# Deep Pepper: Expert Iteration based Chess agent in the Reinforcement Learning Setting

**Sai Krishna G.V.**
MILA, Université de Montreal
Montreal
saikrishnagv1996@gmail.com

**Kyle Goyette**\*
MILA, Université de Montreal
Montreal
kdgoyette@gmail.ca

**Ahmad Chamseddine**\*
École Polytechnique de Montreal
Montreal
ahmad.chamseddine@polymtl.ca

**Breandan Considine**
MILA, Université de Montreal
Montreal
breandan.considine@gmail.com

## Abstract

An almost-perfect chess playing agent has been a long standing challenge in the field of Artificial Intelligence. Some of the recent advances demonstrate we are approaching that goal. In this project, we provide methods for faster training of self-play style algorithms, mathematical details of the algorithm used, various potential future directions, and discuss most of the relevant work in the area of computer chess. Deep Pepper uses embedded knowledge to accelerate the training of the chess engine over a "tabula rasa" system such as Alpha Zero. We also release our code to promote further research.

## 1   Introduction and Motivation

The game of chess has been a long standing challenge in modern artificial intelligence. While Deep Blue ([13], [12]) made headlines in 1996, the first chess engines started beating top human players in the 1970's[10] based on the principles laid by Von Neumann [28], Calude Shannon[22] and Alan Turing[3]. Aided by rapid growth in computational power, computer chess has come a long way since then, with most contemporary engines using Alpha-Beta search [18] (like the Stockfish[4] engine, released in 2008). More recently, there has been significant progress in machine learning applied to chess, lead by DeepMind's Alpha Zero [24]. Perhaps the most impressive is Alpha Zero's ability to learn the game entirely from self-play. Alpha Zero makes use of several ideas in deep learning and reinforcement learning, including function approximation with deep neural networks and more conventional heuristic search algorithms such as Monte Carlo methods. However there has been certain criticism[1] claiming that Alpha Zero was given an unfair advantage against computer chess engines during evaluation, which raised questions on how we could leverage the strengths of both the approaches. We have adapted methods from Alpha Zero, deep networks and Monte Carlo Tree Search methods, with a feature representation that encodes information about the rules of chess, and the mobility of chess pieces.

Deep Pepper attempts to build on some of the principles introduced by Expert Iteration [5], Alpha Go and Alpha Zero by using deep neural network architectures, paired with Monte Carlo Tree Search, but to also diverge from Alpha Zero by avoiding a full board representation, using a hand-crafted feature vector, with the existing Stockfish engine and few other methods to accelerate training. We also

---

\*Equal contribution, random order

provide mathematical details of the algorithm as an extension of Classification based Approximate Policy Iteration (CAPI)[8] framework, and experimental details as the training progresses.

Our code may be found at this URL: `https://github.com/saikrishna-1996/deep_pepper_chess`

## 2 Related work

Chess has been explored for more than 40 years in the world of computer science, as such there exists a large amount of related work, here we shall focus on more contemporary and machine learning(ML) driven approaches.

Some of the initial ML-based chess engines attempted to play like grandmasters by imitation learning on grandmaster games. But, they failed due to the absence of negative dataset in the case of sacrificing major pieces. Some engines attempted to overcome this problem by hard coding in some constraints. But, they still did not achieve strong play because the target policy vector was always one-hot encoded. So just trying to learn a policy from given states is not ideal. However, one has to note that this might work in other games like Go, Othello, Hex or connect4. Unlike all those games, chess has a "tactical" element to it and hence the agent must look ahead server moves to decide the move at any given position. Thus started the era of planning agents in computer chess.

Most of the engines use alpha-beta pruning search with value function evaluation at the leaf nodes of the tree. However, it poses certain problems. Primarily, it is difficult to determine the depth of the search. And if the value function is slightly misaligned, important nodes are overlooked.

Barber et al. [5] is one of the earliest known attempts of using a completely self-play algorithm to learn a board game, demonstrating promising results on the game of Hex. David Silver et al. extended their results to the game of Go [25] (whose performance exceeded their previous version [23]), and further to the game of chess [24]. [27] is an attempt to replicate the results of [25]. Leela Chess attempted to replicate the results of AlphaZero, but is still far from the results reported in [24]. Giraffe [17] is one of the first reinforcement learning based agents which performed at International Master level. Our custom feature representation is inspired by Giraffe's representation. Our pretraining is based on the evaluations provided by Stockfish [4] which is currently the best performing open source engine.

Recent developments on Monte Carlo Tree Search such as memory-augmented MCTS [15] outperformed the original MCTS in the game of Go. Other methods like MCTSnets[9] incorporate simulation-based search inside a neural network, but it remain computationally very expensive.

## 3 Methods

### 3.1 Overview

The Deep Pepper chess engine consists of a neural network (value network) which predicts the value of a given state as well as the optimal move (policy network) at a given state. The policy network is coupled with an MCTS algorithm, which performs a number of simulations of the most probable moves as dictated by an upper confidence tree selection process, initially based on the neural network's predictions. Coupling the neural network architecture with Monte Carlo Tree Search produces a powerful group of training algorithms called expert iteration[5] in the framework of Classification-based Approximate Policy Iteration (CAPI)[8]. Unlike Alpha Zero[24], Deep Pepper is trained by embedding prior knowledge of the game of chess. Deep Pepper leverages the Stockfish engine for early stopping of games to accelerate training, and can be optionally pretrained using games from professional players and Stockfish evaluations. It also uses custom feature representation.

### 3.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is used by Deep Pepper in both policy evaluation and policy improvement. MCTS is a heuristic search algorithm used in Deep Pepper to evaluate the most promising moves given the current board state. The algorithm is broken down into three main phases:

1. **Selection:** Starting from the root node of the current tree, the most promising child node, is selected until an unvisited node is encountered.

2. **Expansion and Evaluation:** Once a leaf node is encountered, the leaf is expanded using Deep Pepper's neural network, which outputs move probabilities for all possible moves on the board, these probabilities are reduced to just legal moves. The value $v$ of the leaf node is computed from our value network.

3. **Back-up:** The statistics for each of the move in the explored tree are updated, including updating each of the action values with the output of the neural network, as well as the number of times each of the move has been taken.

Deep Pepper stores each of the board states as a node, with each node containing information about the number times each of its children were visited $N(s, a)$ (where s is the state, and a the action), the total value (as determined by the neural network or game conditions) of all descendant leaf nodes expanded beneath each node $W(s, a)$, the average value for each node $Q(s, a) = \frac{W(s,a)}{N(s,a)}$ and the initial move probabilities, $P(s, a)$ (as determined by the neural network). Additionally, Dirichlet noise is added to the initial probabilities of the root node during each call of MCTS, to further favor exploration and avoid aggressively repeatable play.

MCTS in Deep Pepper explores states and actions using a variant of Polynomial Upper Confidence Tree's (PUCT)[21]. With PUCT, the selection process is based on estimates on the upper bound of selection probabilities $U(s, a)$ where

$$U(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_k N(s, k)}}{1 + N(s, a)}$$

$c_{puct}$ is a hyper-parameter, called the exploration factor, controlling how much one would like to explore less visited nodes. In this variant, we also add the average value for each given node to the PUCT values. That is, given the current statistics stored within a node, the child node is selected via:

$$A = argmax_a(Q(s, a) + U(s, a))$$

At the end of each iteration (800 simulations of MCTS), MCTS provides a policy to be used in the current game state. This policy is determined by:

$$\pi(s, a) = \frac{N(s, a)}{\sum_a N(s, a)}$$

Optionally, a temperature parameter $\tau$ has been added to account for the exploration-exploitation paradigm. This has been annealed to an infinitesimal value after the opening phase of the game. The policy is now determined via:

$$\pi(s, a) = \frac{N(s, a)^{\frac{1}{\tau}}}{\sum_a N(s, a)^{\frac{1}{\tau}}}$$

### 3.3 State Representation

The value network (which is used in evaluating leaf nodes) and policy network (which is used in determining prior probabilities of every state node in MCTS) require a board representation to be given as input. Some recent methods like Alpha Zero use the complete board representation of dimensions 8x8x73. This primarily poses 2 problems: 1) It is harder to train as it takes lot of time and too many layers, owing to its giant input representation. 2) In chess, two similar-looking boards might have a completely different evaluation. Traditional convolutional neural networks (CNNs) are not equipped to handle this issue. (This can be understood by using the following intuition: Consider a CNN trained to classify the images as cats or dogs. If you modify one pixel in a cat image, it is still supposed to output it as 'cat' (unless it is an adversarial attack[26]). However, in chess, if the white queen is at 'f3' instead of 'g3' the evaluation would change from -500 to +500 (on centi-pawn scale[2]) and CNNs cannot be trained to predict such drastic changes in output with only minor changes in the input.

To counter the above problems, taking inspiration from Giraffe[17], we design a custom 353-dimensional feature representation for given board that is faster to train and immune to adversarial

attacks (to be proven with further experiments and human evaluation). Additionally, we demonstrate that it is a good representation as we observe favorable results during pre-training. Deep Pepper uses embedded knowledge to accelerate the training of the chess engine over a "tabula rasa" system such as Alpha Zero. The state representation used by Deep Pepper hence uses knowledge derived by human experts about the value of pieces, and whether each piece and square is currently threatened and defended, and many other features (as detailed in the appendix).

## 3.4   Neural Network Architecture

We experiment with multiple architectures.

### 3.4.1   PolicyValNetwork-parts

From our feature representation, we connect our global features to h1a neurons(see appendix 7.12), piece centric features to h1b neurons and square centric features to h1c neurons in the first layer. We then concatenate all the outputs from first layer and connect to h2p neurons in the second layer and further this is connected to d-out neurons in the last layer which will give us the softmax policy output. As another head, we also connect the concatenated output from first layer to h2e neurons in the second layer which is further connected to 1 output neuron which gives us the value function. Therefore, we are using value and policy networks as different heads (with last 2 layers being different) in the same large network. This is much faster than the architecture with 2 separate networks, and without significant loss in performance. The number of different neurons used in each layer are detailed in appendix.
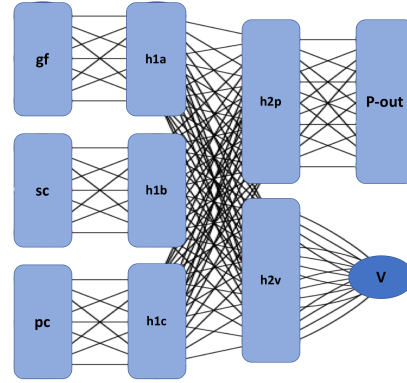


Figure 1: Architecture of PolicyValNetwork-parts

### 3.4.2   PolicyValNetwork-Full

Similar to PolicyValNet-parts, but rather than having separate connections in the first layer, we have a fully connected layer which allows the network to learn more implicit connections between each of the global, square centric and piece centric features. An insignificant performance improvement is noticed which suggests that these extra connections are not effective considering the extra training time.

### 3.4.3   PolicyNetwork-parts and Critic-parts

Here, we use 2 separate networks for policy and value. The architectures are exactly similar to 3.4.1 (PolicyValNetwork-parts) except that, here the first layer is also trained independently for Policy and Value (critic) networks. This architecture has given a minor improvement in the performance.

### 3.4.4   PolicyNetwork-full and Critic-full

It is exactly similar to 3.4.2 except that, here the first layer is also trained independently for policy and value networks. As predicted, there is only a very slight improvement in the performance.

## 3.5   Training Algorithm Pipeline

Deep Pepper was trained using a self-play mechanism. In this format, the system learns a game primarily by playing against itself. The training pipeline can be decomposed into thee main steps:

1. **Game generation**, wherein data is generated to be used to update the network

4

2. **Network training**, wherein the network is improved using back propagation on the afore-mentioned generated games

3. **Improvement testing**, wherein the new network is tested against the previous iteration in a series of games

Monte Carlo Tree Search is used both during the game generation phase as well as the improvement testing phase to select the best move at each position.

Generally, one can view this training procedure as a form of classification-based approximate policy iteration (CAPI). The policy improvement step begins with MCTS performing a search based on the current network, $f_\theta$, which improves upon the current search policy. This new search policy is projected back into the network's function space via back-propagation during network training. Next, the policy evaluation step can be viewed as values of the outcomes of generated games, which are also projected back into the network's function space via back-propagation during each step network training. (similar reasoning is provided in the AlphaZero paper as well).

### 3.5.1 Game Generation

During the game generation phase of the training pipeline, Deep Pepper plays many games against itself, generating data to be used in the network training phase of the algorithm. During this phase, the current iteration of the network, $f_\theta$ is used in conjunction with MCTS to play games from start to finish. During each turn of the game MCTS is called, using the current state as the root node, determining the optimal policy for the current board state. The most probable move is selected as the best action. Furthermore, each of the states and policies, output by MCTS, are stored as expert policies (to be used for training the network). Next, MCTS is called using the next board state as the root node, and the current collected statistics from MCTS are stored and used again during the following batch of simulations.

In order to accelerate training, Stockfish evaluations are used after $k$ half moves, if the Stockfish evaluation indicates a high probability for the current player to win/lose, the game ends with resignation by that player/opponent. Otherwise, the game continues until ending via draw, or checkmate. Once the game is over, the final value is added to each of the saved states and policies. The score at the end of the game is 0 in the case of a draw, +1 if white wins, or -1 if black wins. In the case of resignation, the back up values are positive if black resigned, or negative if white resigned, and are based on the Stockfish evaluation (with appropriate scaling) at the time of resignation.

### 3.5.2 Network Training

During this phase of the training pipeline, the updated policies and values are projected back into the function space $f_\theta$ defined by the neural network. The data created in the game generation phase is a set of triplets saved from each state of each game played. Each triplet, $\{s, \pi(s,a), \nu\}$, where $s \in S$ is the state of the board, $\pi(s,a) : S \to A$ is the policy generated from MCTS taken at the current board state, and $\nu \in \mathbb{R}$ is the backed up end game value as determined via, draw, winner or resignation. Each of these triplets is passed into the neural network, with $\pi(s,a)$ acting as the label for the networks policy output and $\nu$ acting as the label for the value output of the network.

The objective function defined for the network is as follows:

$$J = L_{MSE}(f_\theta, z) + L_{CE}(f_\theta, \pi(s,a)) + \lambda\Omega(\theta)$$

$$J = (f_{\theta val}(s) - z)^2 - \pi(s,a)^T log(f_{\theta pol}(s)) + \lambda||\theta||^2$$

Where $L_{MSE}$ is the mean squared error loss, between the predicted and realized value of the state at the end of the game, $z$ . $L_{CE}$ is the cross entropy loss, for the learned policy, $\pi(s,a)$ and the predicted policy $f_\theta(s)$. And $\Omega(\theta)$ is the $L2$ regularization loss, applied to the network weights. Using KL-Divergence loss instead of $L_{CE}$ reminds us of distillation[11]. The network is trained via back propagation with gradient updates calculated using an Adam optimizer on mini-batches.

### 3.5.3 Improvement Testing

Once the network parameters have been updated, there is an optional check to ensure that the new network has improved over the last iteration. In order to measure network improvements, self-play

is again used as in game generation, however, the previous generation is paired against the new candidate. As before, at each turn of the game, MCTS guided by either the updated network or an older iteration, and used to determine the optimal move at every given turn. The game is played until checkmate, draw or resignation due to Stockfish evaluation. We only save the outcome of each game for ranking, but no other data during this phase of training.

Should the updated network defeat the older iteration by more than half the time, the newly trained network is kept and used in the next round of game generation.

### 3.6 Pretraining

An optional pretraining phase was added to some models in order to offer early guidance in training the network. During pretraining, board positions (taken from professional games of chess), are assigned value by the Stockfish engine. Next, an approximate policy label is generated using the Stockfish engine to compute a value for each possible subsequent board, and calculating an approximate policy by applying the softmax function to the set of generated values. This is because, as detailed before, training of the policy network directly on expert games does not work.

$$\pi_{approx}(s, a) = (\frac{\exp\left(Stockfish(s_1)\right)}{\sum_k \exp Stockfish(s_k)}, \cdots, \frac{\exp\left(Stockfish(s_j)\right)}{\sum_k \exp Stockfish(s_k)})$$

The generated labels are used to train the networks via supervised learning. In this way, the networks are capable of initially analyzing an approximate value for some common board positions. The same cost function is used as during the main training pipeline:

$$J = L_{MSE}(f_\theta, \nu) + L_{CE}(f_\theta, \pi(s, a)) + \lambda\Omega(\theta)$$

which is optimized again using an Adam optimizer.

Overall, pretraining was performed on approximately 100,000 board positions seen in games played by Kasparov, Karpov, Ivanchuk, Aronian and others.

## 4 Results

Results can be seen in Table 1. Experiments were run by testing trained networks against a purely random decision playing process. So far, only three models were tested in this fashion. Each model was pretrained on a set of approximately one hundred board positions, then trained for some number of games afterwards. Pretrain0 refers to a model that was just pretrained, Pretrain50 refers to a model that received 50 games of additional training and so on.

Additionally, models were tested against previous iterations from their own evaluation. Models were shown to improve as training continued as shown in 4. A model that was trained for 100 iterations (and had pretraining) beat a purely pretrained model in 56 percent of games. Further, a pretrained model won against a randomly initialized model in 64% of games.
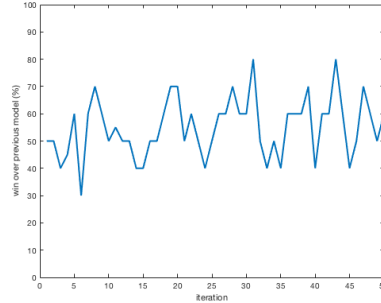


Figure 2: Performance of the current model against it's previous iteration, tested on 10 games at every iteration as the training progresses. We can see that, about 70 percent of the time, our current model is observed to better than it's previous iteration.

## 5 Future Work

### 5.1 Improved Evaluation Mechanisms

When developing a complex system of this nature, evaluation is a critical step to ensuring the system is behaving as expected and meeting desired

Table 1: Trained networks versus random play on 25 games

| Part | |
|---|---|
| Model | Win-rate |
| Pretrain0 | |
| Pretrain50 | 0.55 |
| Pretrain100 | 0.64 |
| Pretrain150 | 0.64 |

performance. At this time, Deep Pepper has been tested only against past models, and models that have had extensive pretraining. Naturally, this limits the scope of evaluation to determining whether the system is improving, but prevents analysis of objective metrics such as ELO. For this reason, a priority of future work will focus on expanding evaluations of models to include ELO rating as an objective metric of skill.

Should ELO based metrics be challenging to test effectively it would also be of interest to test current iterations against past iterations using win percentage as a gauge for growth in the systems skill. This system could be quickly implemented to run parallel to the current system and continuously play a subset of networks against each other, as the network is updated, iterations can be added to the subset of models for evaluation, to show estimated changes in win rates based on number of iterations.

## 5.2 System Optimization

The training pipeline in its current form could be accelerated in a number of ways to reduce the time taken at the primary bottleneck, game generation. More concretely, MCTS takes significant time due to it's high usage of computation resources for determining all legal moves for each board, and calculating $f_{\theta pol}$ and $f_{\theta val}$. These processes could be further optimized by having a large number of processes running simultaneously during game generation. Once a process reaches the expansion phase of the MCTS process, it would stop and wait for a number of other processes to reach the same point. This would reduce the computation cost in using the neural network (as the underlying framework, Pytorch, can be optimized to use a GPU to compute the outputs on a large number of samples).

Training time could also be saved by having each separate MCTS process share node statistics among each other. This change would reduce the time spent expanding identical board positions, which would now be shared among processes, but the impact on total convergence rate of MCTS is yet to be explored.

## 5.3 Hyper-parameter Selection and Network Optimization

At this time, very little hyper-parameter optimization has been performed on the system, leaving room for improvement, since a system of this nature has many hyper-parameters. Initial values for hyper-parameters such as number of MCTS simulations, and PUCT exploration factor were taken from Alpha Zero [24] and Alpha Go [23]. There is no evidence that these values are still optimal given the variation between Deep Pepper and these systems. Additionally, other hyper-parameters, such the number of games to generate between network training have been set to values that have been observed to work empirically, but extensive search for an upper limit of the number of games to use in each epoch has not been performed. Extensive testing should be done to determine the available range of games which can be generated, and the optimal value for maximizing the rate at which the network learns to play.

Furthermore, while several network architectures have been tested, there has not been extensive testing on all available architecture options. Batch normalization[14] has the potential to rapidly increase the speed at which the network learns to play. It would also be of interest to explore alternative activation functions like swish [19], as well as better optimization techniques like [20].

7

# 6 Acknowledgements

# References

[1] On fair comparision between Stockfish and AlphaZero by tord romstad. `https://www.reddit.com/r/chess/comments/7igro1/alphazero_reactions_from_top_gms_stockfish_author/dqyt0fr/`. Accessed: 2018-05-18.

[2] Centi pawn scale - chess programming wiki. `https://chessprogramming.wikispaces.com/Centipawns`, 2009.

[3] Turing's paper machine - garry kasparov. `https://en.chessbase.com/post/reconstructing-turing-s-paper-machine`, 2012.

[4] Stockfish - an open source chess engine. `http://stockfishchess.org`, 2015.

[5] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *CoRR*, abs/1705.08439, 2017.

[6] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *J. Mach. Learn. Res.*, 3:397–422, March 2003.

[7] Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *CoRR*, abs/cs/0703062, 2007.

[8] Amir-massoud Farahmand, Doina Precup, André da Motta Salles Barreto, and Mohammad Ghavamzadeh. Classification-based approximate policy iteration: Experiments and extended discussions. *CoRR*, abs/1407.0449, 2014.

[9] Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. *CoRR*, abs/1802.04697, 2018.

[10] Fred Hapgood. Computer chess bad—-human chess worse. *New Scientist*, pages 827–830, December 1982.

[11] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[12] Feng-Hsiung Hsu. *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA, 2002.

[13] Feng-hsiung Hsu, Murray S. Campbell, and A. Joseph Hoane, Jr. Deep blue system overview. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 240–244, New York, NY, USA, 1995. ACM.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[15] Arbaaz Khan, Clark Zhang, Nikolay Atanasov, Konstantinos Karydis, Vijay Kumar, and Daniel D. Lee. Memory augmented control networks. In *International Conference on Learning Representations*, 2018.

[16] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[17] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *CoRR*, abs/1509.01549, 2015.

[18] Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: Symbols and search. *Commun. ACM*, 19(3):113–126, March 1976.

[19] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *CoRR*, abs/1710.05941, 2017.

[20] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations*, 2018.

[21] Christopher D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):202–230, 2011.

[22] Claude E. Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[23] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[24] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[25] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[26] Jiawei Su, Danilo Vasconcellos Vargas, and Sakurai Kouichi. One pixel attack for fooling deep neural networks. *arXiv preprint arXiv:1710.08864*, 2017.

[27] Yuandong Tian, Jerry Ma*, Qucheng Gong*, Shubho Sengupta, Zhuoyuan Chen, and C. Lawrence Zitnick. Elf opengo. `https://github.com/pytorch/ELF`, 2018.

[28] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928.

## 7 Appendix

### 7.1 Features and Parameters

#### 7.1.1 Feature vector

This is a slightly modified version of our interpretation of Giraffe's[17] feature representation.

**0:** Side to move: 1 for White and 0 for Black

**1-4:** Binary values for castling rights (White queen-side castling, White king-side castling, Black queen-side castling, Black king-side castling in that order).

**5-16:** Number of pieces of each type. White King, White Queen(s), White rooks, White Bishops, White knights, white pawns, black king, black queen(s), black rooks, black bishops, black knights, black pawns in that order.

**17-176:** For every piece, we determine whether the piece is existent (1 slot), normalized x-coordinate (1 slot), normalized y-coordinate (1 slot), lowest valued attacker of that piece (1 slot), and lowest valued defender of that piece (1 slot). So, we use 5 slots for every piece. And, there are a total of 32 pieces. (2 kings, 2 queens, 4 rooks, 4 bishops, 4 knights, 16 pawns). So, a total of 160 slots.

**177-224:** For each major and minor piece, this encodes how far they can go in each direction. So 8 slots for a queen, 4 slots for a rook and 4 slots for a bishop. Since there are 2 queens, 4 rooks and 4 bishops, we need a total of 48 slots.

**225-352:** For each square, we encode the values of the lowest valued attacker and lowest valued defender of that square. Since we have 64 squares, there are a total of 128 slots used here.

### 7.1.2 Network parameters

d-in = 353 (size of input feature representation)

global-features = 17 (includes slots 0-16 described in section 9.1)

piece-centric = 218 (includes slots 17-224 described in section 9.1)

square-centric = 128 (includes slots 225-353 described in section 9.1)

h1a = 32 (no.of neurons in the first hidden layer to be connected to the global features in the input layer for Policy-parts and critic-parts networks)

h1b = 512 (no.of neurons in the first hidden layer to be connected to the piece-centric features in the input layer for Policy-parts and critic-parts networks)

h1c = 480 (no.of neurons in the first hidden layer to be connected to the square-centric features in the input layer for Policy-parts and critic-parts networks)

h2p = 2048 (no.of neurons in the second hidden layer which are fully connected to all the neurons in the first hidden layer in Policy head of PolicyValNetwork-parts and PolicyNetwork-parts)

h2e = 512 (no.of neurons in the 2nd hidden layer which are fully connected to all the neurons in the first hidden layer in Value head of PolicyValNetwork-parts and Critic-parts networks)

h1 = 1024 (neurons in the first hidden layer for policy-full and critic-full networks)

h2 = 2048 (neurons in the second hidden layer for policy-full and critic-full networks)

d-out = 5120 for policy outputs. It is represented as "from square" to "to square" which gives us 64x64 = 4096 possibilities and the additional neurons are added to account for pawn promotions and other miscellaneous moves.

### 7.2 Mathematical Details

### 7.2.1 UCT

These are inspired from Upper Confidence Bound (UCB) based algorithms for k-arm bandit problems [6] . The Upper Confidence Tree (UCT) based algorithm was first introduced in [16]. The objective is to find a path from the root node to the leaf node which maximizes the mean reward defined as $Q_i = E[r_i]$. For non-leaf nodes, the Q-value is defined as $Q_k = max_{i \in T(k)} Q_i$, where, $T(k)$ is the set of all leaf nodes in the subtree with root $k$.

In each iteration $t$, a node is selected (until a leaf node is encountered) which maximizes the $A$-value defined as:

$$A_t(k) = \hat{Q}_{k,t} + c\sqrt{\frac{logN_j(t)}{N_k(t)}}$$

where, c is a positive constant, $N_k(t)$ is the number of times the node $k$ was visited, $N_j(t)$ is the number of times the parent of node $k$ was visited. Note that, in the case of MCTS, each child node is the corresponding state-action pair, and hence $\hat{Q}_{k,t}$ is exactly $Q(s, a)$ where we ignore the iteration number $t$ for convenience in representation. So,

$$\hat{Q}_{t,k} = \frac{1}{N_k(t)} \sum_{s=1}^{t} r_s \mathbb{1}[I_s \in T(k)]$$

This can be compared with:

$$Q(s, a) = \frac{W(s, a)}{N(s, a)}$$

10

Theoretical guarantees for the lower bound on the regret of UCT are given in [16]. But, the worst case analysis given in [7] shows that it could be worse than uniform sampling.

### 7.2.2 Classification Based Approximate Policy Iteration

The formulation of deep pepper fits into the broader class of algorithms called "Classification based Approximate Policy Iteration" (CAPI) introduced in [8] which is given below:

---

**Algorithm** CAPI($\Pi$, $\nu$, $K$)

**Input:** policy space $\Pi$, state distribution $\nu$, number of iterations $K$
**Initialize:** Let $\pi_{(0)} \in \Pi$ be an arbitrary policy
**Loop:** for $\mathbf{k} = 0, 1, \ldots, K$ **do**

- construct a dataset $\mathcal{D}_n^{(k)} = \{X_i\}_{i=1}^n$, with $X_i \overset{\text{i.i.d.}}{\sim} \nu$

- $\widehat{Q}^{\pi_k} \leftarrow \text{PolicyEval}(\pi_k)$

- $\pi_{k+1} \leftarrow \text{argmin}_{\pi \in \Pi} \widehat{L}_n^{\pi_k}(\pi)$ (approximate policy improvement step with action-weighted classification)

---

where, the action-gap weighted empirical loss function $\widehat{L}_n^{\pi_k}(\pi)$ in policy space $\Pi$ is defined as:

$$\widehat{L}_n^{\pi_k}(\pi) \overset{\text{def}}{=} \int_{\mathcal{X}} \mathbf{g}_{\widehat{Q}^{\pi_k}}(x) \cdot \mathbb{1}\{\pi(x) \neq \text{argmax}_{a \in \mathcal{A}} \widehat{Q}^{\pi_k}(x, a)\} d\nu_n$$

$$= \sum_{X_i \in \mathcal{D}_n^{(k)}} \mathbf{g}_{\widehat{Q}^{\pi_k}}(X_i) \cdot \mathbb{1}\{\pi(X_i) \neq \text{argmax}_{a \in \mathcal{A}} \widehat{Q}^{\pi_k}(X_i, a)\}$$

and, the action-gap for a MDP with 2 actions is defined as:

$$\mathbf{g}_Q(x) \overset{\text{def}}{=} |Q(x, 1) - Q(x, 2)|$$

for all $x \in \mathcal{X}$

By using loss distortion lemma, they provide theoretical guarantees for the convergence of CAPI algorithm. They further give directions on extending this to MDPs with multiple actions in each state. By seeing Deep Pepper as an approximation of CAPI (see section 3.5), similar guarantees for convergence can be proven.