SigmaZero
A General Reinforcement Learning Based Chess Engine

Aditya Yadav *210050006*
Hrishikesh Jedhe Deshmukh *210050073*
Kartik Nair *210050083*
Prerak Contractor *210050124*
Rayane Tayache *23V0016*

***Abstract*—The game of chess is very widely studied in artificial intelligence. In 2010s there was a breakthrough in programs learning using reinforcement learning with no human involvement and biases. This improvement was enabled using Monte Carlo Tree Search and deep neural networks for searching and evaluation. In this project, we explore the application of self-play based reinforcement learning techniques to build a chess engine from scratch, inspired by the groundbreaking AlphaZero, that was able to achieve, *tabula rasa*, superhuman performance. We train a program to learn to play chess starting with knowledge of only the rules and learns via self-play only.**

I. Introduction

Chess has been the grand challenge task for a generation of artificial intelligence researchers, culminating in high-performance chess engines that perform at superhuman level. In 1997, DeepBlue defeated the human world champion, Garry Kasparov, 3.5-2.5. Over the following years the strength of chess engines kept increasing steadily past the human level.

However, these programs are heavily specialized to their domain and do not generalise well. These programs evaluate positions using features handcrafted by human grandmasters and carefully tuned weights, combined with a high-performance alpha-beta search[5] that expands a vast search tree using a large number of clever heuristics and domain-specific adaptations. An ambition of artificial intelligence has been to create programs that can learn for themselves from basic rules.[7]

AlphaGo[11] is a program developed by DeepMind to the game of Go. In 2016, AlphaGo became the first computer program to beat a 9-dan Go player. It uses a combination of machine learning and tree search techniques, combined with extensive training, both from human and computer play. It uses Monte Carlo tree search (MCTS), guided by a value network and a policy network, both implemented using deep neural networks. This was followed by AlphaGo Zero which was able to achieve superhuman performance using deep convolutional neural networks and training solely from games of self-play by reinforcement learning. In 2017, DeepMind generalized it's approach into an AlphaZero[10] algorithm that achieved within 24 hours a superhuman level of play in the games of chess, shogi, and Go by defeating world-champion programs, Stockfish, Elmo, and 3-day version of AlphaGo Zero in each case.

Instead of a handcrafted evaluation function and move ordering heuristics, AlphaZero utilises a deep neural network. This neural network takes the board position s as an input and outputs a vector of move probabilities p for each action a, and a scalar value v estimating the expected outcome z from position $s, v \approx E[z|s]$. AlphaZero learns these move probabilities and value estimates entirely from self- play; these are then used to guide its search.

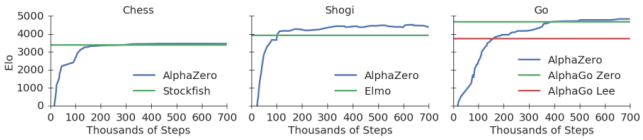


Fig. 1: Training AlphaZero for 700,000 steps. Elo ratings were computed from evaluation games between different players when given one second per move [10]

The aim of this project is to recreate (on a smaller scale) the results of AlphaZero, that is, to train a program to learn to play chess starting from the knowledge of game rules only and learning through self-play exclusively. We also aim to experiment with some modifications while implementing the algorithm

II. Background

A. MCTS

Monte Carlo tree search (MCTS) is a heuristic search algorithm that is commonly used on deterministic games.

It works by building a search tree incrementally, starting from the current state of the game. At each iteration, it performs four steps: selection, expansion, simulation, and backpropagation.

- Selection : A node in the tree is selected, based on a balance between exploration and exploitation. MCTS uses an upper confidence bound (UCB) formula, such as UCT [6] (Upper Confidence bounds applied to Trees), to determine which node to select.
- Expansion: MCTS adds one or more child nodes to the selected node, representing possible actions or moves from that state.
- Simulation/Rollout: MCTS randomly plays out the game from the expanded node until a terminal state or a predefined depth is reached.
- Backpropagation: MCTS updates the information of the nodes visited during the simulation, such as the number of visits and the average reward or value. This information is used to guide the selection of nodes in future iterations.

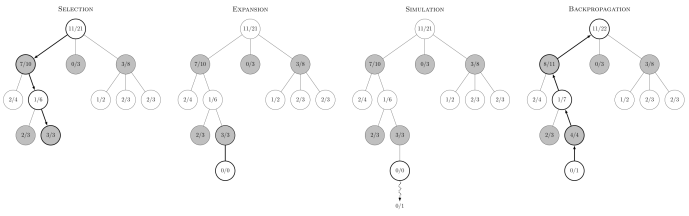


Fig. 2: Standard MCTS [9]

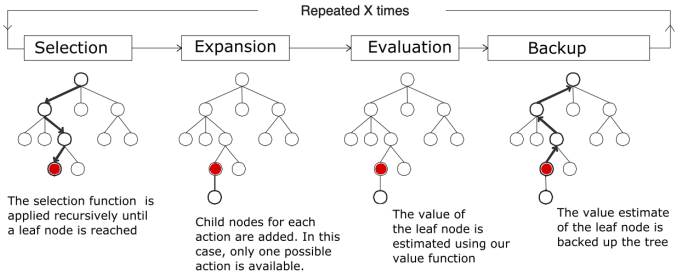


Fig. 3: MCTS for AlphaZero[14]

MCTS repeats these four steps until a time limit or a computational budget is reached. Then, it chooses the best action or move based on the information stored in the root node of the tree.

For the purposes of standard 8x8 chess, however, the Selection and Rollout/Simulation Steps of the MCTS are extremely expensive. This is because

1) The number of actions possible are generally too large.
2) Simulation may involve too many steps as the length of the game may be too long with random moves.

Thus modifications are made to the MCTS algorithm by AlphaZero which are:

1) In the simulation step, a neural network is used to predict the next moves. The probabilities predicted for each action are used as the prior distribution for the MCTS.
2) Instead of a rollout, the value predicted by the neural network is utilised for updating the search tree.

B. Neural Network and Representation

1) Architecture: The neural network consists of a *deep convolutional block*, a *policy head* and a *value head*.

The game state is firstly converted to its tensor representation of shape $M \times 8 \times 8$. This is encoded into a latent space representation with the deep convolutional block.

The convolutional block consists of 19 Residual Blocks[3], each consisting of 2 convolutional blocks and a skip connection. Each convolutional block consists of 2 2D convolutional layers with kernel size 3 and 256 output channels, Batch Norm layers, and a ReLU activation.

This encoding is passed to the policy head and value head which produce a policy output and value output. The policy head consists of a convolutional layer and batch norm layer with ReLU activation followed by a linear layer. The output activation is sigmoid. The value head consists of another convolutional and batch norm layer with relu activation, followed by two linear layers. The output activation is tanh. The output to policy is a $73 \cdot 8 \cdot 8$ flattened tensor with each corresponding to a different action. The legality of these moves are not explicitly enforced here.

2) Input Representation: We represent each state as a stack of 8×8 planes. For a given board, each of the 12 different types of pieces (6 of each colour) correspond to a different plane with a value of 0 or 1 based on the presence of the piece on the corresponding square.

Additional game state variables are represented by filling the corresponding plane uniformly.

The original AlphaZero considers the most recent 8 board states in its history. Additional planes for repetition counter (for each of the 8 states and 2 players), as well as 7 additional planes are maintained (colour, move counter, 4 castling rights, no-progress count). Thus, here $N = 119$.

Given hardware and time constraints we eschewed history representation to reduce the need for more training data. Thus in our case, the input has $N = 19$ planes.

3) Output Representation: The output of the value head is a float that ranges from -1 to $+1$

The output of the policy head is a flattened $73 \cdot 8 \cdot 8$ tensor corresponding to 73 8×8 planes. Each of the plane corresponds to a different type of move - relative to the starting square . And for a given index, it corresponds to the corresponding move on a piece belonging on that square, i.e. for each possibilty for final-square − initial-square. There are 56 "Queen type moves" - i.e vertical, horizontal, and diagonal. There are 8 "Knight Type" moves. For pawns, there are additional possibilities involving promotion. Hence the default is considered as a queen promotion, and 9 additional planes are for each type of underpromotion - rook, knight, bishop and the straight and diagonal moves.

The policy output is interpreted as probabilities for each move.

Note that many of the indices of the output tensor will never correspond to any legal moves. And for a given state, most of the moves will be illegal. Legality is checked by the environment, and not the neural network, and it is expected that it will learn the rules of the game with the appropriate loss function.

III. Methodology & Implementation

We referred to an open-source implementation of AlphaZero by Tuur Vanhoutte[13] along with his bachelor thesis [14] on the same while implementing SigmaZero.

A. MCTS

Monte Carlo Tree Search has three major steps. The evaluation step is ultimately done by agent and thus we are not describing it in detail here. We implemented the entire MCTS algorithm in C++ and interfaced it with Python using the Boost library[1].

1) Selection of child: At every node, we select the *best* child of that node to create a game path. *Best* can be decided by various metrics depending on if we want exploitation or exploration. During exploitation we choose the node corresponding to the edge having maximum value of $\frac{W_e}{N_e+1}$ where W_e is the value of the action-value of the edge and N_e the number of times the edge is visited. To increase exploration, we also inject Dirichlet noise into the probability distribution. Let d be the Dirichlet noise and r be the exploration rate, using it we define an upper confidence bound.

$$UCB = r \cdot d \cdot \sqrt{\frac{N_n}{N_e + 1}}$$

N_n is the number of times we have visited the node that we are currently on, while N_e is the number of times of we have visited the corresponding edge. We either add or subtract the UCB from $\frac{W_e}{N_e+1}$ to decide the next edge in the game path. We add it in case of white's turn, while subtract in case of black's turn. Above procedure is recursively done until we reach a leaf node.

2) Expand: After selecting children recursively, we reach a leaf node. We check if the leaf node is a terminal node (corresponding to win, loss or a draw) i.e. no valid moves are possible. If it does we set its value appropriately and move onto the backpropagation step. If valid moves are possible, we create edges and nodes corresponding to all the valid moves. Newly created nodes are assigned values predicted by the agent. We now move onto the backpropagation step.

3) Backpropagate: We backtrack along the game path that we had stored. The visit counts N_n and N_e for each edge and node encountered on the path is incremented by one. The value of the leaf node is added to the action-value of each edge W_e

Above four steps are repeated for `NUM_ITER` times whose value is typically kept 400 or 800. In our training and evaluation, we set its value to 800.

B. Training on Puzzles

We recognised the inefficiency of training the model solely through end-to-end game simulations. We thus altered the base training protocol so that our network can recognize patterns include checkmate scenarios, piece advantages, threats, and positional advantages faster.

Consequently, we incorporated self-play runs that initialised the board from intermediate positions, leveraging the Lichess puzzle dataset [8] for this purpose. Our training approach involves allocating a fraction p of the training to puzzle-based runs and the remaining fraction $(1-p)$ to end-to-end game simulations. In our experiments we tweak this value of p.

C. Training and Evaluation

1) Training: Training primarily consisted of two aspects: Collecting experiences and optimising model on collected experiences. These two were performed in parallel.

Collecting experiences involved letting two instances of the current best-model play against each other. The initial position was either chosen to be a starting position from a puzzle with probability $p = 0.6$, or the starting board position with probability $1 - p$. After each move, the tuple $\langle s, \boldsymbol{\pi} \rangle$, where s is the state of board before the move, and $\boldsymbol{\pi}$ the posterior probability distribution over legal actions obtained by performing MCTS simulations, is stored in a memory. Each game was played till either termination by draw or win, or truncated after a maximum of $T = 75$ moves, after which the winner is approximated using an evaluation function. The result z is then back-propagated to all the tuples in the memory, and the modified tuples $\langle s, \boldsymbol{\pi}, v \rangle$ are stored in the memory. The memory is then saved to a disk, later to be used in training.

During each training phase, all the experience tuples[1] of the form $\langle s, \boldsymbol{\pi}, v \rangle$ collected so far were aggregated in a replay buffer. Mini-batches of size N were sampled randomly from the replay buffer, and the model was optimised on the sampled experiences. The loss function for the model (parameterised by weights θ) is:

$$L_\theta = (v - z_\theta)^2 - \boldsymbol{\pi}^T \log(\boldsymbol{p}_\theta) + \lambda ||\theta||_2^2 \tag{1}$$

where z_θ is the value estimation, and $\boldsymbol{p}_\theta$ the policy estimation for state s.

Each such backward pass is counted as one time-stamp. One training phase involved optimizing the model for T time-stamps.

The training was done in two stages:

- For the initial stage, the training was done with learning rate $\alpha = 0.02$, mini-batch size $N = 128$ and number of time stamps $T = 40,000$
- For the second stage, the training was done with learning rate $\alpha = 0.002$, mini-batch size $N = 128$ and number of time stamps $T = 100,000$

2) Evaluation: Each training phase was followed by an evaluation phase, where the newly trained model was evaluated against the current best-model. The evaluation consisted of playing $2K$ games between the two agents. K different opening were chosen randomly as starting positions for the game, to add diversity to the evaluation and not let agent be good at few openings only. For each opening, two games were played, by allowing each agent to play once as white and once as black.

The newly trained model replaced the current best-model if it won K or more games. Since playing games during evaluation was a time consuming operation for us, the number of different openings in evaluation phase was set to $K = 3$.

IV. Experiments

Some of the experiments and modifications attempted to try and improve performance or get a speed up for training are:

1) In our first run we trained the model on a large proportion of puzzles over games (Setting the proportion of

[1]These include all the experiences collected so far from all the models. We reused the experiences of the discarded models as well due to time constraints, since collecting new experiences was an expensive operation

puzzles to $p = 0.8$). Since puzzles are relatively short sequences of moves (generally with forced moves) that feature different patterns that appear frequently in real games we expected the neural network used to estimate the value function to learn these patterns like humans do. However, due to an imbalance in the puzzle dataset, most of the puzzles had a winning evaluation for white and the value function learnt this instead of the patterns. As a consequence, it gave a winning evaluation for white in every position. The proportion of puzzles to actual games was hence decreased and regularisation added to the training loss.

2) Initially, a simple evaluation function, which counts the score of each remaining piece and assigns a score of 0.25 to the player with a clear majority, was used to approximate the position in case of truncation. However, this may not be the best evaluation as it only considers the pieces and not other aspects, such as positional advantage, piece development and king threat. Hence, we decided to use Stockfish's position evaluation function [12] to approximate the value of a position [2], in hopes that the value function's training could be accelerated by more accurate estimates in the collected data set.
3) While training we could consider an entire game as a single experience. In this case, the model finds out the outcome only at the end of the game. However, when we backpropagate through the multiple moves of the game it is very difficult to learn which moves led to the result and which ones were counterproductive, for example, the agent could play many good moves but one blunder could lose the game and all the moves will be penalized for a negative result. This slows down the training a lot, therefore, we instead decided to go with each move as a training experience. This allows us to give the model instant feedback rather than wait for the game to finish.

Having incorporated all these changes, the model was trained for roughly one week, collecting experiences from a total of 3227 games, and the plots of the loss function observed for the first stage (with learning rate $\alpha = 0.02$) are shown in figures 4 and 5 and the second stage (with learning rate $\alpha = 0.002$) are shown in figures 6 and 7. The plots show the variation in loss during each time stamp and a running average.

[2]This may be considered as supervision, violating the no supervision rule of AlphaZero

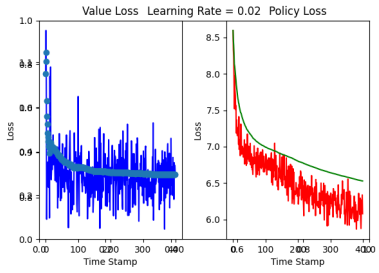


Fig. 4: Loss Function during first training iteration

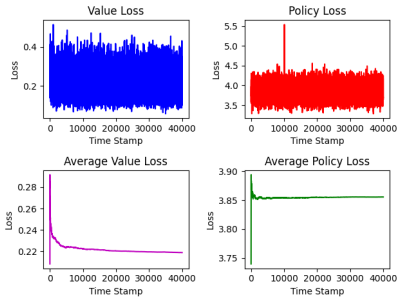


Fig. 5: Converging to a plateau

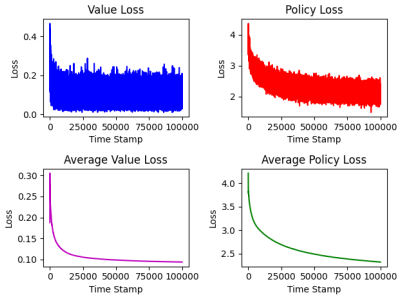


Fig. 6: Decaying learning rate to 0.002

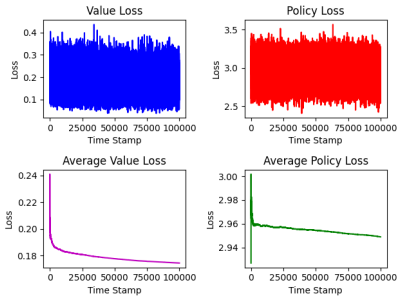


Fig. 7: Final training iteration

V. Results

A. *Policy function learning the rules of chess*

The decrease in the cross entropy loss between MCTS policy (which assigns probabilities only to legal moves) and

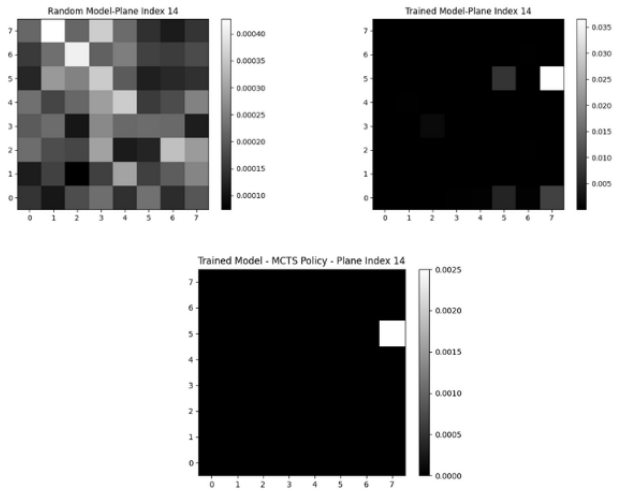


Fig. 8: Board: 4r2k/8/3PP3/P1P5/p3N3/p4P2/2ppp3/K2n1B2 w - - 0 1

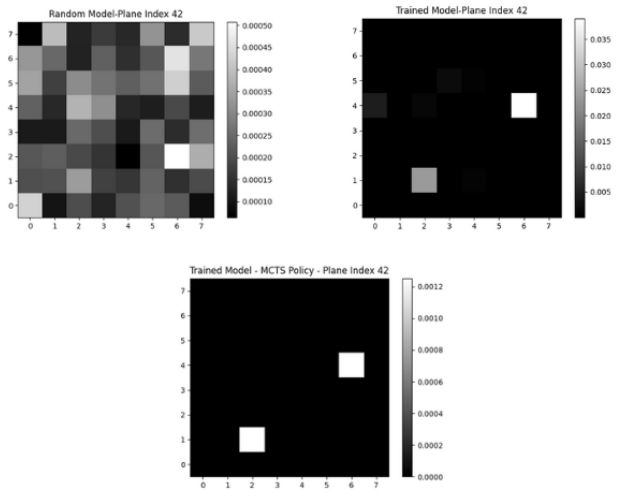
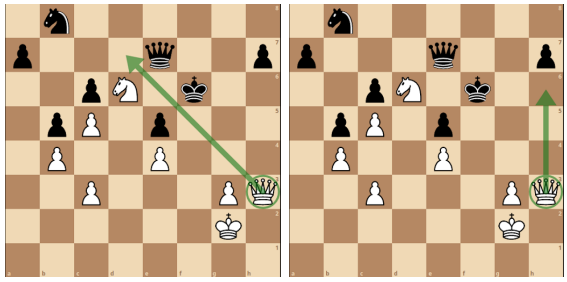


Fig. 9: Board: 4r3/8/qB2P3/pkP2P2/1P1b1P2/8/3pKppp/8 w - - 0 1

the model's policy could be explained by model outputting policies which assign higher probabilities to legal moves. To demonstrate this, a random chess board position is used, and the planes of the action tensor of the trained model are plotted as grey-scale images, with the intensity of a cell being proportional to the probability of the move which the cell corresponds to. The planes were compared with MCTS's action policy planes and a randomly initialised model's action policy planes. The results obtained are shown in figures 8 and 9. It was observed that the trained model assigns higher probabilities to only a few blocks. Some of these still correspond to illegal moves, but it was observed that many of the legal moves also got higher probabilities, from which we can infer that the agent's policy function had started learning to make legal moves.



(a) Random Model: Qd7 (b) Trained Model: Qh6#

Fig. 10

Fig. 11: exf1=N#

B. *One move checkmates*

It was observed that the agent got better at finding one-move mates. A vanilla MCTS working with a randomly initialised model is able to find to find one move check mates in positions having only a few legal moves, but fails when the number of legal moves increases. The trained model however has improved upon this, and is now able to find one move mates in more complex board positions. One such example is shown in figures 10a and 10b.

However, the model is still unable to find longer mate sequences or some other simple tactics.

C. *Pawn Promotion*

It was observed from the evaluation games that the agent learnt that promoting pawn to queen is the best option. However, we observed instances of under promotion if it led to check mate, again validating our hypothesis that the agent had started learning mate in one actions. One such instance is shown in figure 11

However, the agent had not reached human level performance in chess. It still hangs pieces, and does not capture free pieces often. We also observed that the agent prefers king moves. One possible explanation can be attributed to the experiences collected from puzzles, which involved a lot of checks, resulting in a significant number of king moves in the replay buffer. The agent has also not learnt the strategic importance of castling, again due to most puzzles not having castling as a legal move.

A video of a game played by the final agent against a human on our GUI can be found here [3]

[3]Note this is rather slow due to the agent running on CPU.

VI. Future Work and Room for Improvement

We faced constraints related to hardware and compute time while training our model. We also were unable to perform any hyperparameter tuning.

Applying Monte Carlo Tree Search (MCTS) in C++ did provide a significant performance boost over the python implementation, but the task of eliminating other bottlenecks remained. The sequentiality of MCTS posed a challenge, prompting consideration of batch training. Implementing concurrency in some of the steps of MCTS such as expansion proved to be a challenge. A reimplementation of MCTS and the chess environment is preferred, aimed to facilitate more seamless interaction with the agent, eliminating CPU-GPU jumps and avoiding Python chess environment inefficiencies.

The insufficient number of games was identified as a problem. Our agent learnt using only 3227 games compared to 44 million games played by AlphaZero. Given these problems of scale, it may be more reasonable to forego the tabula rasa approach to learning.

One such change is using Stockfish evaluations for learning the value function. This can be done until the MCTS trees stabilize towards near optimal, after which the original training protocol may be used to surpass stockfish. This does induce bias towards existing strategies, but this may be a reasonable trade-off.

We also faced imbalances in the ratio of experiences with white having an advantage v/s black having an advantage, the reason for which we could not explain. Better tuning of puzzles and normal games, along with better protocol for evaluation in case of truncation would be desirable.

Extending the training protocol to a multistep approach could be considered. Our approach to incorporate puzzles in training did indeed have positive effects despite inducing some biases. This may point to some potential benefit in choosing training data appropriately. One may also consider clamping the tree depth to learn ideal short sequences with respect to developong pieces. The feasibility of generating "synthetic" data tailored to expedite learning can also be considered.

Even as we eschewed move history in order to train our network reasonably at the given scale, we do recognise its utility in describing the move state. To obtain a functional engine it may be necessary to incorporate some aspects of board history in the input state. We may be able to strike better balance between expressivity and performance with more experimentation. One may be interested in searching for a better deep representation of the chessboard such as Chess2Vec [4] (which although uses Stockfish for analysis). An unsupervised or semi-supervised Chess Representation Learning formulation should be of interest, such as a Deep Belief Network used for positional embedding in pos2vec of DeepChess[2].

For evaluation purposes, we also suggest the idea of maintaining multiple agents, similar to evolutionary approaches. We note that it may possible for biases towards a particular side may snowball if an Agent only self-plays. We may also only accept an agent if it shows an improvement exceeding a certain margin over 50%.

References

[1] Boost.org. *Boost C++ Libraries*. https://www.boost.org/. Library. Site Accessed on 2023-11-29.

[2] Omid E. David, Nathan S. Netanyahu, and Lior 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. URL: http://dx.doi.org/10.1007/978-3-319-44781-0_11.

[3] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[4] Berk Kapicioglu et al. "Chess2vec: Learning Vector Representations for Chess". In: *arXiv preprint arXiv:2011.01014* (2020).

[5] Donald E. Knuth and Ronald W. Moore. "An analysis of alpha-beta pruning". In: *Artificial Intelligence* 6.4 (1975), pp. 293–326. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(75)90019-3. URL: https://www.sciencedirect.com/science/article/pii/0004370275900193.

[6] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. ISBN: 978-3-540-46056-5.

[7] Matthew Lai. *Giraffe: Using Deep Reinforcement Learning to Play Chess*. 2015.

[8] *Lichess Puzzles Database*. https://database.lichess.org/#puzzles. Last Accessed: 2023-11-29.

[9] Robert Moss. *The 4 steps of Monte Carlo tree search: selection, expansion, simulation, and backpropagation.* Image source: Wikimedia Commons. 2020. URL: https://commons.wikimedia.org/w/index.php?curid=88889583.

[10] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *arXiv preprint arXiv:1712.01815* (2017).

[11] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–503. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[12] *Stockfish*. https://stockfishchess.org/. Comp. software.

[13] Tuur Vanhoutte. *Chess engine with Deep Reinforcement learning*. URL: https://github.com/zjeffer/chess-deep-rl.

[14] Tuur Vanhoutte. *How to Create a Chess Engine with Deep Reinforcement Learning*. Bachelor's Thesis. 20221-22.