Overcoming Perceptual Deficits in Chess via Deep Q-learning Algorithm with Monte Carlo Tree Search

John Rho

Neuro 140 Spring 2023

Hypothesis/Question

Questions:

How do perceptual deficits impact the performance/robustness of a reinforcement learning model in strategy games, like chess?

How do models perform when noise is introduced to the board state (changing transition probabilities)?

Significance: This process simulates players making blunders from recognizing game state incorrectly. We can observe how models recover, possibly like humans.

Hypothesis:

The model will be able to generalize effectively given randomized opponent board states, given there is little to no correlation between the noise and the optimal action. There will likely be an initial dropoff, but with continual learning, the model should be able to update and improve performance.

Brief Literature Review

Research into chess algorithms has coincided with the rise of computers since the mid-20th century. Since the late 1990s when IBM's Deep Blue beat chess champion Gary Kasparov, chess algorithms have largely outperformed the world's best players.

However, there is a lack of literature tackling how chess algorithms perform given uncertainties in perception. Problems of perception are common in other reinforcement learning applications – like self-driving cars. However, there is not much literature on how models adapt to uncertainty in strategy games (e.g., introducing noise to the positions of the opponent's pieces, obscuring their exact location). Hence, the objective of this paper is to explore the models' performance on these tasks.

Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm (Silver et al., 2018)

Teaching algorithms to play chess has been a decades-long endeavor. This breakthrough research paper presents the development and evaluation of AlphaZero, a reinforcement learning algorithm that masters two-player, zero-sum games like chess and shogi through self-play. AlphaZero extends the success of its predecessor, AlphaGo, designed for the game of Go, by using a deep neural network combined with Monte Carlo Tree Search (MCTS) to guide gameplay.

AlphaZero learns exclusively from self-play without prior knowledge of game rules or human-generated data. The algorithm employs (1) a deep neural network for estimating action values and position values, and (2) an MCTS that utilizes these estimates to navigate the search for promising moves. AlphaZero demonstrated superior performance compared to its opponents, often with fewer positions evaluated per second. Additionally, AlphaZero displayed creative and human-like strategies, which differentiated its gameplay from traditional engines.

Building on top of the findings from the research above, my paper will incorporate a similar approach by training a Convolutional Q-network while using a MCTS to guide gameplay. However, due to limitations in compute, rather than learning action values, the model will prioritize learning state values.

Methods

Training took approximately 1-2 hours to complete, and running ~100,000 iterations of testing (validation) took approximately 1-2 hours as well.

Overview and Evaluation Metric:

The reward function was defined as the game outcome. Rewards are calculated from the perspective of the white pieces ("the player") against the black pieces ("the opponent"). +1 indicates white checkmated black. -1 indicates black checkmated white. 0 meant there was a draw (stalemate). We trained over 1,000,000 iterations to teach the model how to play chess.

The state value function was defined as the material balance, which is the net sum of the value of the pieces in the game. White's pieces are positive and black's pieces are negative, with the pawn being the weighted least-valuable piece and the queen being the weighted most-valuable piece. A positive balance is viewed as favorable.

The model was evaluated at each iteration based on a Temporal Difference metric. We can define this equation iteratively as

$$TD\ Error\ =\ reward\ +\ (discount\ factor)\ *\ (next\ state\ value\ estimate)\ -\ (current\ state\ value\ estimate)$$

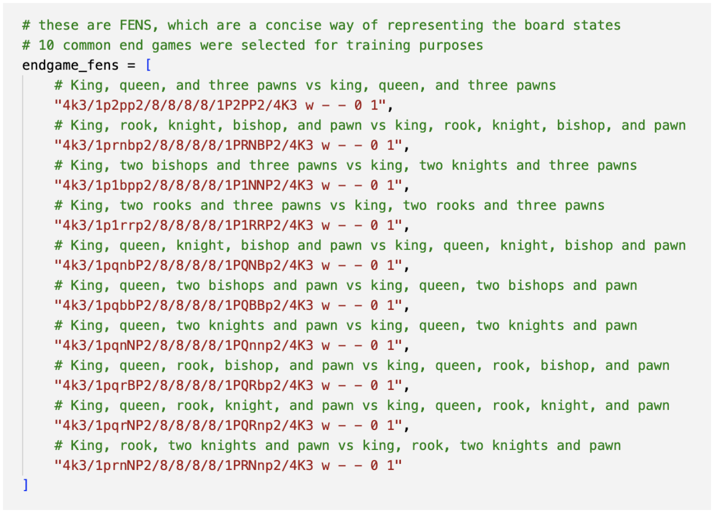
Then,

$$(updated\ value\ estimate)\ =\ (current\ state\ value\ estimate)\ +\ (learning\ rate)\ *\ (TD\ Error)$$

We use the updated value, which is tied to the board state, as a reference for future decision-making. The Q-learning model keeps a record of these state-value pairs. Without diving into the technicalities of MCTS, the model is especially useful given its ability to parse through large state and value spaces while also balancing both exploration and exploitation to find a high-performing decision (Nair, 2017).

Procedure:

1. Train on chess environment with 10 different end games, download weights
    - Weights were calculated from my own training, could vary based on local optima
    - Model used MCTS with modified Q-learning algorithm (learning state-values rather than action values)
        - 100,000 to 1,000,000 iterations, 10K was not enough
    - Opponent was a Greedy Agent that played by picking the highest-value move each turn
2. Test model on a modified chess environment that adds noise in perception (0.20 likelihood), obscuring the exact location of the opponents' pieces
    a. Load weights from training a model on the "regular" noise-free environment
    b. Use the same model architecture
        i. Tested one game at a time, maximum of 100 half-moves per game
    c. Incorporated continual learning so that the model would keep updating based on learning to overcome the noise.



```
# these are FENS, which are a concise way of representing the board states
# 10 common end games were selected for training purposes
endgame_fens = [
    # King, queen, and three pawns vs king, queen, and three pawns
    "4k3/1p2pp2/8/8/8/8/1P2PP2/4K3 w - - 0 1",
    # King, rook, knight, bishop, and pawn vs king, rook, knight, bishop, and pawn
    "4k3/1prnbp2/8/8/8/8/1PRNBP2/4K3 w - - 0 1",
    # King, two bishops and three pawns vs king, two knights and three pawns
    "4k3/1p1bpp2/8/8/8/8/1P1NNP2/4K3 w - - 0 1",
    # King, two rooks and three pawns vs king, two rooks and three pawns
    "4k3/1p1rrp2/8/8/8/8/1P1RRP2/4K3 w - - 0 1",
    # King, queen, knight, bishop and pawn vs king, queen, knight, bishop and pawn
    "4k3/1pqnbP2/8/8/8/8/1PQNBp2/4K3 w - - 0 1",
    # King, queen, two bishops and pawn vs king, queen, two bishops and pawn
    "4k3/1pqbbP2/8/8/8/8/1PQBBp2/4K3 w - - 0 1",
    # King, queen, two knights and pawn vs king, queen, two knights and pawn
    "4k3/1pqnNP2/8/8/8/8/1PQnnp2/4K3 w - - 0 1",
    # King, queen, rook, bishop, and pawn vs king, queen, rook, bishop, and pawn
    "4k3/1pqrBP2/8/8/8/8/1PQRbp2/4K3 w - - 0 1",
    # King, queen, rook, knight, and pawn vs king, queen, rook, knight, and pawn
    "4k3/1pqrNP2/8/8/8/8/1PQRnp2/4K3 w - - 0 1",
    # King, rook, two knights and pawn vs king, rook, two knights and pawn
    "4k3/1prnNP2/8/8/8/8/1PRNnp2/4K3 w - - 0 1"
]
```

Limitations:

Given compute-power limitations, I elected to train the model on 10 different chess end-games. These end-games are documented thoroughly in the codebase (shown on the left)) and consist of different variations of five-on-five matches (e.g., king, queen, and three pawns for both player and opposition).

Datasets:

There was not a need for external datasets because the model learned via self-play. The model learned state-value pairs via Q-learning. The python-chess environment was used to train and test the model. In the *endgame_environment.py* file, I modified the original codebase environment to start at certain endgames. Moreover, I also modified the *reset* function for the environment so that we can continue testing a trained model on a different endgame. The board states were represented as FEN, or Forsyth-Edwards notation, which models chess states as a single text string (for example,"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR" represents the initial position of a chess game). We could pass in modified values for different end states as well, which I was able to determine using an online tool that converted the board state to FEN.

Main Contributions to Codebase:

The primary modifications to the codebase were as follows:

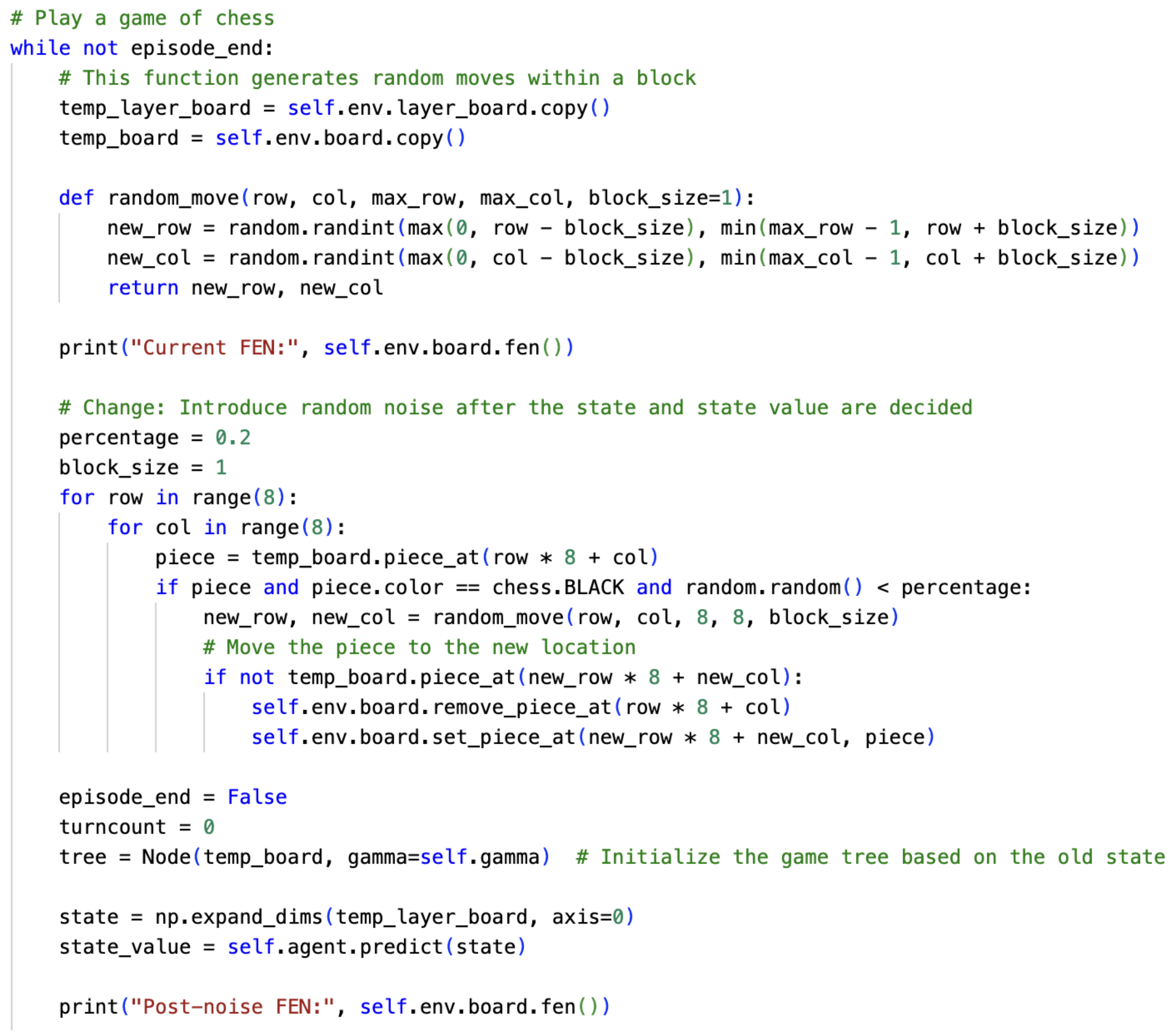
- End-game introduction: A list of 10 possible endgames was generated, as detailed below. Each player had five pieces to start. Endgames were chosen at random during training/testing.
- Model weight preloading while testing: TensorFlow Keras was used to implement a *load_model()* that passed in a .h5 file generated from training.
- Introduction of noise: modified chess environment to obscure pieces (detailed below in the description of the primary codebase)
- Main changes are shown in a fork, also imported directly in the testing codebase
- Script to convert series of FEN into a GIF (link)

Links to Codebase:

1. Primary codebase, with learning, testing, and output files

https://github.com/john7rho/neuro140-revised

This codebase contains the code used to train a Convolutional Q-network algorithm that learns state-value pairs while planning moves using a Monte Carlo Tree Search algorithm. Primary contributions include coding a new environment that introduced random noise to the board state (*randnoise_learn.py* and *randnoise_environment.py*). The changes are such that there is a threshold (0.20) at which a piece has a chance of being moved within a 1-block radius of its current state. This change is not relayed to the white player, whose perception of the board state is before the noise is introduced.



```python
# Play a game of chess
while not episode_end:
    # This function generates random moves within a block
    temp_layer_board = self.env.layer_board.copy()
    temp_board = self.env.board.copy()

    def random_move(row, col, max_row, max_col, block_size=1):
        new_row = random.randint(max(0, row - block_size), min(max_row - 1, row + block_size))
        new_col = random.randint(max(0, col - block_size), min(max_col - 1, col + block_size))
        return new_row, new_col

    print("Current FEN:", self.env.board.fen())

    # Change: Introduce random noise after the state and state value are decided
    percentage = 0.2
    block_size = 1
    for row in range(8):
        for col in range(8):
            piece = temp_board.piece_at(row * 8 + col)
            if piece and piece.color == chess.BLACK and random.random() < percentage:
                new_row, new_col = random_move(row, col, 8, 8, block_size)
                # Move the piece to the new location
                if not temp_board.piece_at(new_row * 8 + new_col):
                    self.env.board.remove_piece_at(row * 8 + col)
                    self.env.board.set_piece_at(new_row * 8 + new_col, piece)

    episode_end = False
    turncount = 0
    tree = Node(temp_board, gamma=self.gamma)  # Initialize the game tree based on the old state

    state = np.expand_dims(temp_layer_board, axis=0)
    state_value = self.agent.predict(state)

    print("Post-noise FEN:", self.env.board.fen())
```

Above is the code introduced to the randnoise_learn.py file. Random noise is used to obscure the perception of the board state for the white player. This changes the transition probabilities for the player.

2. Fork from original reinforcement learning chess codebase

https://github.com/john7rho/RLC

This codebase is a fork from the original, including some changes from my end. To note, all relevant code used/modified from this codebase was included in the primary codebase above.

3. Script to convert FEN into GIF

https://colab.research.google.com/drive/1pYyZu48YdYXxjHE1apBhkxPBiQXLTZeu?usp=sharing

This was used to convert the output of FEN states for the noisy environment into a GIF. This was because the noise made it so that the PGN did not output properly.

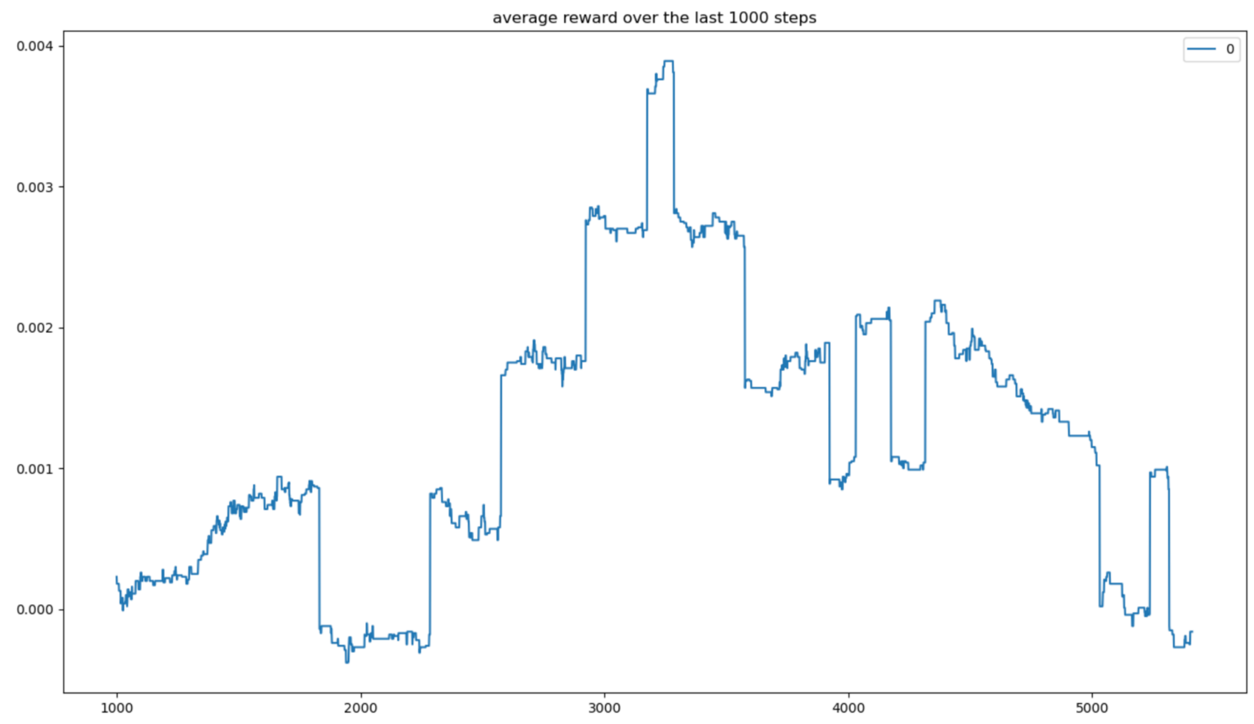
Results

We tested for two primary objectives: maximizing the material balance and reaching checkmate from the white player's perspective. While training the model, the network was updated using Temporal Difference (TD) learning, as detailed in the methodology.

Challenges: I ran into several difficulties while running this project. Primarily, I was unable to run a simulation of a full chess match because of compute limitations. Hence, I elected to focus on end states (with less pieces). Moreover, a large number of iterations were still necessary to train the model to reach an acceptable level of performance – on the order of 100,000 to 1,000,000 or more. This took several hours to train, and the testing of the model was even more time consuming. This is because matches, particularly in the noisy environment, could last for 100 half-moves or more. Qualitatively, the model would learn to simply move the king around to stall a loss, and this dragged out games for much longer than they should have. It is possible that this could be improved with further iterations or modifications to the model structure; if I have more time in the future, I would re-visit other changes to discover a solution to this behavior.

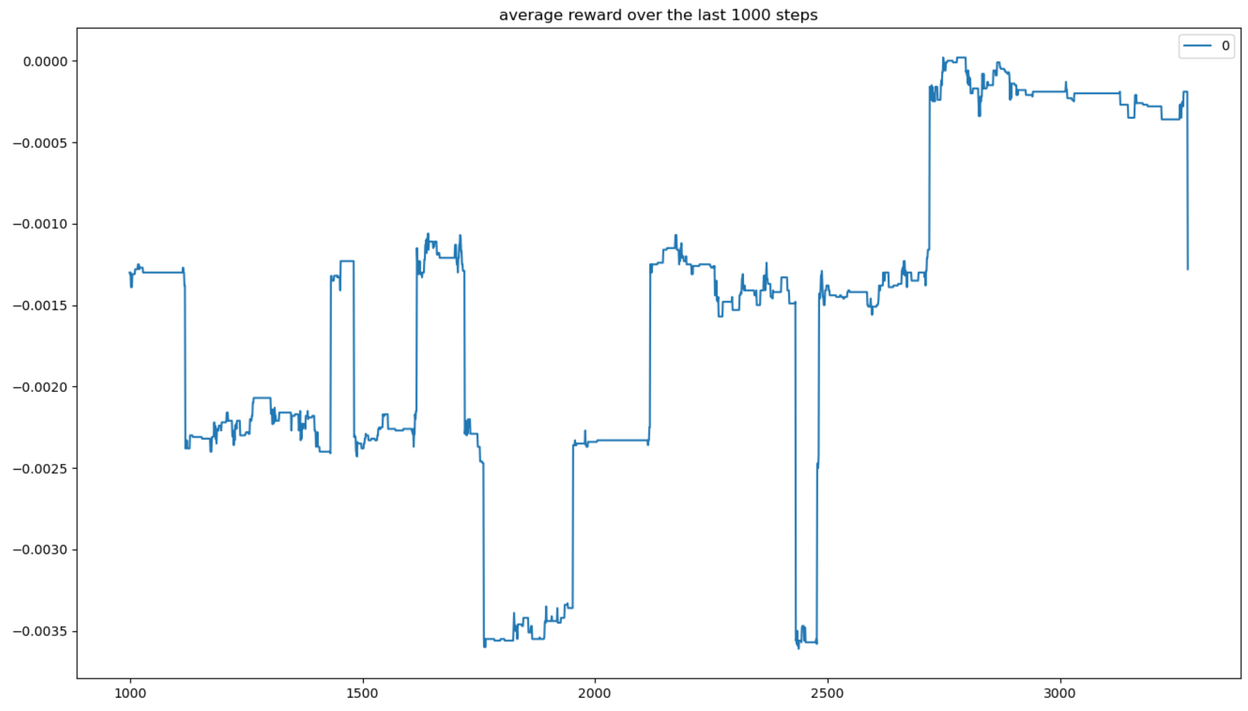
The below was from the continual learning of the Deep Q-network model that was initially trained in a normal, noise-free environment.

Graph 1: Reward Performance in Noise-free Environment (100,000 iterations, noise-free environment, no pre-loaded weights):



The weights from this model were then saved to be inputted into the learning showcased below. Note that performance is essentially net zero, though still largely in the positive range. Moreover, the number of steps is larger for this environment compared to below, suggesting that many more of the games are "stalled" in that they reach the maximum number of half-steps without reaching a conclusive end (leading to 0 reward).

Graph 2: Reward Performance in Noisy Environment (100,000 iterations, noisy environment, pre-loaded weights):



Note that this was after preloading model weights into the network. The model started with much lower performance, given that it had to revise its weights to the new environment. Note that the performance, however, oscillates frequently, and losses outnumber wins.

Demos

Noise-free environment

- Demo 1a: checkmate, reward 1
- Demo 1b: draw, reward 0

Noisy environment

- Demo 2a: timeout (draw), reward 0
- Demo 2b: loss, reward -1

Codebase Walk-Through

- Link: https://youtu.be/FRRbnCf442Q

Full folder of GIFs linked here.

Discussion

The data suggest that the state-based model, when originally trained on a noise-free environment, is able to continually learn to overcome uncertainty in perception. The model can play chess endgames at a level similar to a model in a noise-free environment. As shown in Graph 2, performance is initially extremely poor and generally declining (negative reward), but then it is able to overcome the initial difficulties after learning. This leads to a rise in performance to reach draws.

However, the overall performance is poor within the random noise environment. Comparatively the noise-free environment also yields mostly draws, but the performance graph is still largely in the positive range, whereas the noisy environment has performance that is more loss-heavy (and hence negative), especially early on. This is expected since black has full perception for the full game. Moreover, quantitatively, we can also **postulate that the near-zero reward is also due to the games stalling for longer and longer**. Note that we limit the games to 100 half-moves for performance reasons (real-world average is 80 half-moves). We can see this "stalling" in demos qualitatively as well, such as in Demo 2, in which the players stall until the 100 half-moves are done.

Hence, we can theorize that there could be a greater convergence of a winning strategy for extremely large game lengths (200 half-moves or more). Given more compute or time, it would be interesting to run this study for 500,000 - 1,000,000 iterations with games of length 200 half-moves or more. With the current model and training, it is interesting to see that **the model learns to prioritize reaching a draw (stalemate)** over reaching a checkmate.

In the future, I would also like to evaluate the model performance to directly compare with existing models and humans. This would require training in a full chess environment, which would require greater computational resources. Current state-of-the-art research into chess algorithm behavior has explored how to make models perform more like human players, trained on a significant volume of human-played games from websites like chess.com. Future research could involve collecting data on how humans perform playing chess given perceptual deficits; these data can then be compared to an expansion of the results based on the methodology proposed in this paper.

Works Cited

Küchelmann, T., Velentzas, K., Essig, K., Koester, D., & Schack, T. (2022). Expertise-dependent perceptual performance in chess tasks with varying complexity. Frontiers in Psychology, 13. https://doi.org/10.3389/fpsyg.2022.986787

Lazaric, A., de Cote, E. M., & Gatti, N. (2007). Reinforcement learning in extensive form games with incomplete information. Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems. https://doi.org/10.1145/1329125.1329180

Loquercio, A., Segu, M., & Scaramuzza, D. (2020). A general framework for uncertainty estimation in deep learning. IEEE Robotics and Automation Letters, 5(2), 3153–3160. https://doi.org/10.1109/lra.2020.2974682

McIlroy-Young, R., Sen, S., Kleinberg, J., &amp; Anderson, A. (2020). Aligning superhuman AI with human behavior. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery* &amp; Data Mining. https://doi.org/10.1145/3394486.3403219

Nair, S. (2017, December 29). *A Simple Alpha(Go) Zero Tutorial*. Simple alpha zero. Retrieved May 4, 2023, from https://www.web.stanford.edu/~surag/posts/alphazero.html

Sigman, M. (2010). Response time distributions in Rapid Chess: A large-scale decision making experiment. Frontiers in Neuroscience, 4. https://doi.org/10.3389/fnins.2010.00060

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., &amp; Hassabis, D. (2018). A general reinforcement learning algorithm that Masters Chess, Shogi, and go through self-play. Science, 362(6419), 1140–1144. https://doi.org/10.1126/science.aar6404

X. Zhou, "Application of Machine Learning in Games with incomplete information," ICMLCA 2021; 2nd International Conference on Machine Learning and Computer Application, Shenyang, China, 2021, pp. 1-4.