Implementation of ChadZero : A self-learning model in chess inspired by AlphaZero

Chadi MOUNTASSIR,

University of Abdelmalek Essaadi, ENSAH, department of data engineering

Abstract:

This report delves into the intricate implementation of ChadZero, shedding light on its unique approach to learning and decision-making in chess throughout self-learning. Furthermore, we explore the limitations and challenges faced during the development process. By combining elements of MCTS and neural networks, Chadzero aims to bridge the gap between domain-specific engines and AlphaZero-like general-purpose models. Understanding these intricacies contributes valuable insights to the ongoing discourse on advancing artificial intelligence in the realm of chess.

Introduction:

Chess, a domain extensively studied in artificial intelligence, has seen the rise of renowned engines like Stockfish and Deep Blue, developed through in-depth analysis of grandmaster plays, representing domain-specific chess engines. In stark contrast, AlphaZero disrupts this paradigm by achieving superhuman levels without prior knowledge of the game beyond its rules.

[1] In 2017 DeepMind published The paper titled "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm" our model stands purely on this paper.

ChaZero, presented in this report, is a basic self-learning chess model inspired by the groundbreaking AlphaZero algorithm. Driven by the constraints of traditional chess engines, our project aspires to create an AI system that autonomously learns and makes strategic decisions in the intricate game of chess. The methodology integrates the Monte Carlo Tree Search (MCTS) algorithm with a neural network architecture designed for efficient learning through self-play and reinforcement learning.

Task definition/modeling:

Our goal is to build an agent for chess games. Our agent interacts with the game environment through a sequence of observations, actions and rewards. We use the chess library to emulate the environment for these games. The game is played in discrete time steps. At every time step, the agent chooses an action, based on the current state or randomly, from a possible set of actions. The emulator then simulates this action and brings the game to a new state. It also returns any reward received during this step. Hence, we can model our problem as follows:

• State s: We consider the state at any time instance to be the board, in most cases the board is represented in fen notation since it represents all our board information in a small size (71 characters) representing the board state at that instance.

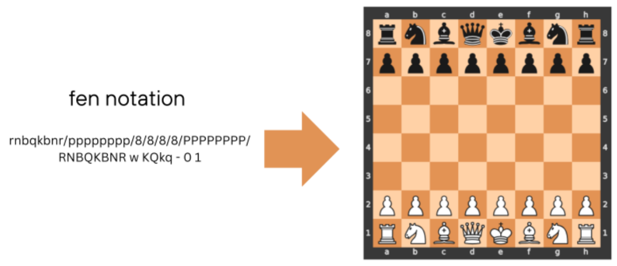

Figure 1: fen notation to board. Fen notation encode the whole board and its full information into a sequence of characters, making the data of board easy to store.

• Action a: The agent has many possible moves the agent can play in the game. of course the set of actions depends on the current state the agent is in.

• Reward r: The reward is returned by the environment after the agent performs an action. We clip the reward so that it lies in the range [-1,1]. We want to make the agent learn to play the game in such a way such that it maximizes the total score.

To find the next state and reward the agent only has to consider the current board state and the action taken. This is known as a Markov Decision Process

(MDP) and we shall find ways to optimize the policy in this process.

concepts:

Reinforcement Learning:

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make decisions by interacting with an environment. The agent takes actions within the environment, receives feedback in the form of rewards or penalties, and then adjusts its strategy which is represented by its policy at every step to maximize the cumulative reward over time. The goal of reinforcement learning is to enable the agent to discover the optimal sequence of actions that lead to the most favorable outcomes.

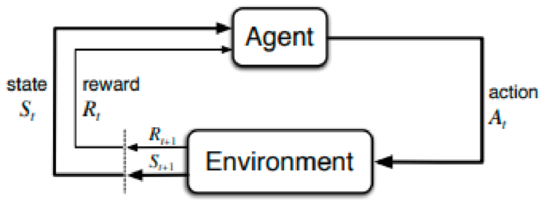

Figure 2: Agent Environment interaction modeled [2]

Convolutional neural networks:

A Convolutional Neural Network (CNN) is a type of deep neural network designed for processing and analyzing visual data, such as images and videos. CNNs have proven highly effective in tasks like image recognition, object detection, and image classification.

Dynamic Programming:

Dynamic programming refers to a class of algorithms that solve or approximate the optimal policy for a Markov Decision Process (MDP). Dynamic programming methods are used to find the optimal policy by iteratively evaluating and improving the value functions associated with different policies.

The basic idea of dynamic programming in RL revolves around the Bellman equation, which expresses the relationship between the value of a state or state-action pair and the values of its successor states or state-action pairs.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\Big] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big], \quad \text{for all } s \in \mathcal{S}, \end{aligned}$$

The bellman equation for the state value

The equation is used to iteratively to enhance the values of policies and values mutually until they converge to the optimal values.

the problem with Dynamic Programming is that we needs full knowledge of the game. in other words we have to thow every possible combination of state/action which is impossible in our case since the total numbre of possible moves in chess surpasses the number of atoms in the universe.

Monte Carlo Tree Search:

Monte Carlo Tree Search (MCTS) is a decision-making algorithm commonly used in artificial intelligence for sequential decision problems, such as game playing and planning. MCTS is particularly popular in games like chess, Go, and other board games. the algorithm is divided into 4 steps as we are going to see later on.

Unlike DP, MCTS does not require a complete knowledge of the game rules or an evaluation function, making it applicable to chess.

description/general idea:

As mentioned before, AlphaZero learns through self-play. Self-play is a technique in which an AI system, typically in the context of games, plays against itself to generate training data. It learns by experiencing a wide range of scenarios and adapting its strategy based on the outcomes of these self-generated games.

In the context of chess, the term "play" indicates that the model must choose a move based on the board state, requiring it to evaluate the board at that state and select the best move. To accomplish this, the model uses a ResNet architecture to process the board and outputs (figure 3) two values: the policy value, which is a vector of probabilities indicating the best move, and a reward value that represents the feedback the model receives after taking an action in a particular state.

Choosing the best move certainly involves predicting the output of a move, highlighting the necessity of a searching algorithm. As mentioned before, we will use MCTS as our searching algorithm.

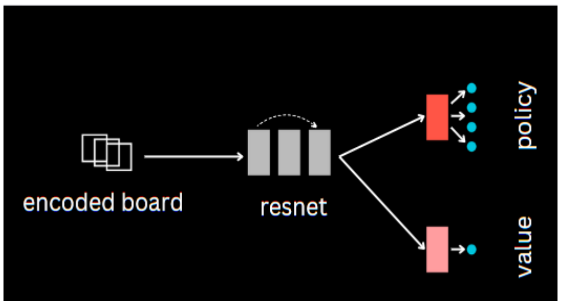

Figure 3 : model architecture, the encoded board is passed through resnet , then passed to two separate neural networks to output the policy and value.

To further enhance the learning process, AlphaZero employs evaluation metrics, after gathering play data, such as Cross Entropy and Mean Squared Error (MSE). Cross Entropy is utilized to measure the dissimilarity between the predicted move probabilities and the actual moves played during self-play. Additionally, MSE is employed to assess the disparity between the predicted reward values and the actual rewards received.

MODEL ARCHITECTURE:

1) CNN:

To construct our AlphaZero model, we need to build three crucial components: the start block, the backbone, and the value and policy head.

The start block comprises a 3x3 convolutional layer followed by batch normalization and then ReLU activation.

The backbone adopts a residual network (ResNet), essentially consisting of pairs of convolutional networks and batch normalization. The key feature of ResNet is the incorporation of "skip connections" or "shortcut connections." These connections enable the model to learn residual functions, representing the difference between the desired output and the current prediction. The presence of skip connections addresses challenges associated with training very deep neural networks.

The value and policy head are composed of a sequence of convolutional layers, batch normalization, ReLU activation, and a fully connected layer. This layer outputs the probability of every square in the case of policy and the value of the reward.

By structuring our model in this way, we aim to create a robust architecture for AlphaZero, leveraging convolutional layers, batch normalization, and skip connections to capture complex patterns and facilitate the training of deep neural networks.

2) searching algorithm (MTCS):

Monte Carlo Tree Search (MCTS) is a decision-making algorithm commonly used in chess-playing programs. In plain terms. The algorithm represents the possible moves and game states as a tree structure, starting with the current position as the root node.

The algorithm is broken down into four important phases:

Selection:

The algorithm begins by selecting the best move to explore. It does this by balancing between known promising moves and exploring new ones [3]. In our case we used the UCB formula:

$$\mathrm{UCB} = \mathrm{q}_{\mathrm{value}} + exploration_{weight} \times$$

$$\frac{(\log(parents_{visits} + 1\,)^{1/2}}{visits + 1}$$

With the value of q_{value} :

$$q_{value} = \frac{\frac{node_{wins}}{node_{visits}} + 1}{2} \quad \text{If} \;\; node_{visits} \neq 0$$

$$else : q_{value} = 0$$

The selection process involves navigating the tree based on a formula that considers both the win rate of a move and the number of times it has been explored.

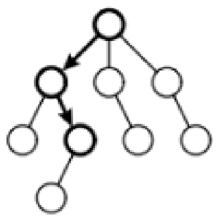

Figure 4: selection phase in MTCS

Expansion:

Once a move is selected, the algorithm considers possible next moves from that position. It expands the tree by adding child nodes representing these new moves.

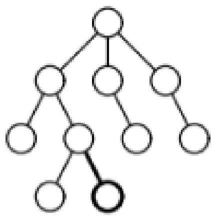

Figure 5: selection phase in MTCS

Simulation (Rollout):

To evaluate a move, the algorithm performs simulated games, or rollouts, from the current position. These simulations are random and continue until the game reaches a terminal state (win, lose, or draw).

Back propagation:

The results of the simulated games are back propagated up the tree. The win rates of the moves are updated based on the outcomes of the simulations.

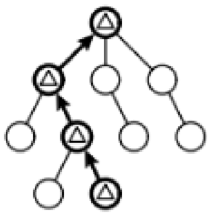

Figure 6: back propagation in MCTS

Selection, expansion, simulation and back propagation are repeated for a predefined number of iterations or until a certain condition is met.

After the iterations, the algorithm selects the move that has been explored and performed well the most.

Modifications:

We merged this algorithm with our model. We removed the simulation step since it's up to our model to evaluate, the output of the model is took in consideration in the UCB formula

$$\mathrm{UCB} = \mathrm{q}_{\mathrm{value}} + exploration_{weight} \times \mathrm{node}_{\mathrm{probability}} \times \frac{(\log(parents_{visits} + 1\,))^{\frac{1}{2}}}{node\_visits + 1}$$

3) The Model:

During the learning phase, ChadZero refines its chess-playing skills through a two-step process: self-play and neural network training.

In the self-play phase, ChadZero engages in simulated chess games against itself using the Monte Carlo Tree Search (MCTS) algorithm. This involves selecting moves based on an evaluation of potential outcomes and progressively building a database of game states, move policies, and game results.

Following self-play, ChadZero transitions to the training phase. The accumulated game history serves as a training dataset. The neural network model is then trained using this dataset, enhancing its ability to make strategic decisions. The training involves optimizing the model parameters by minimizing a combined loss that includes policy cross-entropy loss and value mean squared error loss.

This iterative learning process continues over multiple iterations, with ChadZero refining its chess strategies by continually playing and learning from its own experiences. Periodically, the model's progress is saved in checkpoint files, allowing for potential resumption of learning from previous states and facilitating the long-term improvement of ChadZero's chess-playing capabilities.

Limitations:

ChadZero encounters challenges related to computational complexity. The processes it utilizes, particularly Monte Carlo Tree Search (MCTS) during self-play and neural network training, demand substantial computational resources.

MCTS involves exploring a tree structure and simulating games, making it computationally intensive due to the extensive search space in chess.

Training the neural network requires processing datasets with game states, policies, and values. Complexity depends on dataset size, neural network architecture, and training epochs.

The overall performance of the algorithm is impacted by the combined complexity of MCTS and neural network training, making it resource-intensive.

Scalability may be a concern when applying the algorithm to more complex games or aiming for state-of-the-art performance. Issues arise with increasing search depth or dealing with larger neural network architectures.

To address these challenges, optimization techniques, such as parallelization can be explored to strike a balance between computational demands and algorithm performance.

Results:

Given limited computational power on a local machine, the training results of ChadZero may reach supperhuman levels (figure2) . matter of fact alphazero, in no longuer than 4 hours reached the level of stochfish. however AlphaZero was trained solely via self-play using 5,000 first-generation TPUs to generate the games and 64 second-generation TPUs to train the neural network.

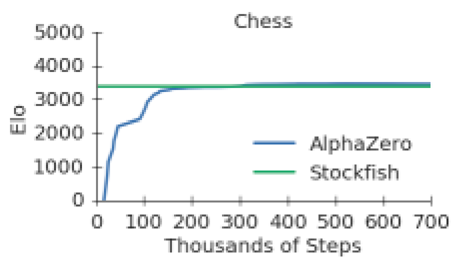

Figure 2 : progression AlphaZero's elo throughout thousands of steps. AlphaZero surpassed the level of stockfish after 300 thousandds of steps)

Conclusion:

The implementation of ChadZero reflects a commendable effort to replicate the success of AlphaZero in chess playing. The model architecture, incorporating a start block, ResNet backbone, and value and policy head, follows the proven design principles of deep neural networks. The utilization of Monte Carlo Tree Search (MCTS) during self-play, along with the integration of the UCB formula for move selection, enhances the decision-making process and strategic exploration.

However, it's crucial to acknowledge the challenges posed by computational complexity, particularly in MCTS and neural network training. Limited computational power may result in longer training times and necessitate careful consideration of resource-efficient optimization techniques.

Acknowledgment:

I would like to thank Dr. Aziz KHAMJANE, my mentor for this project, because of whom i was able to discover these beautiful and usefull techniques.

References:

[1] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. MIT Press, 2018.

[2] DeepMind. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv preprint arXiv:1712.01815. Retrieved from https://arxiv.org/abs/1712.01815.pdf

[3] Dominik Klein. 2022. Neural Networks for Chess : The magic of deep and reinforcement learning revealed