32 * game.row_count * game.column_count, game.action_size)
        )
        self.valueHead = nn.Sequential(
            nn.Conv2d(num_hidden, 3,  kernel_size=3, padding=1),
            nn.BatchNorm2d(3),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(3 * game.row_count * game.column_count, 1),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.startBlock(x)
        for resBlock in self.backBone:
            x = resBlock(x)
        policy = self.policyHead(x)
        value = self.valueHead(x)
        return policy, value
```

Figure 19: Class `Node`

We then implemented the MCTS algorithm adapted to the AlphaZero algorithm. The steps of the algorithm are the same as the classical MCTS algorithm, however the step of the simluation of a playout doesn't exist anymore.

```python
class MCTS:
    def __init__(self, game, args, model):
        self.game = game
        self.args = args
        self.model = model

    @torch.no_grad() # don't want to use policy and value for training
    #faster
    def search(self, state):
        #define root node
        root = Node(self.game, self.args, state)
        for search in range(self.args['num_searches']):
            node = root
            # selection
            while node.is_fully_expanded():
                node = node.select()
                #chcek if the node is a terminated one
                #action taken -> action of the opponent
            value, is_terminal = self.game.get_value_and_terminated(node.state, node.action_taken)
            value = self.game.get_opponent_value(value)

            # expansion
            if not is_terminal:
                policy, value = self.model(
                    torch.tensor(self.game.get_encoded_state(node.state)).unsqueeze(0)
                )
                policy = torch.softmax(policy, axis=1).squeeze(0).cpu().numpy()
                legal_actions = self.game.get_legal_actions(node.state)
                policy += legal_actions
                policy /= np.sum(policy)

                #just want to get the float from the value head
                value = value.item()

                node.expand(policy)
                #we remove the simulation part

            # backpropagation
            node.backpropagate(value)

        # return visit_counts
        action_probs = np.zeros(self.game.action_size)
        for child in root.children:
            action_probs[child.action_taken] = child.visit_count
        action_probs /= np.sum(action_probs)
        return action_probs
```

Figure 20: `MCTS` algorithm adapted to AlphaZero

Finally we implemented the `AlphaZero` class, that contains the 3 main steps of the algorithm :

- Self-play step `selfPlay` : the algorithm plays against itself and learn from the obtained outputs of these playouts.
- Training phase `train` : The model is trained on the outputs of the `selfPlay` function.
- Learning phase `learn` : the checkpoints of the model are saved in model_iteration.pt file.

```python
class AlphaZero:
    def __init__(self, model, optimizer, game, args):
        self.model = model
        self.optimizer = optimizer
        self.game = game
        self.args = args
        self.mcts = MCTS(game, args, model)

    def selfPlay(self):
        memory = []
        player = 1
        state = self.game.get_initial_state()

        while True:
            neutral_state = self.game.change_perspective(state, player)
            action_probs = self.mcts.search(neutral_state)

            memory.append((neutral_state, action_probs, player))
            action = np.random.choice(self.game.action_size, p=action_probs)
            state = self.game.get_next_state(state, action, player)
            value, is_terminal = self.game.get_value_and_terminated(state, action)

            if is_terminal:
                returnMemory = []
                for hist_neutral_state, hist_action_probs, hist_player in memory:
                    hist_outcome = value if hist_player == player else self.game.get_opponent_value(value)
                    returnMemory.append((
                        self.game.get_encoded_state(hist_neutral_state),
                        hist_action_probs,
                        hist_outcome
                    ))
                return returnMemory

            player = self.game.get_opponent(player)

    def train(self, memory):
        random.shuffle(memory)
        for batchIdx in range(0, len(memory), self.args['batch_size']):
            sample = memory[batchIdx: min(len(memory)-1,batchIdx + self.args['batch_size'])]
            state, policy_targets, value_targets = zip(*sample)

            state, policy_targets, value_targets = np.array(state), np.array(policy_targets),
 np.array(value_targets).reshape(-1,1)
            state = torch.tensor(state, dtype = torch.float32)
            policy_targets = torch.tensor(policy_targets, dtype = torch.float32)
            value_targets = torch.tensor(value_targets, dtype = torch.float32)

            out_policy, out_value = self.model(state)
            policy_loss = F.cross_entropy(out_policy, policy_targets)
            value_loss = F.mse_loss(out_value, value_targets)
            loss = policy_loss + value_loss

            #minimize the loss by backpropagating
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()


    def learn(self):
        for iteration in range(self.args['num_iterations']):
            print("iteration :", iteration)
            memory = []

            self.model.eval()
            for selfPlay_iteration in trange(self.args['num_selfPlay_iterations']):
                print("selfPlay_iteration :", selfPlay_iteration)
                memory += self.selfPlay()

            self.model.train()
            for epoch in trange(self.args['num_epochs']):
                print("epoch :", epoch)
                self.train(memory)

            torch.save(self.model.state_dict(), f"model_{iteration}.pt")
            torch.save(self.optimizer.state_dict(), f"optimizer_{iteration}.pt")
```

Figure 21: `AlphaZero` class

9.2 Results

As part of the experiment, we decided to compare the policy of two models :

- A simple MCTS algorithm that contains the 4 standard steps :
  - Selection
  - Expansion

  - Simulation
  - Backpropagation
- AlphaZero algorithm

The state is the following : Player 1 has placed a pawn in the second column, and Player 2 has placed a pawn in the sixth column. Here are the policies obtained by those two models for the next action of player 1 :

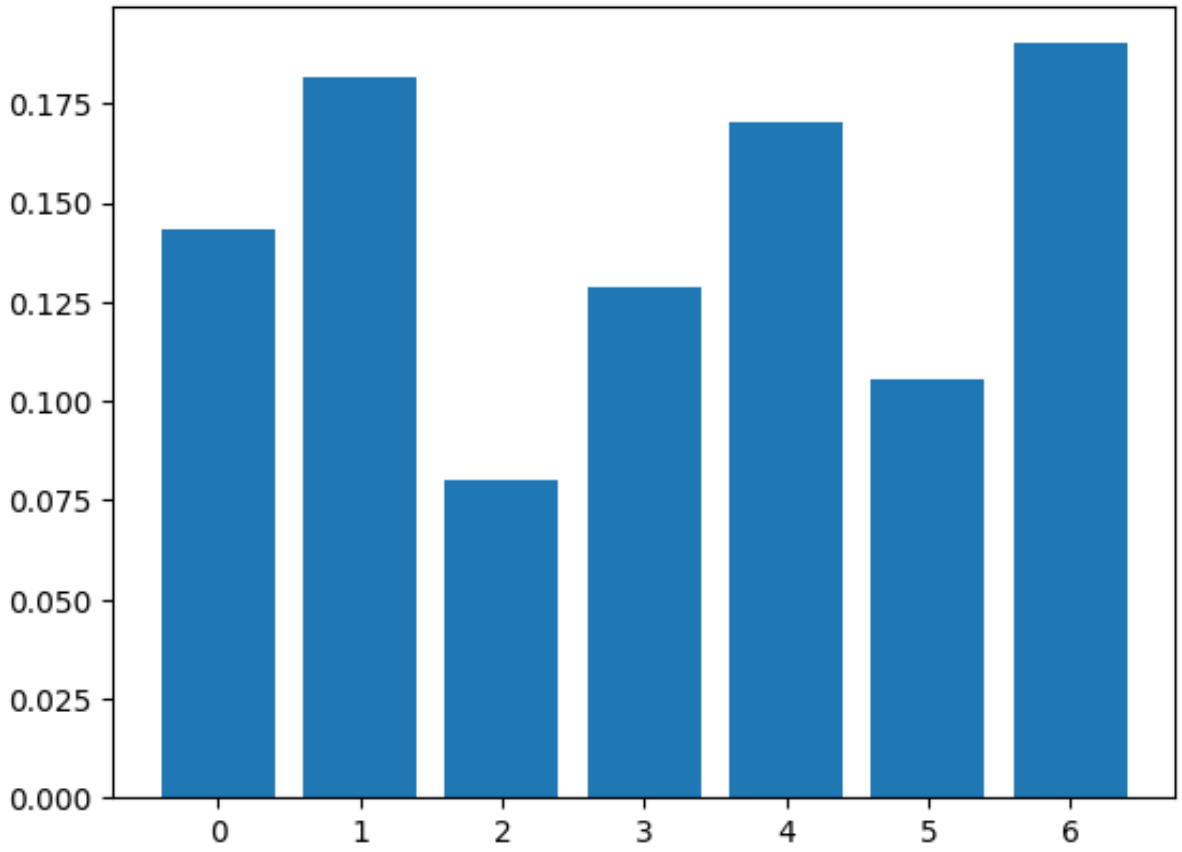

Figure 22: Policies for the next position obtained with `MCTS` algorithm

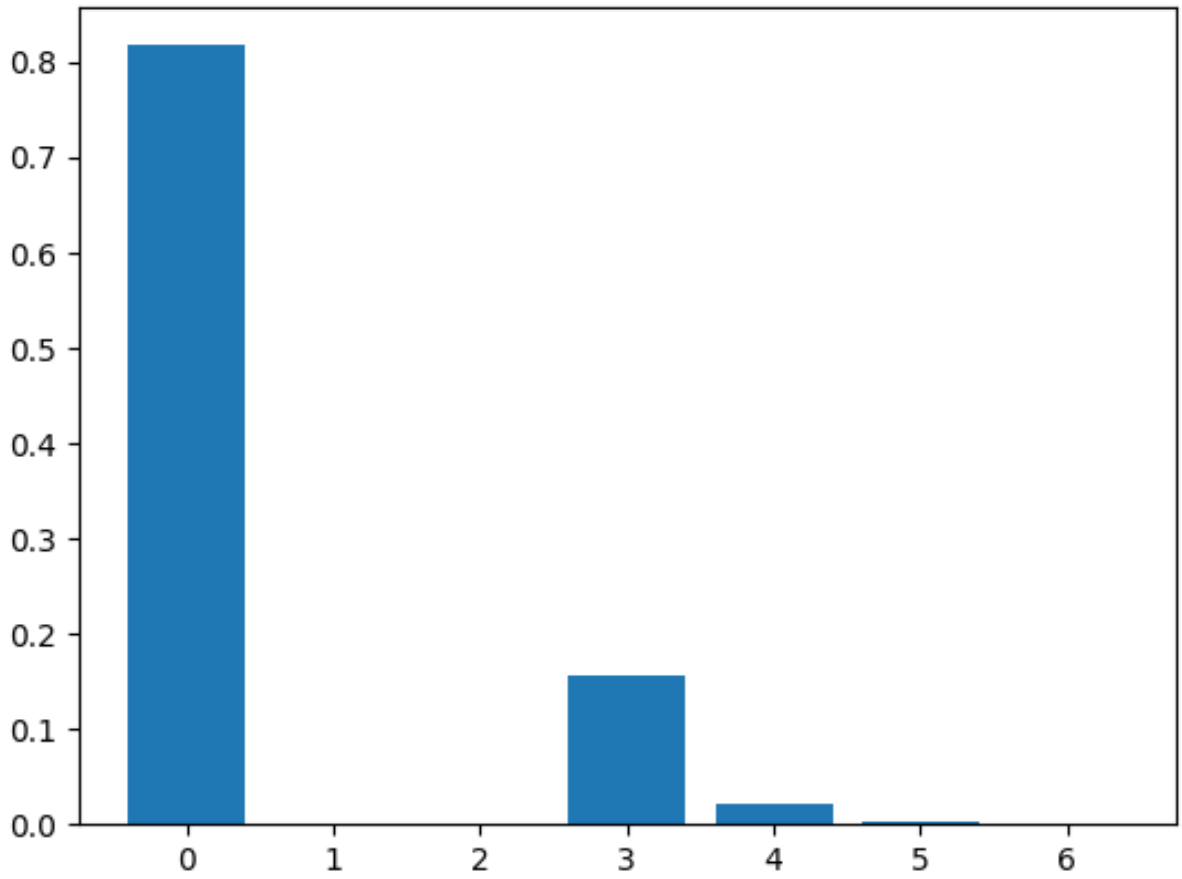

Figure 23: Policies for the next position obtained with `AlphaZero` model

We notice that the classical MCTS model doesn't not really give a sure policy for the next action, as the probabilities to put a pawn in each column are nearly equal. On the contrary to the AlphaZero model we implemented, a probability of 0.8 is generated to place a pawn in the first column. Therefore, as expected the AlphaZero model is more accurate as it gives a more "thoughtful" decision than a random one. However, since we trained the model only on a few epochs, the model is still not very efficient.

References

[1] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. arXiv: 1712.01815 [cs.AI].

[2] S. Thrun, "Learning to play the game of chess," in *Neural Information Processing Systems*, 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:12298610.

[3] J. Baxter, A. Tridgell, and L. Weaver, "Learning to play chess using temporal differences," *Machine Learning*, vol. 40, pp. 243–263, 2000. [Online]. Available: https://api.semanticscholar.org/CorpusID:10389798.

[4] J. Veness, D. Silver, W. T. B. Uther, and A. D. Blair, "Bootstrapping from game tree search," in *Neural Information Processing Systems*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:7493916.

[5] O. E. David, N. S. Netanyahu, and L. Wolf, "Deepchess: End-to-end deep neural network for automatic learning in chess," in *Lecture Notes in Computer Science*. Springer International Publishing, 2016, pp. 88–96, ISBN: 9783319447810. DOI: 10.1007/978-3-319-44781-0_11. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-44781-0_11.

[6] S. D., H. A., and M. C. et al., *Mastering the game of go with deep neural networks and tree search*, 2016. [Online]. Available: https://doi.org/10.1038/nature16961.

[7] J. Schrittwieser, "Mastering atari, go, chess and shogi by planning with a learned model," 2020.

[8] R. Förster. "Alphazero from scratch – machine learning tutorial." (2023), [Online]. Available: https://www.youtube.com/watch?v=wuSQpLinRB4.