# Double Deep Q-Learning Chess Agent with Action Replay

**Thomas Inman**
260947857
thomas.inman@mail.mcgill.ca

**Marie Ezra Marin**
261053813
ezra.marin@mail.mcgill.ca

## Abstract

The game of chess is a popular game with various artificial intelligence (AI) algorithms created with the purpose of playing. We aimed to create an agent that uses the state of the board as the state value function. The agent implements a double deep Q-learning model with an action replay and alpha-beta pruning. We utilize a convolutional neural network (CNN) as our state value network. In comparison, AlphaZero and Stockfish use a much larger action value network. We trained the agent based on two methods: self play and real games, with a goal of finding difference in performance between these methods of training.

## 1 Introduction

The game of chess has an incredible number of game states: approximately $10^{40}$ possible game states. Creating an AI to play chess is a daunting task, especially when factoring in the number of parameters and training time required. Thus, our goal was to create a model that uses far less parameters and requires less resources to train. To simplify our model, we forked the git repository Andoma [2]. In the course of this paper we will refer to ELO (the player chess rating system) as a measure of our agents ability to play.

Our agent is a variation of the double deep Q-Learning agent; specifically, instead of an state-action network, we use a state value network combined with alpha beta pruning. The intuition behind this choice is to reduce the number of trainable parameters used, and thus the resources required to train our networks. As mentioned above the state space of chess is abundantly large, thus representing the board state is a important stage of creating a well functioning chess agent. We use a 3-dimensional matrix to represent the board state (specifics discussed in methodology). Both networks use the same CNN into dense layers structure as their state value function. The training consists of self play or real game data. The testing phase consists of playing against the Stockfish bot.

## 2 Background

The backbone of our agent was based on the GitHub repository Andoma [2]. The Andoma repository acts as a chess engine providing us with basic features. The chess engine supplies methods that to convert Forsyth-Edwards Notation (FEN) into a board state. This is an important step as FEN is the standard notation for chess games, which allows us to use FEN data from past games. The chess engine allows us to retrieve a list of acceptable moves. The intuition behind this is to train the agent over legal moves, removing the need of the agent learning the basic rules. This gives us more time to train the model on the best positions to be in, rather than learning the rules of chess. Andoma provides a user interface (UI) that allows us to visualize the board. This helped in the experiment

Preprint. Under review.

phase and also allowed us to play against the bot ourselves.

Part of our experiments includes playing against a virtual opponent. We chose to use Stockfish, a well known chess bot, to play against our agent. Stockfish provides a library in python, allowing us to select the skill level, ELO, depth, and other parameters for the Stockfish bot. This gives us further control on our agents adversary. Due to our model being much weaker, in terms of number of parameters and training data used, we opted to use weaker versions of Stockfish in order to measure the results of different parameter selection.

The goal of our convolutional neural network (CNN) is to output a score (state value) given a board state. The rationale behind using a CNN is due to the state representation of the chess board. Since we represent the chess board as a $(12 * 8 * 8)$ matrix, with each $(8 * 8)$ channel representing color and piece type, a 2D CNN appears to be a powerful model. Additionally, a CNN would allow the learning of spacial relationships [1], which is instrumental in chess. While CNN's are typically used for image recognition and classification, utilizing a CNN with our state representation should provide more opportunity for feature extraction compared to a typical neural network. This signifies that the CNN is better equipped to handle our state representation. The implementation of the CNN is dependent on the TensorFlow library.

## 3 Methodology

### 3.1 State representation

To represent the state of the board we created a three-dimensional matrix. The matrix is divided into 12 channels, representing a piece type and color. The first 6 channels represent the location of the agent's pieces and the second six represent the opponents pieces. Since there are six piece types (pawn, rook, knight, bishop, queen, and king), each channel represents a chess board for a distinct color and piece type. The first channel, for example, is an $8 * 8$ binary matrix (chess board) with a value of one meaning a white pawn is located there and 0 meaning no white pawn is located on the tile. The intuition behind this representation is to allow the CNN to learn the different pieces and their value in specific locations. We can further improve this representation by adding the attacking squares of a player. This would mean that the CNN would get two additional channels representing the attacking squares for the agent and the opponent.

### 3.2 Agent

For the reinforcement learning agent, we are using double deep Q-learning model with action replay, and a Deep CNN as the underlying network. Due to our model outputting a value for the board state, we used a modified version of the Q-Learning which incorporated minimax with alpha-beta pruning. When selecting the best move, we use minimax; in contrast, when exploring moves we use the output of our CNN to select the moves which would give us an advantage: either the highest value for white or the lowest for black, and we then save the transition inside our memory. Since we use an action replay, we update the weights batch wise. We use two networks, where the second neural network (named targetModel) is used to predict the future rewards, which we then update only at the end of the episodes. This is done because in our trials with our initial neural network and agent schema, we had more information encoded in the input, and we output the probabilities of choosing a certain move which we used to make the chess engine decide. An issue is that this method would get stuck in very long cycles of preferring to repeat moves than to actively try to play to win, and even when tweaking hyper-parameters, it took a very large amount of iterations for it to break that habit. We are implemented a decaying epsilon model, but instead of starting with all random moves (because of the high cost in time and computing power of self-play simulations), we use epsilon chance of using a hard-coded evaluation function [6] and 1-epsilon chance of getting the evaluation from our model.

As stated above the agent uses a CNN as its state value function. CNN takes a $(12 * 8 * 8)$ matrix as input which represents the board state. From this input we use a few convolution layers (the number of layers being a parameter of our model). Then we flatten the results and add dense layers to our model. The final layer has an output size of one, representing the state value. The CNN was trained with an $AdamW$ optimizer, provided by TensorFlow. The figures *(c)* and *(d)* (without and with normalization) in the appendix represent the structure of our CNN. Note that the number of parameters in the CNN depend on the input parameters and whether or not normalization was used.

Since our algorithm relies on the board state rather than a policy, we implemented minimax with alpha beta pruning and move selection. This allows our model to play at various depths, thus providing a parameter for difficulty level. The move ordering and alpha-beta pruning was implemented by the Andoma repository. Using alpha-beta pruning and move ordering shortens the average runtime of our agent permitting larger depth usage.

### 3.3 Training and Data

The agent was trained with two methods: real game data and self play. The data for the real games was collected from LiChess. LiChess is a popular chess application that contains many full games utilizing FEN. Using FEN allows us to use past games played by Grand Masters, Masters, and even lower ELO games. Training our model with various levels of game-play means our model will be trained in a fashion that allows for an exploration of moves. Additionally, we can examine the differing results based on training style. We also use data from PGN mentor as part of our training set [7].

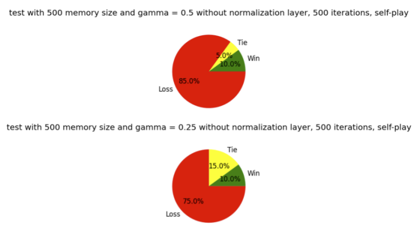
## 4 Experiment

As stated above, we used the Stockfish bot as part of our experiment section. This encompassed two types of play: full chess games and pre-positioned games. We use this style because it allows us to verify how our agent performs in a full game and how our agent performs with checkmates, captures, and other chess principles. Pre-positioned games are a good measure of our agents strength because of known solutions to these positions. For example, a starting board state where our agent can checkmate in three moves. In contrast, full games are used to measure the average performance of our agent. The full games, however, come with several disadvantages, namely, a long runtime and skewed results. The results are skewed due to the low number of test games played by our agent and Stockfish.

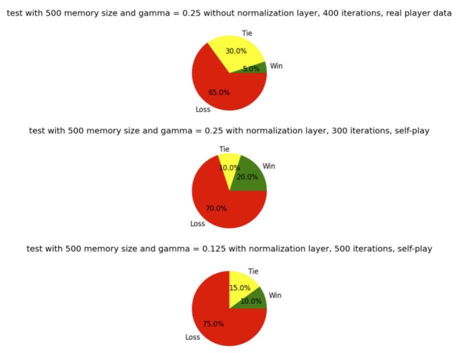
### 4.1 Results

Each of the tests on the models is a pie chart of the full games played against the Stockfish agent. The Stockfish agent was set to a depth of one. This was done in order to test our agents networks rather than test the alpha beta pruning method. Additionally, setting our model and our opponents (Stockfish) to a depth of one saves time during testing. We tested four different parameter settings: gamma, normalization layer (with and without), number of training iterations, and training style (self play or real game data). We vary gamma between $1/8$ and $1/2$. It should be noted that the test phase consists of only 20 games per hyper-parameter setting. This is due to the amount of time required to run each game: around 15 minutes.

We observe that the best performing model (based on win rate) has a gamma of $1/8$, a normalization layer, 500 training iterations, and was trained on self play. The model with the lowest loss rate has a gamma of $1/4$, no normalization layer, 400 training iterations, and was trained with real game data [7]. Although the results of our models vary, we still obtain sub-optimal results when facing the Stockfish opponent. However, when playing models ourselves we observed that our agent does not make random moves, giving us tricky moves at time.



(a)



(b)

### 4.2 Future improvements

Our agent, as a proof of concept, performs adequately. Further improvements must be made to achieve good results: modifying our board representation, increasing training iterations, improving the speed of our agent, and using a larger network. The board representation can be improved by adding two more channels indicating the attacking squares for both colors. This change would give our agent more chess specific information, leading to better feature extraction. Additionally, the board state can be represented using a bitboard rather than a numpy matrix. This is, however, not implementable using the Tensorflow library. The speed of our agent was a major bottleneck in the testing and training phase. Due to the amount of resources required for the training and testing phase, we were unable to run a large number of training episodes. With more resources (time and hardware), we could train more episodes and achieve better results. A larger network can be used to increase performance, however, it comes with the cost of longer training times. Another potential improvement to our model would be to use a combination of self play and real game data. Starting with real game data and ending with self play could result in better results.

## 5 Conclusion

The implementation of a double deep Q-learning agent for chess is an interesting area of research. Creating a chess bot with limited resources proved to be a challenging problem. However, we achieved an agent that performs adequately well. Alterations to the board representation, CNN, and training can be made with access to more resources, leading to better results. Our agent uses a fraction of parameters and resources compared to those used in Stockfish and AlphaZero. More testing is required to find the better performing network structures and hyper-parameters. In conclusion an agent utilizing double deep Q-learning, alpha-beta pruning, and state-value network can be implemented and compete with lower level bots and players.

## References

[1] B. Wang, R. Ma, J. Kuang, and Y. Zhang, (2020) "How Decisions Are Made in Brains: Unpack 'Black Box' of CNN With Ms. Pac-Man Video Game," *IEEE*, vol. 2020, no. 7, pp. 2645-2653, 2020. DOI: 10.1109/ACCESS.2020.3013645.

[2] C. Healey. (2019) andoma. GitHub Repository. Retrieved from `https://github.com/healeycodes/andoma`.

[3] S. Maharaj, N. Polson, and A. Turk, "Chess AI: Competing Paradigms for Machine Intelligence," *Entropy*, vol. 24, no. 4, p. 550, 2022. DOI: 10.3390/e24040550.

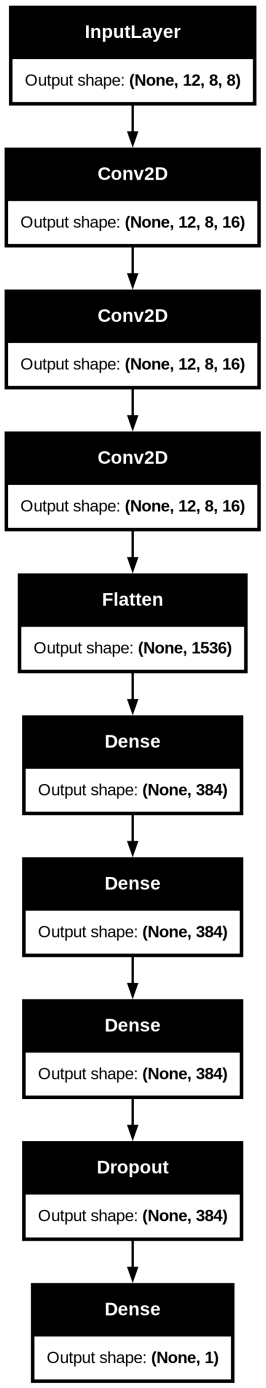
[4] Ze-Li Dou, Liran Ma, Khiem Nguyen, and Kien X. Nguyen. (2020). 2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC). In *Proceedings of the IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, November 6-8, 2020. DOI: 10.1109/IPCCC50635.2020.9391562.

[5] M.-A. Dittrich and S. Fohlmeister. (2021). "A deep q-learning-based optimization of the inventory control in a linear process chain." *Production Management*, vol. 15, pp. 35–43, 23 November 2020. DOI: 10.0000/production-management.

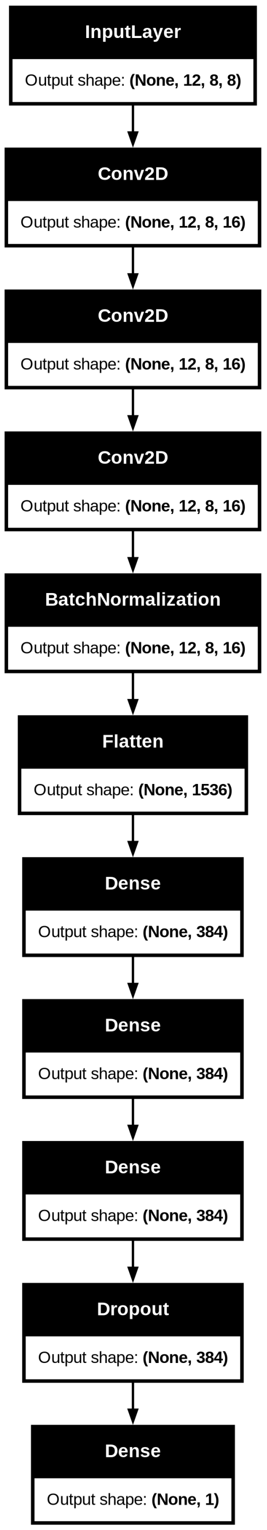
[6] Chess Programming Wiki. (2021). "Simplified Evaluation Function." Retrieved from `https://www.chessprogramming.org/Simplified_Evaluation_Function`.

[7] PGN Mentor. (2022). Retrieved from `https://www.pgnmentor.com/`.

## 6 Appendix


InputLayer
Output shape: (None, 12, 8, 8)
Conv2D
Output shape: (None, 12, 8, 16)
Conv2D
Output shape: (None, 12, 8, 16)
Conv2D
Output shape: (None, 12, 8, 16)
Flatten
Output shape: (None, 1536)
Dense
Output shape: (None, 384)
Dense
Output shape: (None, 384)
Dense
Output shape: (None, 384)
Dropout
Output shape: (None, 384)
Dense
Output shape: (None, 1)

(c) Convolutional Neural Network Layers


InputLayer
Output shape: (None, 12, 8, 8)
Conv2D
Output shape: (None, 12, 8, 16)
Conv2D
Output shape: (None, 12, 8, 16)
Conv2D
Output shape: (None, 12, 8, 16)
BatchNormalization
Output shape: (None, 12, 8, 16)
Flatten
Output shape: (None, 1536)
Dense
Output shape: (None, 384)
Dense
Output shape: (None, 384)
Dense
Output shape: (None, 384)
Dropout
Output shape: (None, 384)
Dense
Output shape: (None, 1)

(d) Convolutional Neural Network Layers with Normalization