SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.021 Artificial Intelligence

SigmaZero: Transfer Learning for Chess960

Abram Tan	1005057
Benjamin Luo Yehao	1005368
Foo Chuan Shao	1005549
Raymond Harrison	1005329
Shaun Hin Fei	1005446

GitHub Repository: https://github.com/DidItWork/Sigma-Zero
FICS Dataset: https://www.ficsgames.org/download.html

Weights:https://sutdapac-my.sharepoint.com/:f:/g/personal/benjamin_luo_mymail_sutd_edu_sg/EmlUJ6kaxjdKmSseXeqKeXEB_oO8OFYyYRJvr4W63ibw0A?e=fvN6vI

Table of Contents

1. Project Description

Project Objective:
The objective of this project is to develop SigmaZero, an advanced AI model based on the AlphaZero architecture, adapted specifically to master Chess960, also known as Fischer Random Chess. The aim is to evaluate the effectiveness of transfer learning by comparing the time and resources required to fine-tune an existing AlphaZero model against the demands of training a model from scratch for Chess960. We seek to determine how transfer learning affects the accuracy and efficiency of the model, ultimately gauging its utility in significantly reducing training time while maintaining, or possibly enhancing, performance levels.

Problem Description:
Chess960, characterized by its 960 possible starting positions, disrupts conventional chess strategies, demanding a highly adaptable AI capable of strategic innovation. SigmaZero is tasked to understand and thrive in this complex environment, evolving from a foundational stage of zero inherited knowledge to a proficient level capable of competing with skilled human players. This task of developing strategic depth in a variable setup poses a significant intellectual challenge.

Current Challenge:
The principal challenge is training SigmaZero from scratch to comprehend and excel in the dynamic realms of Chess960. SigmaZero must commence learning through self-play, devoid of any predefined strategies or game insights typical of traditional chess models. This model's training from ground zero requires extensive computational effort and time, as it needs to progress through countless gameplay iterations, evolving from novice to expert.

Proposed Approach:
The project proposes a transfer learning strategy to accelerate the training process of SigmaZero. By fine-tuning a pre-trained AlphaZero model, originally designed for traditional chess, SigmaZero can swiftly assimilate and modify complex strategic paradigms to fit the unique starting configurations of Chess960. This approach not only anticipates a substantial reduction in developmental time and resources but also serves as a test bed for assessing the impact of transfer learning on the performance of AI models in highly variable environments. The success of SigmaZero, employing this methodology, will be rigorously evaluated through systematic testing against both sophisticated AI adversaries and experienced human players, ensuring its viability and competitiveness.

2. Dataset Description

2.1. Data Sources & Generation

In the development of SigmaZero, two distinctive sets of data are utilized, each serving a specific training objective to enhance the model's performance in both standard and Chess960 configurations.

Vanilla Chess (Supervised)
The first dataset comprises historical game records from the Free Internet Chess Server (FICS), which provides a variety of standard chess games. This dataset is composed of a multitude of game records from traditional chess, offering a rich variety of gameplay styles, strategies, and outcomes. This foundational dataset allows SigmaZero to gain a comprehensive understanding of conventional chess, which is crucial before it adapts these strategies to the more complex and varied scenarios encountered in Chess960. [1]

Chess960 (Self-generated)
The second dataset is generated internally through self-play specifically for Chess960. SigmaZero autonomously plays games against itself, starting from each of the 960 possible initial configurations. This process generates a diverse set of game scenarios, capturing unique positions, moves, and outcomes necessary for adapting to the complexities of Chess960.

2.2. Data Composition

The input and generated data for this project will adhere to the format utilized by AlphaZero, as detailed in [2]. According to this source, the data is structured in tensor format, which effectively captures the complex array of game states and potential moves.

Each tensor encapsulates critical information about the game's current state, historical moves, and potential future consequences. This structured representation allows SigmaZero to analyze and predict outcomes with a depth of strategic insight akin to that developed by AlphaZero for standard chess configurations.

- **P1 Piece & P2 Piece (12 planes each)**
  Each type of piece for players 1 and 2 (P1 and P2) would still require 6 planes each to represent their positions on the board. This would be done in the order of Pawn, Knight, Bishop, Rook, Queen, King. The order of P1 and P2 would be swapped depending on the current player. It will always be the Player then the opponent.
- **Repetitions (2 planes each)**
  Two planes are used to capture states that have occurred once and twice before, which is a standard rule in chess to claim draws due to threefold repetition.
- **Colour (1 plane)**
  A single plane is used to indicate whose turn it is to move, which is a binary feature with 0 for white and 1 for black.

- **Total Move Count (1 plane)**
  This plane records the number of half-moves (each player's move counting as one half-move) made in the game so far, to determine if the fifty-move rule is applicable.
- **Castling Rights (4 planes)**
  Chess960 has unique castling rules, but they can be encoded within the same two planes as standard chess because there are still only four types of castling rights (kingside and queenside for both players), irrespective of the starting positions.
- **No-Progress Count (1 plane)**
  Similar to the total move count, this plane keeps track of the number of moves since the last pawn advance or capture, which is crucial for enforcing the fifty-move rule.

The tensor will be a total of 119 channels. It captures the positions of Player 1 and Player 2 pieces, as well as any board state repetitions, across 8 sequential half-moves, totaling 112 channels. This records the recent tactical developments within the game. The remaining 7 channels represent static features like turn color, move count, castling rights, and no-progress count, crucial for evaluating the game state. If fewer than 8 half-moves have occurred, the initial channels are set to zero, ensuring the tensor accurately reflects the current stage of the game.

3. Model Description

A major component of the AlphaZero algorithm is the policy network, which takes in the board state and outputs a value associated with how favourable the board state is and the probabilities of the actions that should be taken from the board state to lead to a positive outcome. As mentioned before, the utilisation of a neural network to evaluate board states removes the need for the simulation step in MCTS.

3.1. Model Architecture

The evaluation of the board state and generation of a policy for possible actions is derived from ResNet [3]. The network takes in the board state as a 3D tensor and passes it through a main backbone consisting of ResNet blocks followed by a policy head which outputs the policy vector and a value head which outputs a scalar value of the board state. Its architecture is shown in Fig. 1 below.

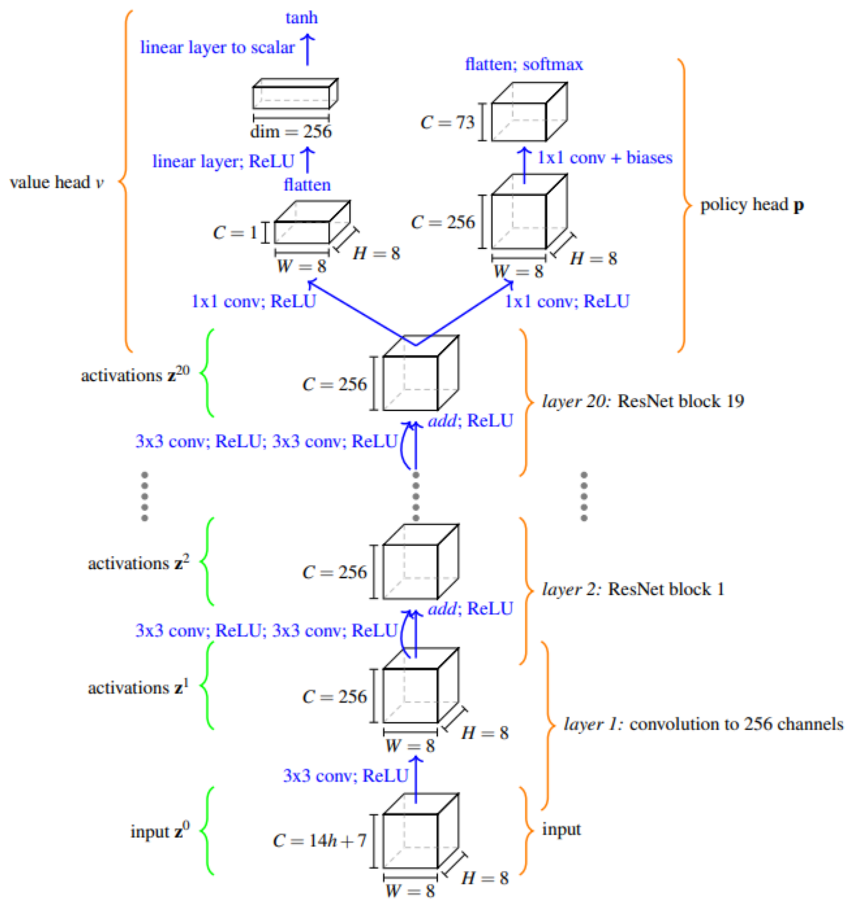


Figure 1. The AlphaZero network. Each 3×3 convolution indicates the application of 256 filters of kernel size 3×3 with stride 1. A ResNet block contains two rectified batch-normalized convolutional layers with a skip connection. In the input $\mathbf{z}^0$, a history length of $h = 8$ plies is used, encoding the current board position and those of the seven preceding plies. The input is a $8 \times 8 \times 119$-dimensional tensor.

Fig. 1. Convolutional architecture of AlphaZero network

The model takes advantage of convolutions to encode the board state as the properties of the former such as translational invariance and locality preservation are useful priors in evaluating the positional states in board games like chess.

Having separate heads based on the same main branch instead of two separate networks for evaluating the board state and producing an action policy respectively also allow the network to learn correlations between the value of the board state and the action policy which is a good prior to have.

3.2. Monte-Carlo Tree Search (MCTS)

MCTS is a popular algorithm in the space of complex decision making problems as it performs particularly well even with enormous search spaces, the lack of domain knowledge and shows asymptotic optimality, among other reasons [4]. Hence, it is no mystery why MCTS was used as part of the AlphaZero architecture. The steps to MCTS are shown below.

1. Selection

Starting at the root node (the current state of the game), the algorithm selects child nodes down to a leaf node (a node that has not yet been fully expanded). The selection is guided by the child node with the highest upper confidence bound (UCB), a policy that balances exploration of less visited nodes and exploitation of nodes known to be advantageous. The selection algorithm can be described as:

$$\text{UCB} = \frac{W_i}{N_i} + p_i c \frac{\sqrt{N_i}}{1 + n_i}$$

Where c is a constant,
p_i is the policy of the child node,
n_i is its simulation count

2. Expansion

Once a leaf node is selected, the MCTS algorithm is now in the expansion phase. The MCTS algorithm obtains a policy from the neural network, and for each action probability pair in the policy, the children of the leaf node is/are stored as one of the leaf node's attributes and the probability as the child's prior.

3. Simulation

In typical MCTS, a simulation of the game is played randomly starting from the selected node until a terminal state is reached. However, this step is skipped in our MCTS with the neural network since the policy neural network already determines the value for the selected node.

4. Backpropagation

After a node has been evaluated, the results are propagated back up the tree. The win probabilities (values) from the neural network are used to update the nodes through which the selection passed. This update adjusts the sum of values and the visit count stored at each node, affecting their selection in future iterations of the search.

These steps are repeated many times and as the tree grows in size, the value sum at each node generally becomes better estimates of the true value of each node as more and more simulations are run (with a higher value meaning a more favorable state). This process is performed for 800 iterations.

4. Training Methodology

4.1. Vanilla Chess Training

Initially, games were generated through self-play where the board state, outcome, and actions taken are stored as training data. During training, the policy network is fed the board states from the training data and losses are generated by comparing the produced policy and values to those collected during training. The losses are then backpropagated through the model for it to learn to make the right moves and predict the right outcomes based on the board.

This cycle of self-play and training repeats, with the updated network being used to produce games during self-play. However, generating each game and training accordingly took too long, about 5 minutes per game. To yield a significant amount of games for the model to learn efficiently, we decided to train the policy network on examples of people playing chess instead.

Games played by 2000+ ELO[1] players were retrieved from the FICS database [1] and formatted into suitable training data. A total of 60 epochs were trained on 15000 games with a batch size of 2048 with each epoch lasting 400 steps. The initial learning rate is set to 0.005 with a step learning rate decay of 0.95 per 500 steps.

4.2. Transfer Learning to Chess960

After supervised learning on vanilla chess, we now have a model that can play chess decently which can be used to generate games through self-play again. We decided to go through with the process of self-play and training for Chess960 - a similar variant of chess in which the pieces in the back rank are shuffled randomly - to see if the model can achieve improved performance compared to without additional self-supervised training.

Self-play games are generated with a minimal 10 searches due to time constraints. 210 games are played and then finetuned for 5 epochs for 1 cycle. We ended the finetuning after 20 cycles of self-play and finetuning which lasted 10 hours. Training losses can be seen to decrease in Fig. 2.

[1] The ELO rating system calculates the relative skill levels of players in zero-sum games. As an example, a player whose rating is 100 points greater than their opponent's is expected to score 64% and if the difference is 200 points, then the expected score for the stronger player is 76% and so on. Hence ELO is an estimate of a player's relative skill level, and every increase in ELO is an increase in the expected wins of a player.

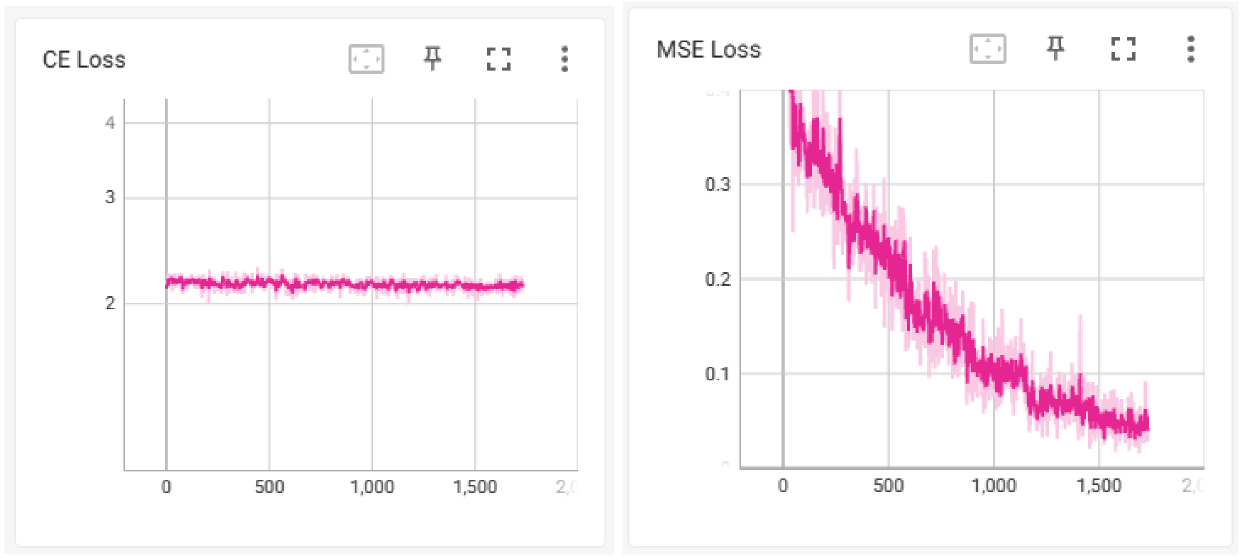


Fig. 2. Training Losses per cycle of self-play and training.

4.3. Loss Function

The loss function for the model is

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

where z represents the value of the node, π signifies the action chosen, p and v denote the policy and the value yielded from the model's output, respectively, and c stands for a constant term.

This loss function serves a dual purpose. Firstly, it endeavors to minimize the disparity between the model's policy p and the policy deduced from Monte Carlo Tree Search (MCTS). Secondly, it aims to diminish the distinction between the model's output value v and the value z obtained through MCTS exploration.

By minimizing this loss, the neural network endeavors to refine its policy and value predictions, aligning them more closely with the outcomes derived from MCTS exploration. This convergence fosters improved decision-making and strategic acumen within the model, ultimately enhancing its gameplay performance.

4.4. Optimizer

The choice of optimizer significantly influences the efficiency and efficacy of the model's training process. In this implementation, the Adam optimizer from the `torch.optim` module is employed. The learning rate is set to 0.005, a value carefully selected to balance the trade-off between rapid convergence and stability during training.

Adam optimizer stands out for its adaptive learning rate methodology, which dynamically adjusts learning rates for each parameter based on the magnitudes of recent gradients and exponentially decaying average of past gradients. This adaptive nature enables Adam to navigate complex loss landscapes more effectively, facilitating faster convergence and mitigating issues such as vanishing or exploding gradients.

By leveraging the Adam optimizer, the training process benefits from improved convergence speed and stability, ultimately contributing to the model's ability to learn and generalize effectively from the training data.

4.5. Hyperparameters

A list of training parameters for the supervised vanilla chess training as well as the self-supervised chess960 training are summarised in the tables below.

Parameter	Vanilla Chess	Chess960
Batch Size	2048	128
Learning Rate	0.005	0.0001
Weight Decay (Adam)	0.0001	0.0001
Epochs	60	20
Steps per epoch	400	300
Learning Rate Decay	0.95 per 500 steps	0.95 per 500 steps

5. Model Evaluation

The trained_weights for this project can be found here.

5.1. Vanilla Chess Testing

Testing of the model followed the iterative play against increasingly better versions of Stockfish, a well-known chess engine, which provided a gauge for the playing strength of the model. The Stockfish chess engine was chosen as it is widely recognised and used on online versions of chess as an analysis and evaluation tool due to beating out other chess engines in the annual Top Engine Chess Championships. By selectively limiting Stockfish's skill level, time to think, and depth to search for a move through configurations in the chess engine, we were able to handicap the strong chess engine to an approximate playing strength of a beginner to intermediate level at around 1000 to 1500+ ELO.

The ranges for the skill level are 0 to 20, time to think at 10 milliseconds to 1 second, and depth of 1 to 10 ply. Through standardizing the levels of play to specific benchmarks derived from various online sources, we approximated the levels of Stockfish to the following:

Level of Playing Strength	Skill Level	Time Limit	Search Depth	Estimated ELO
0	0	1	5	1376
1	1	1	5	1462
2	2	1	5	1547
3	3	1	5	1596
4	4	1	5	1718
5	5	1	5	1804
6	6	1	5	2012
7	7	1	5	1993
8	8	1	6	2127
9	9	2	7	2270
10	20	10	50	3100

Table 1. Skill levels of Stockfish

The choice for having 10 levels is inspired by the 8 levels of difficulties found on lichess. However, unlike the lichess distribution of Stockfish, we reduced the delta between successive levels even further after a few initial tests showed that the playing strength of the model was around 1600+ ELO for more fine-grained steps at lower levels. Also of note is the final level which aims to maximize the default configurations of Stockfish and act as a hard stop to the testing phase since the ELO of the engine at these settings are well above 3000+ ELO. By iteratively increasing the limits to Stockfish, and playing a best of 5 format per level where the model must win at least 2.5 or more points on Stockfish (+1 win, +0.5 draw, 0 lose), the final playing strength of the model was estimated.

Running the model against stockfish using an evaluation script, the following results are a brief summary of data obtained for both vanilla chess and Chess960:

Model	Game Mode	Stockfish Level of Playing Strength	Estimated ELO	Model Win	Model Loss	Model Draw	Games[2]	Points
supervised_model_15k_40.pt[3]	Vanilla	3	1596	2	1	2	DWLDW	3.0/5.0
		4	1718	1	2	2	LDDDL	1.5/5.0
supervised_model_15k_45.pt	Vanilla	3	1596	3	2	0	WLWLW	3.0/5.0
		4	1718	1	2	2	DWDLL	2.0/5.0
supervised_model_15k_40.pt[4]	Chess960	3	1596	1	4	0	WLLLL	1.0/5.0
supervised_model_15k_45.pt	Chess960	3	1718	2	3	0	WLLLW	2.0/5.0

Table 2. Results against Stockfish levels for supervised vanilla chess agent

The supervised model at 40 epochs won fewer than the necessary 2.5 points against Stockfish at level 4, and performed slightly worse than the model trained to 45 epochs, hence we will focus on the result of the model at 45 epochs.

Since the supervised model at 45 epochs won 2 points against stockfish skill level 4 before failing to proceed, the model ELO is extrapolated to be between 1600 and 2000 which is the estimated ELO of the commonly configured skill level 3 and 6, specifically around 1700+ ELO. This fact is corroborated by the model winning against stockfish in the best of 5 at level 3, and

[2] W represents a win, D represents a draw, and L represents a loss, in order of the games the model played against Stockfish

[3] It should be noted that the supervised model at 40 epochs ran with the value for constant C as 1 instead of 2 which seemed to improve performance as well.

[4] It should be noted that the supervised model at 40 epochs ran with the value for constant C as 1 instead of 2 which seemed to improve performance as well.

holding a commendable 2 draws at level 4. Hence, through the reinforcement learning and fine tuning, the model was able to achieve a relatively high playing strength that would take an average human player 2 to 3 years of study to achieve.

5.2. Chess960 Testing

A similar testing procedure is repeated for Chess960.

Model	Game Mode	Stockfish Level of Playing Strength	Estimated ELO	Model Win	Model Loss	Model Draw	Games[5]	Points
RL_960_0.pt	Chess960	0	1376	3	1	0	WWLW	3.0/5.0
		1	1462	0	3	1	DLLL	0.5/5.0
RL_960_5.pt	Chess960	0	1376	0	3	2	LDLDL	1.0/5.0
RL_960_15.pt	Chess960	0	1376	1	0	0	WLLL	1.0/5.0
RL_960_20.pt	Chess960	0	1376	0	3	0	LLL	0.0/5.0
supervised_model_15k_45.pt	Chess960	0	1376	3	0	0	WWW	3.0/5.0
		1	1462	3	1	0	WWLW	3.0/5.0
		2	1596	1	1	3	LWDDD	2.5/5.0
		3	1718	2	3	0	WLLLW	2.0/5.0

Table 3. Results against Stockfish levels for supervised and unsupervised Chess 960 agents

5.3. Results Discussion

The model was tested independently by our team by playing vanilla chess against it after training, and was observed to be able to repeatedly produce several well-known chess openings in its repertoire, namely the Ruy Lopez, in particular the Closed and Exchange Variations of the Morphy's Defense, Indian Defence, particularly the Nimzo-Indian and transpositions into the Queen's Gambit Declined, and the Sicilian Defence, including the Steinitz variation. These openings suggest that the model is able to make sound decisions in the opening of the game and that the supervised learning for vanilla chess is working as expected.

[5] W represents a win, D represents a draw, and L represents a loss, in order of the games the model played against Stockfish

The model’s capability to perform at a competitive level in Chess960 demonstrates its proficiency in adapting to novel game setups, indicating an approximate ELO rating of 1600+ from the results against Stockfish in the Chess960 format. This underscores the effectiveness of supervised learning and transfer learning in honing the model's strategic decision-making, allowing it to discern optimal moves and principles based on different board configurations in the initial setup.

The performance of the Reinforcement Learning does not improve the performance of the chess agent in Chess960, potentially due to the low number of games generated during self-play and seen during each training epoch, the low number of searches used during self-play to save time may also have resulted in sub-optimal moves and training data. As such, by learning from sub-optimal data, the performance of the model gets more random and unpredictable despite the advantages of getting examples of the new Chess960 game mode.

However, given the ability of training losses to decrease during Self-play Reinforcement Learning, we believe that with a higher number of self-play games generated with sufficient searches using MCTS, we can achieve performance improvements.

Our hypothesis regarding the efficiency of transfer learning in accelerating training compared to starting from scratch on Chess960 is corroborated by the reduced total training time. Training the model from the outset on Chess960 would have necessitated a similar number of games as the supervised vanilla chess model of 15000 with each game taking about 5 minutes to generate, or approximately 137 years to generate on the high-end consumer hardware that we are using, an RTX 4080 GPU. Hence, the training time through supervised training which took roughly 10 hours for 20 epochs of self-play of Chess960 cuts the amount of training time for Chess960 significantly, affirming the advantage of leveraging pre-existing knowledge in related domains.

5.4. Comparison with State-of-the-Art Models

When comparing our model to state-of-the-art (SOTA) models like AlphaZero and Leela Chess Zero, which rely solely on neural networks for playing chess, several notable distinctions emerge regarding training duration and achieved ELO ratings.

While precise training durations for AlphaZero and Leela Chess Zero in the context of Chess960 aren't publicly available, we can use their training durations for standard chess as a reference point. AlphaZero, for instance, completed its training on 44 million games in just 9 hours, leveraging 5000 specialized tensor processing units tailored for generating game data efficiently. On the other hand, Leela Chess Zero adopts open-source distributed computing, engaging in self-play at a staggering rate of approximately 1 million games per day. As of April 2024, it has amassed an impressive tally of over 2.5 billion self-play games.

In contrast, our model underwent training on a single RTX 4080 GPU, culminating in a total training time of 10 hours. Given the similarities in task between vanilla chess and Chess960, it's reasonable to infer that if AlphaZero or Leela Chess Zero were to undergo transfer learning for Chess960, the process would likely be exponentially faster than what we achieved with our resources. Comparatively, the swift training duration of our model to reach a competitive ELO rating in the Chess960 transfer learning task underscores the substantial advantage of employing transfer learning within this project.

By leveraging pre-existing knowledge and skills acquired from vanilla chess, our model effectively circumvents the need for extensive self-play generation, highlighting the efficiency and efficacy of transfer learning in adapting to novel chess variants such as Chess960. This not only underscores the adaptability and versatility of neural networks but also showcases the potential for rapid skill acquisition and performance enhancement through transfer learning paradigms.

6. Setting up Graphical User Interface (GUI)

This section of the project report will guide you through setting up the software environment necessary to run the graphical user interface (GUI).

6.1. Pre-Requisites

We will be using Miniconda as the environment manager, but you can adapt the steps for any similar tool you might prefer.

Ensure Miniconda is installed on your system. If not installed, you can download it from Miniconda's official website. This project is developed using Python 3.11, so it is advisable to use a compatible version of Miniconda.

After installing miniconda, clone our project repository by running:

```
git clone https://github.com/DidItWork/Sigma-Zero
```

6.2. Environment Setup and Running Application

1. **Create the Environment:** Navigate to the root directory of the project file where you can find 'environment.yml'. This file lists all the necessary packages and their specific versions required to run the application.

   Create the Conda environment using the following command:

   ```
   conda env create -f environment.yml
   ```

2. **Activate the Environment:** After creating the environment, activate it using:

   ```
   conda activate sigmazero
   ```

3. **Run the Apllication:** Start the application using Streamlit by running:

   ```
   streamlit run Home.py
   ```

6.3. Troubleshooting

Dependency Errors: If you encounter errors related to missing packages or version conflicts, ensure that the environment.yml file includes all necessary dependencies with correct versions. Ensure that you are using the right version of streamlit:

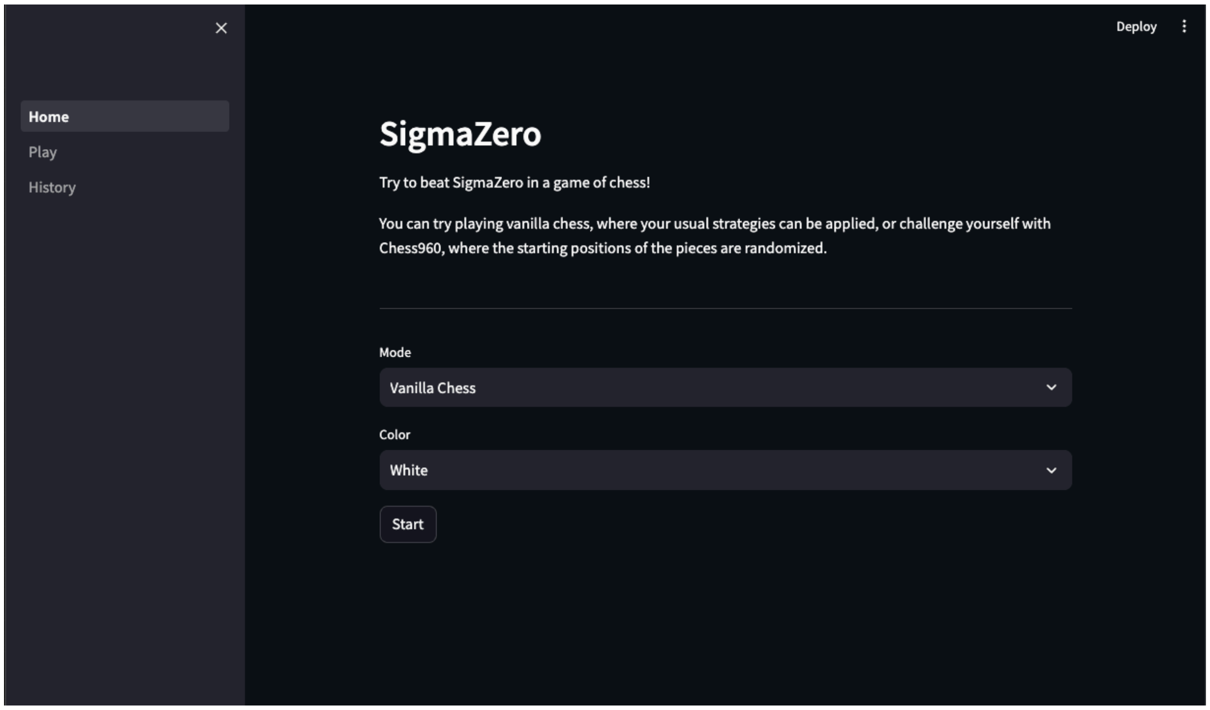
```
pip install streamlit==1.33.0
```

Environment Activation: Make sure you activate the correct Conda environment before attempting to run the application. If the environment name is incorrect, check the name specified in the environment.yml file.

7. GUI Demonstration

The SigmaZero Chess GUI is intuitively designed to facilitate users in engaging with an AI to play chess. Upon launch, the homepage offers the option to select from two game types: "Vanilla Chess," which adheres to traditional chess rules, and "Chess960," which presents a variant with randomized starting positions. Users can also choose their preferred color—either white or black—before beginning the game.

7.1. Home Page



SigmaZero

Try to beat SigmaZero in a game of chess!

You can try playing vanilla chess, where your usual strategies can be applied, or challenge yourself with Chess960, where the starting positions of the pieces are randomized.

Mode: Vanilla Chess

Color: White

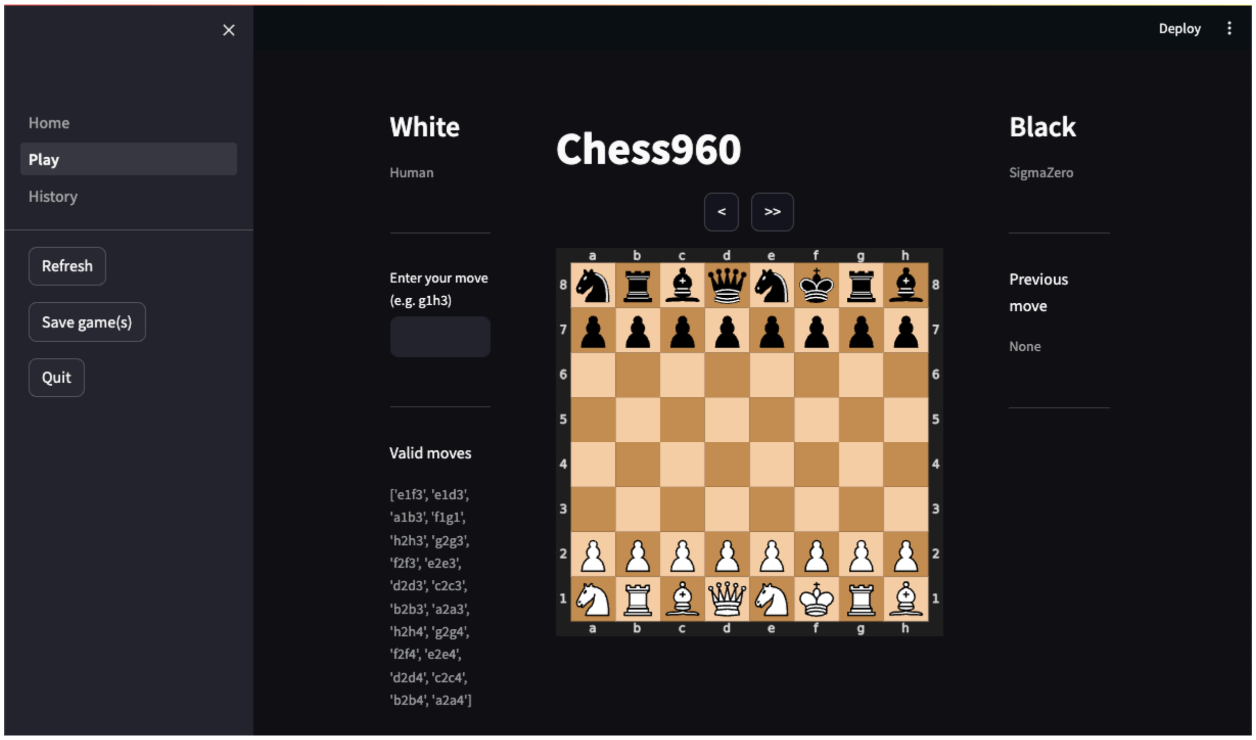
Fig. 3. Home Page

Layout and Design
The Home Page of SigmaZero Chess provides a simple and user-friendly entry point to the game. It is the starting point where the user makes their initial choices. The dropdown menus are designed to be straightforward, ensuring that players of all levels can quickly understand and make their selections without any confusion.

Functionality and Options
At the center of the homepage, users are presented with a dropdown menu labeled "Mode" to select the type of chess game they want to engage in. The options available are "Vanilla Chess" for the classic game experience and "Chess960" for a variation with randomized starting positions.

Below the game type dropdown, there is another dropdown menu for "Color," allowing the player to choose whether to play as white or black. This choice determines the player's starting pieces and position on the board.
After the selections are made, the 'Start' button initiates the game with the chosen settings. Clicking this button transitions the user to the Play page, where the chessboard and game mechanics are displayed.

7.2. Play Page



```
['e1f3', 'e1d3',
'a1b3', 'f1g1',
'h2h3', 'g2g3',
'f2f3', 'e2e3',
'd2d3', 'c2c3',
'b2b3', 'a2a3',
'h2h4', 'g2g4',
'f2f4', 'e2e4',
'd2d4', 'c2c4',
'b2b4', 'a2a4']
```

Fig. 4. Play Page

Layout and Design
The Play page is where the action unfolds. It features a centralized chessboard for visual feedback that updates the game state after each move, flanked by functional panels on either side for game control and information.

Functionality and Options
To the left of the chessboard, there's a dedicated input box where players can type their moves in algebraic notation (e.g., e2e3, b1c3, etc). Below the move input, a dynamically updated list shows all the legal moves available to the player. This feature aids in decision-making and prevents illegal moves.

User Interaction
Players interact with the Play page primarily through the move input box. If an invalid move is typed in, a warning pop-up will appear saying that the move is invalid.

To the right of the chessboard is the portion of the screen reserved for SigmaZero. It shows the previous move taken by the AI.

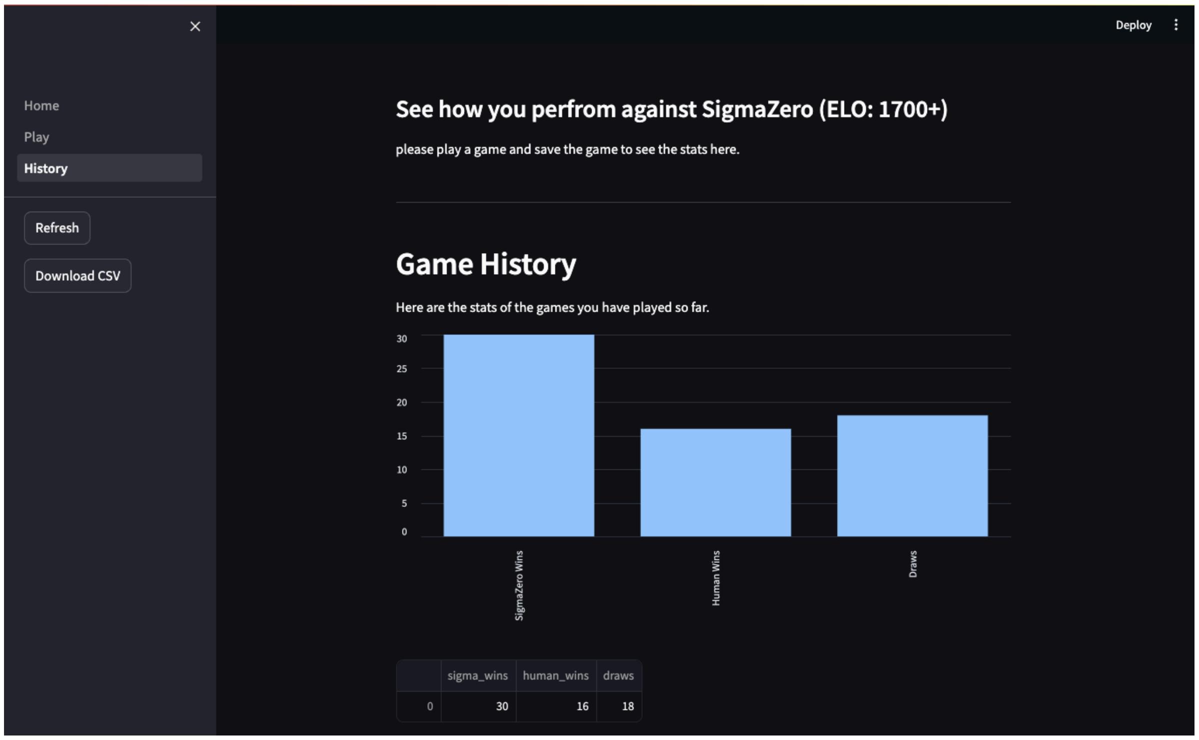
On top of the chessboard, you will see these buttons:

1. **<:** Allows players to view their past moves history one step at a time
2. **>>:** Allows players to fast-forward to the move they made last.

At the sidebar, you will see these buttons:

1. **Refresh:** Allows players to reload the game state, useful for updating the display if there are any discrepancies.
2. **Save game(s):** saves the current game progress into the database, a crucial feature for tracking performance over time. This data can then be visualized in the 'History' page later on.
3. **Quit:** Exits the current game session and returns the user to the Homepage.

7.3. History Page



See how you perfrom against SigmaZero (ELO: 1700+)

please play a game and save the game to see the stats here.

Game History

Here are the stats of the games you have played so far.

	sigma_wins	human_wins	draws
0	30	16	18

Fig. 5. History Page

Layout and Design
The History page serves as a statistical dashboard for the player, featuring a simple and effective layout. A graph at the center displays the user's performance data, with a text reminder above it to encourage game saving for accurate statistics.

Functionality and Options
The bar chart is the centerpiece of this page, illustrating the number of games won by SigmaZero (the AI), the human player, and the number of draws. This allows us to track how SigmaZero is performing against the human players who have tried our AI.

At the sidebar, you will see these buttons:

1. **Download CSV:** Allows users to export their game data, allowing for offline analysis and record-keeping.

2. **Refresh:** Updates the display to reflect the most current game history and statistics.

Visualization of Results
The bar chart offers an at-a-glance understanding of user performance, making data consumption quick and easy.

8. Conclusion

This study embarked on the ambitious project of developing SigmaZero, a highly advanced AI designed to master Chess960 by leveraging the existing architecture and learned behaviors of AlphaZero. The primary goal was to explore the effectiveness of transfer learning in significantly reducing the time required to train an AI model on a complex variant of chess, compared to training from scratch.

Our approach involved a two-pronged data acquisition strategy: initially harnessing a robust dataset from the Free Internet Chess Server (FICS) to introduce SigmaZero to traditional chess strategies through supervised learning, followed by employing a self-play methodology tailored to Chess960. This combination allowed SigmaZero to build on foundational chess knowledge while adapting to the unpredictable nature of Chess960's numerous starting positions.

The results of our experiment were highly encouraging. We developed a fully-integrated pipeline for supervised model training and reinforcement learning through self-play. With better compute and more examples, the performance of knowledge transfer from vanilla chess to Chess960 through self-play can be verified.

In conclusion, this project underscores the potential of transfer learning as a transformative approach in AI development, especially in applications requiring a rapid adaptation to new conditions without the need for extensive retraining. The success of SigmaZero suggests that similar methodologies could be applied effectively across other domains where learning efficiency and time constraints are critical factors. This study not only advances our understanding of AI learning capabilities but also sets a precedent for future research in AI training efficiency and adaptability.

The trained_weights for this project can be found here and the code for the project can be found here.

9. References

[1] Ludens@freechess.org, “FICS games database - download,” Search, https://www.ficsgames.org/download.html (accessed Apr. 15, 2024).

[2] D. Silver et al., “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” arXiv.org, https://arxiv.org/abs/1712.01815 (accessed Apr. 15, 2024).

[3] T. McGrath et al., “Acquisition of chess knowledge in AlphaZero,” Proceedings of the National Academy of Sciences, vol. 119, no. 47, Nov. 2022, doi: 10.1073/pnas.2206625119.

[4] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, “Monte Carlo Tree Search: a review of recent modifications and applications,” Artificial Intelligence Review, vol. 56, no. 3. Springer Science and Business Media LLC, pp. 2497–2562, Jul. 19, 2022. doi: 10.1007/s10462-022-10228-y.