Deep Dive on Checkers Endgame Data

by

Jiuqi Wang

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

In this thesis, we investigate applying deep learning techniques to learn the win-loss-draw results contained in the databases of the checkers-playing program CHINOOK. Our initial objectives were to (1) compare a deep-learning-based compression scheme versus the custom algorithm used in CHINOOK, and to (2) extract human-understandable features from the data. We have implemented the data processing pipeline, the neural network and its training loop, and an experimentation infrastructure. Our experiment results suggest that (1) training the neural network with a small random subset of the target database can achieve a high accuracy; (2) using the learned network with a naïve one-ply minimax search can further improve the robustness of the predictor most of the time; (3) transfer learning from one database to another one is feasible; (4) dynamically switching between the model and the one-ply search can give a better result than using either exclusively. We conclude that the neural network equipped with search does a decent job compressing the endgame databases, but the custom algorithm is hard to beat. Extracting features that are useful not only to the neural network but also to humans is a tricky task that requires more sophisticated and creative techniques. Our work is the first effort in this direction.

Dedication

This thesis is dedicated to my beloved parents and friends and all the musicians whose work keeps me company during the late night hours.

"It's better to burn out than to fade away." – Neil Young, *Hey Hey, My My*

Acknowledgements

My tremendous gratitude goes to my supervisors, Dr. Martin Müller and Dr. Jonathan Schaeffer, for guiding me through every stage of this work with professionalism, patience and care, offering their essential help with my Ph.D. application and preparing me as an early career researcher. My achievements would not be possible without their dedication.

I would also like to thank my group members for being incredibly supportive and friendly, especially Zeyi Wang, who assisted me in tackling technical issues.

Lastly, I would like to thank my girlfriend, Vicky Chen, who keeps reminding me of my worth and pulling me out of negative thought loops.

Contents

List of Tables

List of Figures

1 Introduction

Checkers is a combinatorial game that has been popular among not only human players but also artificial intelligence (AI) research for decades, thanks to the simplicity of its rules. Although checkers is not a game of chance, its number of possible positions is roughly 5×10^{20} – a daunting sum making it impossible for humans or even modern computers to analyze exhaustively. Research on AI using checkers as a benchmark date back to the 1950s[53, 54]. As a result of advancing research and ever-developing computer hardware, checkers playing programs became stronger and stronger, eventually reaching and defeating the best human players[57]. The endgame databases built for the checkers playing system CHINOOK eventually became the foundation for a long-running program that was able to solve checkers. Perfect play by both sides leads to a draw[56]. Checkers is the most complex game in terms of search space that has been solved to date.

An endgame database contains positions near the end of the game – endgame positions – with their computed analysis – usually the outcome of the game if both players play optimally in the subsequent moves. Thus, game-playing programs equipped with an endgame database only need to search until they reach a position in the database to determine the game result (win, loss, draw), potentially speeding up the program and enhancing accuracy of the search. Since the game of checkers has an overwhelmingly large search space, endgame databases only capture a small fraction of all possible states. Nevertheless, this small portion makes a big difference in game playing and solving. The endgame databases for solving checkers contain all positions with a total of up to 10 pieces on the board. There are roughly 40 trillion (4×10^{13}) entries in total in the databases, compressed into 256 GB of data on the disk[55].

The checkers endgame databases contain a vast amount of knowledge about how to play the endgames perfectly. The problem is that the databases are "data rich" and "knowledge poor". To date, no one has been able to turn the 40 trillion positions in the databases into a handful of human-understandable heuristics. This is a major challenge of AI – turning computational results into useful human knowledge.

Deep learning is a subfield of machine learning that focuses on studying and applying deep neural networks[33], a class of computational models inspired by the human brain. Typical deep neural networks contain parameterized layers of computational units called neurons where the outputs of one layer are the inputs of the next. Generally speaking, training a deep neural network refers to tuning the parameters of the layers with respect to a well-defined objective function using techniques such as backpropagation. A trained network can capture the correlations (if any) between the input data and the output result and can often generalize to unseen data remarkably well. Deep neural networks have revolutionized the field of AI thanks to their unprecedented scalability, applicability and generalizability. They have triumphed in computer vision[19, 30], natural language processing[21, 22, 64], data mining[46], and many other fields.

End-to-end learning is one of the main characteristics of deep learning[59]. It refers to training a learning system represented by a single model such that the inputs to the system are raw data with minimum human engineering and the outputs are the ultimate targets of the model. The intermediate outcomes between the layers are known as features or representations. A deep neural network can automatically extract features useful for prediction given enough training data. It is widely acknowledged in the deep learning community that automatic feature extraction is a crucial property that distinguishes deep learning from previous learning methods. It is also what makes end-to-end learning possible. One of the foundational examples of applying deep neural networks in image classification is ALEXNET, where the authors employed end-to-end learning to train the deep neural network and pushed forward the state-of-the-art by a large margin[30].

In the checkers database context, we have 40 trillion input states (positions) and three output states (winning, losing, or drawn position). What can we learn from a neural network that reflects this large dataset?

1.1 Motivation and Goals of this Thesis

Given a giant endgame database, standard compression algorithms (such as gzip) do not take advantage of domain-specific properties. Furthermore, the program using the data may require thousands of accesses per second. Thus, finding a value in the database requires a fast decompression algorithm for single positions (gzip cannot do this). Apart from that, the database is in the form of lookup tables, implying that it does not know the value of the positions that have not been computed. These observations incentivize us to leverage modern deep learning techniques to train a deep neural network to compress and generalize the database. The neural network should have a much smaller number of parameters than that of the positions it captures. Therefore, the neural network has to share weights and extrapolate to unseen data.

Although deep neural networks are good at approximation and generalization, they can rarely achieve perfection – they can make wrong predictions. Hence, we propose to use a look-ahead search procedure using the neural network as an oracle to compensate for this imperfection and improve the overall robustness of the compression algorithm. One benefit of the search procedure is potentially better performance without additional knowledge. Clearly, the search will induce extra computational costs. How to tradeoff between the cost of computation and accuracy requires empirical studies.

In our project, we chose the CHINOOK checkers endgame databases as our research subject thanks to its accessibility and detailed documentation. We seek to answer several research questions through experiments around the central theme of compressing checkers' endgame databases using deep neural networks. The foremost one is if a neural network can discover patterns and learn the mapping from positions to outcomes in the first place. If the network can improve on the training set, is it merely memorizing (as is done in the current CHINOOK endgame compressed databases representation), or does it understand some underlying patterns? Can the trained model generalize to unseen positions? Confronted with the motivations, we came up with a list of ideas worth exploring:

1. We train the neural network with a small subset of the whole database to make it suitable for our limited computational resources and test its generalization ability.

2. We would like to see if the neural network trained end-to-end can achieve even better compression than the hand-designed one used in CHINOOK.

3. If possible, we want to identify patterns important for compression and might also be meaningful as human knowledge.

4. We are curious if a look-ahead search can improve the overall performance at the cost of extra computation compared to directly querying the same neural network.

1.2 Problem Setting

We formulate our project as a supervised learning problem. Nevertheless, there are some distinctive properties that differentiate our problem from ordinary supervised learning tasks:

1. Our dataset is noise-free because it is an endgame database with proven results associated with each board position. Each position-outcome pair appears only once in the dataset. In most "real-world" scenarios, the collected data is usually noisy, and one should be careful not to capture the noise in a model.

2. We have all the data in the "population" because the positions in the databases are exhaustive. In the real world, it's virtually hopeless to collect all the data on the population because the world is prohibitively vast and complex, and new data may appear continually. As a result, the training data only represents a small portion of the whole population for those tasks, and one must rely on models to get the best out of training data to infer the unseen data as well as possible.

3. Board positions as raw inputs are sensitive to local changes. When dealing with images in computer vision problems, such as in image classification, the label would likely remain the same even if a few pixels get altered. For example, when training a classifier for dogs and cats, the labels will likely stay the same if the pixels corresponding to the ears of the animals get corrupted. However, that's not the case for checkers' board positions. Even a tiny shift in one of the pieces may result in a completely different outcome. Two similar boards differing only by one piece could be a win and a loss, respectively.

1.3 Contributions

In this thesis project, we present a deep-learning pipeline for the CHINOOK endgame databases and experimental results, including:

1. A program that reads and processes the database records and transforms them into NumPy arrays[16] on the disk. NumPy arrays are efficiently convertible to either PyTorch tensors[47] or JAX NumPy arrays[6].
2. Python implementations of the neural network in both PyTorch[47] and JAX[6].
3. Python functions that generate figures to visualize the meta-information of the datasets such as the class distributions.
4. A training procedure that trains the neural network with different choices of learning rates, batch sizes, optimizers and loss functions.
5. A Python function that performs a small search using the neural network. Adding search allows the system to reduce the error rate on unseen data.
6. A TensorBoard[1] integration that visualizes and monitors the training procedure.
7. An evaluation program that evaluates the performance of the trained model and search, generates informative tables and visualizes misclassified positions.
8. Experiments that demonstrate the competence of deep neural networks to compress the databases, generalize to unseen positions, and how search can help reduce errors.

The paper *Deep Dive on Checkers Endgame Data*[66], which contains many of the results from this thesis, has been accepted to the IEEE Conference on Games 2023.

2 Background

2.1 Computer Game Playing and Solving

In the field of artificial intelligence research, computer game playing has a long and rich history. Games usually provide a more well-defined, controllable and simple environment than most real-world problems. They are popular benchmarks for AI algorithms. However, the knowledge and insights obtained from computer game playing are by no means confined to the games themselves. They are highly transferable to other domains and often have real-world implications[10, 27].

In computer game playing, the research objective often involves developing more powerful, efficient, robust and general algorithms. Based on these goals, a game-playing algorithm is typically evaluated on a set of criteria, for example, the winning rate of the algorithm against existing players (computer or human), the computational resource the algorithm consumes, exploitable weaknesses of the algorithm, and the applicability of the algorithm to other games.

Computer game solving is relevant to game playing but is a different concept. Solving a game means finding the minimax value of the game – the game outcome if all players play perfectly. There are three levels of game solving – ultra-weakly solved, weakly solved and strongly solved games. For ultra-weakly solved games, the minimax result of the game is known but no optimal strategy. For weakly solved games, the minimax outcome and optimal play starting from the beginning of the game are known. Strongly solved games have the minimax value and an optimal strategy for any valid position of the game. For example, Awari was strongly solved by searching the entire state space and building a database[49, 50]. Hex was ultra-weakly solved by combining mathematically proven arguments that the first player cannot lose and drawing is impossible[17]. Afterwards, Hex was weakly solved for 10×10 and smaller boards[3]. Checkers, our subject of study, was weakly solved using search and endgame databases[56].

2.1.1 Evaluation

Evaluation is the process of determining the chance of winning the current game state for each player. For most game-playing algorithms, the evaluation function is a crucial part of the program because it quantifies the "goodness" of the current and future board positions, directly influencing the player's decision. The evaluation can either return a game-theoretic value or a heuristic value. A game theoretic value guarantees the minimax game result if all players play perfectly from the current state to the end of the game. On the other hand, when the game-theoretic value of a position is unknown (as is usually the case for most positions in most games), some form of application-specific knowledge – heuristics– is combined to form an assessment. The knowledge could be based on human experience, statistics, or computer-generated patterns. In addition, the game-theoretic values and heuristics can be computed on-demand or cached in tables. The computation of the game-theoretic value of a position is often unrealistic for early game positions due to the astronomically large search space. Thus, heuristic evaluation is the only choice in most cases. Nevertheless, one cannot obtain any guarantee of the game result from using heuristic evaluation functions.

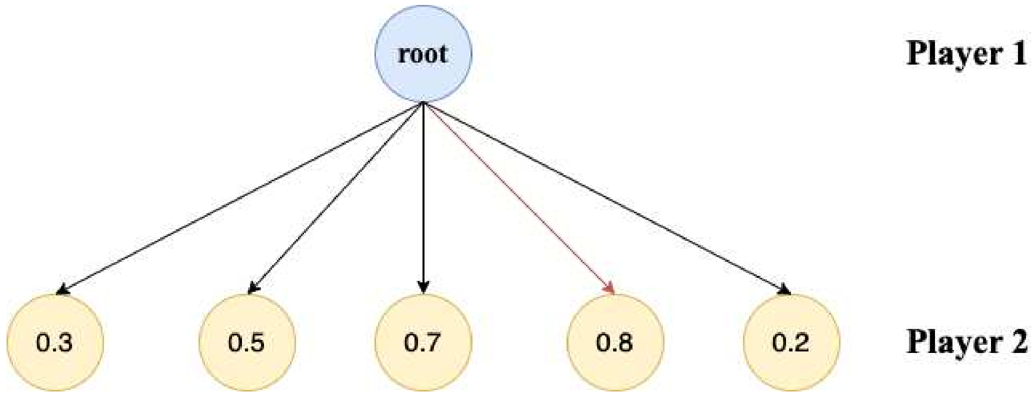


Figure 1: Example of One-Ply Search. The numerical values represent evaluation scores from player 1's perspective. The fourth child has the highest score of 0.8 and thus should be selected by player 1.

2.1.2 Search

Search is a class of algorithms widely used in game playing and solving. It is analogous to how humans plan in a game scenario by looking ahead several steps. A ply refers to one move taken by one player. A state refers to a unique and unambiguous configuration of the game. The identical game state always has the same board position, but the reverse is not necessarily true – the same board position may resolve to different outcomes depending on the game history (sequence of states/actions leading to the current position). Most search algorithms in the context of game playing and solving build a game tree – a tree structure rooted in the current state. One node of the tree represents one state of the game. Each node has branches (corresponding to the legal moves in the position) that connect to the states one ply away. The results are propagated back to the root node after evaluating the leaf nodes. The program then analyzes the information gathered at the root to decide its next move. Figure 1 illustrates a one-ply search. The naïve minimax search algorithm recursively expands the search tree and evaluates the children at each node. The program chooses the move that leads to the maximum value. Examples of more advanced search algorithms are alpha-beta search[29] and proof number search[2]. The former algorithm prunes branches that can be proven to be irrelevant in deciding on the best move. The latter uses properties of the search tree to decide which node to expand in every iteration. These algorithms were crucial tools in solving the game of checkers.

2.2 Checkers

2.2.1 Rules

Setup

- The checkerboard is an 8×8 square board with vertically and horizontally alternating dark and light squares. Only the dark squares are used.
- The game pieces used in checkers are disk-shaped in either black or white colour.

- There are two types of pieces in checkers – *checker* and *king*. Both players begin with *checker* pieces only.
- At the beginning of the game, each player places 12 *checker* pieces on the dark squares of the first three rows at the bottom from their perspective. Figure 2 shows a checkerboard at the beginning of the game.
- The two players (referred to as Black and White) alternate taking turns.
- Black opens the game.

Move

In the player's turn, they can move one of their *checker* pieces diagonally forward one square or one of their *king* pieces diagonally forward or backward one square, provided they meet the following criteria:

- The destination square is empty. In other words, there are no other pieces blocking the target piece.
- There are no capturing moves for the player.

Capture

- To capture an opponent piece, the player moves one of their pieces two squares diagonally in the same direction, leaping over the captured piece, and landing on an empty square. The captured opponent piece is removed from the board.
- If, after one capture, the piece lands on a position where it can immediately capture another one, then it should continue capturing until it can no longer do it.
- A *checker*, like its move rule, can only capture forward. A *king* can capture forward and backward. This rule applies to both single and consecutive captures.
- The player must capture if it is possible. This is called the forced capture rule.

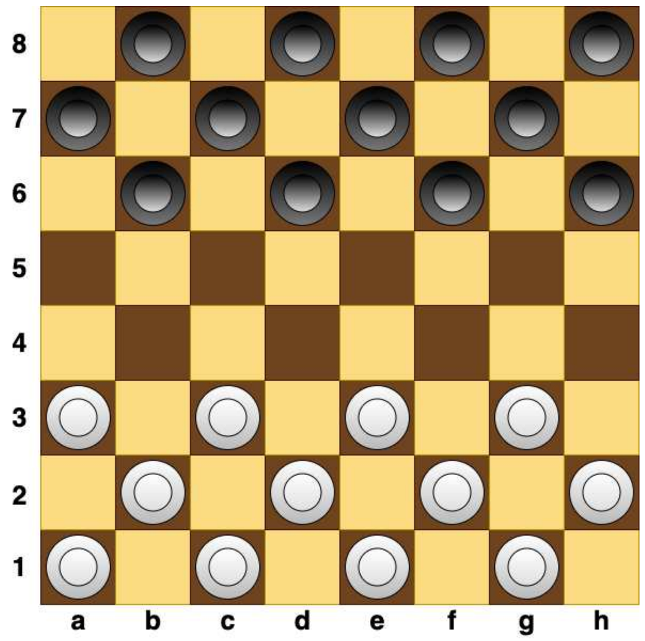


Figure 2: Checkerboard at the Beginning of the Game.

- If the player has multiple capture strategies in one turn, they can choose to execute one of them freely. This does not violate the forced capture rule.
- Figure 3 illustrates a capture scenario for White.

Promotion

- A *checker* is promoted to *king* if it reaches the furthest row from the player's point of view.
- A *king* can then move and capture forward and backward diagonally.
- Once a *checker* is promoted, the turn terminates automatically for the player, even if they can immediately capture a piece using the newly promoted *king*.

Game ending

- The game ends when one player cannot make any move in their turn or all of their pieces have been captured. In these cases, the aforementioned player loses the game, and their opponent wins.

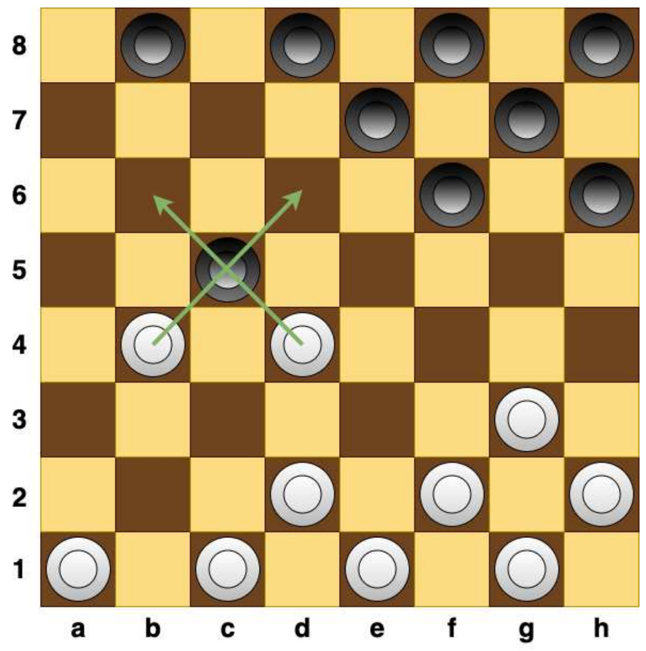


Figure 3: Checkers Capture. If we label the columns a-h and rows 1-8, then White can capture in two ways – move the piece on $(b,4)$ to $(d,6)$ or the piece on $(d,4)$ to $(b,6)$. Either way, the black piece on $(c,5)$ will be captured.

- There are no standardized rules for ending a game with a drawn result. In practice, the following two rules are often adopted: The same position repeats three times; There are no captures for either side in 40 moves.

2.2.2 History of Checkers AI

Arthur Samuel's Checkers Program

Arthur Samuel's checkers-playing program [53, 54] is recognized as the first program that applied heuristic search and machine learning principles to play checkers. Its victory against a skilled amateur player was a monumental moment for artificial intelligence and was widely publicized.

Samuel chose checkers over chess due to its simplicity of rules, which allowed him to put more emphasis on learning techniques. The program applied what we call the heuristic search technique today – the computer looks ahead a few moves and evaluates the resulting board positions to decide what action to take at the present board. He used a linear function as the evaluation function by combining the values of a list of carefully selected features. When performing the search, the program builds a tree and carries out a minimax procedure. Samuel tried several strategies to improve the strength of the program based on this framework. He explored rote learning and learning by generalization in much greater detail in his early report. Rote learning involves caching the more frequent positions with their respective values to enable searching to greater depth effectively during the real game. Learning by generalization involves a self-play procedure that modifies the parameters of the linear function to improve its evaluation function. Samuel claimed that rote learning is especially good at playing opening (the first moves of the game) and end (last moves of the game) games, whereas learning by generalization is good at playing middle games (moves that bridge the end of the opening to the start of the endgame). Besides these two techniques, he also utilized what we consider discounting today to give preference to the traces of play that win quicker in an advantageous circumstance or lose slower otherwise. Samuel's machine learning work was a precursor of modern-day reinforcement learning[65].

In a later version of the work[54], Samuel made various improvements to the program, including an alpha-beta pruning procedure[29], a book learning procedure analogous to what we know as supervised learning today, and a signature table to replace the linear function. The improved version performed much better than the previous version, though according to the author, there were still some defects to address.

A Brief History of CHINOOK

The development of CHINOOK began in 1989 as a project to better understand heuristic search algorithms[57]. The team soon realized they could make CHINOOK strong enough to compete for the world championship. CHINOOK was allowed to play in the 1990 U.S. championship and earned the right to play for the world championship. In 1992, CHINOOK played a World Checkers Championship match against the world champion Marion Tinsley and lost the match with two wins for CHINOOK, four wins (one by default) for Tinsley and 33 draws[58]. After the match, the two sides agreed on a rematch in 1994. The team then worked on improving the four major components of CHINOOK – search, evaluation function, endgame databases and opening book. Specifically,

- an upgrade of hardware allowed a deeper search for CHINOOK.
- A tactic table helped handle specific error-prone situations.
- A much more comprehensive and precise opening book strengthened the opening plays.
- They also computed more positions and fixed errors in the endgame databases.

In 1994, CHINOOK earned the world championship title by forfeit against Tinsley after six draws. After the 1994 rematch, CHINOOK played against several human champions and checkers AIs. By 1996, it was clear that CHINOOK was better than any human or AI player at the time.

The next ambition of the CHINOOK team was to solve the game of checkers with search and endgame databases. They kept growing the number of positions in the databases using a network of workstations non-stop until the ten-piece database was complete (all positions with ten or fewer pieces on the board)[31, 55]. In 2007, they finally solved the game of checkers with the conclusion that checkers is a drawn game if both players play without making a mistake[56].

2.2.3 Endgame Databases

John Nunn's Effort to Distill Knowledge from Chess Endgame Databases

Given the massive number of positions in endgame tablebases, extracting knowledge from this data is a task suitable for computers. For example, there are 25 billion chess positions with five pieces on the board (two of which are kings). Although viewed as "simple" positions, many give rise to complex sequences of play to preserve the game-theoretic result. A few brave humans have manually tried to turn this data into human-understandable heuristics. In the 1990s, chess grandmaster John Nunn devoted several years to analyzing these chess endgames. The result was a trilogy of books, each 320 pages long, and each containing a compendium of rules, generalizations, and exceptions designed to help humans understand these classes of positions[42, 43, 44].

Here are some examples of Nunn's data mining from his first book, *Secrets of Rook Endings*[44]. This analysis covers the case of king, rook, and pawn versus king and rook.

- White has a b-pawn on the 7th rank with White's king in front of the pawn. "The situation with the pawn on the 7th rank and the king in front of it is extremely common... In the case of the b-pawn White always wins, except for a few exceptional positions where he loses the pawn immediately." ([44], page 109)

- White has a c-pawn on the 5th rank, with Black's king in front of the pawn. "In general this is a draw, but White can sometimes win with a very favorable initial position." ([44], page 208)

The books introduce hundreds of new heuristics for humans to learn and master. While this dramatic reduction of data complexity has been acclaimed as an important contribution to human knowledge, Nunn's super-human patience and dedication to the task at hand cannot be repeated for more complex endgames.

Building CHINOOK*'s Endgame Databases*

Unlike John Nunn's effort to compile human endgame knowledge in a book, the CHINOOK endgame databases were computed exhaustively by a network of computers over 15 years[31, 55]. The number of pieces on the board strictly decreases in a game of checkers. Taking advantage of this nature, the CHINOOK team applied retrograde analysis to compute the endgame databases more efficiently[31]. That is, starting from only one piece on the board, the program builds the two-piece database relying on the one-piece database, and so on. When computing the database for a specific piece number, the algorithm will resolve capture moves first because they will guarantee to end in positions with fewer pieces already present in the previous databases. After that, the algorithm will iteratively resolve non-capture positions until complete. Using this methodology, the CHINOOK team computed the endgame databases for all positions with 2-10 pieces on the board, summarized in Table 1.

Number of Pieces	**Total Number of Positions**
1	120
2	6,972
3	261,224
4	7,092,774
5	148,688,232
6	2,503,611,964
7	34,779,531,480
8	406,309,208,481
9	4,048,627,642,976
10	34,778,882,769,216
total	39,271,258,813,439

Table 1: A Summary of CHINOOK's Endgame Databases by Number of Pieces on the Board.

Uses of Endgame Databases in Other Games

Awari is a game strongly solved by building a database that stores the outcome for every possible position of the game[49]. The state space of Awari has over 889 billion positions (10^{12}). The team that solved it designed and executed retrograde analysis on a parallel computer with 144 processors as efficiently as possible – a significant engineering effort. After roughly 51 hours of computation, they built the entire database and announced that Awari is a drawn game if both players play perfectly.

Nine Men's Morris is a game weakly solved with the help of an endgame database[11]. The game consists of three phases – opening, midgame and endgame; Each has its own set of rules. The program that solved it has two parts – an 18-ply search for the opening phase and a database for the midgame and the endgame. Taking advantage of game-specific knowledge such as symmetries, the team reduced the total number of positions needed for storage roughly 16-fold. They built a database of approximately 10^{10} positions using retrograde analysis. Afterwards, they used alpha-beta pruning[29] to search for the value of the root position. Through about three weeks of computation on a desktop computer, they solved Nine Men's Morris and concluded that the game is a draw at the initial position with optimal play.

2.3 Deep Learning

Designing an endgame database that is storage and lookup efficient is never a trivial task. It often requires designers to leverage application-specific knowledge and invent special tricks to make it work. The rise of deep learning[33] provides us with a promising tool – deep artificial neural networks – to discover an alternative way to represent endgame knowledge in place of the complete position-value table. Since each position only maps to one value, the database effectively defines a function. We can theoretically approximate the endgame database to arbitrary precision using deep neural networks because they are universal function approximators[23]. In addition, deep neural networks are extraordinarily good at generalization, which implies that parameters needed to represent the neural network may require much less memory than the table format – compressing the database. In terms of access speed, modern deep-learning libraries and hardware are good at parallelizing the neural network inference – speeding up batched access significantly. Therefore, deep neural networks are competitive candidates to compress endgame databases while achieving remarkable access speed.

Inspired by the mechanism of the human brain, the most fundamental unit of processing in a neural network is called a neuron. A valid neuron is composed of an input connection, an activation function, and an output connection. We often refer to the array of neurons at the same depth as a layer. The input connection receives and aggregates signal from the previous layer. An activation function is a nonlinear function applied to the aggregation of the input signal. Nonlinearity is necessary for neural networks, or the entire neural network is equivalent to a linear model. Popular activation functions include sigmoid, tanh, rectified linear unit (ReLU), etc. The output connection propagates the activation to the subsequent layer the neuron connects to. In a feedforward neural network, the connections are loop-free, and the processing happens sequentially. Therefore, except for the input layer that receives the raw input, all input connections receive the activation from the previous layer. Likewise, all output connections except for the output layer send the activations to the next layer. Each connection has a real value called weight that gets multiplied by the input activation during forward propagation. Besides the activation, the neuron may also receive a real constant called bias to shift the input signal by some amount. When we say training a neural network, most of the time we are implying changing the values of the weights and biases with respect to some objective.

2.3.1 Multi-layer Perceptron

A multi-layer perceptron (MLP)[51] is arguably the most basic form of deep neural network – the entire model is composed of dense layers. A dense or fully-connected layer has each neuron connected to the previous layer pairwise. In other words, we can represent the weights of a dense layer as a real-valued matrix with one dimension aligning with the number of neurons in the previous layer and the other dimension equivalent to that of the current layer. If we denote the input to the i-th layer as $x^{[i]}$, the weights and biases of the i-th layer as $W^{[i]}, b^{[i]}$, and the activation function as $f^{[i]}$, then the output $o^{[i]}$ of the i-th layer is

$$o^{[i]} = f^{[i]}(W^{[i]}x^{[i]} + b^{[i]})$$

2.3.2 Convolutional Neural Network

Convolutional neural networks (CNN)[32] trained via back propagation[34] have revolutionized computer vision and game-playing. A particular structure in a CNN is the convolutional layer. A convolutional layer has a number of parameterized windows called filters (kernels). Each filter convolves with the input feature during forward propagation by element-wise multiplying with the current region, summing the products, and then shifting along an axis by some positions to the next area. Each sum defines one entry of the output feature map corresponding to that filter. The shifted amount is called a stride, a hyperparameter that controls the resolution of the generated feature maps. Figure 4 illustrates a convolution.

The convolutional layers are the main reason behind the extraordinary performance of CNNs on unstructured data such as images. The first advantage of the convolutional layer is weight sharing – the same filter is applied to the entire feature map. The outcome is a network with much fewer parameters than its MLP counterpart that assigns a weight to each pixel. Another strength of the convolutional layer is detecting local features because the filters look at the input one region at a time, capturing those within their scope. In contrast, the input to an MLP is linear, where the spatial relations cannot be retained after linearization. To further reduce the number of parameters to learn, one can apply pooling[12] after the activation function of convolutional layers. This reduces the size of the intermediate feature maps, saving computation in the convolutional layers and cutting down the size of the vector fed to the classifier.

One can roughly divide a CNN into a feature extraction body and a classifier. The feature extraction body consists of blocks of convolutional layers followed by an activation function followed by perhaps pooling. The classifier can be viewed as an MLP.

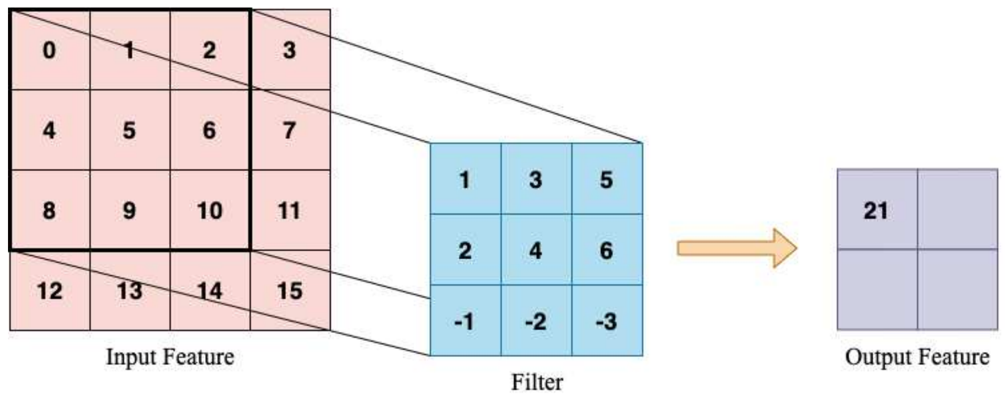


Figure 4: Convolution on a 4×4 Feature Map with a 3×3 Filter. This example shows the outcome of the convolution on the top left 3×3 region.

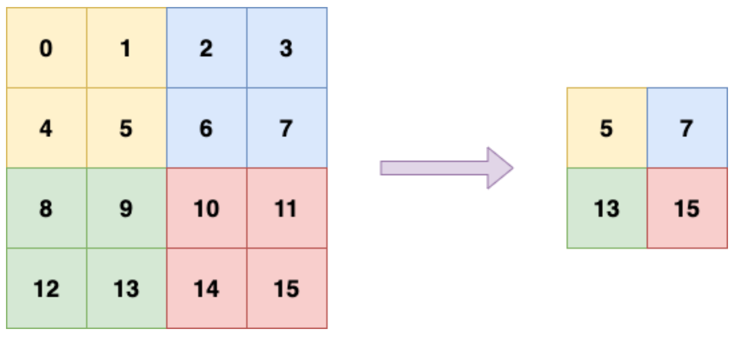


Figure 5: Pooling. This example demonstrates 2×2 maximum pooling on a 4×4 feature map.

2.3.3 Residual Neural Network

Even though CNNs have achieved outstanding performance, they suffer from accuracy saturation – when we make the network deeper and deeper using more convolutional layers, the final accuracy increases to a certain point and then drops[18, 19]. This is intriguing because deeper CNNs could perform at least as well as their shallow counterparts by learning identity mappings $f(x) = x$ in the earlier layers. He *et al.* [19] hypothesized that identity mappings are hard to learn with nonlinear activation functions. Another difficulty when training a deep CNN is the vanishing gradient problem[13] – the gradients become smaller and smaller as they backpropagate to the shallower layers and eventually become negligible. This phenomenon makes training the first few layers unacceptably inefficient, hindering further improvements.

Deep residual neural networks (ResNet)[19] considerably improved upon plain CNN with residual connections that have negligible additional computational costs. Rather than learning the mapping $\mathscr{H}(x)$ directly, the residual layer fits the mapping $\mathscr{F}(x) = \mathscr{H}(x) - x$ such that the original mapping can be recovered by $\mathscr{H}(x) = \mathscr{F}(x) + x$. Figure 6 illustrates an example of a residual connection with two weight layers. There are two advantages of learning the residuals instead of the original mappings. Firstly, residuals are easier to optimize – the authors hypothesized that driving the residual to zero is easier than learning the identity mappings. Secondly, the residual connections help mitigate the vanishing gradient problem by allowing the gradients to travel faster to the shallower layers. As a result, one can build a residual neural network tens and hundreds of blocks deep without seeing a degradation in performance.

2.4 Learn to Play Games from Humans and Self-Play

In this section, we discuss three popular types of learning in games:

- opening book learning
- supervised learning from human data
- learning via self-play

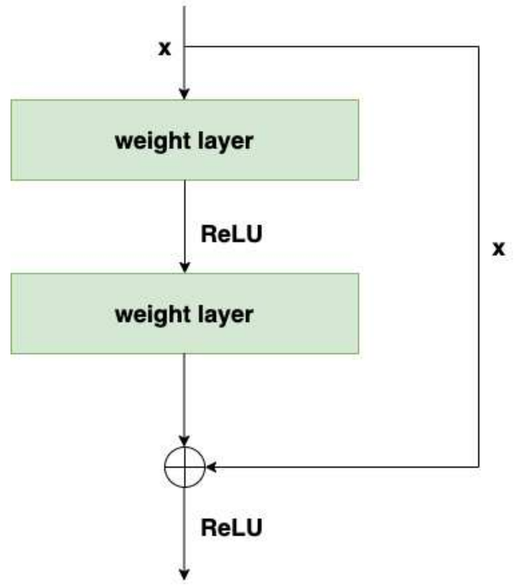


Figure 6: A Residual Block with Two Weight Layers and ReLU Activation Functions.

In games, the "book" is usually a strategy for the opening phase of the game. It is handled differently than the rest of the game because there are human sources for the best sequences of moves. Thus, rather than having the computer decide on the best moves (which will introduce errors), computer “books” are usually built from information contained in human books on the openings.

Book learning has been used for a long time in game AI because it quickly equips the program with sophisticated knowledge distilled from top players. The famous chess-playing engine DEEP BLUE[7] used an opening book of about 4,000 positions hand-crafted by chess Grandmasters. These openings emphasize the positions that DEEP BLUE played well. In addition, it also had an extended book built from a 700,000-position database. CHINOOK[57] also used an opening book to overcome the weaknesses in its opening play present in an earlier version. The book is a database of lines of openings acquired from the published literature.

Learning from self-play implies generating the training data by playing against a copy of the program itself. The program improves over time, and so does the copy. By learning from its own experiences, the program eventually masters the game. The first published version of ALPHAGO defeated a human champion in an even game of

Go[62]. The ALPHAGO team followed the path taken by past researchers and started off by incorporating human knowledge into their system via supervised learning from human games. However, the crucial part of the algorithm was learning from self-play. The program improved by playing against itself after initializing its policy network with expert plays (trained on expert game data). Later, ALPHAZERO[63] mastered a number of games from scratch, purely via self-play. Removing the expert knowledge and letting the program learn everything on its own turned out to be even better. The final evolution of self-play systems was MUZERO, where the program learned to play without even being given the rules[61].

2.5 Compress CHINOOK Endgame Databases

Because of the need for real-time decompression, the CHINOOK databases used a custom-designed compression algorithm[31, 55]. The results are impressive – 40 trillion positions (10^{13}) are compressed into 256 billion (10^{11}) bytes: a staggering 156 positions are encoded in each byte or roughly 20 positions per bit! Needless to say, this level of compression will be hard to beat. In our work, we intend to achieve compression by approximating the endgame databases of checkers using end-to-end learning.

3 Formal Concepts

In this section, we formalize concepts related to the endgame databases, neural networks, one-ply search, and evaluation metrics needed for our task.

3.1 CHINOOK Endgame Databases

3.1.1 Content of the Databases

The databases store the perfect play outcomes of checkers positions with respect to Black to move. The outcome is one of {draw, win, loss}, represented numerically. Thus, we define a function o that maps the outcome for position x to an integer:

$$o(x) = \begin{cases} 0, & \text{if } x \text{ is a draw} \\ 1, & \text{if } x \text{ is a win} \\ 2, & \text{if } x \text{ is a loss} \end{cases} \tag{1}$$

For a position x, the program that returns $o(x)$ also computes a property of x called the flag that takes a value in {none, capture, threatened capture}. This application-dependent property is used to improve database compression.

- Capture means that Black has at least one capturing move in position x. Due to the checkers forced capture rule, if one or more capture moves are possible, then one of them must be chosen.

- Threatened capture indicates that there are no Black capture moves in x, but if it were White to move, White could make a capture move.

- The none flag implies the position has neither a capture nor a threatened capture. These positions are said to be quiescent – their evaluation is stable and unlikely to change drastically after a move.

The flags are also represented numerically. We use function f to map each flag to an integer:

$$f(x) = \begin{cases} 0, & \text{if } x \text{ is none} \\ 1, & \text{if } x \text{ is capture} \\ 2, & \text{if } x \text{ is threatened capture} \end{cases} \tag{2}$$

To query a position for White, one needs to rotate the board 180 degrees and switch the colour of the pieces on the board. In addition, since checkers is a zero-sum game, a win for Black means a loss for White, and vice-versa. The same goes for the flag – capture for one means threatened capture for the other.

3.1.2 Organization of the Databases

The CHINOOK project computed the game result for trillions of positions. Given the technology available when it started in 1989, it was infeasible to perform such an enormous computation in one run. Hence, the positions were divided into smaller subsets and the results combined to produce the complete database.

Figure 7 shows a sample position *p* that will be used in this section. The CHINOOK endgame databases were organized in increasing granularity:

- **by number of pieces**

  Each partition includes all positions that have the same number of pieces on the board regardless of the colour or the type of the pieces. *p* is in the 5-piece partition.

- **by number of pieces for each side**

  This partition paradigm splits the previous one by considering the number of pieces for Black and White on the board, organized as [black piece count][white piece count]. *p* is in the partition encoded by *23*.

- **by number of piece types**

  This partition paradigm further splits the previous one by distinguishing the piece types for each side, organized as [black *king* count][white *king* count][black *checker* count][white *checker* count]. *p* is in the partition encoded by *1211*.

- **by leading *checker* rank**

  This partition paradigm is the finest of the CHINOOK endgame databases. It splits the previous one by the rank of the most advanced *checker* (the *checker* closest to becoming a *king*) for each side, organized as [black *king* count][white *king* count][black *checker* count][white *checker* count].[leading black *checker* rank][leading white *checker* rank]. The rank is defined as the row index of the piece from its player's point of view. The rank ranges from 0 to 6 for a *checker* piece since it gets promoted automatically to a *king* when reaching rank 7. *p* is in the partition encoded by *1211.23*.

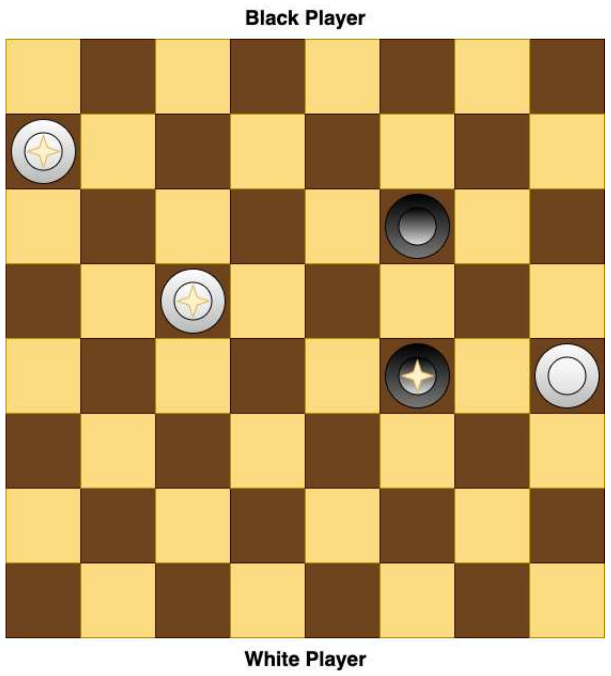


Figure 7: A Sample Position for Partition Demonstration. This position can be found in the 5-piece partition by number of pieces, in the 23 partition by number of pieces for each side, in the 1211 partition by number of piece types, or in the 1211.23 partition by leading *checker* rank.

3.2 Neural Network

3.2.1 Definition

We can define our neural network mathematically as a function g parameterized by a real vector θ that maps from the input space $\mathscr{X}$ to the output space $\mathscr{Z}$:

$$g_\theta : \mathscr{X} \mapsto \mathscr{Z} \tag{3}$$

For this specific problem, we let the output of the network be the logits such that $\mathscr{Z} = \mathbb{R}^3$. Each dimension corresponds to one of the three possible outcomes of the game. The logit, also named the log odds, is an unnormalized score used by the *softmax* function[15] to generate a probability distribution for classification. The *softmax* function σ is defined as:

$$\sigma(z)[i] \doteq \frac{\exp^{z[i]}}{\sum_j \exp^{z[j]}} \tag{4}$$

Furthermore, define a function h that maps board positions in the databases to the input space $\mathscr{X}$ of the neural network. Similarly, define a function q that maps the numerical outcome to the output space $\mathscr{Z}$. In our case, q is a one-hot encoding function $q : x \in \{0,1,2\} \mapsto \hat{x} \in \mathscr{Z}, \hat{x}[i] = \mathbb{I}(x = i)$, where $\mathbb{I}$ is the indicator function.

Given a board position x, the probabilities of the game outcome predicted by the neural network are

$$P(i|x) = \sigma(g_\theta(h(x)))[i], \forall i \in \{0,1,2\} \tag{5}$$

3.2.2 Training

We formulate our problem as supervised learning – training our neural network on data consisting of examples and their class labels.

Dataset

Our dataset is a collection of position-outcome pairs. A dataset with N samples is $\mathscr{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$, where x_i is a position and y_i is the associated outcome. We split the entire dataset into a training, a validation and a test set. The learning algorithm only uses the training set to improve its performance. The validation set serves as a benchmark for how well the model performs on unseen data. After the training is complete, we report the final performance of the model on the test set.

Loss Function

A loss function $\mathscr{L}$ is a mapping $\mathscr{L} : \mathscr{Z} \times \mathscr{Z} \mapsto \mathbb{R}$. Given a board position x and its outcome y, we compute the loss l as

$$l = \mathscr{L}(g_\theta(h(x)), q(o(y))) \tag{6}$$

The loss function mainly measures how well our neural network models a single data instance. It should also be differentiable with respect to the parameters θ of the neural network.

Cost Function

The cost function $\mathscr{C}$ is the empirical mean of the loss on a collection of data, serving as an objective to optimize. Given a dataset $\mathscr{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$, the cost c is

$$c = \mathscr{C}(\mathscr{D}, g_\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathscr{L}(g_\theta(h(x_i)), q(o(y_i))) \tag{7}$$

Note that $\mathscr{C}$ is also differentiable with respect to θ. In general, our objective is to find θ^* such that

$$\theta^* = \arg\min_\theta \mathscr{C}(\mathscr{D}, g_\theta) \tag{8}$$

Optimizer

An optimizer *opt* takes θ and the gradient of $\mathscr{C}$ with respect to θ and outputs the updated θ':

$$\theta' = opt(\theta, \nabla_\theta \mathscr{C}(\mathscr{D}, g_\theta)) \tag{9}$$

with the goal that $\mathscr{C}(\mathscr{D}, g_{\theta'}) < \mathscr{C}(\mathscr{D}, g_\theta)$.

Training Loop

An epoch in the context of deep learning is defined as one complete pass through the training set. The most basic training loop is shown in Algorithm 1.

Algorithm 1 A Basic Training Loop

```
Require: $numEpoch, \theta, g, \mathscr{D}, \mathscr{C}, opt$
Ensure: $numEpoch > 0$
  $epoch \leftarrow 1$
  while $epoch \leq numEpoch$ do
      $grad \leftarrow \nabla_\theta \mathscr{C}(\mathscr{D}, g_\theta)$
      $\theta \leftarrow opt(\theta, grad)$
      $epoch \leftarrow epoch + 1$
  end while
  return $\theta$
```

3.2.3 Inference

Given the parameters θ of the neural network and a board position x, the prediction of the outcome in the numerical representation is

$$\hat{y} = \arg\max_i g_\theta(h(x))[i], i \in \{0, 1, 2\} \tag{10}$$

We don't need to compute the probability distribution of the outcomes given the logit vector z thanks to the fact that σ is monotonic such that $\arg\max_i z[i] = \arg\max_i \sigma(z)[i]$.

3.3 Search

To employ a one-ply search using the neural network, assume that given a position x, procedure *getChildren* returns the list of all possible subsequent positions $\{x'_1, x'_2, \ldots, x'_N\}$ after making a move in x. These positions are children of x in the search tree. Here we assume that they have already been converted to the standard query form, corresponding to Black's turn to move. If the returned list is empty ($N = 0$), then Black cannot move from the current position – White wins. Otherwise, we infer the outcomes $\{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_N\}$ of each child using the neural network. If any child is losing, then the prediction for the parent position x is a win. If no child is losing but at least one of them is a draw, then the prediction for x is also a draw. We predict x to be a loss for Black if all children are winning. Algorithm 2 provides the pseudocode for this search procedure.

Figure 8 displays an example of how the one-ply search works: The neural network infers that each child of the root position leads to a draw for Black if both players make no mistakes; The one-ply search returns a draw for the root position because there is no winning move and at least one drawing move for Black.

Algorithm 2 One-ply Search

```
Require: $x, \theta, g, h, predict, getChildren$          ▷ draw = 0, win = 1, loss = 2
  $children \leftarrow getChildren(x)$
  if $children$ is empty then
      return 2
  else
      $logits \leftarrow g_\theta(h(children))$
      $outcomes \leftarrow predict(logits)$
      if 2 is in $outcomes$ then
          return 1
      else if 0 is in $outcomes$ then
          return 0
      else
          return 2
      end if
  end if
```

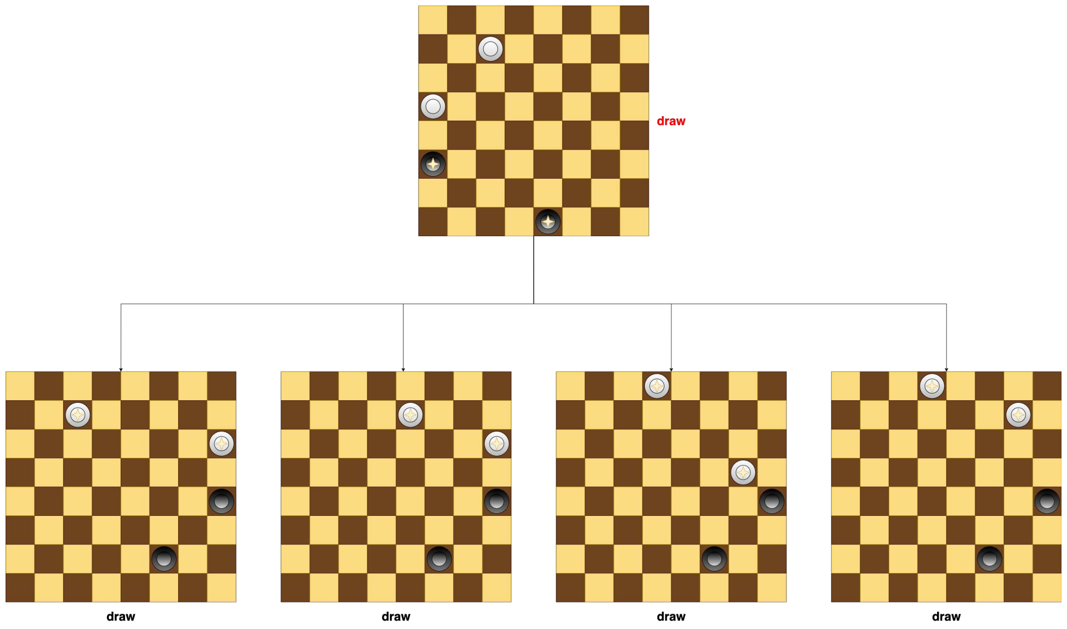


Figure 8: Example of One-Ply Search. The neural network predicts each child of the root position to be a draw for Black. Thus, the one-ply search predicts that the root position is also a draw for Black. Note that internally, all positions are represented as Black to move. In the diagram, the root position is shown from Black's perspective. After a move is made, it is White to move, but the position is converted to look like it is Black to move (the four positions one ply from the root).

3.4 Evaluation Metrics

In this section, we introduce and formalize the evaluation metrics that measure the performance of the neural network and the search algorithm from multiple perspectives.

3.4.1 Confusion Matrix

Given a collection of prediction-label pairs $\{(\hat{y}_1, y_1), (\hat{y}_2, y_2), \ldots, (\hat{y}_N, y_N)\}$, we can build a matrix $M \in {\mathbb{Z}^+}^{\kappa \times \kappa}$, where κ is the cardinality of the set of possible outcomes ($\kappa = 3$ in our case), such that M_{ij} is the number of instances of class i predicted to be class j. The matrix M is called a confusion matrix[48]. It provides an informative and intuitive summary of the behaviour of the prediction model on the dataset. The main diagonal entries of M count the correctly classified instances, while the off-diagonal entries summarize the incorrectly predicted ones.

Figure 9 is a mock confusion matrix for our problem. Summing up every cell, there are 100 positions in the test set of this example. The diagonal shows there are 45, 21, and 23 correctly classified positions for Draw, Win and Loss, respectively. There are a total of 11 misclassified positions in the off-diagonal entries. For example, $M_{1,2}$ indicates two drawing positions are predicted as wins.

3.4.2 Accuracy

The accuracy is defined as the number of correctly classified samples divided by the total number of samples in the dataset. Given a collection of prediction-label pairs $\{(\hat{y}_1, y_1), (\hat{y}_2, y_2), \ldots, (\hat{y}_N, y_N)\}$,

$$accuracy = \frac{\sum_{i=1}^{N} \mathbb{I}(\hat{y}_i = y_i)}{N} \tag{11}$$

We can also compute the accuracy based on the confusion matrix M:

$$accuracy = \frac{\sum_{i=1}^{\kappa} M_{i,i}}{\sum_{i=1}^{\kappa} \sum_{j=1}^{\kappa} M_{i,j}} \tag{12}$$

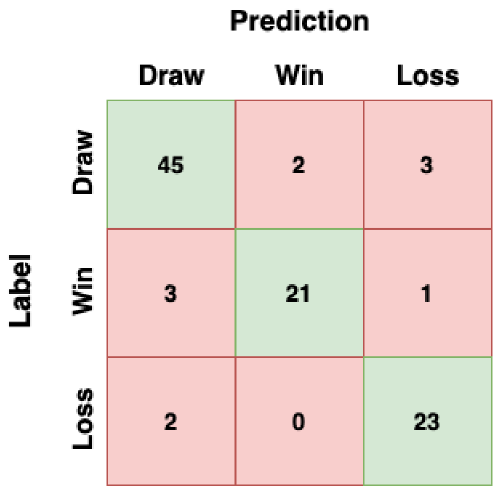


Figure 9: Confusion Matrix for Predicting the Outcomes of the Checkers Positions. The counts of the correctly predicted ones are in the main diagonal. The off-diagonal entries count the misclassified ones.

The confusion matrix in Figure 9 indicates an accuracy of 89%: 89 positions in the diagonal entries divided by a total of 100.

3.4.3 Precision

The precision for a class is defined as the number of correctly classified samples in that class divided by the total number of samples predicted in that class. Given a collection of prediction-label pairs $\{(\hat{y}_1, y_1), (\hat{y}_2, y_2), \ldots, (\hat{y}_N, y_N)\}$,

$$precision[i] = \frac{\sum_{j=1}^{N} \mathbb{I}(y_j = i \wedge \hat{y}_j = y_j)}{\sum_{j=1}^{N} \mathbb{I}(\hat{y}_j = i)} \tag{13}$$

We can also compute the precision based on the confusion matrix M:

$$precision[i] = \frac{M_{i,i}}{\sum_{j=1}^{\kappa} M_{j,i}} \tag{14}$$

Taking the draw label in Figure 9 as an example, we can compute its precision by dividing $M_{1,1}$ by the sum of the first column, giving a result of 90%.

3.4.4 Recall

The recall for a class is defined as the number of correctly classified samples in that class divided by the total number of labels in that class. Given a collection of prediction-label pairs $\{(\hat{y}_1, y_1), (\hat{y}_2, y_2), \ldots, (\hat{y}_N, y_N)\}$,

$$recall[i] = \frac{\sum_{j=1}^{N} \mathbb{I}(y_j = i \wedge \hat{y}_j = y_j)}{\sum_{j=1}^{N} \mathbb{I}(y_j = i)} \tag{15}$$

We can also compute the recall based on the confusion matrix M:

$$recall[i] = \frac{M_{i,i}}{\sum_{j=1}^{K} M_{i,j}} \tag{16}$$

Taking the win label in Figure 9 as an example, we can compute its recall by dividing $M_{2,2}$ by the sum of the second row, giving a result of 84%.

4 Deep Learning and Evaluation Methods

4.1 Board Representation

Inspired by the work on ALPHAZERO[63], we represent each board position as a stack of binary planes where each plane represents a specific piece type. For each plane, the dimension is 8×8 – the same as the checkers board. The positions where a piece of that type exists are encoded as 1, while the rest are 0. Since we have four piece types in checkers, the representation for one board position is of dimension $(4, 8, 8)$, organized as *(black king, white king, black man, white man).* Figure 10 shows an example of a board position and its representation.

4.2 Neural Network Architecture

We apply a residual neural network[19] to learn the outcomes in the CHINOOK endgame databases. In the following sections, we describe the building blocks and the final structure of our neural network.

4.2.1 Batch Normalization

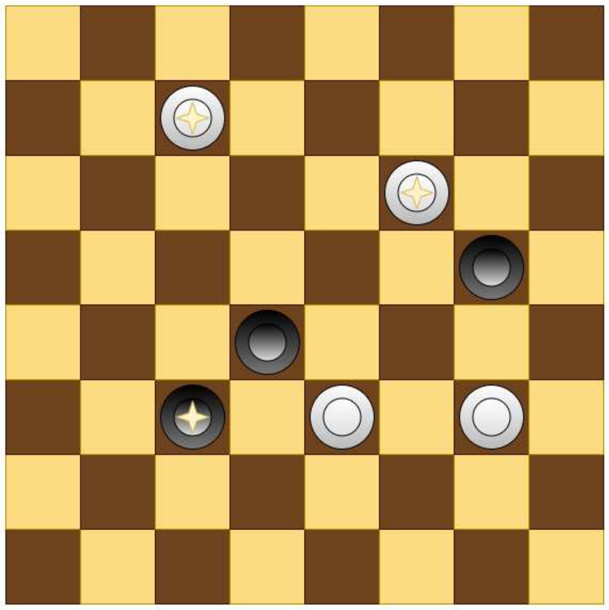
Batch normalization[25] is a technique to normalize the input data to the neural network layers using mini-batch statistics in order to speed up and stabilize the training. We found it necessary in our tasks – without batch normalization, training the neural network takes a long time to converge and can even diverge in some scenarios. Formally, given a d-dimensional input $x = (x^{[1]}, x^{[2]}, \ldots, x^{[d]})$, we normalize each dimension by

$$\hat{x}^{[k]} = \frac{x^{[k]} - \mathbb{E}[x^{[k]}]}{\sqrt{\sigma^2[x^{[k]}]}} \tag{17}$$

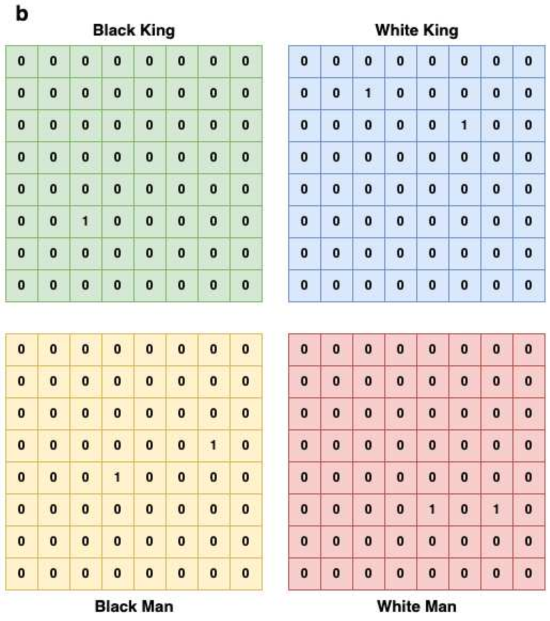
where σ^2 is the variance. In addition to the normalization, we also need a pair of learnable parameters $\gamma^{[k]}$ and $\beta^{[k]}$ that scale and shift $\hat{x}^{[k]}$:

$$y^{[k]} = \gamma^{[k]}\hat{x}^{[k]} + \beta^{[k]}. \tag{18}$$

a



b



Black King

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

White King

0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Black Man

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

White Man

c

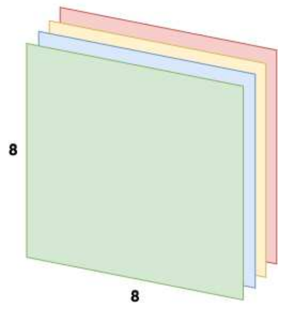


Figure 10: Board Representation Example. **a**. The original board position. **b**. The values of each plane corresponding to the piece type. **c** The representation of the original position as a stack of planes.

Since samples arrive in mini-batches $\mathscr{B} = \{x_1, x_2, \ldots, x_m\}$ during training, we use each mini-batch to estimate its mean and variance:

$$\mu_{\mathscr{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{19}$$

$$\sigma^2_{\mathscr{B}} = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathscr{B}})^2 \tag{20}$$

However, queries do not usually come in mini-batches during validation and testing. Even if they come in batches, we should still treat them individually. Therefore, the estimation method for training does not work in these phases. To resolve this, we prepare an estimation of the mean and the variance beforehand during training – that is – we keep an exponential moving average of these two parameters. For each mini-batch $\mathscr{B} = \{x_1, x_2, \ldots, x_m\}$:

$$\hat{\mu}_{t+1} = \eta \hat{\mu}_t + (1-\eta)\mu_{\mathscr{B}} \tag{21}$$

$$\hat{\sigma}^2_{t+1} = \eta \hat{\sigma}^2_t + (1-\eta)\sigma^2_{\mathscr{B}} \tag{22}$$

where η is the decay rate. We initialize $\hat{\mu}_0$ and $\hat{\sigma}^2_0$ to 0. This way, we can use the final $\hat{\mu}$ and $\hat{\sigma}^2$, together with the learned γ and β, to normalize and transform the input in validation and testing.

Through experiments, we found that it is best to apply batch normalization only after the activations of the convolutional layers. Batch normalization on the classifier hinders the final performance.

4.2.2 Activation

We use Mish [41] as the activation function of our neural network:

$$\text{Mish}(x) = x\tanh(\text{softplus}(x)) = x\tanh(\ln(1+e^x)) \tag{23}$$

We selected Mish after comparing it with several popular activation functions such as ReLU[30], SELU[28], and tanh[33] on our 2-5 piece databases benchmark. Mish is the most robust and best-performing one for our task in terms of convergence rate and peak accuracy.

4.2.3 Structure

We conducted extensive experiments on the 2-5 piece databases to pick our architecture from a set of candidates with a different number of residual blocks, channels of the convolution layers, and neurons in the fully connected layers. There was a number of considerations when we chose the final configuration:

- the neural network has to learn the data reasonably well while having an acceptable size;
- we cannot keep growing the size of the neural network to gain a slight performance increase because we want to achieve good compression;
- the model cannot be too simple either else it may sacrifice the performance too much.

Therefore, our final architecture achieves a balance between performance and size. The final structure of our residual network consists of an input convolution layer, five residual blocks (each having two convolution layers), and a classifier made of two fully connected layers. Figure 11 illustrates the overall structure of the residual neural network and one residual block. Table 2 summarizes the specifications of each type of layer in the neural network and the total parameter count. There are a total of 533,163 parameters that need to be trained in this network.

Layer	**Input Dim**	**Output Dim**	**Batch Norm**	**Kernel**	**Padding**
input conv	$(N,4,8,8)$	$(N,8,8,8)$	Yes	$(3,3)$	same
1st res conv	$(N,8,8,8)$	$(N,8,8,8)$	Yes	$(3,3)$	same
2nd res conv + res out	$(N,8,8,8)$	$(N,8,8,8)$	Yes	$(3,3)$	same
linear	$(N,512)$	$(N,512)$	No	N/A	N/A
output	$(N,512)$	$(N,3)$	No	N/A	N/A
Total Parameters: 533,163					

Table 2: Specifications of Each Type of Layer of the Residual Neural Network.

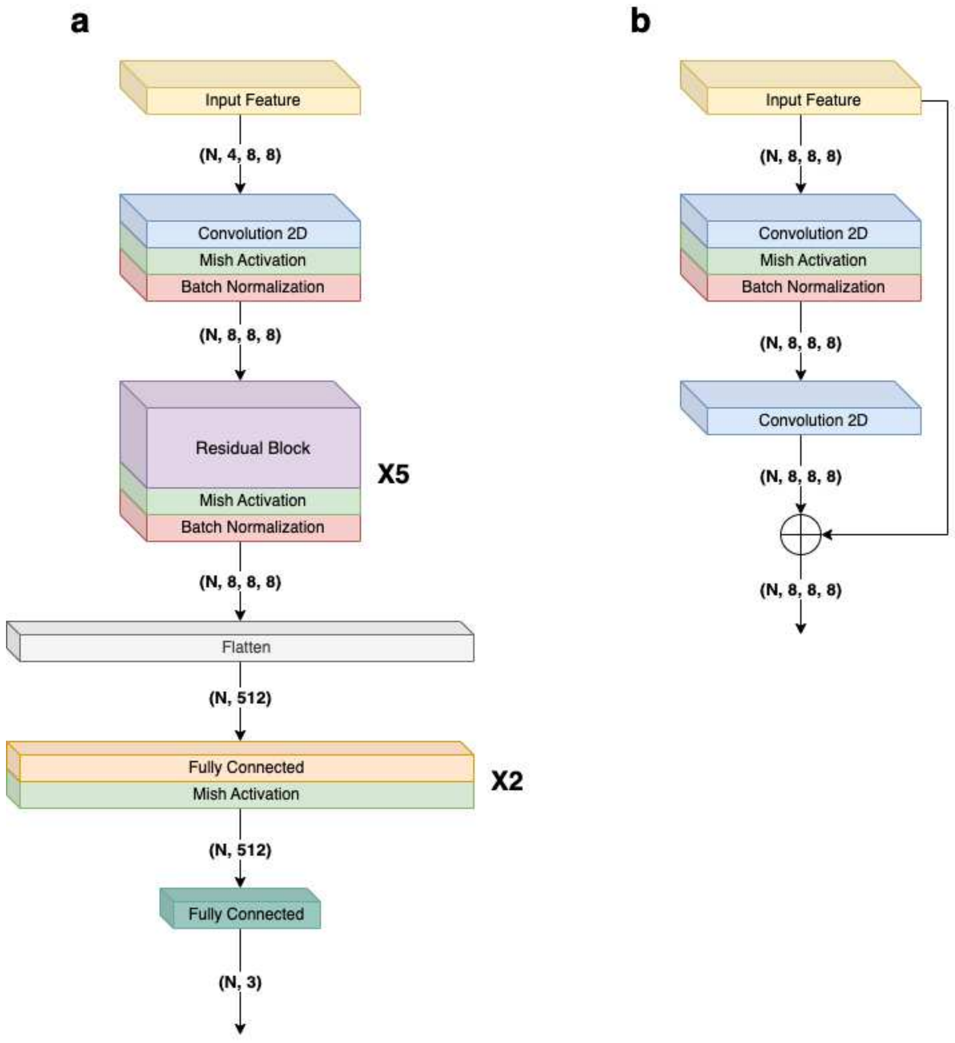


Figure 11: Residual Neural Network Structure. **a**. The overall structure. **b**. Structure of one residual block.

4.3 Loss Function

Our loss function is the weighted cross entropy loss[4]. Given a predicted probability distribution $[p_1, p_2, p_3]$, a true distribution $[y_1, y_2, y_3]$, a weight vector $[w_1, w_2, w_3]$, and a weight exponent α, the loss is:

$$\text{Weighted CE Loss} = -\sum_{i=1}^{3} w_i^{\alpha} y_i \log(p_i) \tag{24}$$

The weight vector is important to remedy the effect of unbalanced datasets. Let N_1, N_2, N_3 denote the number of positions for each outcome class in the dataset, respectively. We calculate the weight vector as follows:

$$w_i = \frac{N_1 + N_2 + N_3}{N_i} \tag{25}$$

where we assume $0 < N_1, N_2, N_3$. Whenever $N_i = 0$, we let $w_i = 0$. α controls the degree of the weights – larger exponents emphasize the weighting and vice-versa.

We also considered the focal loss[35] which has another weighting scheme:

$$\text{Focal Loss} = -\sum_{i=1}^{3} w_i^{\alpha} y_i (1 - p_i)^{\gamma} \log(p_i) \tag{26}$$

where γ is another exponent controlling the term $(1 - p_i)$. The focal loss gives less weight to the samples that the neural network can predict well and more to the poorly classified ones. In practice, we notice that focal loss results in unstable training and is unable to deliver better overall performance on our benchmark for a range of hyperparameters. Hence, we use the weighted cross entropy loss.

4.4 Optimizer

For the optimizer, we use rectified Adam (RAdam)[36], a variant of Adam[26] that is more robust.

We tested RAdam against a set of candidates, including Adamax[26], Adagrad[9], and Adamw[37] for our task. The experiments showed that the other optimizers had comparable or inferior performance to RAdam in terms of convergence rate and final accuracy. Therefore, we chose RAdam as our default optimizer.

There are two hyperparameters of RAdam – exponential decay rate of the first and the second moment of past gradients – β_1 and β_2. We found that the default configuration $\beta_1 = 0.9$, $\beta_2 = 0.999$ is good for our experiments.

4.5 Hyperparameter Selection

The hyperparameters involved include the learning rate, the mini-batch size, the decay rate for batch normalization, the weight exponent α for the loss function, and the decay rates β_1, β_2 for the optimizer. The large number of combinations prevents us from using an exhaustive grid search. As a result, we treated the hyperparameters independently and optimized them separately; That is, we choose a set of candidates for each hyperparameter and benchmark the performance for each without changing other hyperparameters; We lock the best candidate and move on to the next hyperparameter once the experiments finish. Table 3 summarizes the set of hyperparameters used for experiments.

primary	learning rate = 0.0002 mini-batch size = 1,024
batch normalization	decay rate = 0.99
loss function	$\alpha = 0.5$
optimizer	$\beta_1 = 0.9$ $\beta_2 = 0.999$

Table 3: Choice of Hyperparameters.

4.6 Evaluation of Neural Networks and Search

In this section, we introduce the evaluation metrics and figures we use to gauge the performance of the neural network and the one-ply search in training and testing.

4.6.1 Training

During model training, we record the overall accuracy, per-outcome class accuracy and per-flag class accuracy every five epochs on both the training set and the validation set:

$$accuracy_{overall} = \frac{\text{\# of correct predictions}}{\text{\# of positions}} \tag{27}$$

$$accuracy_{outcome}[i] = \frac{\text{\# of correct predictions of outcome class } i}{\text{\# of positions of outcome class } i} \tag{28}$$

$$accuracy_{flag}[i] = \frac{\text{\# of correct predictions of flag class } i}{\text{\# of positions of flag class } i} \tag{29}$$

This way, we can monitor the overall progress of training and the performance of the model on each class of outcome and flag. The per-outcome class accuracy is also called the recall. Figure 12 shows a typical example of the learning curves of training a neural network with overall and per-flag class accuracies.

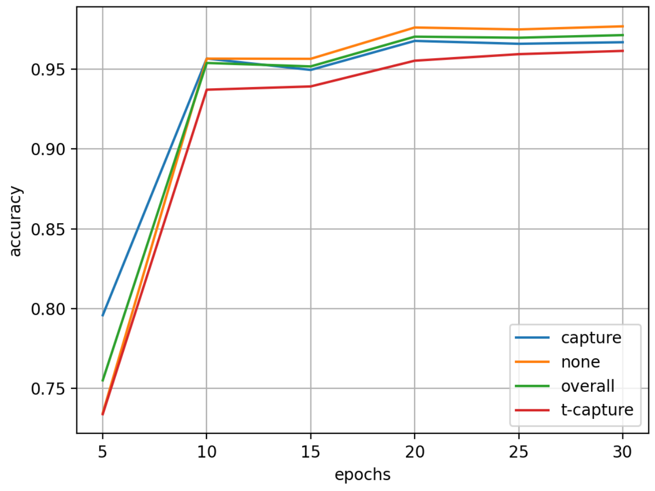


Figure 12: Example Learning Curves of the Neural Network with Overall and Per-flag Class Accuracies. (t-capture denotes threatened capture.)

4.6.2 Testing

For testing, we present two more informative ways to evaluate and visualize the performance of both the neural network and the one-ply search.

Using the normalized confusion matrix shown in Figure 13, we can analyze the distribution of the predictions across the label classes. The rows are the ground truth, and the columns are predictions. Each cell holds the percentage of the positions over the test set. We can compute accuracy, precision and recall based on this matrix.

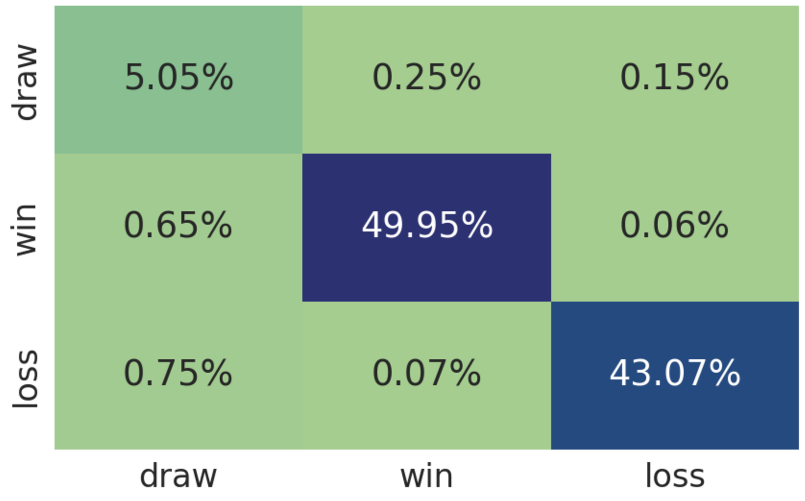


Figure 13: Example of a Normalized Confusion Matrix.

The second type of measure is a correlation table that correlates the neural network performance with that of the search. It partitions the set of evaluation results into four disjoint categories – the neural network and search are both correct, only the neural network is correct, only the search is correct, and both are incorrect. Table 4 gives an example. Each entry contains the count and percentage of one category. Summing up the rows/columns gives the correct and incorrect numbers for the search/neural network.

	Model Correct	Model Incorrect	
Search Correct	25,661,576 (95.09%)	507,088 (1.88%)	96.97%
Search Incorrect	522,139 (1.93%)	294,825 (1.09%)	3.02%
	97.02%	2.97%	

Table 4: Example of a Correlation Table.

5 Implementation

We provide some implementation details of the crucial components of this project in this section.

5.1 Computing Environment

The computing environment where we implement, run and experiment with our program includes:

- 10 Intel(R) Xeon(R) E5-2650 v4 @ 2.20GHz CPUs
- 78 GB system memory
- 2 TB network file system
- 1 Nvidia Titan RTX GPU with 2.4 GB memory

5.2 Data Preparation

The pipeline that reads positions from the databases and saves them in the neural-network-ready format at the end has several steps:

1. A program called front prepared by J. Schaeffer can access the databases by the piece type plus leading *checker* rank and outputs a binary "byte file." Each byte file holds the board positions and their corresponding outcomes and flags. The first 8 bytes of the file represent the total number of positions in this slice. The remaining portion consists of consecutive chunks of 13 bytes, where each chunk stores information about one board and its outcome and flag. Of the 13 bytes, the first 4 bytes represent the white pieces, the second 4 bytes represent the black pieces, the third 4 bytes represent the kings, and the last byte holds the outcome and the flag. Each bit of the 4 bytes (32 bits) maps to one of the 32 playable squares of a checkerboard – a piece is present if the corresponding bit is on and absent otherwise. In the last byte, bits 0 and 1 store the outcome, while bits 4 and 5 store the flag.

2. Utilities in data_utils.py read, parse and process the byte file to recreate the board positions, outcomes and flags. Let W, B, and K denote the bit vector for the white, black, and *king* pieces for one position. We need B AND K to extract black *king*s and B AND (NOT K) to restore the black *checker*s. The same logic applies to the white *king*s and *checker*s. For the outcome and the flag, we use a mask and shift to extract them. Once we have obtained the bit vectors, we store a board position as a $(4,8,8)$-dimensional NumPy array which serves as input for the neural networks. Figure 14 illustrates this process.

3. The last stage of the pipeline is in Python script process.py. It uses the above utilities to generate all the required positions, outcomes and flags, aggregates and transforms them into NumPy arrays, splits them into partitions of specified length, and saves each in a .npz file on the disk. A .npz file is a zipped archive of multiple NumPy arrays – a format suitable for our use case. Each file contains an array of positions, an array of outcomes, and an array of flags. process.py also creates a Python dictionary that maps each file path to the meta-data that keeps the count of each outcome-flag pair in the file.

5.3 Neural Network

We implement the neural network in both PyTorch[47] and JAX[6] + Haiku[20]. Both are popular deep learning frameworks yet differ in many ways. We compare and contrast them in detail in the appendix. As an illustration, code snippets 1 and 2 demonstrate how to implement our residual block in JAX and PyTorch, respectively.

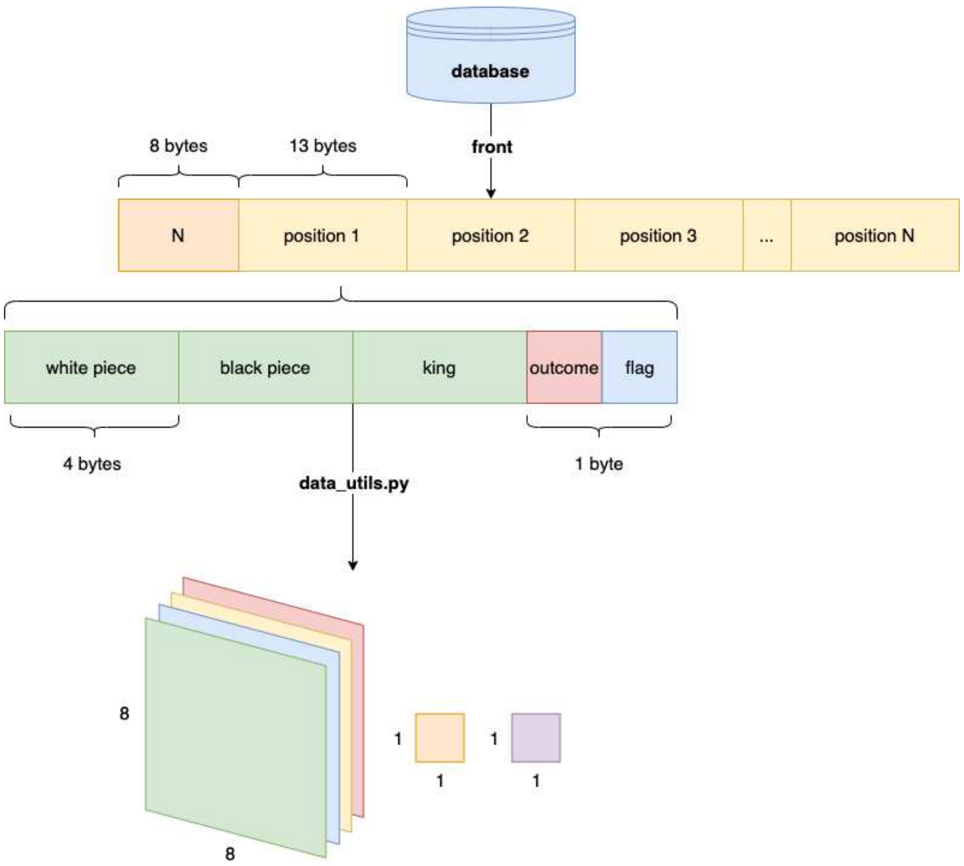


Figure 14: Data Pipeline. The front program retrieves a slice of checkers data from the databases and generates a byte file. The utilities in data_utils.py transform the byte representations into neural network input representations for the board or numerical values for the outcome and flag.

```
import haiku as hk
import jax
class ResBlock(hk.Module):
    def __init__(self, is_train: bool, name: str):
        super().__init__(name)
        self.is_train = is_train
        '''
        This method initializes a convolutional layer.
        output_channels: define the # of feature maps
        output by this convolutional layer
        kernel_shape: define the size of the filter;
        the height and the width of the filter are equal
        if the argument is a scalar
        data_format: the order of the axis; 'NCHW' stands
        for (batch size, channel #, height, width)
        '''
        self.conv1 = hk.Conv2D(output_channels=8,
                               kernel_shape=3,
                               data_format='NCHW')
        # initialize the activation function
        self.activation = Mish()
        '''
        This method initializes a
        batch normalization layer.
        create_scale: whether to use a scale factor gamma
        create_offset: whether to use an offset beta
        decay_rate: exponential decay rate
        data_format: same mechanism as introduced earlier
        '''
        self.batch_norm = hk.BatchNorm(create_scale=True,
                                       create_offset=True,
                                       decay_rate=0.99,
                                       data_format='NCHW')
        self.conv2 = hk.Conv2D(output_channels=8,
                              kernel_shape=3,
                              data_format='NCHW')
    # the method gets called during forward propagation
    def __call__(self, x):
        residual = self.activation(self.conv1(x))
        residual = self.batch_norm(residual, is_training=
                                       self.is_train)
        residual = self.conv2(residual)
        return x + residual
```

Code Listing 1: Illustrative Implementation of the Residual Block in JAX.

```
from torch import nn
class ResBlock(nn.Module):
    def __init__(self):
        super().__init__()
        '''
        This method initializes a convolutional layer.
        in_channels: defines the # of input features
        out_channels: defines the # of output features
        kernel_size: defines the size of the filter;
        the height and the width of the filter are equal
        if the argument is a scalar
        padding: defines the padding size around the input
        stride: defines the # of positions to shift before
        the next convolution operation.
        '''
        self.conv1 = nn.Conv2d(in_channels=8,
                               out_channels=8,
                               kernel_size=3,
                               padding=1,
                               stride=1)
        # initialize the activation function
        self.activation = Mish()
        '''
        This method initializes a
        batch normalization layer.
        num_features: defines the # of input features
        momentum: defines the momentum (1-decay_rate)
        '''
        self.batch_norm = nn.BatchNorm2d(num_features=8,
                                         momentum=0.01)
        self.conv2 = nn.Conv2d(in_channels=8,
                               out_channels=8,
                               kernel_size=3,
                               padding=1,
                               stride=1)
    # the method gets called during forward propagation
    def forward(self, x):
        residual = self.activation(self.conv1(x))
        residual = self.batch_norm(residual)
        residual = self.conv2(residual)
        return x + residual
```

Code Listing 2: Illustrative Implementation of the Residual Block in PyTorch.

For optimizers, we use the torch.optim package[47] for PyTorch and Optax[5] for JAX. Both implementations use the PyTorch data loader for data loading.

5.4 One-ply Search

We design our one-ply search program such that the inference on the children nodes occurs in batches to reduce the data transfer overhead from CPU to GPU and vice-versa. We implement a function *generateNextMove* that, given a position, produces all the possible subsequent boards by making a legal move as Black at the current position. Given a mini-batch of positions $\mathscr{B} = \{x_1, x_2, \ldots, x_N\}$, the naïve way to implement the one-ply search is to find the children of the first position, call the neural network to predict the outcomes of its children, determine the outcome of the first position, then move on to the next one in the mini-batch. Nevertheless, this implementation usually induces significant data transfer between the CPU and GPU and takes negligible advantage of the parallelization of neural network inference on the GPU. Our solution is to find the children of every position in $\mathscr{B}$ first, then concatenate the children to form another batch $\mathscr{B}_{children} = \{generateNextMove(x_1), generateNextMove(x_2), \ldots, generateNextMove(x_N)\}$. In this way, we can send $\mathscr{B}_{children}$ in mini-batches to the neural network on the GPU and save data transfers. We keep a list of (start index, end index) pairs $\mathscr{I} = \{(i_1, j_1), (i_2, j_2), \ldots, (i_N, j_N)\}$ to retrieve the predictions of the children positions from the returned batch. Figure 15 illustrates this batched one-ply search process and the devices where different computations occur.

In some cases, the program doesn't need the neural network to know the outcome. The first case is when *generateNextMove* returns an empty list, where the position is a loss automatically. The second case is when one of the children has no black pieces, where the position is a win. We incorporate these special cases in our algorithm.

5.5 Experimental Framework

We implement a base experiment that can do the following tasks:

- load the configuration files
- create a directory structure for experiment results and configure the output paths
- initialize the loss function, optimizer and benchmark writer

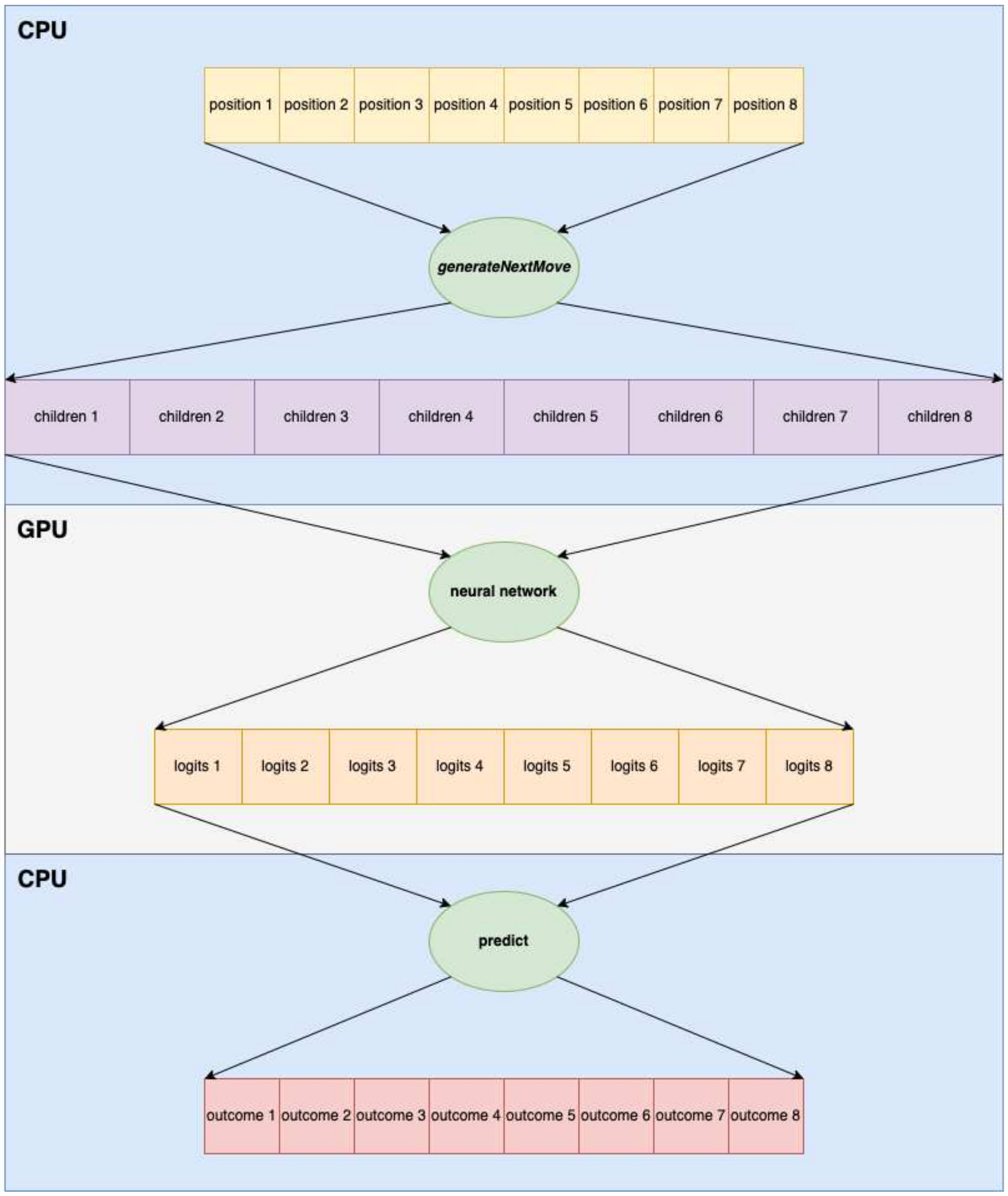


Figure 15: Batched One-Ply Search.

- plot the meta-data using Matplotlib[24] on TensorBoard[1] for visualization
- train the neural network and plot the learning curve on TensorBoard for monitoring
- evaluate the neural network and search, generate the confusion matrix, the error distribution table, and the correlation table with Matplotlib, and save misclassified positions in a text file

These tasks are universal to any experiment. We only need to modify some components to create a new trial.

6 Experiments

We have conducted experiments to explore and investigate the applicability of deep neural networks in learning the CHINOOK endgame database. We tested the generalization of the ResNet and proposed a method to dynamically choose between the model and the one-ply search for inference. We present the experiments in the order of their development.

6.1 Sampling of the 2-7 Piece Databases

Exhaustively presenting all the positions in the databases to the neural network quickly becomes infeasible with our hardware as we move beyond the 6-piece database. As a concrete example, training our network with 70% of the 5-piece database ($\approx$ 97.6 million positions) for 30 epochs takes almost nine hours. We argue that it is still impractical to exhaust all positions, even with better hardware, when dealing with trillions of data points. Furthermore, our neural network is not a particularly large one. Fortunately, deep neural networks are known to generalize well to unseen data.

In this experiment, we investigate the performance of our ResNet when we train it with a small random subset of the target databases. While training on more data usually yields better performance with deep neural networks, we are curious to find out how sensitive our model is to the size of the training set on this particular problem.

6.1.1 Dataset of the Sampling Experiment

The dataset for the sampling experiment consists of 10% of the entire 2-6 piece databases and 0.1% of the four pieces vs. three pieces partition of the 7-piece database, sampled uniformly. In the remainder of the section, we refer to this dataset as the 2-7 piece dataset. Beginning with a split of 70% for training, 10% for validation and 20% for testing, we shrink the training set by a factor of ten for each consecutive trial, down to 0.1% of its original size, while keeping the size of the validation and test sets constant. Table 5 summarizes the number of positions for each trial. The plot in Figure 16 reveals the distribution of the outcomes of the positions in the baseline 2-7 piece dataset. The downsampled datasets preserve the relative distributions because of uniform sampling.

	Baseline	**10%**	**1%**	**0.1%**
Training	202,238,136	20,223,804	2,022,368	202,229
Validation	28,891,154	-	-	-
Test	57,782,317	-	-	-

Table 5: Summary of the 2-7 Piece Dataset for the Sampling Experiment.

6.1.2 Results of the Sampling Experiment

We record accuracy and recall of our ResNet on the training and validation sets during training, saving a checkpoint every 5 epochs. We run each trial until the validation accuracy converges. After training, we pick the checkpoint achieving the highest validation accuracy to run on the test set.

The confusion matrices in Figure 17 show the distribution of the mistakes across different outcomes. Most misclassifications happen in drawing positions, with them being classified as winning or losing and vice-versa. This statement holds for both the neural network and the one-ply search, regardless of the sampling ratio. The highest recall for Draw achieved by either the model or the search is approximately 93.52% compared to 98.53% for Win and 96.71% for Loss. We speculate that it is more difficult for the neural network to do well in the Draw class because there are fewer drawing positions in the dataset, and the decision boundaries for Draw are trickier to learn than for the other two outcomes.

Table 6 displays the correlation table for each trial. Both the model and the one-ply search achieve an accuracy of over 90% for every task in this experiment, with search outperforming the raw neural network in three out of four cases. The ResNet can attain an accuracy of above 95% when trained with a little more than 10% of the total number of positions in the baseline dataset.

The model correctly scores 95.46% with the full baseline dataset, but as the amount of training data decreases, so does the performance. Yet with only 0.1% of the baseline, the ResNet still learns enough to score an impressive 90.79% accuracy.

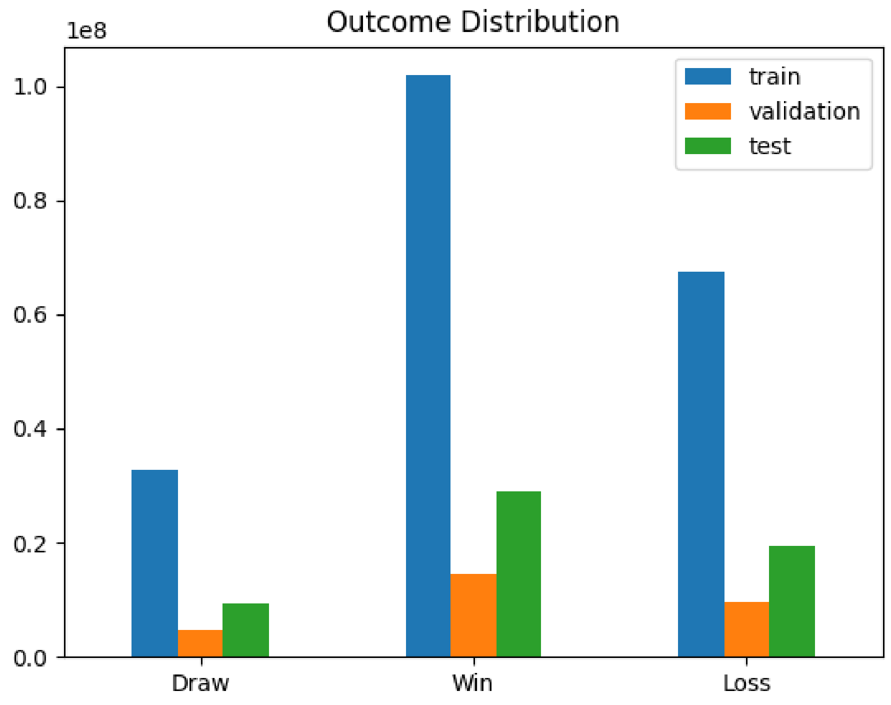


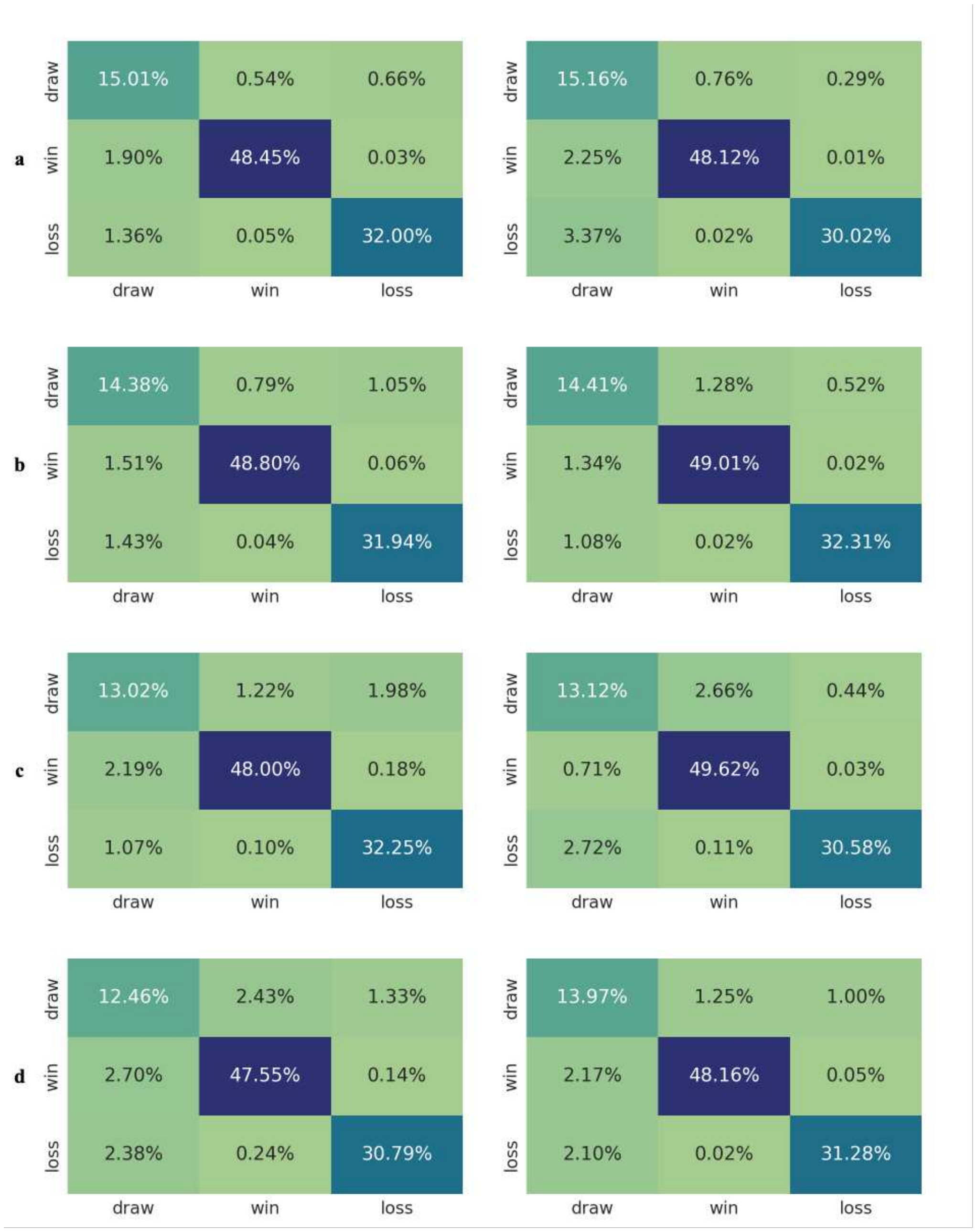
Figure 16: Outcome Distribution of the Baseline 2-7 Piece Dataset.

The search performance shows an interesting pattern – the accuracy of the one-ply search does not peak at the baseline dataset. Instead, it happens on the 10% baseline task with an accuracy of 95.72%. The one-ply search with the best-performing model only yields an accuracy of 93.30%. Even the least accurate model trained on the 0.1% baseline training set results in a 93.40% accuracy when used with the search. These results imply that the search accuracy may not solely rely on the quality of the model. One possibility is that the one-ply search alters the distribution of the positions input to the neural network for inference. The models trained with more training data overfit on the original distribution, making them more accurate when presented with the same distribution but less robust otherwise.

This experiment demonstrates that sampling is a feasible technique to train a deep neural network to learn the mappings in the CHINOOK endgame databases without enumerating every position. Our ResNet can achieve high accuracy ($> 95\%$), even trained with less than 1% of the entire population. In addition, the one-ply search usually improves upon the neural network, suggesting its capability to increase the robustness of the prediction when the model is imperfect.

Baseline	Model Correct	Model Incorrect	
Search Correct	52,321,877 (90.55%)	1,591,546 (2.75%)	93.30%
Search Incorrect	2,839,778 (4.91%)	1,029,116 (1.78%)	6.69%
	95.46%	4.53%	
10%	Model Correct	Model Incorrect	
Search Correct	53,309,487 (92.26%)	2,002,055 (3.46%)	95.72%
Search Incorrect	1,648,779 (2.85%)	821,996 (1.42%)	4.27%
	95.11%	4.88%	
1%	Model Correct	Model Incorrect	
Search Correct	51,183,270 (88.58%)	2,745,657 (4.75%)	93.33%
Search Incorrect	2,708,708 (4.69%)	1,144,682 (1.98%)	6.67%
	93.27%	6.73%	
0.1%	Model Correct	Model Incorrect	
Search Correct	50,455,079 (87.32%)	3,515,282 (6.08%)	93.40%
Search Incorrect	2,004,385 (3.47%)	1,807,571 (3.13%)	6.60%
	90.79%	9.21%	

Table 6: Correlation Tables for the Sampling Experiment.



a (neural network)	draw	win	loss
draw	15.01%	0.54%	0.66%
win	1.90%	48.45%	0.03%
loss	1.36%	0.05%	32.00%

a (one-ply search)	draw	win	loss
draw	15.16%	0.76%	0.29%
win	2.25%	48.12%	0.01%
loss	3.37%	0.02%	30.02%

b (neural network)	draw	win	loss
draw	14.38%	0.79%	1.05%
win	1.51%	48.80%	0.06%
loss	1.43%	0.04%	31.94%

b (one-ply search)	draw	win	loss
draw	14.41%	1.28%	0.52%
win	1.34%	49.01%	0.02%
loss	1.08%	0.02%	32.31%

c (neural network)	draw	win	loss
draw	13.02%	1.22%	1.98%
win	2.19%	48.00%	0.18%
loss	1.07%	0.10%	32.25%

c (one-ply search)	draw	win	loss
draw	13.12%	2.66%	0.44%
win	0.71%	49.62%	0.03%
loss	2.72%	0.11%	30.58%

d (neural network)	draw	win	loss
draw	12.46%	2.43%	1.33%
win	2.70%	47.55%	0.14%
loss	2.38%	0.24%	30.79%

d (one-ply search)	draw	win	loss
draw	13.97%	1.25%	1.00%
win	2.17%	48.16%	0.05%
loss	2.10%	0.02%	31.28%

Figure 17: Normalized Confusion Matrices for the Sampling Experiment. Each cell shows the percentage of positions over the test set. The rows are ground truths. The columns are predictions. The entries on the main diagonal correspond to correct predictions. Left Column: neural network. Right Column: one-ply search. **a.** baseline **b.** 10% of baseline **c.** 1% of baseline **d.** 0.1% of baseline.

6.2 Inter-database Generalization

We have learned from the sampling experiment that our deep neural network can achieve a compelling accuracy on the test set when trained with a tiny bit of the whole target database. An important question to ask is whether the generalization is useful to the databases that do not appear in the neural network's training set. For example, is a model trained on the 4-piece database capable of inferring the outcomes of the positions with six pieces on the board? If this is true, then it would be an important result. It would demonstrate the ability of the neural network learning to extrapolate to larger (and more complex) datasets.

In the inter-database generalization experiment, we design two tasks to try to answer the above questions. The first task is training the neural network on the 4-piece database until convergence and testing it on the 6-piece database. The second one is training on the 5-piece database and testing the resultant model on the four pieces vs. three pieces partition of the 7-piece database. We will refer to the dataset sampled from this partition simply as the 7-piece dataset for the remainder of this section. Our choice to train the ResNet on an even/odd-piece database is based on the concern of material advantage – one of the players has a material advantage for all the positions in odd-piece databases, leading to a different distribution of outcomes from the even ones.

Note that the inter-database generalization experiment differs from the subsequent transfer learning one. Here, we apply our trained models directly on the target databases without any modifications. In transfer learning, we add an adaptation phase, retraining on the new data.

6.2.1 Dataset of the Inter-database Generalization Experiment

We use the entire 4-piece and 5-piece databases with a 7:1:2 train-validation-test split ratio. The test sets are generated with 30% of the 6-piece database and 100% of the sampled four pieces vs. three pieces partition of the 7-piece database, split according to the 7:1:2 ratio. Table 7 records the number of positions in each split for each dataset used.

	4 Piece	**5 Piece**	**6 Piece**	**7 Piece**
Training	4,346,521	97,615,565	N/A	N/A
Validation	620,931	13,945,080	N/A	N/A
Test	N/A	N/A	153,162,235	38,110,526

Table 7: Summary of the Datasets for the Inter-Database Generalization Experiment.

6.2.2 Results of the Inter-database Generalization Experiment

We train our models on the 4-piece and 5-piece datasets until convergence and test their performance on the 6-piece and 7-piece datasets, respectively.

The confusion matrices shown in Figure 18 demonstrate that again, the mistakes happen more frequently in the drawing positions. The model recall of the drawing position is 74.24% in the 4-piece to 6-piece task. The model misclassifies 28.67% of the losing positions as drawing. The one-ply search in the same task achieves a recall of 83.38% in the drawing positions but misclassifies winning positions as Draw this time, accounting for 21.69% of all Win. The recalls of Win are 98.33% for the model and 77.03% for the search. For Loss, the recalls are 63.27% for the model and 92.63% for the search. In the 5-piece to 7-piece task, the model recall for Draw is 20.61%. The recall of Draw for the search is 7.98% – much lower than the model. The model has a recall of 98.44% for Win – 3.3% higher than the search and 91.96% for Loss – 5.22% lower than the search.

The correlation tables in Table 8 reveal that the neural network and the search can still reach an accuracy above 80%, despite the lower performance on the drawing positions. Furthermore, the one-ply search gets 0.6% higher accuracy than the model in the 4-piece to 6-piece task, whereas the model's accuracy is 2.13% higher in the other experiment.

The results suggest that generalization does happen across different databases. However, the performance of both the neural network and the one-ply search is significantly less impressive than in the sampling experiment. This outcome is within our expectations because the input data distribution in the training phase does not align with the test phase, a "covariate shift."

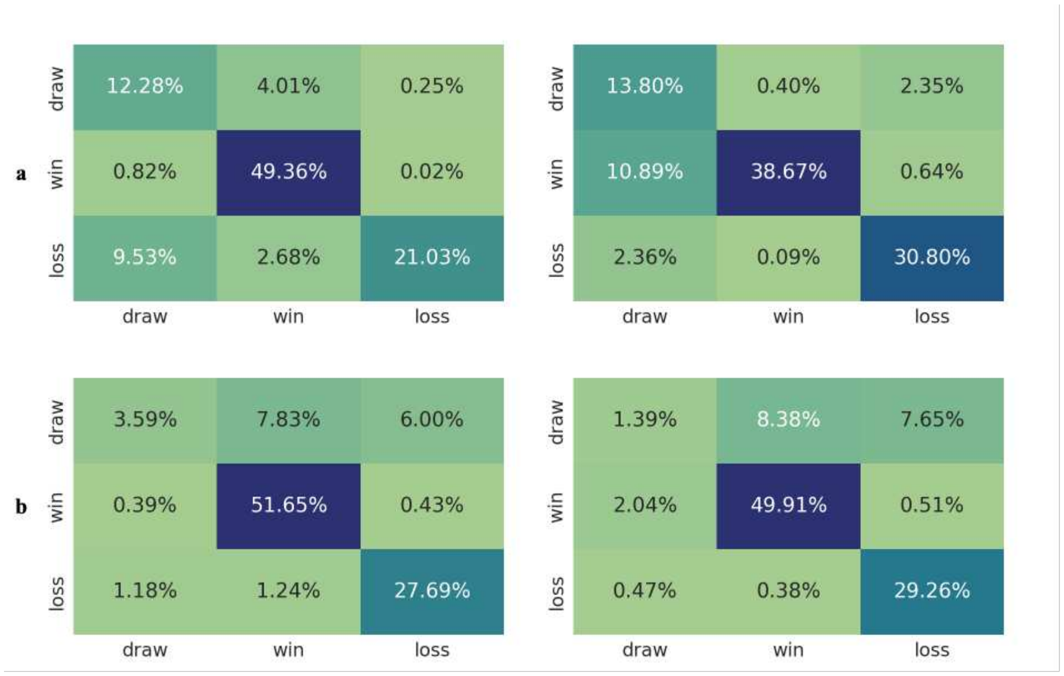


Figure 18: Normalized Confusion Matrices for the Inter-Database Generalization Experiment. Each cell shows the percentage of positions over the test set. The rows are ground truths. The columns are predictions. The entries on the main diagonal correspond to correct predictions. Left Column: neural network. Right Column: one-ply search. **a.** train on the 4-piece dataset and test on the 6-piece dataset **b.** train on the 5-piece dataset and test on the 7-piece dataset.

4 Piece → 6 Piece	Model Correct	Model Incorrect	
Search Correct	107,115,684 (69.94%)	20,430,421 (13.34%)	83.28%
Search Incorrect	19,514,041 (12.74%)	6,102,089 (3.98%)	16.72%
	82.68%	17.32%	
5 Piece → 7 Piece	Model Correct	Model Incorrect	
Search Correct	29,301,935 (76.89%)	1,397,825 (3.67%)	80.56%
Search Incorrect	2,302,548 (6.04%)	5,108,218 (13.40%)	19.44%
	82.93%	17.07%	

Table 8: Correlation Tables for the Inter-Database Generalization Experiment.

6.3 Transfer Learning

The results for directly applying the neural network trained on one database to another are reasonable but not impressive. Can we mitigate the performance degradation caused by covariate shift and still adapt it to unseen databases to reuse its knowledge? We use transfer learning[15, 45], by applying the knowledge gained in solving one problem to another different but related task. Transfer learning often involves two stages – pre-training and fine-tuning. One pre-trains the neural network on some precursor task and then fine-tunes (usually only the later layers) it on the target problem. In our setting, if the features learned by the earlier layers of the deep neural network are not database-specific, then it should be able to adapt to a new unseen database by only reconfiguring its classifier. Based on this intuition, we design our transfer learning experiment as follows:

1. We first pre-train two ResNet models, one on the 2-5 piece dataset and the other on the 2-6 piece dataset.

2. Then we freeze (stop gradient) the residual blocks of the two models and only fine-tune their classifiers on the 7-piece dataset for a few epochs.

3. We also prepare another model as a control by freezing its residual blocks right after random initialization and training only its classifiers on the 7-piece dataset for enough epochs. We refer to this model as the *random feature model*. The purpose of the *random feature model* is to eliminate the possibility that the neural network can learn well by arbitrarily mapping the input to a higher dimensional space.

4. Lastly, we train a *reference model* directly on the target 7-piece dataset for a sufficient number of iterations. The *reference model* will serve as the baseline with no transfer learning.

6.3.1 Dataset of the Transfer Learning Experiment

Our 2-5 piece and 2-6 piece datasets contain 100% and 30% positions randomly sampled from the target databases, respectively. We use only 30% of the 2-6 piece databases because using the entire database empirically gives us similar or even worse results but triples the training time. Both datasets follow a 7:1:2 train-validation-test split ratio. The 7-piece dataset is the same one used in the inter-database generalization experiment. The detailed summary of the datasets is in Table 9.

	2-5 Piece	**2-6 Piece**	**7 Piece**
Training	102,101,750	566,698,372	133,386,843
Validation	14,585,964	80,956,902	19,055,263
Test	N/A	N/A	38,110,526

Table 9: Summary of the Datasets for the Transfer Learning Experiment.

6.3.2 Results of the Transfer Learning Experiment

We train two models on the 2-5 piece and 2-6 piece datasets until convergence, respectively. Table 10 contains their correlation tables before fine-tuning. Interestingly, the model pre-trained on the 2-5 piece dataset reaches an accuracy of 80.22% while the other model pre-trained on the 2-6 piece dataset has an accuracy of 75.36%, even though the 2-6 piece dataset has more training data and positions with more pieces on the board. This phenomenon is likely due to the outcome distribution of the 2-5 training set being closer to the 7-piece test set. The accuracies are comparable with the previous inter-database generalization experiment. The one-ply search raises the performance in both cases but not to a competitive level as in the sampling experiments.

We then freeze the residual blocks of our pre-trained models and fine-tune their classifiers on the 7-piece dataset for only five epochs. This fine-tuning results in high accuracy for both pre-trained models. Figure 19 displays their learning curves. The accuracy rises beyond 94% after merely one epoch of training and continues to increase by another 0.5% in five epochs. The final accuracy centers around 95%. An interesting observation is that the training and validation accuracies of the model pre-trained on the 2-6 piece dataset are consistently higher than those of the one pre-trained on the 2-5 piece dataset, even though the former model performs worse before transfer learning begins. The final performance on the test set also reflects this difference, as illustrated in Table 10. We hypothesize that adding the 6-piece positions improves the quality of the features learned by the neural network by introducing richer signals and more complex piece interactions.

Figure 20 presents the learning curves of the *random feature model*. The fine-tuned classifier is able to correctly classify approximately 90% of the random features after 50-60 epochs – considerably slower and worse than the pre-trained models. The correlation table in Table 11 confirms that. Thus, we can conclude that training the residual blocks to discover patterns is crucial for better performance and faster convergence.

Finally, we train our *reference model* until convergence. Table 12 shows the correlation table for the best checkpoint. We observe that the transfer learning models achieve comparable performance with the directly trained one.

Our two main findings through this experiment are:

1. Reusing previously trained models on larger databases is viable through transfer learning. One only needs to fine-tune their classifiers with samples from the target database.

2. Some features extracted by the deep neural network are likely high-level and database-independent. The impressive final performance of the transfer learning models and the suboptimal performance of the *random feature model* support this hypothesis.

6.4 Combining Neural Network and Search Inference

The previous experiments demonstrate that the neural network inference and the one-ply search do not always agree. While in some tasks, the one-ply search can get an overall better accuracy at the end, in other cases, using the model alone is better, or the advantage is negligible. Therefore, using only one for inference implies a considerable opportunity cost. Is there a way to dynamically switch between the model and the search position-wise based on some criteria such that the ultimate classifier is more accurate than either?

As a context for our next experiment, we first define the highest probability from the distribution output by the neural network given a position as its confidence for

2-5 Piece Pre-train	Model Correct	Model Incorrect	
Search Correct	29,313,893 (76.92%)	1,682,679 (4.42%)	81.34%
Search Incorrect	1,258,510 (3.30%)	5,855,444 (15.36%)	18.66%
	80.22%	19.78%	
2-6 Piece Pre-train	Model Correct	Model Incorrect	
Search Correct	25,383,593 (66.61%)	5,454,699 (14.31%)	80.92%
Search Incorrect	3,334,511 (8.75%)	3,937,723 (10.33%)	19.08%
	75.36%	24.64%	
2-5 Piece $\rightarrow$ 7 Piece	Model Correct	Model Incorrect	
Search Correct	34,695,990 (91.04%)	1,019,743 (2.68%)	93.72%
Search Incorrect	1,463,083 (3.84%)	931,710 (2.44%)	6.28%
	94.88%	5.12%	
2-6 Piece $\rightarrow$ 7 Piece	Model Correct	Model Incorrect	
Search Correct	34,326,432 (90.07%)	969,666 (2.54%)	92.61%
Search Incorrect	1,971,422 (5.17%)	843,006 (2.21%)	7.38%
	95.24%	4.75%	

Table 10: Correlation Tables of the Pre-trained Models Before and After Fine-tuning on the Target 7-Piece Dataset.

Random Feature Model	Model Correct	Model Incorrect	
Search Correct	26,906,527 (70.60%)	1,926,654 (5.06%)	75.66%
Search Incorrect	7,515,449 (19.72%)	1,761,896 (4.62%)	24.34%
	90.32%	9.68%	

Table 11: Correlation Table of the Trained *Random Feature Model*.

Reference Model	Model Correct	Model Incorrect	
Search Correct	30,257,729 (79.39%)	816,636 (2.14%)	81.53%
Search Incorrect	6,268,606 (16.45%)	767,555 (2.01%)	18.46%
	95.84%	4.15%	

Table 12: Correlation Table of the Trained *Reference Model*.

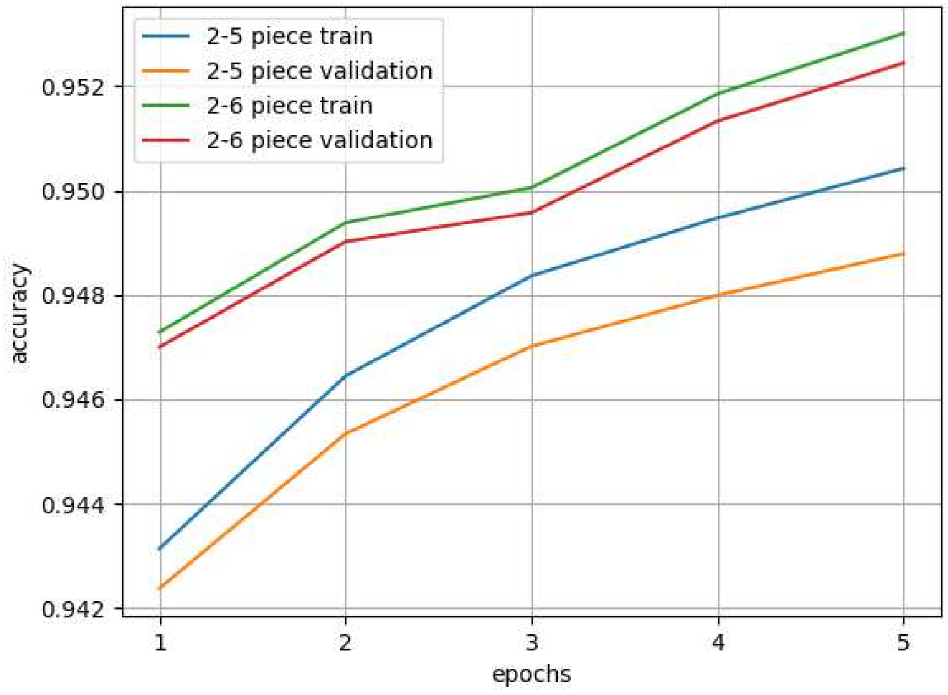


Figure 19: Learning Curves of Fine-tuning Pre-trained Models to the Target Database.

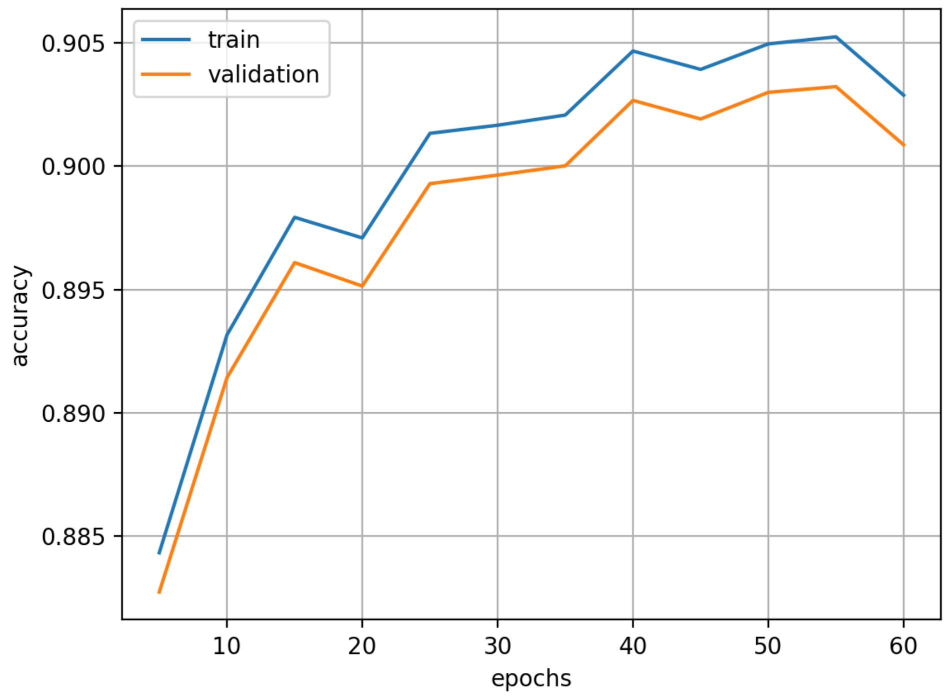


Figure 20: Learning Curves of the *Random Feature Model*.

that position. For instance, if the model outputs a distribution [0.80 (Draw), 0.15 (Win), 0.05 (Loss)] given an input board representation, its confidence for this board position is 80%.

We discover a correlation between the neural network's quality of prediction and confidence – the higher the confidence, the more likely the ResNet's inference is correct. We reuse the best baseline model from the sampling experiment, whose correlation table is in Table 6. Using this neural network, we generate Table 13, a summary of the mean confidences of the correctly and incorrectly classified positions in the 2-7 piece validation set with respect to the outcome class. One can realize a significant difference between the mean confidence between the correct and incorrect predictions. The difference across the outcome classes is not as striking. We also plot the distribution of the number of correct/incorrect positions over bins of confidence levels in Figure 21. The higher the confidence, the higher the ratio of the number of correct to incorrect inferences.

Exploiting this property of the neural network, we propose using a confidence threshold to dynamically switch between the neural network and the one-ply search for each position. If the confidence of the neural network for a position is higher than the threshold, then we accept the model's decision. Otherwise, we perform a one-ply search and use the search result.

Outcome	**Mean Confidence Correct**	**Mean Confidence Incorrect**
Draw	94.26%	74.86%
Win	98.54%	77.31%
Loss	97.71%	73.88%

Table 13: Mean Confidence Table. The columns are average confidences of the neural network for the correctly classified and misclassified positions, respectively. The rows show the mean for both cases with respect to different outcome classes.

6.4.1 Results of the Threshold Approach

Among thresholds 70%, 80% and 90%, 80% gives the best performance. Thus, we use 80% as the confidence threshold to test our algorithm on the 2-7 piece test set. Table 14 displays the statistics of using the model only, the search only and the

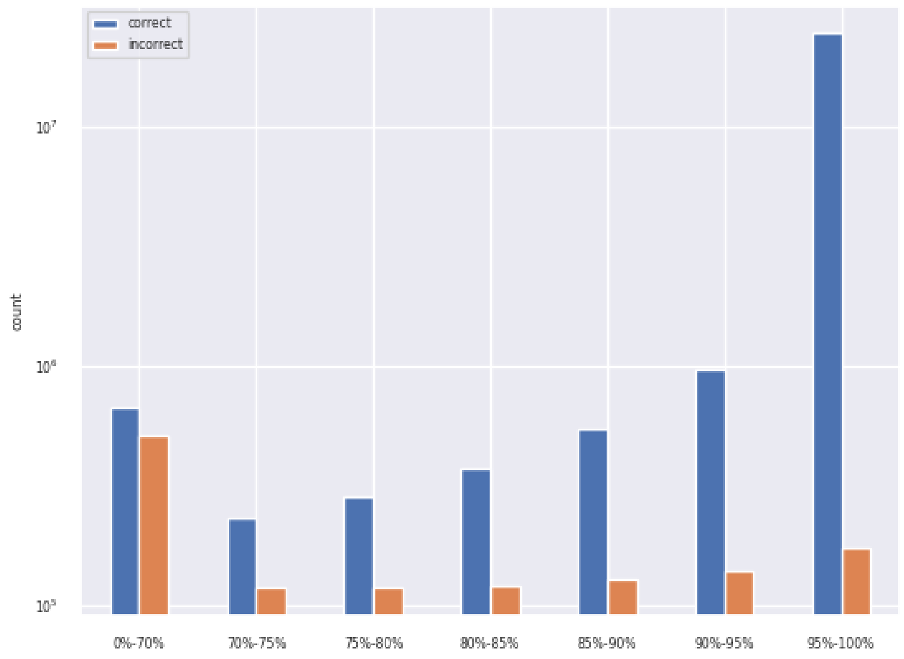


Figure 21: Distribution of Positions Across Confidence Levels.

threshold algorithm that merges the two dynamically. Interestingly, the search-only method performs worse than the model-only but complementing the model inference with search according to the threshold results in 55,441,847 correct mappings out of 57,782,317 positions in the test set – 280,192 (0.5%) more than model-only and 1,528,424 (2.65%) more than search-only. This evidence supports the effectiveness of our threshold algorithm.

In this experiment, we have proposed and empirically verified a simple algorithm that dynamically merges the strengths of the neural network and the one-ply search to create a predictor better than solely relying on either. Our algorithm exploits the classification confidence property of the ResNet and does not use any domain-specific checkers knowledge.

Method	**Correct**	**Total**	**Accuracy**
Model-only	55,161,655	57,782,317	95.46%
Search-only	53,913,423	57,782,317	93.30%
Above threshold - Model used	52,782,198	53,899,336	
Below threshold - Search used	2,659,649	3,882,981	
Threshold	55,441,847	57,782,317	**95.95%**

Table 14: Statistics of the Neural Network, the One-ply Search, and the Confidence Threshold Method.

6.5 Compression

We have demonstrated how good the neural network and the one-ply search are in capturing the CHINOOK endgame databases and how to improve them via our confidence threshold method. However, how effective are they quantitatively in terms of compression? We run an experiment on the 2-5 piece databases to answer this question. We use the entire 2-5 piece databases, split again according to a 7:1:2 ratio, then train our ResNet on the training set until convergence. The best accuracy achieved using the confidence threshold approach is 99.377%. If its performance is roughly the same on the complete 2-5 piece databases, it should get approximately 144,950,938 correct predictions out of 145,859,644 positions. That means it would get 908,706 incorrect predictions, accounting for 4.33 MB, if we store them in a separate table with 5 bytes per position: We use 5 bits for the location, one for the colour and one for the type (king/checker) for one piece of a position. Each position has at most five pieces on the board. Thus, 5 bytes per position is sufficient. The theoretical size of our ResNet is 2.13 MB. Therefore, the total size of the 2-5 piece endgame databases compressed according to our scheme is about 6.46 MB. The custom compression algorithm takes only 1,590,303 bytes (1.52 MB) to store the 2-5 piece databases. We thus conclude that the custom compression scheme is hard to beat with our more general neural network-based compression, despite the high accuracy achieved by the latter.

6.6 Summary

In this chapter, we have presented four experiments and their results to demonstrate the strengths and weaknesses of the neural network and the one-ply search in fitting the CHINOOK endgame databases and possible improvements. Our results show that:

- Training from a subset as small as 1% of the entire dataset is sufficient for the deep neural network to map above 90% of all the positions in the dataset to correct outcomes.
- The neural network trained from one database can perform reasonably well out-of-the-box on another database it has never seen. However, it can achieve comparable performance to the ones trained directly on the target database by reusing the learned convolutional features, and further training (fine-tuning) only its classifier's parameters for a few epochs on the target database.
- There is a positive correlation between the neural network's chance to correctly predict a position's outcome and its confidence, defined as the highest probability from the distribution output by the model on that position. By exploiting this correlation, we have developed a mechanism to dynamically switch between the inference made by the neural network and the one-ply search. The resultant algorithm is superior to exclusively applying the neural network or the one-ply search.

In addition, we also report through experimentation that our compression scheme cannot beat the custom algorithm used to compress the CHINOOK endgame databases, although the accuracy of this trained model is over 99%.

7 Conclusion

This project investigates applying deep neural networks to endgame databases for the game of checkers. Our objectives include compressing the endgame databases of Chinook, a checkers playing program and extracting features from the neural network. The implication of the first objective is a general algorithm that is more memory-efficient than a highly customized lookup table. The second one would potentially advance human players' understanding of the game and may one day help people discover novel insights and strategies. We also offer perspectives on how our problem differentiates from "common" supervised learning tasks and self-play algorithms.

In accomplishing our goals, we have developed a complete pipeline to interface with the Chinook endgame databases and generate datasets compatible with our deep learning frameworks. We have discovered and implemented the most suitable residual neural network that balances the computational and memory overhead with the performance through experimentation. We also provide the best optimizer and loss function to our knowledge alongside the optimized hyperparameters through empirical studies.

The four experiments we have conducted are – sampling, inter-database generalization, transfer learning, and merging the model and search inference. They demonstrate three primary discoveries:

1. Although the size of the endgame databases is prohibitively large for exhaustive enumeration when training the neural network, we find that the model can achieve impressive accuracy even with a tiny random subset. Furthermore, a one-ply search can usually perform better when the model is imperfect.

2. Pre-training a neural network on one database and applying it to a distinct one gives fair but less impressive results. However, by fine-tuning only the classifier of the pre-trained model for a few epochs with the target database, it quickly attains similar accuracy as the model trained directly on the target database. We argue that it is evidence that the neural network can learn general features of checkers that are universally applicable to any database of Chinook.

3. We define the term confidence and have successfully observed that the neural network is more accurate in the positions where it is confident. Considering that the one-ply search can complement the model's imperfection, we propose to use a confidence threshold to dynamically switch between the model and the one-ply search and succeed in getting an accuracy higher than either.

We conclude that the custom hand-tuned compression scheme is hard to beat with neural networks, especially for smaller databases. Nevertheless, the gap is not impossible to close. Our work supports that the neural network is likely to discover general patterns of checkers, but we are unclear on how to acquire that knowledge such that humans can interpret it. We leave this question to future work.

Other future work shall improve and extend our current work methodologically and experimentally:

- Methodologically, one could propose a more advanced neural network architecture that achieves a better generalization without increasing its size. In addition, one can test or invent alternative loss functions and optimizers. Another direction is work on the search algorithm because the current system uses the most naive one-ply minimax search. Furthermore, we lack a way to automatically and intelligently select a threshold for confidence. The current version relies on trial and error. Last but not least, one could attempt to develop knowledge acquisition techniques to understand the patterns learned by the neural network.

- Experimentally, it would be interesting to scale to larger databases, given more computational resources and efficient implementations. We also need novel experiments to verify our hypothesis regarding the properties of the learned features when we train the models on different databases. Do the positions with more pieces on the board provide more signals resulting in better features? Setting up experiments to investigate why Draw is more difficult to learn than the other two outcomes is also an interesting direction.

Our work is the first attempt to compress and extract features from endgame databases using deep learning techniques to our best knowledge. The lack of previous work poses challenges in this field. On the other hand, this research direction offers numerous opportunities and potential. We hope our work provides motivation and valuable insights to future projects.

References

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2015. URL: `http://download.tensorflow.org/paper/whitepaper2015.pdf`.

[2] L Victor Allis, Maarten van der Meulen, and H Jaap Van Den Herik. "Proof-Number Search". In: *Artificial Intelligence* 66.1 (1994), pp. 91–124.

[3] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. "Solving Hex: Beyond Humans". In: *Computers and Games*. Ed. by H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–10. ISBN: 978-3-642-17928-0.

[4] Yuri Sousa Aurelio et al. "Learning from Imbalanced Data Sets with Weighted Cross-Entropy Function". In: *Neural Processing Letters* 50.2 (Oct. 1, 2019), pp. 1937–1949. DOI: `10.1007/s11063-018-09977-1`.

[5] Igor Babuschkin et al. *The DeepMind JAX Ecosystem*. 2020. URL: `http://github.com/deepmind`.

[6] James Bradbury et al. *JAX: Composable Transformations of Python+NumPy Programs*. Version 0.3.13. 2018. URL: `http://github.com/google/jax`.

[7] Murray Campbell, A.J. Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/S0004-3702(01)00129-1`.

[8] Sharan Chetlur et al. "cuDNN: Efficient Primitives for Deep Learning". In: *CoRR* abs/1410.0759 (2014). arXiv: `1410.0759`.

[9] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." In: *Journal of Machine Learning Research* 12.7 (2011).

[10] Alhussein Fawzi et al. "Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning". In: *Nature* 610.7930 (Oct. 1, 2022), pp. 47–53. DOI: `10.1038/s41586-022-05172-4`.

[11] Ralph Gasser. “Solving Nine Men’s Morris”. In: *Computational Intelligence* 12.1 (1996), pp. 24–41. DOI: https://doi.org/10.1111/j.1467-8640.1996.tb00251.x.

[12] Hossein Gholamalinezhad and Hossein Khosravi. “Pooling Methods in Deep Neural Networks, A Review”. In: *arXiv preprint arXiv:2009.07485* (2020).

[13] Xavier Glorot and Yoshua Bengio. “Understanding the Difficulty of Training Deep Feedforward Neural Networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[14] *Glossary*. URL: https://docs.python.org/3/glossary.html#term-duck-typing.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[16] Charles R. Harris et al. “Array Programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[17] Ryan B. Hayward and Bjarne Toft. *Hex: The Full Story*. CRC Press, 2019.

[18] Kaiming He and Jian Sun. “Convolutional Neural Networks at Constrained Time Cost”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015.

[19] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[20] Tom Hennigan et al. *Haiku: Sonnet for JAX*. Version 0.0.9. 2020. URL: http://github.com/deepmind/dm-haiku.

[21] Geoffrey Hinton et al. "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 82–97. DOI: `10.1109/MSP.2012.2205597`.

[22] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: `10.1162/neco.1997.9.8.1735`.

[23] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer Feedforward Networks are Universal Approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/0893-6080(89)90020-8`.

[24] John D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: `10.1109/MCSE.2007.55`.

[25] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 448–456.

[26] Diederik P Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[27] Akihiro Kishimoto et al. "Depth-First Proof-Number Search with Heuristic Edge Cost and Application to Chemical Synthesis Planning". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: `https://proceedings.neurips.cc/paper/2019/file/4fc28b7093b135c21c7183ac07e928a6-Paper.pdf`.

[28] Günter Klambauer et al. "Self-Normalizing Neural Networks". In: *Advances in Neural Information Processing Systems* 30 (2017).

[29] Donald E. Knuth and Ronald W. Moore. "An Analysis of Alpha-Beta Pruning". In: *Artificial Intelligence* 6.4 (1975), pp. 293–326. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/0004-3702(75)90019-3`.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386.

[31] Robert Lake, Jonathan Schaeffer, and Paul Lu. "Solving Large Retrograde Analysis Problems Using a Network of Workstations". In: (1993). DOI: 10.7939/R3J09W57M.

[32] Yann LeCun, Yoshua Bengio, et al. "Convolutional Networks for Images, Speech, and Time Series". In: *The Handbook of Brain Theory and Neural Networks* 3361.10 (1995), p. 1995.

[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521.7553 (May 1, 2015), pp. 436–444. DOI: 10.1038/nature14539.

[34] Yann LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[35] Tsung-Yi Lin et al. "Focal Loss for Dense Object Detection". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017.

[36] Liyuan Liu et al. "On the Variance of the Adaptive Learning Rate and Beyond". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=rkgz2aEKDr.

[37] Ilya Loshchilov and Frank Hutter. "Fixing Weight Decay Regularization in Adam". In: *CoRR* abs/1711.05101 (2017). arXiv: 1711.05101.

[38] David Luebke. "CUDA: Scalable Parallel Programming for High-Performance Scientific Computing". In: *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 2008, pp. 836–838. DOI: 10.1109/ISBI.2008.4541126.

[39] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. "Autograd: Effortless Gradients in NumPy". In: *ICML 2015 AutoML Workshop*. Vol. 238. 2015, p. 5.

[40] Charles C. Margossian. “A Review of Automatic Differentiation and Its Efficient Implementation”. In: *WIREs Data Mining and Knowledge Discovery* 9.4 (2019), e1305. DOI: https://doi.org/10.1002/widm.1305.

[41] Diganta Misra. “Mish: A Self Regularized Non-Monotonic Neural Activation Function”. In: *arXiv preprint arXiv:1908.08681* (2019).

[42] John Nunn. *Secrets of Minor-piece Endings*. Batsford chess book. Batsford, 1995. ISBN: 9780713477276. URL: https://books.google.ca/books?id=BWBoAAAACAAJ.

[43] John Nunn. *Secrets of Pawnless Endings*. Batsford chess book. Batsford, 1994. ISBN: 9780713475081. URL: https://books.google.ca/books?id=yba7AAAACAAJ.

[44] John Nunn. *Secrets of Rook Endings*. Batsford chess book. Batsford, 1992. ISBN: 9780713471649. URL: https://books.google.ca/books?id=eeQAAQAACAAJ.

[45] Sinno Jialin Pan and Qiang Yang. “A Survey on Transfer Learning”. In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (2010), pp. 1345–1359. DOI: 10.1109/TKDE.2009.191.

[46] Zheyi Pan et al. “Urban Traffic Prediction from Spatio-Temporal Data Using Deep Meta Learning”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD ’19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 1720–1730. ISBN: 9781450362016. DOI: 10.1145/3292500.3330884.

[47] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[48] Karl Pearson. *On the Theory of Contingency and Its Relation to Association and Normal Correlation*. Vol. 1. Cambridge University Press, 1904.

[49] J.W. Romein and H.E. Bal. “Awari Is Solved”. In: *ICGA Journal* 25.3 (2002), pp. 162–165.

[50] J.W. Romein and H.E. Bal. “Solving Awari with Parallel Retrograde Analysis”. In: *Computer* 36.10 (2003), pp. 26–33. DOI: 10.1109/MC.2003.1236468.

[51] Frank Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.” In: *Psychological Review* 65.6 (1958), p. 386.

[52] Amit Sabne. *XLA : Compiling Machine Learning for Peak Performance*. 2020.

[53] Arthur L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229. DOI: 10.1147/rd.33.0210.

[54] Arthur L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers. II—Recent Progress”. In: *IBM Journal of Research and Development* 11.6 (1967), pp. 601–617. DOI: 10.1147/rd.116.0601.

[55] Jonathan Schaeffer et al. “Building the Checkers 10-Piece Endgame Databases”. In: *Advances in Computer Games: Many Games, Many Challenges*. Ed. by H. Jaap Van Den Herik, Hiroyuki Iida, and Ernst A. Heinz. Boston, MA: Springer US, 2004, pp. 193–210. ISBN: 978-0-387-35706-5. DOI: 10.1007/978-0-387-35706-5_13.

[56] Jonathan Schaeffer et al. “Checkers Is Solved”. In: *Science* 317.5844 (2007), pp. 1518–1522. DOI: 10.1126/science.1144079.

[57] Jonathan Schaeffer et al. “CHINOOK The World Man-Machine Checkers Champion”. In: *AI Magazine* 17.1 (Mar. 1996), p. 21. DOI: 10.1609/aimag.v17i1.1208.

[58] Jonathan Schaeffer et al. “Man Versus Machine for the World Checkers Championship”. In: *AI Magazine* 14.2 (Mar. 1993), p. 28. DOI: 10.1609/aimag.v14i2.1040.

[59] Jürgen Schmidhuber. “Deep Learning in Neural Networks: An Overview”. In: *Neural Networks* 61 (2015), pp. 85–117. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003.

[60] Samuel S. Schoenholz and Ekin D. Cubuk. "JAX M.D. A Framework for Differentiable Physics". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020. URL: https://papers.nips.cc/paper/2020/file/83d3d4b6c9579515e1679aca8cbc8033-Paper.pdf.

[61] Julian Schrittwieser et al. "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model". In: *Nature* 588.7839 (Dec. 1, 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4.

[62] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 1, 2016), pp. 484–489. DOI: 10.1038/nature16961.

[63] David Silver et al. "Mastering the Game of Go without Human Knowledge". In: *Nature* 550.7676 (Oct. 1, 2017), pp. 354–359. DOI: 10.1038/nature24270.

[64] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf.

[65] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018. URL: http://incompleteideas.net/book/RLbook2020.pdf.

[66] Jiuqi Wang, Jonathan Schaeffer, and Martin Müller. *Deep Dive on Checkers Endgame Data.* to appear in IEEE Conference on Games (CoG) 2023.

A Appendix

A.1 Deep Learning Frameworks

In this work, the definition, implementation and training of deep neural networks are described for two deep learning frameworks, PyTorch and JAX. The former has been one of the most popular choices by practitioners and researchers in recent years, while the latter has been gaining popularity steadily among researchers since its release. While the two are both useful for a common purpose – training a deep neural network – they differ in various aspects examined in the following subsections.

A.1.1 Overview

PyTorch

PyTorch[47] is a full-fledged framework that includes modules for building neural networks, popular loss functions and optimizers, all under a common namespace. It was inspired and based on Torch, a scientific computing framework based on the Lua programming language. PyTorch provides both Python and C++ interfaces, although the former is the most polished and the dominant choice for its users. It has become one of the top choices for deep learning thanks to its ease of use, speed, and robust ecosystem.

JAX

JAX[6] is described as a high-performance machine-learning framework that combines Autograd[39] and XLA[52].

Autograd is a Python package capable of differentiating native Python and NumPy[16] code. It can take the gradients of functions written in Python and NumPy without the analytical form of the gradient function. Furthermore, unlike numerical differentiation techniques, which compute empirical gradients, Autograd computes the gradient function precisely.

XLA stands for accelerated linear algebra, a specialized compiler. It analyzes the computation graph, optimizes it and produces efficient machine code for devices such as CPU and GPU and custom accelerators such as TPU (tensor processing unit).

One of the main features of JAX is its composable transformation capability, where transformations like JIT(just-in-time compilation) and automatic differentiation can be arbitrarily composed. For instance, one may have defined a custom function in native Python. The gradient transformation can transform it into another one that computes its gradients. By composing the gradient function using JIT, the outcome is yet another function that retains the functionality but is optimized for performance.

JAX itself is not a complete deep-learning framework but more like a library for high-performance scientific computing. It lacks support for building and training neural networks. Fortunately, reliable external libraries Haiku[20] and Optax[5] are publicly available for this purpose. Furthermore, since JAX heavily emphasizes performance, it is also gaining popularity among a wide variety of scientific computing domains, such as differentiable physics[60].

A.1.2 Installation

When installing PyTorch on an Nvidia GPU-enabled machine, the distribution comes with built-in CUDA[38], cuDNN[8], etc. As a result, one can usually use PyTorch out of the box.

On the other hand, JAX only has the equivalent distributions for Linux machines. Users often need to set up the environment manually after the installation and could be frustrated by the technical challenges.

A.1.3 Programming Style

PyTorch follows the imperative programming paradigm, and Python users are typically comfortable with this style. Users abstract their deep neural networks into objects and train their models by modifying their internal states. Under this paradigm, most of the internal mechanisms get abstracted away. Users can quickly build, train and test their models in a few lines of function calls, and the framework manages everything in the background.

JAX employs the functional programming paradigm, where the users need to write pure functions. In writing a neural network, the state of the network, the forward propagation function and the gradient computation function are all separated. To do this requires more work on the user side, but it pays off later for a logically cleaner code and the applicability of JAX's composable transformations.

A.1.4 Tensor Representation

The fundamental data structure of PyTorch is a tensor (a multi-dimensional matrix containing elements of a single data type). Tensors are mutable data structures. Most tensor operations in PyTorch mimic the API of NumPy, and the conversion between NumPy arrays and PyTorch tensors needs to be explicit.

JAX represents tensors as JAX NumPy arrays. These are immutable types to retain the functional programming nature of JAX. JAX NumPy arrays employ duck-typing[14] to NumPy arrays. Users can make function calls on JAX NumPy arrays using the identical NumPy API most of the time to achieve the same effect. The conversion between the two types can be done implicitly.

A.1.5 Automatic Differentiation

There are mainly two modes of automatic differentiation (AD) – forward mode and reverse mode. The latter is commonly referred to as backpropagation when the target function outputs a real scalar. Since the loss functions in deep learning usually map to scalars, backpropagation virtually replaces reverse mode AD when discussing training a deep neural network. Forward mode AD calculates and keeps the value and the gradient of intermediate activations in the forward computation and repeats for each input variable. Reverse mode AD only keeps the intermediate values in the forward pass. Then it calculates the gradients and applies the chain rule in the backward step for each output variable. The selection between the two modes usually depends on the dimension of the function's input relative to its output[40].

PyTorch performs automatic differentiation (AD) by dynamically building a computation graph and then calculating and aggregating the gradients. In the earlier releases, it only supported reverse-mode AD. The latest releases include forward-mode AD in Beta testing. Computing higher-order gradients is not trivial in PyTorch and often requires writing loops.

In JAX, there is technically no concept of an explicit computation graph. The gradient transformation transforms the original function into another one that computes the first-order gradient with respect to the first argument of the function call. JAX has had support for both forward-mode and backward-mode AD since its release. Furthermore, higher-order differentiation is as easy as composing transformations in JAX.

A.1.6 Working with Accelerators

In PyTorch, the user needs to transfer data across different devices explicitly through code, for example, CPU to GPU and vice-versa. To use TPU, users need the PyTorch/XLA package that enables PyTorch models to run on XLA devices.

On the other hand, JAX is accelerator agnostic, meaning that the same code can run on any device without any change unless explicitly specified. Furthermore, optimized JAX code eventually compiles to XLA, so that it is compatible with CPU, GPU and TPU.