Technische Universität Berlin

Chair Neurotechnology

Faculty IV
Marchstraße 23
10587 Berlin
https://www.tu.berlin/neuro

Bachelor Thesis

Learning Explainable Move Prediction for Chess

Alexander Herbrich

Matriculation Number: 468688
October 29, 2024

Supervised by
Prof. Dr. Benjamin Blankertz

Secondary Supervisor
Dr. Stefan Fricke

Abstract

State-of-the-art chess engines, such as Stockfish and Leela Chess Zero, have achieved unparalleled performance by employing advanced models and techniques like neural networks and reinforcement learning. Despite their superhuman level of performance, these methods have essential limitations in terms of interpretability and energy-efficient training. This thesis proposes a novel approach to chess move prediction, that utilizes Bayesian statistics, factor graphs, and the sum-product algorithm. The proposed method emphasizes interpretability while also maintaining reasonably efficient training. Furthermore, the method introduces uncertainty measures into the prediction process, opening interesting avenues for further investigation. While the current model demonstrates high interpretability and reasonable accuracy, this work also explores potential enhancements to further improve accuracy. The thesis details the design, implementation, and testing of this move prediction function, emphasizing its ability to predict strong moves by learning from a dataset of high-Elo games. Additionally, it briefly discusses the model's potential to contribute to the future development of chess engines and, more broadly, to the advancement of interpretable artificial intelligence systems. By bridging the gap between raw computational power and human-like reasoning, this approach provides valuable insights into the decision-making process of AI systems, specifically in the context of chess move prediction.

Zusammenfassung

Hochmoderne Schachprogramme wie Stockfish und Leela Chess Zero haben durch den Einsatz fortschrittlicher Modelle und Techniken wie neuronaler Netzwerke und Reinforcement Learning eine beispiellose Leistung erreicht. Trotz ihres Erfolgs und ihrer übermenschlichen Leistungsfähigkeit weisen diese Methoden wesentliche Einschränkungen hinsichtlich Interpretierbarkeit und energieeffizientem Training auf. Diese Arbeit schlägt einen neuartigen Ansatz zur Vorhersage von Schachzügen vor, der auf Bayesscher Statistik, Faktorgrafen und dem Summen-Produkt-Algorithmus basiert. Der vorgeschlagene Ansatz legt den Schwerpunkt auf Interpretierbarkeit und gewährleistet gleichzeitig ein relativ effizientes Training. Darüber hinaus führt die Methode Unsicherheitsmaße in den Vorhersageprozess ein, was interessante Ansätze für weitere Untersuchungen eröffnet. Während das aktuelle Modell eine hohe Interpretierbarkeit und eine solide Genauigkeit aufweist, untersucht diese Arbeit auch potenzielle Verbesserungen zur Steigerung der Genauigkeit. Die Arbeit beschreibt das Design, die Implementierung und die Testung dieser Zugvorhersagefunktion, mit einem Fokus auf ihre Fähigkeit, gute Züge nach dem Lernen aus einem Datensatz von Partien mit hohem Elo vorherzusagen. Darüber hinaus wird kurz das Potenzial des Modells erörtert, zur zukünftigen Entwicklung von Schachprogrammen beizutragen und im weiteren Sinne zur Entwicklung interpretierbarer KI-Systeme. Durch die Überbrückung der Lücke zwischen roher Rechenleistung und menschlichem Denken bietet dieser Ansatz wertvolle Einblicke in den Entscheidungsprozess von KI-Systemen, konkret im Kontext der Vorhersage von guten Schachzügen.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

Over the past few decades, chess engines have achieved remarkable progress. One of the most iconic moments in AI history occurred in 1997 when IBM's Deep Blue defeated the world chess champion Garry Kasparov in a standard six-game match — the first time a computer had ever accomplished such a feat [1]. Since then, advancements have led to state-of-the-art chess engines like Stockfish and Leela Chess Zero, which have widened the gap between human and machine performance even further[1]. Today, these engines are not only essential tools for analyzing expert-level chess games[2] but also for helping amateurs improve their skills - widely accepted as superhuman chess-playing machines. However, this raises intriguing questions: What makes chess engines so effective? And how interpretable are the decisions made by these engines, especially given that we rely on them to analyze weaknesses in human play?

1.2. Chess AI Landscape

Stockfish

Stockfish is considered one of the best chess engines in the world, consistently placing first in all major Top Chess Engine Competition (TCEC) events since 2020 [4]. Originally, Stockfish was a highly optimized, heuristic-based engine utilizing alpha-beta search with a complex handcrafted evaluation function [5, 6]. However, with the rise and popularity of neural networks, Stockfish underwent significant changes to its evaluation function in 2020. Initially developed in the context of computer shogi [7], the efficiently updatable neural network (NNUE) architecture was integrated into Stockfish, moving from a solely handcrafted evaluation function to a neural network enhanced solution, leading to a significant performance boost [8]. Over the next three months, Stockfish improved by over 100 Elo points—a leap equivalent to about two years of traditional development at the time [9].
The following is a brief description of the NNUE used in Stockfish, as outlined in the official Stockfish NNUE documentation [11]. The most notable neural network architecture

[1] This is evident from the significant advancements in AI, which have dramatically boosted the strength of chess engines. The increasing gap between humans and machines can be seen through Elo ratings and notable matches, such as Grandmaster Nakamura's AI-assisted 1.5-0.5 loss to Stockfish in 2014 [2]. For more on the impact of AI advancements on chess engine strength, see [3].

[2] Chess engines are routinely used during tournaments to provide real-time analysis, often highlighting mistakes or missed opportunities by the players.

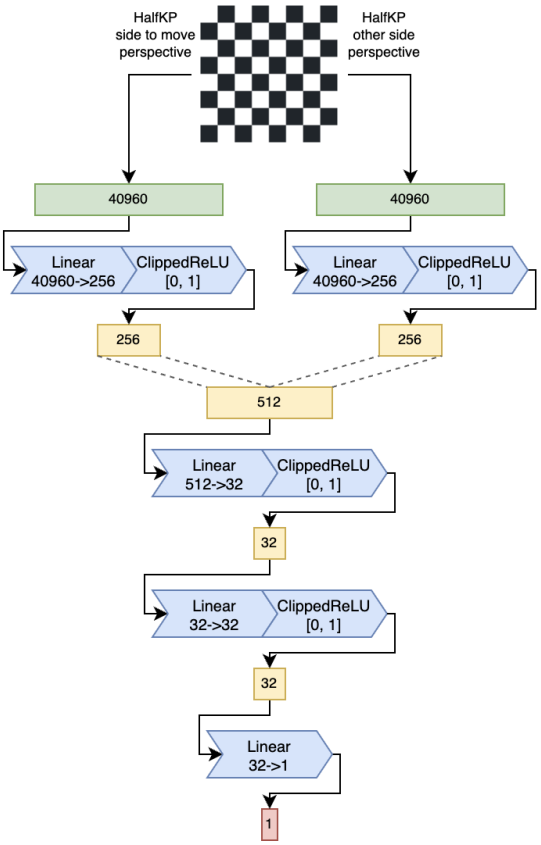


Figure 1.1.: Overview of the HalfKP architecture, the initial neural network architecture used in Stockfish. The network comprises of 2x40960x256, 512x32, 32x32, 32x1, totaling 20,988,960 parameters. *Figure source:* [10]

used in Stockfish is the HalfKP structure, as shown in Figure 1.1. In this architecture, the chessboard is encoded in a specific manner before being fed into the neural network. The encoding works as follows: On a chessboard, there are two kings — one white and one black — and five piece types for each color, i.e. ten non-king pieces. In the input layer there are two "halves" (one for each king), in which there is a node for every king-square to piece-square combination which will be set to either 0 or 1. For instance, if the white king is on *b3* and a black pawn is on *a6*, the node *b3-a6-bp* in the white king input layer "half" is set to 1. This process is repeated for all white king and piece combinations as well as all black king and piece combinations - all other input nodes are set to 0. This results in 64×64×10 nodes for the white king and an additional 64×64×10 nodes for the black king, totaling 81,920 nodes[3]. The structure and number of inner layers in the

[3]We were not able to find an official explanation for the architecture name "HalfKP," but based on our understanding, it likely refers to the two **halves** of the input layer, each centered around one of the **K**ings (considering all 64 possible positions for each). These halves are then combined with the positions of the ten non-king **P**ieces, thus the name "HalfKP".

neural network are somewhat specific to the exact architecture. In the final layer, the output layer, a single node corresponds to the evaluation of the position.
The neural network is trained using gradient descent, guided by a loss function that measures the difference between its output (the value of the single node in the last layer) and the evaluation generated by Stockfish's traditional alpha-beta search[4]. Essentially, the network learns to predict the evaluation value that the alpha-beta search would produce. When the NNUE-based evaluation is integrated into the alpha-beta search, the neural network effectively doubles the search depth, significantly enhancing the engine's performance.
A crucial feature that makes this neural network practical is its efficient updatability. The encoding of the chessboard sets specific input nodes to either 0 or 1, depending on the positions of pieces on the board. As a result, when a move is made or unmade in the context of search, only a very small number of input nodes need to update their values. This allows for an efficient updating process of the output node's value, without requiring a full pass through the neural network, which would be necessary in a traditional approach.

Leela Chess Zero

Leela Chess Zero is a relatively new chess engine that utilizes reinforcement learning and deep neural networks. In 2018, it made history as the first neural network-based chess engine to win the main event at the TCEC [4]. Developed as an open-source project, Leela Chess Zero was created to replicate and validate the remarkable results of Google's AlphaZero, whose source code was never publicly released despite its groundbreaking performance [6]. Like AlphaZero, Leela Chess Zero starts with only the basic rules of chess and learns to play entirely by playing against itself — a process known as reinforcement learning. The core algorithms and principles underlying AlphaZero were published in the seminal 2018 paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" [13]. Leela Chess Zero's structure closely resembles that of AlphaZero and is best understood through its underlying algorithm: Monte Carlo Tree Search (MCTS).
Monte Carlo Tree Search is an algorithm for evaluating game positions by simulating random play-outs, updating the values of encountered positions based on the game's outcome — win, loss, or draw. This iterative process, as described in detail by Browne et al. [15], consists of four main stages:

1. **Selection**: In this phase, the algorithm traverses the tree recursively, selecting child nodes at each level until it reaches an expandable node. An expandable node

[4]We were unable to find a source explicitly clarifying how these evaluations were derived. However, the official NNUE release states that the network was "trained on the evaluations of millions of positions at moderate search depth" [8], likely from games played against itself. As of 2024, on Stockfish's GitHub, there are downloadable datasets from past training [12], including evaluations from self-play of Stockfish in 2021 (which by then, was already able to use NNUE). Therefore, it is likely that NNUE was further trained on evaluations generated by alpha-beta search that had already integrated NNUE, leading to continuous improvement of the model.

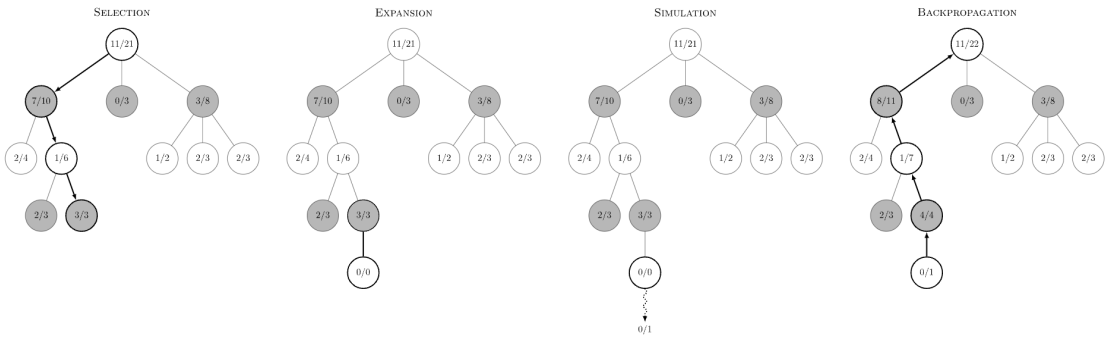


Figure 1.2.: Diagram illustrating the steps of Monte Carlo Tree Search (MCTS). The process begins with the **Selection** phase, where the tree is traversed from the root to a leaf node (= node that has unexpanded nodes, i.e. there exist moves that have never been played from the corresponding game state). In the **Expansion** phase, one of these unplayed moves is selected at random, played, and the resulting game state is added as a new node in the tree. The **Simulation** phase follows, where the outcome of the game is simulated by playing random moves from this node onward. Finally, in the **Backpropagation** phase, the results of the simulation are propagated back up the tree, updating the values of the nodes along the way. *Figure source:* [14]

corresponds to a non-terminal game state with unvisited child nodes. The selection policy determines which child node to choose at each level. A popular policy is the UCT (Upper Confidence Bound applied to Trees) policy. This policy selects the child node with the highest value (winning probability) plus a correction factor that favors less frequently explored moves, reflecting greater uncertainty. Formally:

$$\mathrm{UCT}(i) = v_i + 2 \cdot C \cdot \sqrt{\frac{2 \cdot \ln N}{n_i}} \tag{1.1}$$

where v_i is the value of child node i, n_i is the number of visits to child node i, N is the number of visits to the parent node, and $C > 0$ is a constant that balances *exploration* (trying new moves) and *exploitation* (choosing moves known to be effective). If $n_i = 0$ for a child node, $\mathrm{UCT}(i)$ is set to infinity, ensuring that every child node is expanded at least once.

2. **Expansion**: When an expandable node is reached, the algorithm selects an unvisited node (indicated by an infinite UCT value) and adds it to the tree.

3. **Simulation**: The game is played out randomly from the newly added leaf node in a process known as a simulation or random rollout. The result of this simulation can be a win, loss, or draw, typically represented as $+1$, -1, and 0, respectively.

4. **Backpropagation**: The outcome of the simulation is backpropagated up the tree, updating the values of the nodes along the path. Nodes associated with winning

outcomes have their values increased, while nodes associated with losing outcomes have their values decreased. During backpropagation, the value, such as +1 for a win, is propagated upwards while flipping it's sign, since a win for one player counts as a loss for the opponent.

As summarized in Klein's work "Neural Networks for chess" [6], Leela Chess Zero follows AlphaZero's idea, and enhances the Monte Carlo Tree Search by integration of neural networks, specifically in the following way:

1. **Value Approximation:** Instead of relying on random rollouts to estimate the values of leaf nodes, Leela Chess Zero employs a neural network to approximate these values. In theory this should lead to a faster convergence to the "real" values of the nodes.

2. **Dynamically Weighted UCT:** The expansion of already visited nodes is improved beyond the traditional UCT approach by using a dynamically weighted UCT policy. This policy adjusts the correction factor based on a prior probability learned by the neural network, reflecting how promising a move is for further exploration. The updated policy can be expressed as:

$$\text{Weighted-UCT}(i) = v_i + p_i \cdot \sqrt{\frac{\ln N}{n_i}} \tag{1.2}$$

where p_i represents the prior probability assigned by the neural network to the move corresponding to child node i. In this approach, the dynamic prior probability p_i replaces the constant factor C used in the classic UCT method as seen in Equation (1.1).

The neural network in Leela Chess Zero functions as follows: the input is a special encoding of the game state[5], and the output is divided into two main components, referred to as "heads": the **value head** and the **policy head**[6]. The value head is a single node and estimates the probability of winning from the given game state, effectively assessing the position's value[7]. The policy head, on the other hand, consists of many nodes and assigns a value to each possible move, determining the likelihood of selecting that move, with these values converted into probabilities using a softmax function.

[5]For a detailed explanation of how the game state is encoded, see Section 4.4.1 in Klein's "Neural Networks for Chess" [6]. In brief, the input layer comprises 112 planes, each of size 8x8 (which can be flattened into a single long array). The first 6 planes encode the positions of the player's six piece types, followed by 6 planes for the opponent's pieces, and this pattern is repeated 7 times to capture also the last 7 positions in the game. Additionally, special planes are included to encode information such as castling rights, repetition rules, and other game-specific details.

[6]Strictly speaking, as noted in Klein's work [6], the neural network originally extends AlphaZero's neural network by a third component, the "Moves Until Terminal State"-head. However, this component is excluded here because it is not essential for understanding the core architecture of Leela Chess Zero. Additionally, as Klein mentions, Leela Chess Zero's architecture is subject to ongoing development.

[7]Again, strictly speaking, Leela Chess Zero uses three nodes (win, loss, draw) instead of one node like AlphaZero, which is not essential for understanding the core architecture.

The neural network in Leela Chess Zero is trained through the following process: Initially, the network's weights are set randomly[8]. Training data is then generated by thousands of games of self-play[9]. During self-play, the values of each node (game state) in the search tree and the playout probabilities of each move while MCTS are recorded. These playout probabilities are calculated by dividing the number of visits to each node by the total visits to all possible moves. The network's weights are then adjusted using gradient descent, guided by a loss function that measures how closely the policy head's output matches the MCTS-derived playout probabilities and how well the value head's output (for game-state-inputs) aligns with actual game outcomes [13]. Over time, and with extensive training, the network converges to weights that somewhat accurately predict playout probabilities and game state values. Therefore in practice, this neural network enhanced MCTS, leads to more precise exploration of promising moves while minimizing the exploration of less relevant ones.

1.3. Objective

Over the past few years, neural networks have gained significant popularity due to their impressive performance across various tasks [17] - as also for chess engines, as described above. However, two major disadvantages have become increasingly apparent: **interpretability** and **energy efficiency** [18, 19]. The growing complexity and size of neural networks, along with the rapid development of new architectures — spawning an entire field of research known as deep learning — have exacerbated these issues. While neural networks are not a new concept, having been known since 1950s with the introduction of the perceptron proposed by Rosenblatt [20], it is only with the recent rise in availability of computational power that training these increasingly complex neural networks has become feasible.
The high energy consumption required for training neural networks has emerged as a significant limiting factor. For example, GPT-3's training demanded an immense amount of energy, estimated of consuming 1287MWh [21], equating to the daily power consumption of approximately 95.000 households - or in other words, the amount of power a household consumes over a duration of 260 years[10]. Similarly, while AlphaZero was initially touted as efficient in Google's press release, requiring only 24 hours to reach superhuman play [13], the actual computation involved was vast, involving 5000 custom-built first generation TPUs and 64 second generation TPUs [23]. Leela Chess Zero, particularly as an open-source project, also faced this challenge, which led to the development of distributed computing solutions. In Leela Chess Zero's case, users could download a client and con-

[8] A neural network is a parameterized function, where the term 'weight' refers to its parameters. The goal of training is to learn these weights, i.e. to find values for these weights such that inputs to the function map to meaningful outputs.

[9] In self-play, both players use Monte Carlo Tree Search (MCTS) enhanced by the **same** neural network. By default, the training client downloads the latest "best" version of the network, though one can manually download and specify specific versions for data generation through self-play [16].

[10] Assuming the average yearly consumption of 4.978 kWh for a Three- and more person household. *Source:* [22]

tribute their CPU power to the training process. By September 2024, after six years, Leela Chess Zero has played nearly 2 billion games against itself, averaging nearly one million games daily, and amassing countless CPU hours[11]. On a smaller scale, Stockfish also demands substantial computational resources for training its neural networks. Since the launch of FishTest, Stockfish's official testing framework, in 2013, over 15,000 CPU years[12] have been utilized, highlighting the immense computational requirements[13]. As a result, energy consumption has become a critical constraint in neural network research, driving a trend toward research for more efficient networks and machine learning methods [19, 26, 27].

Interpretability is another significant challenge for neural networks, especially when compared to more transparent machine learning techniques. One of the main obstacles is the sheer number of weights in a neural network, which makes understanding the model's internal workings difficult (for example, the HalfKP architecture in Stockfish has roughly 21 million weights; see Figure 1.1). Additionally, neural networks often consist of tens or even hundreds of layers, with architectures that can appear somewhat arbitrary, making them inherently less interpretable. Even when attempting to interpret the weights of a neural network, the task is far from straightforward. For instance, in the case of the widely-used MNIST dataset — a collection of digitized handwritten digits — one might expect the network's weights to correspond to interpretable features, such as detecting circles or lines in specific regions of an image, and combining these to form recognizable digits. However, in practice, these weights are often hard to interpret[14]. This "black-box" nature of neural networks has become a focal point in recent research [18]. The explainability of neural networks has become increasingly relevant in areas such as autonomous driving, and medical diagnostics, where understanding the decision-making process of neural networks is crucial [29].

This thesis addresses these issues by exploring a novel approach to chess move prediction - among other things through the learning of a board evaluation function. Despite the ever-growing, somewhat unnecessarily large, gap between computer and human chess play, the interpretability and efficiency of training chess engines have been ever since declining. This work proposes a solution aimed at achieving relatively high accuracy in chess move prediction while enhancing interpretability. Although the model seeks efficient training, we do not explore this as extensively as the interpretability aspect. Instead, we focus on avoiding the extensive computational resources typically required by large neural networks, which often demand hundreds or thousands of compute hours across multiple servers. The proposed approach leverages Bayesian inference combined with the sum-

[11] Numbers are taken from the official Leela Chess Zero training web interface. *Source:* [24]

[12] Numbers are taken from the official Stockfish's testing web interface (Fishtest). *Source:* [25]

[13] It is important to note that this includes non-NNUE-related commits to the code base, though isolating NNUE-specific usage is challenging.

[14] You can observe this yourself by training a neural network and visualizing the weights in each layer. The resulting weights will often seem random and difficult to interpret. While we do not conduct this visualization here, a demonstration of this can be found in the popular YouTube series on neural networks by 3Blue1Brown (Grant Sanderson), specifically in the video "Gradient descent, how neural networks learn | Chapter 2, Deep learning" at minute 14:02. *Source:* [28]

product algorithm applied to specially designed factor graphs. As we will see, by design, these factor graphs are inherently more interpretable than traditional neural network models. While this method may result in less optimal play, it compensates by offering a more transparent alternative. Additionally, this method provides a unique measure of uncertainty in the learned move prediction function, demonstrating a promising path for future research.

1.4. Scope

While this thesis aims to develop more interpretable and potentially more efficient models for chess engines, the focus is not on achieving the highest possible accuracy or creating a superhuman-performing engine like Stockfish. Instead, the thesis concentrates on a specific aspect: a move prediction function. The focus will not extend to other components, such as search algorithms like Alpha-Beta or Monte Carlo Tree Search, which are vast areas of study with numerous optimizations that fall beyond the scope of this work.

The accuracy of the proposed model will not be evaluated based on the performance of the chess engine against other engines, as this approach would not allow for the isolated testing of the move prediction function which is the primary focus of this work. Chess engine performance is influenced by many factors, including board representation, search algorithms and heuristics, and SIMD optimizations (**S**ingle **I**nstruction, **M**ultiple **D**ata), making it difficult to attribute performance gains solely to the learnt move prediction.

Rather, to assess the accuracy of the model, the move prediction function will be trained on a dataset of high-Elo human chess games, and its performance will be measured by how accurately it predicts the moves made in those games. This approach effectively evaluates the model's imitation play — how well it can replicate observed game-play after training. In the later chapters, we will explore potential methods to enhance the model's accuracy and discuss how this new move prediction function could be integrated into search algorithms, considering the uncertainty measures introduced by the model. Overall, we will use chess as an application for this explainable prediction approach, without attempting to create a fully superhuman chess engine, which would exceed the scope of this thesis.

Regarding energy efficiency, our focus will be on the "trainability" of the proposed model, as well as its implications for larger models. However, we will not extend our observations to a comprehensive evaluation of the method's overall efficiency or how training efficiency compares to prediction accuracy. Additionally, we will not directly compare energy consumption to other methods, such as neural networks, as this would require a direct comparison between the proposed method and neural network approaches using the same model design — something that is not a key focus of our work. Instead, our emphasis is on developing a model that enhances interpretability while remaining (accurate and) efficient, specifically in terms of not requiring extensive training to converge to a "trained" model.

1.5. Outline

We will begin with a chapter on the **Fundamentals**, providing a brief overview of probability theory and its associated notation. Next, we will introduce factor graphs and their graphical representations to simplify our discussion of the model. Additionally, we will explore the sum-product algorithm, detailing its operation and practical considerations. We will focus particularly on the Normal distribution, which will be prominently used in our model.
In the next chapter, the **Concept** chapter, we will discuss the idea behind the model and introduce two specific model architectures alongside their graphical representations. We will delve into the specific factors in the corresponding factor graphs and how updates are performed within the context of the sum-product algorithm. During this process, we will derive the factor equations and highlight the inherent interpretability these factors offer. We will also highlight optimizations and practical details necessary for the efficient execution of the sum-product algorithm.
In the next chapter on **Implementation**, we will cover the programming languages and environments utilized. We will also introduce *Poppy*, the custom chess engine developed for this thesis, designed to be efficient with fast move generation through advanced techniques to support the training required. Further, we will discuss the implementation of the factor graph and its components, the sum-product algorithm and also the testing environment and the methods used to assess model accuracy.
The following chapters, **Results** and **Discussion**, will present and interpret the outcomes of the move prediction function, exploring it's implications, limitations and possible improvements.
In the final chapter, the **Conclusion**, we will summarize the thesis, provide closing remarks, and highlight potential future directions.

2. Fundamentals

In this chapter, we will cover the fundamental concepts and notation necessary for understanding the model described in the subsequent sections. The explanations and concepts presented here primarily draw from Bishop's "Pattern Recognition and Machine Learning" [30], as well as Ralf Herbrich's lectures for the course "Introduction to Probabilistic Machine Learning" at the Hasso Plattner Institute in Potsdam.[1]

2.1. Basic Concepts in Bayesian Statistics

Definition 1 (Random Variable)**.** A random variable X is a real-valued function $X : \Omega \to \mathbb{R}$ that assigns each possible outcome A from sample space Ω a real value $x \in \mathbb{R}$. In essence, a random variable translates outcomes from the sample space into real numbers. If the image (see Equation (2.1)) of a random variable X is finite, it is referred to as a *discrete* random variable. Conversely, if the image of X is infinite, it is referred to as a *continuous* random variable. Formally, we define the image of a function $f : A \to B$ as:

$$\text{image}\,(f) := \{b \in B \mid \exists a \in A \text{ s.t. } f(a) = b\}\,. \tag{2.1}$$

Example 1.1. Consider tossing a coin N times. Let the random variable X represent the number of heads that appear. If $N = 2$, the mapping $X : \Omega \to \mathbb{R}$ is as follows:

$$(\text{tail}, \text{tail}) \mapsto 0\,, \qquad (\text{tail}, \text{head}) \mapsto 1\,, \qquad (\text{head}, \text{tail}) \mapsto 1\,, \qquad (\text{head}, \text{head}) \mapsto 2\,.$$

Using (2.1) it follows that $\text{image}\,(X) = \{0, 1, 2\}$, i.e. X is a *discrete* random variable.

Definition 2 (Probability)**.** Probability is a measure of how likely it is for an event A from sample space Ω to occur, and is represented by a number $P(A)$ ranging from 0 and 1. A probability of 0 means the event is impossible, conversely a probability of 1 means the event is certain. For a random variable X, probability relates to the likelihood that X takes on a specific value x, written as $P(X = x)$.

- For *discrete* random variables, the likelihood of X taking on a value x is described by the *probability mass function*, as detailed in Definition 3.
- For *continuous* random variables, the likelihood of X falling within a particular range is given by the *probability density function*, as detailed in Definition 4.

[1] Video footage of all lectures in the summer term 2024 can be found here [31].

Definition 3 (Probability Mass Function)**.** The probability mass function, or for short PMF, is a function $p : \mathbb{R} \to [0, 1]$ that assigns each value $x \in \text{image}(X)$ the probability $p(x) \in [0, 1]$ that the random variable X takes on the value x. The PMF is defined only for discrete random variables. The probabilities of all possible values $x \in \text{image}(X)$ must **sum** to 1:

$$\sum_{x \in \text{image}(X)} p(x) = 1\,. \tag{2.2}$$

Definition 4 (Probability Density Function)**.** The probability density function, or for short PDF, is the continuous counterpart of the probability mass function. Since the image of a continuous random variable X is infinite, the probability of any single outcome $x \in \mathit{image}(X)$ is technically zero. Therefore, instead of assigning probabilities to x, the probability density function provides a relative likelihood for a continuous random variable taking on the value x. More generally, the probability density function is a function $f : \mathbb{R} \to \mathbb{R}^{\geqslant 0}$ that maps to non-negative real values, which unlike the probability *mass* function, possibly exceed the range $[0, 1]$. Similarly to how the probability *mass* function must sum to 1, as seen in Equation (2.2), the probability density function must **integrate** to 1:

$$\int_{-\infty}^{\infty} f(x)\ \mathrm{d}x = 1\,. \tag{2.3}$$

The probability density function allows us to calculate the probability that X falls within a range $[a, b]$ through integration:

$$P(a \leqslant X \leqslant b) = \int_a^b f(x)\ \mathrm{d}x\,.$$

Intuitively, the probability of X taking on a specific value $x \in \mathbb{R}$ can be seen as the integration over an infinitely small interval:

$$P(X = x) = \lim_{\Delta \to 0} \left[\int_x^{x+\Delta} f(t)\ \mathrm{d}t \right].$$

Remark 1. In this work, we use the terms *probability distribution*, *probability mass function*, and *probability density function* interchangeably. The term *distribution* is used more broadly to refer to both the PMF for discrete random variables and the PDF for continuous random variables. In some contexts, we will simply use the term *distribution* when referring to the distribution of a random variable X, without explicitly specifying whether it is a PMF or PDF.

Remark 2. In this work, we will use the following notations for summation and integration, as demonstrated in Equations (2.2) and (2.3), respectively:

$$\int f(x)\,\mathrm{d}x \qquad \text{(for integration over } x \in \mathbb{R})\,,$$
$$\sum_x x \cdot f(x) \qquad \text{(for summation over } x \in \text{image}(X))\,.$$

These notations will be used when the context makes it clear that we are summing or integrating over all possible values of x, without explicitly specifying the domain or range of x.

Definition 5 (Expectation and Variance)**.** Given a continuous random variable X over sample space $\mathbb{R}$ with density $p(\cdot)$ and a discrete random variable Y over sample space Ω with probability mass function $q(\cdot)$, the expectation $\mathbb{E}[\cdot]$ and variance $\mathbb{V}[\cdot]$ are defined by

$$\mathbb{E}[X] = \int x \cdot p(x) \, \mathrm{d}x \qquad \mathbb{E}[Y] = \sum_y y \cdot q(y)\,, \tag{2.4}$$

$$\mathbb{V}[X] = \int (x - \mathbb{E}[X])^2 \cdot p(x) \, \mathrm{d}x \qquad \mathbb{V}[Y] = \sum_y (y - \mathbb{E}[Y])^2 \cdot q(y)\,. \tag{2.5}$$

Example 5.1. Consider the case of a slot machine. Intuitively, the expectation answers the question: *"What is the average return I can expect?"*, whereas the variance addresses the question: *"How much do individual outcomes tend to deviate from the expected value?"*.

Remark 3. Note that we introduced the notation $p(\cdot)$. This dot notation will be used throughout this thesis to indicate that we are not specifying the particular input. The input could be any variable, such as x, y, z, and so on. This notation will be used, especially in contexts where the specific variable is clear from the context.

Definition 6 (Marginal, Conditional, and Joint Probability Distributions)**.** Let X and Y be two random variables.

1. **Marginal Distribution:** The *marginal* distribution of X, or simply distribution of X, denoted $p(x)$, represents the probability distribution of X regardless of the value of Y.
2. **Conditional Probability:** The *conditional* probability, denoted $p(x \mid y)$, represents the probability distribution of X given that Y is known to take a specific value y.
3. **Joint Distribution:** The joint distribution, denoted $p(x, y)$, describes the probability that both X and Y simultaneously take on specific values x and y.

The relationship between the joint distribution, conditional probability, and marginal distributions is given by:

$$p(x, y) = p(x \mid y) \cdot p(y) = p(y \mid x) \cdot p(x)\,. \tag{2.6}$$

Definition 7 (Law of Total Probability)**.** The Law of Total Probability provides a method for calculating the marginal distribution of a random variable X by considering all possible values of another random variable Y. It states that the distribution of X can

be obtained by summing (or integrating, in the continuous case) over all possible values of Y. Specifically, for discrete and continuous random variables, the law is expressed as follows:

$$p(x) = \sum_y p(x \mid y) \cdot p(y) = \sum_y p(x, y) \quad p(x) = \int p(x \mid y) \cdot p(y) \, \mathrm{d}y = \int p(x, y) \, \mathrm{d}y \,. \tag{2.7}$$

Definition 8 (Bayes Theorem)**.** Bayes Theorem provides a method for calculating the probability distribution $p(y \mid x)$, given $p(x \mid y)$ and $p(y)$. Specifically, the law is expressed as follows:

$$p(y \mid x) = \frac{p(x \mid y) \cdot p(y)}{p(x)} \,. \tag{2.8}$$

Using Equation (2.7) for discrete and continuous random variables, the formula extends to

$$p(y \mid x) = \frac{p(x \mid y) \cdot p(y)}{\sum_y p(x \mid y) \cdot p(y)} \qquad \text{and} \qquad p(y \mid x) = \frac{p(x \mid y) \cdot p(y)}{\int p(x \mid y) \cdot p(y) \, \mathrm{d}y} \,. \tag{2.9}$$

Definition 9 (Prior, Posterior, and Likelihood)**.** In the context of Bayes' Theorem, we commonly refer to three key concepts: *prior*, *posterior*, and *likelihood*. These terms are defined as follows:

$$\underbrace{p(y \mid x)}_{\text{posterior}} \propto \underbrace{p(x \mid y)}_{\text{likelihood}} \cdot \underbrace{p(y)}_{\text{prior}} \,. \tag{2.10}$$

Notice that we describe this relationship as *proportional to* ($\propto$), since we ignore the normalization term $p(x)$ (see Equation (2.8)).

Definition 10 (One-Dimensional Gaussian Distribution)**.** A continuous random variable $X \in \mathbb{R}$ is said to be distributed according to a one-dimensional Gaussian distribution if its probability density function is given by either $\mathcal{N}\left(\cdot; \mu, \sigma^2\right)$ or $\mathcal{G}\left(\cdot; \tau, \rho\right)$, where the parameters are defined as $\mu \in \mathbb{R}$, $\sigma \in \mathbb{R}^+$, $\tau \in \mathbb{R}$, and $\rho \in \mathbb{R}^+$. The probability density functions are expressed as follows:

$$\mathcal{N}\left(x; \mu, \sigma^2\right) := \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \,, \tag{2.11}$$

$$\mathcal{G}\left(x; \tau, \rho\right) := \sqrt{\frac{\rho}{2\pi}} \cdot \exp\left(-\frac{\tau^2}{2\rho}\right) \cdot \exp\left(\tau \cdot x - \rho \cdot \frac{x^2}{2}\right) \,. \tag{2.12}$$

If $\mu = \tau = 0$ and $\sigma^2 = \rho = 1$ we simply write

$$\mathcal{N}\left(x\right) := \mathcal{N}\left(x; 0, 1\right) \,, \tag{2.13}$$

$$\mathcal{G}(x) := \mathcal{G}\left(x; 0, 1\right) \,. \tag{2.14}$$

These two formulations are equivalent under the transformations:

$$\tau = \frac{\mu}{\sigma^2} \qquad \text{and} \qquad \rho = \sigma^{-2} \,, \qquad \text{or} \tag{2.15}$$

$$\mu = \frac{\tau}{\rho} \qquad \text{and} \qquad \sigma^2 = \rho^{-1} \,. \tag{2.16}$$

Additionally, if $X \sim \mathcal{N}\left(\cdot; \mu, \sigma^2\right)$, then

$$\mathbb{E}[X] = \mu \qquad \text{and} \qquad \mathbb{V}[X] = \sigma^2. \tag{2.17}$$

We refer to μ as the *mean* and σ^2 as the *variance*, as is standard. We denote τ as the *precision-mean* and ρ as the *precision*.

Remark 4. The notation $X \sim \mathcal{D}$ is used to indicate that the random variable X follows the probability distribution $\mathcal{D}$. In this context, the symbol $\sim$ is read as "is distributed according to."

Theorem 1 (Gaussian Multiplication Theorem)**.** *Given two Gaussian densities* $\mathcal{G}(\cdot; \tau_1, \rho_1)$ *and* $\mathcal{G}(\cdot; \tau_2, \rho_2)$ *we have*

$$\mathcal{G}(x; \tau_1, \rho_1) \cdot \mathcal{G}(x; \tau_2, \rho_2) = c \cdot \mathcal{G}(x; \tau_1 + \tau_2, \rho_1 + \rho_2) \,, \tag{2.18}$$

where c is a normalization constant. This means that multiplying two Gaussian densities results in a (potentially) non-normalized Gaussian density, specifically one where the new precision-mean is the sum of the precision-means, and the new precision is the sum of the precisions.

Proof. For a detailed proof, see Appendix A. □

Theorem 2 (Gaussian Division Theorem)**.** *Given two Gaussian densities* $\mathcal{G}(\cdot; \tau_1, \rho_1)$ *and* $\mathcal{G}(\cdot; \tau_2, \rho_2)$ *with* $\rho_1 > \rho_2$

$$\frac{\mathcal{G}(x; \tau_1, \rho_1)}{\mathcal{G}(x; \tau_2, \rho_2)} = c \cdot \mathcal{G}(x; \tau_1 - \tau_2, \rho_1 - \rho_2) \,. \tag{2.19}$$

where c is a normalization constant. This means that dividing two Gaussian densities results in a (potentially) non-normalized Gaussian density, specifically one where the new precision-mean is the difference of the precision-means, and the new precision is the difference of the precisions.

Proof. For a detailed proof, see Appendix A. □

Limit Gaussian Distributions There are two limit cases we would like to consider for $\mathcal{N}\left(\cdot; \mu, \sigma^2\right)$:

1. **Dirac Delta**. The limit case of σ^2 converging to zero and $\mu = 0$. The resulting function is called a Dirac delta and is formally defined as follows[2].

   Definition 11 (Dirac Delta)**.** The Dirac delta function, denoted $\delta(\cdot)$, is a normalized function defined as zero everywhere except at 0. Formally, it is defined by

   $$\delta(x) := \lim_{\sigma^2 \to 0} \mathcal{N}\left(x; 0, \sigma^2\right) \,. \tag{2.20}$$

[2]Note that if one seeks a Dirac delta with mean $\mu \neq 0$, this can be realized by subtracting the mean μ from the argument x because of the identity $\mathcal{N}\left(x; \mu, \sigma^2\right) = \mathcal{N}\left(x - \mu; 0, \sigma^2\right)$.

2. **Uniform**. The limit case of σ^2 converging to $+\infty$ and $\mu = 0$. The resulting function is effectively a constant function and is formally defined as follows.

   Definition 12 (Gaussian Uniform)**.** The Gaussian uniform function $\mathcal{U}(\cdot)$ is a normalized function defined as constant everywhere. Formally, it is defined by

$$\mathcal{U}(x) := \lim_{\sigma^2 \to +\infty} \mathcal{N}\left(x; 0, \sigma^2\right) . \tag{2.21}$$

Cumulative Distribution Function An important quantity is the cumulative distribution function (CDF) of the Gaussian density.

Definition 13 (Gaussian Cumulative Distribution Function)**.** Given parameters $\mu \in \mathbb{R}$, $\sigma \in \mathbb{R}^+$, the cumulative distribution function $\Phi\left(\cdot; \mu, \sigma^2\right)$ of a Gaussian $\mathcal{N}\left(\cdot; \mu, \sigma^2\right)$ for any $t \in \mathbb{R}$ is defined by

$$\Phi\left(t; \mu, \sigma^2\right) := \int_{-\infty}^{t} \mathcal{N}\left(x; \mu, \sigma^2\right) \, \mathrm{d}x \,, \tag{2.22}$$

$$\Phi(t) := \Phi(t; 0, 1) . \tag{2.23}$$

2.2. Bayesian Inference

2.2.1. Background

At the core of machine learning is the idea of learning from data, which typically involves recognizing patterns [30]. This concept can be understood intuitively through the following examples:

Example 1. Suppose we were given random data, like trying to predict the outcome of a roulette game based on the last million spins. In this case, prediction would be impossible because the data is inherently random and lacks any pattern. There's no underlying relationship between past spins and future outcomes.

Example 2. On the other hand, imagine someone told us the height of a person. Even without seeing them, we could make a rough estimate of their weight. While the estimate might not be perfect, we intuitively understand that taller people tend to weigh more, and shorter people tend to weigh less. In this case, we've identified a clear pattern.

Example 3. Similarly, consider recognizing digits in images. If shown images of the digits "4" to "6," most of us could correctly classify these numbers. We do this by recognizing specific patterns — such as the straight lines and corners in the digit "4" or the circular shape of the "6".

In abstract terms, there must be some decision-making process, or a "black box," that takes *inputs*, like a person's height or an image of a digit, and produces *outputs*, such as a weight or digit.

The process of linking inputs to outputs is described by a *mapping*, or more formally, a *function*. In machine learning, the goal is to learn this function. While in theory we could store this function by storing every possible input-output pair, this approach becomes impractical or impossible when dealing with large or even infinite input spaces. Instead, we utilize **parametric functions**, which are characterized by a *finite set* of parameters. Consequently, learning the function reduces to learning these parameters.

Example 4. In the Example 2, we might expect a *linear* relationship between height and weight. A parametric function that describes this relationship could look like:

$$f(\text{height}) = m \cdot \text{height} + c = \text{weight}\,.$$

Here, the function is parameterized by a slope m and an offset c. These parameters model the relationship between height and weight, and in machine learning, we aim to learn these parameters (also referred to as *weights*) through data, i.e many examples of height-weight-combinations.

Specifically, we aim to identify the function (parameters) that best explains our data — essentially, the function under which the observed data is most probable. This approach is commonly known as maximum likelihood estimation [30]. Moreover though, our interest might not be limited to a single "best" function. Instead, we may seek to understand how probable every possible function is for the observed data.

2.2.2. Formal Definition

Bayesian inference is one such machine learning method used to determine the probability distribution over function parameters θ, given the observed data D — formally expressed as $p(\theta \mid D)$. As detailed in "The Bayesian Choice" [32], at the core of Bayesian inference is Bayes' Theorem: Given a prior distribution $p(\theta)$ and a function which describes the likelihood $p(D \mid \theta)$, Bayes' Theorem (see Definition 8) allows us to compute the posterior distribution $p(\theta \mid D)$. Once we have the posterior distribution, it can be used for prediction.

Remark 5. In this work, we will use the terms *function* and *model* interchangeably. While both refer to a mapping from inputs to outputs, the term *model* will often seem more appropriate in later chapters, as it better captures the broader concept of a function.

Definition 14. Formally, we define all relevant variables and distributions as follows:

- $D \in (\mathcal{X} \times \mathcal{Y})^n$, the **training data** consisting of n labeled training examples (x_i, y_i).
- θ, the **parameters** of the function (=model).
- $p(\theta)$, the **prior distribution** of the model parameters θ before any data is observed. We also call this the *prior belief* or simply *prior* over θ. This distribution is specified by the model designer during the design process. More details are provided in Section 2.3, specifically in Examples 5 and 6.

- $p(D \mid \theta)$, the **likelihood** of the observed data D conditional on the model parameters θ. Intuitively, the likelihood measures how well the observed data can be explained given different model parameters. Again, this distribution is model-specific and its formal definition is determined during the model design process. More details are provided in Section 2.3, specifically in Examples 5 and 6.

- $p(D)$, the **marginal likelihood** of the observed data D marginalized over the parameters θ. By application of the Law of total probability (see Definition 7), for the continuous case it is formally given by:

$$p(D) = \int p(D \mid \theta) \cdot p(\theta) \ \mathrm{d}\theta \,. \tag{2.24}$$

  Intuitively, it can be thought as a "weighted average" of $p(D \mid \theta)$ for all possible $p(\theta)$, indicating how well the model, in it's entirety, explains the data.

- $p(\theta \mid D)$, the **posterior distribution** of the parameters, after incorporating the observed data with our prior belief. This is the distribution we aim to learn. It is formally given by Bayes Theorem (see Definition 8):

$$p(\theta \mid D) = \frac{p(D \mid \theta) \cdot p(\theta)}{p(D)} \propto p(D \mid \theta) \cdot p(\theta) \,. \tag{2.25}$$

2.3. Factor Graphs

2.3.1. Motivation and Overview

As discussed in Section 2.2, Bayesian inference is a method used to learn the posterior probability distribution $p(\theta \mid D)$. However, the shape of the posterior distribution is significantly influenced by the actual design of the parameterized function f_θ. This design includes the choice of parameters, their interactions, and the overall structure of the model. This raises an intriguing question: How do the posterior distribution and the parameterized function relate to each other?
The answer lies in the following observation: both the prior $p(\theta)$ and likelihood component $p(D \mid \theta)$ are sufficient to derive the posterior (see Equation (2.9)) and both depend on the parameters θ. Therefore, by carefully modeling the prior and likelihood, we can incorporate the specific design of f_θ into the posterior.

Example 5. Consider Example 4, where we modeled height x and weight y using a linear relationship:

$$f_\theta(x) = \theta_1 \cdot x + \theta_2 = y \,.$$

In this model, we might incorporate f_θ into prior and likelihood in the following way:

- For the **likelihood** $p(D \mid \theta)$, we might choose:

$$p(D \mid \theta) = p((x, y) \mid \theta) = \mathcal{N}\left(y; f_\theta(x), \beta^2\right) = \mathcal{N}\left(y - f_\theta(x); 0, \beta^2\right) \,.$$

This likelihood function measures how close the observed value y is to the predicted value $f_\theta(x)$ using a Normal distribution. It assigns high probability to parameters for which $f_\theta(x)$ and y are close to each other, and low (asymptotically zero) probability to parameters for which $f_\theta(x)$ and y are far apart.

- For the **prior** $p(\theta)$, we might choose:

$$p(\theta) = \mathcal{N}(\theta; \boldsymbol{\mu}, \boldsymbol{\Sigma}) \ .$$

This prior function represents our initial beliefs about the parameters θ as a multivariate Gaussian distribution. Here, each parameter θ_1 and θ_2 is expected to be around $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, with some (co-)variance specified by $\boldsymbol{\Sigma}$. The specific values and the details of the multivariate distribution are less important; what is crucial is that the choice of prior can significantly influence the resulting posterior distribution. The likelihood $p(D \mid \theta)$ for a specific θ might be relatively high, even though the prior $p(\theta)$ might indicate that this specific θ has low probability of being the "correct" θ.

In this simple scenario, the factors - prior and likelihood -, were modeled by Normal distributions, i.e. the posterior as simply the product of two Normal distributions (ignoring normalization over $p(D)$). However, both factors, the prior and likelihood, may themselves be composed of multiple factors, making the overall model more complex. In the following, we will introduce a second, more complex example that will be referenced in later sections and chapters. This example is closely related to the model we will explore in future chapters, which is used for predicting moves in chess. Introducing this similar model now will help provide an early intuition about the concepts and methods we will use later.

Example 6. The following example is inspired by Herbrich's work, "TrueSkill™: a Bayesian skill rating system" [33]. Consider a scenario where we want to estimate the skill levels of two players s_1 and s_2 based on the outcome of their head-to-head match $y \in \{-1, 1\}$. For clarification, the s_i's correspond to the parameters θ, and the outcome y to the observed data D. In this case, we might come up with the following model, i.e. factorization into prior and likelihood:

- For the **prior** we might choose:

$$p(s_1, s_2) := p(s_1) \cdot p(s_2) = \mathcal{N}(s_1; \mu_1, \sigma_1) \cdot \mathcal{N}(s_2; \mu_2, \sigma_2) \ ,$$

i.e. the prior as the product of two *mutually independent* Normal distributions, each representing the prior belief about each player i's skill s_i with a mean μ_i and variance σ_i.

- For the **likelihood** $p(y \mid s_1, s_2)$, we might choose:

$$\begin{aligned} p(y \mid s_1, s_2) &:= \int\int p(y, p_1, p_2 \mid s_1, s_2) \, \mathrm{d}p_1 \, \mathrm{d}p_2 \\ &:= \int\int p(p_1 \mid s_1) \cdot p(p_2 \mid s_2) \cdot \mathbb{I}(y \cdot (p_1 - p_2) > 0) \, \mathrm{d}p_1 \, \mathrm{d}p_2 \\ &:= \int\int \mathcal{N}(p_1; s_1, \beta) \cdot \mathcal{N}(p_2; s_2, \beta) \cdot \mathbb{I}(y \cdot (p_1 - p_2) > 0) \, \mathrm{d}p_1 \, \mathrm{d}p_2 \,. \end{aligned} \tag{2.26}$$

  In this likelihood function we introduce intermediate variables p_1 and p_2 which represent the performances of the players in the match. These performances are normally distributed around each player's skill level, with some added noise β, modeling the natural assumption that players do not perform equally good in every game. The indicator function $\mathbb{I}(y \cdot (p_1 - p_2) > 0)$ ensures that the difference in performances $p_1 - p_2$ aligns with the observed match outcome y (i.e., if y is 1, then p_1 should be greater than p_2, and if y is -1, then p_1 should be less than p_2). Since $p(y \mid s_1, s_2)$ is a function of y (and s_1 and s_2), we need to integrate out all other variables, which in this case are p_1 and p_2.

In other words, our model is fully described by the joint probability function:

$$\begin{aligned} p(s_1, s_2, p_1, p_2, y) &= p(y, p_1, p_2 \mid s_1, s_2) \cdot p(s_1, s_2) \\ &= p(s_1) \cdot p(s_2) \cdot p(p_1 \mid s_1) \cdot p(p_2 \mid s_2) \cdot \mathbb{I}(y \cdot (p_1 - p_2) > 0) \,. \end{aligned} \tag{2.27}$$

In this specific case, we are not interested in the intermediate variables p_1 and p_2, which can be marginalized out by integrating them away (similar to how it was done in the likelihood term (2.26)), resulting in $p(s_1, s_2, y)$.

Remark 6. As one might notice, the **joint** distribution $p(s_1, s_2, y)$ is **not** the **posterior** distribution $p(s_1, s_2 \mid y)$, the distribution we are actually after. Rather, the joint distribution is only *proportional* to the posterior distribution (see Bayes Theorem (2.8)). An important insight here is that since y is observed, it is fixed and can be treated as a constant. For instance, if we observe $y = 1$, meaning our data indicates that player one won the match, a more appropriate formulation of the joint probability distribution might actually be $p(s_1, s_2, y = 1)$, i.e. it is only a function of s_1 and s_2. Thus, formulating the joint probability distribution (as a product of the prior and likelihood) is sufficient to describe our model, as the posterior $p(s_1, s_2 \mid y = 1)$ is merely a re-scaled version of this, in specific $p(s_1, s_2, y = 1)/p(y = 1)$ using Bayes Theorem.

Even though this model is still relatively simple, formulating such models for larger, more complex systems can become tedious. This is where factor graphs come into play, providing a graphical way to represent the dependencies between variables and the factors that make up the joint probability distribution.

2.3.2. Formal Definition and Notation

Definition 15 (Factor Graph (Frey et al. [34]))**.** Given a product of m functions $f_1, f_2, ..., f_m$, each over a subset of n variables $x_1, x_2, ..., x_n$, a factor graph is a bipartite graphical model with m factor nodes and n variable nodes where an undirected edge connects f_i and x_j if and only if the function f_i depends on x_j.

Example 15.1. Consider the function f that factorizes as follows:

$$f(x_1, x_2, x_3) = f_1(x_1) \cdot f_2(x_2) \cdot f_3(x_1, x_2, x_3)\,.$$

The corresponding factor graph to f is shown in Figure 2.1.

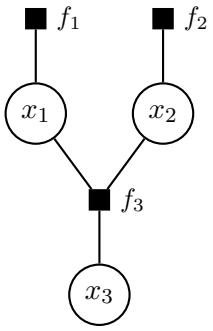


Figure 2.1.: Illustration of factor graph notation, where factors are represented by black rectangles and variables by circular nodes.

Example 15.2. The factor graph shown in Figure 2.2 corresponds to the Example 6 provided earlier, where we modeled the skills of two players based on the outcomes of their matches. This graphical representation makes it easier to visualize the dependencies between the variables and the structure of the joint probability distribution. Note that in this case, we assume that the match outcome $y = 1$ was observed, as can be seen for the center factor $\mathbb{I}(p_1 - p_2 > 0)$. For the case $y = -1$ the factor function would have been $\mathbb{I}(p_2 - p_1 > 0)$ (see Equation (2.26) for reference).

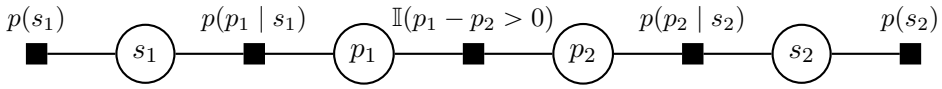


Figure 2.2.: Factor graph representing the probabilistic model of the head-to-head match outcome between two players, as described in Example 6. The graph illustrates the dependencies between the skill levels (s_1 and s_2) of the players and their respective performances (p_1 and p_2) and the observed match outcome $y = 1$.

2.4. Sum-Product Algorithm

2.4.1. Motivation

We previously demonstrated how modeling a complex function to describe our joint probability distribution (proportional to the posterior) may lead to a factorization of this distribution into smaller components, each dependent on only a subset of variables. These variables can be categorized into two groups: *observed* and *unobserved* variables.

Example 7. In the model, which we introduced in Example 6, these variables were the skill levels s_1, s_2 the match performances p_1, p_2 and the match outcome y.

- The variable y is **observed** and can be treated as a constant since we can replace y in the joint probability with its actual value (as was done in Example 15.2).
- The variables s_1, s_2, p_1 and p_2 are **unobserved**. Our main focus is on the skill variables s_1 and s_2, while p_1 and p_2 were introduced as part of the likelihood component. However, we don't directly need or care about p_1 and p_2; these kind of variables are what we refer to as *latent* unobserved variables.

The sum-product algorithm, introduced by Kschischang, Frey and Loeliger in their paper "Factor Graphs and the Sum-Product Algorithm" [35], is a method used to compute the marginal distribution of each *unobserved* variable, conditional on the *observed* ones, given a joint distribution over all variables.
In general, using the Law of Total Probability (see Definition 7), the process of deriving a marginal distribution for an unobserved variable requires integrating over all other unobserved variables, which unfortunately scales exponentially with the number of such variables.

Example 8. Consider n Bernoulli distributed random variables $X_1, ..., X_n$ (Bernoulli meaning that they act in a binary way, taking on only values 0 or 1). The marginal distribution of X_i is given by:

$$p(x_i) = \sum_{X_1} \cdots \sum_{X_{i-1}} \sum_{X_{i+1}} \cdots \sum_{X_n} p(x_1, ..., x_n)\,.$$

In this case, a (naive) computation of the marginal distribution, assuming $n = 256$, would involve 2^{255} - approximately 10^{80} — summations, which is roughly equivalent to the number of atoms in the observable universe[3].

The sum-product algorithm leverages the fact that the joint probability distribution may be factorized into smaller functions. Since not every function depends on every variable, we do not need to (naively) sum over all possible combinations of unobserved variables. The factorized structure allows the algorithm to minimize the number of necessary summations effectively.

[3]This estimate can be (or rather, is typically) derived from cosmological parameters, which are for example presented in "Planck 2018 results-VI. Cosmological parameters" [36] or "Wilkinson Microwave Anisotropy Probe (WMAP) observations" [37]

Example 9. For instance, consider the joint probability distribution $p(x_1, ..., x_n) = \prod_i p(x_i)$. In this case we could have rewritten the summation as:

$$\begin{aligned} p(x_i) &= \sum_{X_1} \cdots \sum_{X_{i-1}} \sum_{X_{i+1}} \cdots \sum_{X_n} \prod_j p(x_j) \\ &= p(x_i) \cdot \sum_{X_1} p(x_1) \cdots \sum_{X_{i-1}} p(x_{i-1}) \sum_{X_{i+1}} p(x_{i+1}) \cdots \sum_{X_n} p(x_n) \,. \end{aligned}$$

This reduces the number of summations from $O(2^n)$ to $O(n)$.

2.4.2. Message Passing

In this section, we detail the message-passing algorithm, specifically the sum-product algorithm, as introduced by Kschischang et al. [35]. This algorithm follows a set of straightforward rules to compute marginal distributions on factor graphs by sending so-called *messages* between the nodes. While the algorithm yields exact results for tree-structured graphs, the same does not hold for general graphs that may contain loops. However, as noted by Kschischang et al., the algorithm can still provide good approximations even in the presence of loops. For now, we will focus on its application to tree structures, where the algorithm operates without issue.

To clarify how the update rules are derived and to gain a better understanding of what exactly is meant by *message passing*, we will use a specific example. The following explanation largely follows the approach in Aji and McEliece's paper "The Generalized Distributive Law" [38]. Both the sum-product algorithm and the generalized distributive law are closely related, and as both Kschischang et al. and Aji-McEliece noted in their papers (published around the same time), they can essentially be derived from one another. Readers who are not interested in the derivation and intuition behind the sum-product algorithm may skip ahead to Subsection 2.4.3, though it is not suggested.

Example 10. Consider a joint probability function f of the discrete random variables X_1, X_2, X_3, X_4 and X_5 and it's corresponding factor graph as shown in Figure 2.3.

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) = f_1(x_1) \cdot f_2(x_1, x_2) \cdot f_3(x_2) \\ \cdot f_4(x_2, x_3, x_4) \cdot f_5(x_3) \cdot f_6(x_3, x_5) \cdot f_7(x_4) \cdot f_8(x_5) \,. \end{aligned} \tag{2.28}$$

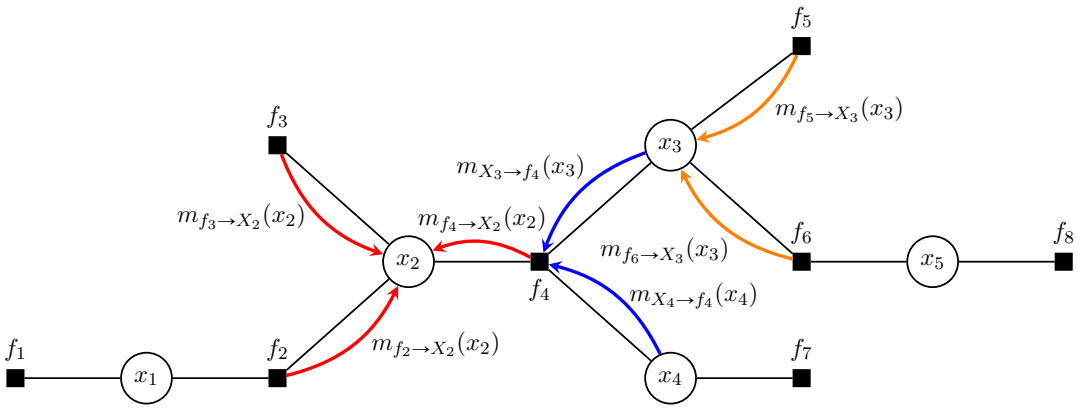


Figure 2.3.: Factor graph for the function f in Example 10. A few selected messages $m_{f_j \to X_i}(x_i)$ and $m_{X_i \to f_j}(x_i)$ used in the derivation of the sum-product algorithm are shown to aid in understanding the message-passing process.

Marginals and Messages: Let us assume we want to calculate the marginal distribution for X_2. Formally, this requires summation over X_1, X_3, X_4 and X_5:

$$
\begin{aligned}
p(x_2) &= \sum_{x_1}\sum_{x_3}\sum_{x_4}\sum_{x_5} f_1(x_1) f_2(x_1, x_2) f_3(x_2) f_4(x_2, x_3, x_4) f_5(x_3) f_6(x_3, x_5) f_7(x_4) f_8(x_5) \\
&= \underbrace{\left[\sum_{x_1} f_1(x_1) \cdot f_2(x_1, x_2)\right]}_{m_{f_2 \to X_2}(x_2)} \cdot \underbrace{[f_3(x_2)]}_{m_{f_3 \to X_2}(x_2)} \\
&\quad \cdot \underbrace{\left[\sum_{x_3}\sum_{x_4}\sum_{x_5} f_4(x_2, x_3, x_4) \cdot f_5(x_3) \cdot f_6(x_3, x_5) \cdot f_7(x_4) \cdot f_8(x_5)\right]}_{m_{f_4 \to X_2}(x_2)} .
\end{aligned} \tag{2.29}
$$

Notice how we can rewrite the joint probability as a product of sums, which is the foundation of the "sum-product" algorithm. Specifically, each sum is a function solely of x_2. We refer to each of these sums (or functions) as a *message* from a factor to a variable, denoted $m_{f_j \to X_i}(x_i)$. Specifically, $m_{f_j \to X_i}(x_i)$ is given as the sum over all variables in the sub-tree rooted at factor f_j. As shown in Equation (2.29), the marginal $p(x_2)$ is simply the product of all incoming messages from neighboring factors (see also Figure 2.3, where the messages are highlighted in red).

Messages from Factor to Variable: We can further refine the message equation for messages from factors to variables. Consider the message equation:

$$\begin{aligned} m_{f_4 \to X_2}(x_2) &= \sum_{x_3} \sum_{x_4} \sum_{x_5} f_4(x_2, x_3, x_4) \cdot f_5(x_3) \cdot f_6(x_3, x_5) \cdot f_7(x_4) \cdot f_8(x_5) \\ &= \sum_{x_3} \sum_{x_4} f_4(x_2, x_3, x_4) \cdot \underbrace{[f_7(x_4)]}_{m_{X_4 \to f_4}(x_4)} \cdot \underbrace{\left[\sum_{x_5} f_5(x_3) \cdot f_6(x_3, x_5) \cdot f_8(x_5) \right]}_{m_{X_3 \to f_4}(x_3)} . \end{aligned} \tag{2.30}$$

As seen in Equation (2.30), we can express the message from a factor to a variable in a more elegant form. Intuitively, sending a message from a factor (in this case, f_4) to a variable (in this case, X_2) involves summing over all neighboring variables (X_3 and X_4, but not X_2) and weighting these by the incoming messages from those variables (denoted as $m_{X_3 \to f_4}(x_3)$ and $m_{X_4 \to f_4}(x_4)$). Figure 2.3 illustrates this process, where the incoming messages feeding into factor f_4 are highlighted in blue. More generally, these messages, sent from variables to factors, are denoted as $m_{X_i \to f_j}(x_i)$ and represent the sum over all variables in the sub-tree rooted at X_i.

Messages from Variable to Factor: Finally, we can also refine the message equation for messages from variables to factors. Consider the message equation:

$$\begin{aligned} m_{X_3 \to f_4}(x_3) &= \sum_{x_5} f_5(x_3) \cdot f_6(x_3, x_5) \cdot f_8(x_5) \\ &= \underbrace{[f_5(x_3)]}_{m_{f_5 \to X_3}(x_3)} \cdot \underbrace{\left[\sum_{x_5} \cdot f_6(x_3, x_5) \cdot f_8(x_5) \right]}_{m_{f_6 \to X_3}(x_3)} . \end{aligned} \tag{2.31}$$

As shown in Equation (2.31), a message from a variable (in this case, X_3) to a factor (in this case, f_4) is computed by multiplying the incoming messages from neighboring factors (f_5 and f_6), excluding the message from the factor to which the message is being sent (f_4). Figure 2.3 illustrates this process, where the relevant messages are highlighted in orange.

Intuition: In general, a message can be understood as a function that encapsulates all the information from the sub-tree rooted at the point where the message originates. Consequently, computing a marginal simply involves multiplying all incoming messages from **direct** neighbors **only**, since these messages already carry the relevant information from other variables and factors in the network. Similarly, a message from a factor to a variable, such as from f_4 to X_2 (which is a function of x_2 and highlighted in red in Figure 2.3), is computed as follows: f_4 is a function of x_2, x_3, and x_4. Therefore, we sum out x_3 and x_4 and incorporate the information about these variables from their respective sub-trees via the incoming messages, which themselves are functions of x_3 and x_4 (highlighted in blue in Figure 2.3).

2.4.3. Update Rules

Definition 16 (Sum-Product Algorithm [35, 38])**.** We arrive at the following set of equations:

$$p(x) = \prod_{f \in \mathrm{ne}(x)} m_{f \to X}(x)\,, \tag{2.32}$$

$$m_{f \to X}(x) = \sum_{x' \in \mathrm{ne}(f) \setminus \{x\}} \cdots \sum_{x'' \in \mathrm{ne}(f) \setminus \{x\}} \left[f(x, x', ..., x'') \cdot m_{X' \to f}(x') \cdots m_{X'' \to f}(x'') \right], \tag{2.33}$$

$$m_{X \to f}(x) = \prod_{f' \in \mathrm{ne}(x) \setminus \{f\}} m_{f' \to X}(x)\,. \tag{2.34}$$

This approach is significantly more efficient than a straightforward summation over all variables, as it allows local summations to be computed in $O(2^{d_{max}})$, where $d_{max} = \max_f |\mathrm{ne}(f)|$, rather than the more computationally expensive $O(2^n)$, where n is the total number of variables. Since $\mathrm{ne}(f)$ represents a subset of the n variables, it is clear that $d_{max} \leqslant n$. As a result, we can compute all messages (and marginals) in time proportional to the number of edges E in the graph, yielding an overall runtime of $O(E \cdot 2^{d_{max}})$.
Extending this approach to continuous variables simply involves replacing summation with integration.

2.4.4. Normalization in Marginal Computation

Returning to our Example 6, where we modeled the skills of two players based on their match outcomes, we defined the joint probability distribution over the skill levels s_1 and s_2, the latent variables p_1 and p_2, and the match outcome y:

$$p(s_1, s_2, p_1, p_2, y) = p(s_1) \cdot p(s_2) \cdot p(p_1 \mid s_1) \cdot p(p_2 \mid s_2) \cdot \mathbb{I}(y \cdot (p_1 - p_2) > 0)\,.$$

The sum-product algorithm enables us to derive the marginal distributions $p(s_1)$, $p(s_2)$, $p(p_1)$, and $p(p_2)$. While we refer to these distributions as marginal distributions, one may wonder why we, for the example s_1, do not express them as $p(s_1, y)$ implying a connection to the observed data y, especially since we are marginalizing out only the unobserved variables. Specifically, we could write:

$$p(s_1, y) = \int\int\int p(s_1, s_2, p_1, p_2, y)\ \mathrm{d}s_2\ \mathrm{d}p_1\ \mathrm{d}p_2\,. \tag{2.35}$$

However, since y is observed, $p(s_1, y)$ effectively becomes a function of s_1 alone. Therefore, it is conventional to simply denote this as $p(s_1)$. More importantly though, our focus is not on deriving $p(s_1, y)$, but rather the posterior distribution $p(s_1 \mid y)$, the probability of s_1 given the observation of y. Formally, we could derive this posterior as

$$p(s_1 \mid y) = \frac{p(s_1, y)}{p(y)}\,.$$

As already mentioned in Remark 6, given that y is constant (observed), the normalization factor $p(y)$ is also a constant and therefore does not affect the shape or relative values of $p(s_1, y)$; it merely scales the distribution. This understanding underscores why we typically do not place excessive emphasis on the distinctions among the notations $p(s_1)$, $p(s_1, y)$, and $p(s_1 \mid y)$: the functions are essentially equivalent for our purposes, depending solely on s_1 and not on y — this should not be confused with the fact that y is part of the joint probability function (as a constant), which in turn influences the shape of the marginal of s_1, i.e. different observations y lead to different posterior distributions $p(s_1 \mid y)$.
Consequently, as long as the marginals computed by the sum-product algorithm are correctly normalized, i.e., integrate to one, they can be understood as valid posterior probability distributions, conditioned on the observed data. In this work, normalization is achieved through the use of Normal distributions, which are inherently normalized. As we will see, in each update step of the sum-product algorithm (applying Equations (2.32), (2.33) and (2.34)), the resulting distributions will be calculated as new Normal distributions, ensuring that everything remains normalized at every step (see Section 3.3 for the derivation of factor-specific update rules).

2.4.5. Practical Implementation and Considerations

Scheduling: The recursive structure of the update rules in the sum-product algorithm (see Definition 16), leads to an efficient update schedule that computes the marginals for all variables by passing a single message in each direction along every edge. This schedule is termed the Generalized Forward/Backward schedule by Kschischang et al. [35]. The steps are as follows:

1. Initialize all messages $m_{f \to X}(x)$ and marginals $p(x)$ with a constant function (i.e., uniform distribution, like a Gaussian Uniform; Definition 12)

2. Choose an arbitrary root variable x_k in the factor graph.

3. Update all messages $m_{f \to X}(x)$ from the leaves of the tree rooted at x_k *upwards* to the root x_k. After completing this step, the marginal $p(x_k)$ will be correctly updated.

4. Update all messages $m_{f \to X}(x)$ from the root x_k to the leaves *"downwards"*, to update the remaining marginals.

It is important to note that different strategies can be applied when deriving messages and marginals in the factor graph. If only a subset of marginals is needed, it may not be necessary to follow a general update schedule like the one outlined above, allowing for more efficient computations tailored to the specific marginals of interest.

Storage Efficiency: Using the update equation (2.32) for marginals, we can rewrite this in a more interpretable way:

$$p(x) = \prod_{f \in \text{ne}(x)} m_{f \to X}(x) = m_{f' \to X}(x) \cdot \underbrace{\prod_{f'' \in \text{ne}(x) \setminus \{f'\}} m_{f'' \to X}(x)}_{m_{X \to f'}(x)} . \tag{2.36}$$

In other words, the marginal of x can also be computed by multiplying it's incoming message $m_{f \to X}$ from factor f and outgoing message $m_{X \to f}$ to factor f. For storage efficiency, we therefore do not need to store all marginals and messages (both incoming and outgoing). Rather we can store only two of the three components - the missing component can be computed through multiplication or division. In this work, we store the marginals and the incoming messages $m_{f \to X}$. The cost of performing multiplication or division to derive the missing distribution is minimal with well-chosen distributions, in our case the Normal distribution, outweighing the storage efficiency gains.

Approximate message passing: In the practical application of message passing, several challenges arise. One significant issue is that marginals and messages can be somewhat arbitrary functions; the only requirement is that marginals must be valid distributions, meaning they must integrate to one. This is impractical, as we need a way to store and handle the marginals and messages efficiently. To address this, it is common to choose an "approximate" parametric distribution. In our case, as hinted before, we will assume our marginals to be normally distributed, parameterized by their mean and variance.
As shown in the update equation (2.32), the marginal is the product of its incoming messages $m_{f \to X}$. In the case of Normal distributions, this approach has an additional benefit: according to Theorem 1, the product of two Normal distributions is again a Normal distribution. Since our normally distributed marginals are products of messages, it is only logical to consider our incoming messages $m_{f \to X}$ to be normally distributed as well. Furthermore, examining the update rule (2.34) for outgoing messages $m_{X \to f}$, if we assume the incoming messages $m_{f \to X}$ to be normally distributed, this would also result in Normal distributions for the outgoing messages $m_{X \to f}$.
However, when we inspect the update rule (2.33) for the incoming messages $m_{f \to X}$, which we want to model as Normal distributions, specifically:

$$m_{f \to X}(x) = \sum_{x' \in \text{ne}(f) \setminus \{x\}} \cdots \sum_{x'' \in \text{ne}(f) \setminus \{x\}} \left[f(x, x', \ldots, x'') \cdot m_{X' \to f}(x') \cdots m_{X'' \to f}(x'') \right],$$

it does not necessarily follow that these messages will indeed be normally distributed. For an arbitrary factor function f (for example, $\mathbb{I}(y \cdot (p_1 - p_2) > 0)$ in Example 6), the resulting distribution may not be a Normal distribution.
This is where the concept of **approximate message passing** becomes important. Approximate message passing has been extensively studied (Minka provides a thorough overview of key algorithms [39]), and although we do not explore it in depth in this thesis, we will introduce the central idea. Approximate message passing involves approximating a probability distribution p by another (simpler) parametric distribution q,

typically chosen to minimize some divergence measure. In our case, we focus on minimizing the so-called **Kullback-Leibler (KL) divergence**, $\mathrm{KL}(p, q)$, where p is the true distribution and q is the approximation.
In particular, when necessary — such as when $f = \mathbb{I}(x > 0)$ — we will approximate the "true" non-Gaussian message $m_{f \to X}$ as follows. From equation (2.36), we see that the marginal distribution $p(x)$ is given by $m_{f \to X}(x) \cdot m_{X \to f}(x)$. Therefore, the "true" marginal $p(x)$ will also be non-Gaussian. We now find the best Gaussian approximation $\tilde{p}(x)$ for $p(x)$ through a method known as Moment Matching (see Theorem 3). With this approximation, we can calculate the approximate message $\tilde{m}_{f \to X}$ by dividing the approximate marginal by the outgoing message:

$$\tilde{m}_{f \to X}(x) = \frac{\tilde{p}(x)}{m_{X \to f}(x)} \,. \tag{2.37}$$

Theorem 3 (Moment Matching)**.** *Given any distribution $p(\cdot)$ over a random variable X and another distribution $q(\cdot)$ over X which is Gaussian $\mathcal{N}\left(\cdot; \tilde{\mu}, \tilde{\sigma}^2\right)$, the KL divergence $\mathrm{KL}(p, q)$ is minimized by matching the first two moments of the distribution $p(\cdot)$ to the first two moments of the Gaussian distribution $q(\cdot)$. Specifically, the moments are given by:*

$$\tilde{\mu} = \mathbb{E}\left[X\right], \tag{2.38}$$

$$\tilde{\sigma}^2 = \mathbb{E}\left[X^2\right] - \mathbb{E}\left[X\right]^2 . \tag{2.39}$$

In other words, the Gaussian distribution which minimizes the KL divergence, is the one, which matches the mean and variance of the approximated distribution [39].[4]

In theory, **if** the message $m_{f \to X}$ is a valid distribution (i.e. integrates to one), we could instead directly approximate the message as a Gaussian $\tilde{m}_{f \to x}$ using Moment Matching. However, since messages can take on arbitrary forms, we cannot guarantee that this will always be the case. For instance, the message $m_{f \to X}(x) = \mathbb{I}(x > 0)$ does not integrate to one, nor can it be normalized to do so.
Therefore, we choose to approximate the true marginal, which, as the product $m_{f \to X}(x) \cdot m_{X \to f}(x)$, is often more amenable to normalization. For example, if $m_{f \to X}(x) = \mathbb{I}(x > 0)$ and $m_{X \to f}$ is Gaussian, then the marginal can be normalized to one, making it a valid distribution. Consequently, Moment Matching can be applied; in fact, this will be done for the above example in Section 3.3.4 on the so-called Greater-Than Factor.
Moreover, we aim for the best possible approximation of the marginals, which may not necessarily result from the product of the outgoing message and the best approximation of the incoming message, even if such an approximation is feasible[5]. The marginals are

[4] This result holds more generally for any distribution within the exponential family, not just Gaussian distributions

[5] While we do not have a specific source to prove this claim, it can be inferred from the derivations in Section 3.3.4, particularly in Figures 3.7b and 3.7a. In those figures, the best approximations of the marginal lead to different incoming messages; this should not not occur since the best approximation for the incoming message would have to be the same, if the product of the outgoing message and the best approximation of the incoming message would in fact lead to the best approximation of the marginal.

what we really care for while the messages are more of a means to an end.
Due to the approximation involved, we can no longer simply apply a forward-backward schedule once over the factor graph as described in Paragraph 2.4.5. Instead, we will need to iterate through all affected marginals and messages (see Subsection 3.2.3). The algorithm terminates when we observe convergence — that is, when changes in the approximated marginals fall below a predefined minimal threshold. This form of approximate message passing is also known as *expectation propagation* [40].

3. Concept

3.1. Overview and Rationale

3.1.1. Motivation

In this chapter, we introduce two models designed to evaluate chess moves, drawing inspiration from the way human chess masters approach decision-making. Extensive research and anecdotal evidence suggest that top players do not analyze every possible move like brute-force algorithms. Instead, they recognize patterns on the board through a process known as "chunking", which allows them to use intuition and experience to focus on a smaller set of promising moves, rather than relying solely on pure calculation [41, 42, 43]. Richard Réti, one of the leading chess players of his time, in his book Modern Ideas in Chess (1923), highlighted this approach, stating, "The layman thinks that the superiority of the chess master lies in his ability to think out 3 or 4, or even 10 or 20, moves ahead. Those chess lovers who ask me how many moves I usually calculate in advance, when making a combination, are always astonished when I reply, quite truthfully, 'as a rule not a single one.'" [44]. Given this, we hypothesize that there must be a way to measure the *importance* of certain moves, without excessive reliance on brute-force look-ahead calculations.

3.1.2. Modeling

Our goal is to develop a model that accurately learns the *importance* of each move. Accuracy, in this context, means that when the model is tasked with predicting the move made by an expert for a given position, it assigns the highest importance to the expert's move with high precision. Learning such a measure of importance is not a new concept. In their 2006 paper, "Bayesian Pattern Ranking for Move Prediction in the Game of Go", Stern et al. introduced a Full Ranking Model that used an importance measure for move prediction. As they note, the model "show[ed] excellent prediction performance as indicated by its ability to perfectly predict the moves made by professional Go players in 34% of test positions." [45]. In the following, we will describe two general models that adopt this ranking-based approach for the game of chess.

Urgency Model

Following the approach in Stern et al.'s work, we frame move prediction as a *ranking problem*, similar to the ranking problem in Example 6 in Section 2.3, where players were ranked based on a skill measure s_i derived from match outcomes y. In this case, however,

we rank multiple moves by a measure of importance, which we term *urgency* u_i, based on expert players' decisions m_k. Specifically, we analyze a dataset of expert games and treat the move made, m_k, at each position as the "winner" of a match between all possible moves $m_1, ..., m_n$ for that position. The formal description of our model is provided in Section 3.2.1 and illustrated in Figure 3.1. We will refer to this model as the **Urgency Model**.

BoardVal Model

In the following we introduce a second, yet related model. Instead of ranking moves in isolation, we propose ranking the resulting board positions. In other words, we shift from assigning urgency to moves, to assigning **board value** to the resulting board states. This approach, modeling moves as the board positions they produce, aligns with the methodology proposed by Stern et al. [45].
However, this approach introduces new challenges. While the number of possible moves in the Urgency Model is relatively small, and therefore learning an urgency for every such move is feasible, the number of potential board positions is exponentially larger (upper bounded by $13^{64} \approx 10^{71}$ board positions). Storing a value for every possible board state is therefore not feasible, and it would also not provide insights into why one board is valued more highly than another. If we could store every possible board state, we would effectively solve chess, making the model unnecessary.
In Stern's et. al. work [45], this issue was addressed by approximating the value of board positions through smaller "sub-boards". Rather than learning the value for every possible board, they focused on a reduced set of sub-boards, significantly decreasing the storage complexity. In Go, a move involves simply the placing of a stone at a specific vertex. The resulting board was approximated by progressively larger sub-boards centered around this vertex — starting from the immediate neighbors, then expanding to include two-field neighbors, three-field neighbors, and so on, up to the entire board in extreme cases. The set of potential sub-boards - the pattern base - was pre-extracted using a Bloom filter, which captured only patterns that appeared a minimum number of times across the dataset. Smaller sub-boards were more frequent because they could appear in many locations on the board, whereas larger patterns were included only if they were common, typically only occurring for early game positions.
Unfortunately, this approach cannot be directly applied to chess for several reasons. In chess, a move involves relocating a piece from one square to another, rather than placing a piece on a fixed vertex. Additionally, each vertex (square) in chess can have one of 13 possible states (occupied by 12 different pieces or empty), compared to just three in Go. This leads to an exponential increase in the number of possible sub-patterns for a fixed pattern size, which makes the pattern extraction process impractical. Lets say we were to extract frequent sub-boards at the destination square: if we limit the number of patterns to be extracted to a manageable number — say, several million — we end up with only very small patterns, which we believe would fail to capture the long-range interactions between pieces. For instance, a rook or queen can influence squares up to seven spaces away, making small sub-patterns insufficient for representing complex board

states.
Therefore, we propose a different solution: instead of approximating the resulting board with a single, maximally-large sub-pattern, we interpret the board as a combination of smaller, more abstract features. For a very simple feature set, we might define a feature set based on the presence of each piece on every square, and the value of the board could be the sum of the values for these piece-square combinations. If, for example, a white king is on *a4* and a black pawn is on *b3*, the board's value would be the sum of the values assigned to these piece-square pairs. While this is a simple model, it provides a useful foundation. We will later demonstrate that the model is flexible in its choice of features, and as we'll see, the selection of features plays a crucial role in the model's overall performance.
The formal description of this new model is provided in Section 3.2.2 and illustrated in Figure 3.2. We will refer to this model as the **BoardVal Model** (or Board Value Model for long).

3.2. Model Structure

3.2.1. Urgency Model

Consider a board b with a set of legal moves $\boldsymbol{m}$, where $n := |\boldsymbol{m}|$ represents the number of possible moves $m_1, ..., m_n$. We assume that the moves are ordered such that m_1 is the expert's chosen move. Each move m_i is associated with an urgency of play u_i, of which the expert observes a noisy version $\tilde{u}_i$. The move m_1 is associated with the highest latent urgency $\tilde{u}_1$. The model is formally defined by the joint probability:

$$p(\boldsymbol{u}, \tilde{\boldsymbol{u}}, \boldsymbol{d}) := \prod_{i=1}^{n} f(u_i) \cdot \prod_{j=1}^{n} g(u_j, \tilde{u}_j) \cdot \prod_{k=2}^{n} h(\tilde{u}_1, \tilde{u}_k, d_k)$$

where

$$\begin{aligned} f(u_i) &= \mathcal{N}\left(u_i; \mu_i, \sigma_i^2\right) , \\ g(u_j, \tilde{u}_j) &= \mathcal{N}\left(\tilde{u}_j; u_j, \beta^2\right) , \\ h(\tilde{u}_1, \tilde{u}_k, d_k) &= \delta\left(d_k - (\tilde{u}_1 - \tilde{u}_k)\right) \cdot \mathbb{I}(d_k > 0) . \end{aligned}$$

The corresponding factor graph for the joint probability is depicted in Figure 3.1.
Note how the expert decision m_1 is modeled indirectly into the joint probability distribution by the correct ordering of the moves. The model consists of several "node layers", each representing different variables and their dependencies.

1. **Urgency nodes:** In the first layer, we have an *urgency node* u_j for every possible move on the board (the first row of variable nodes in Figure 3.1). We assume that each urgency u_j is normally distributed, and this prior belief is formally expressed by the **1D-Gaussian Factor** (positioned above each node u_j in Figure 3.1).

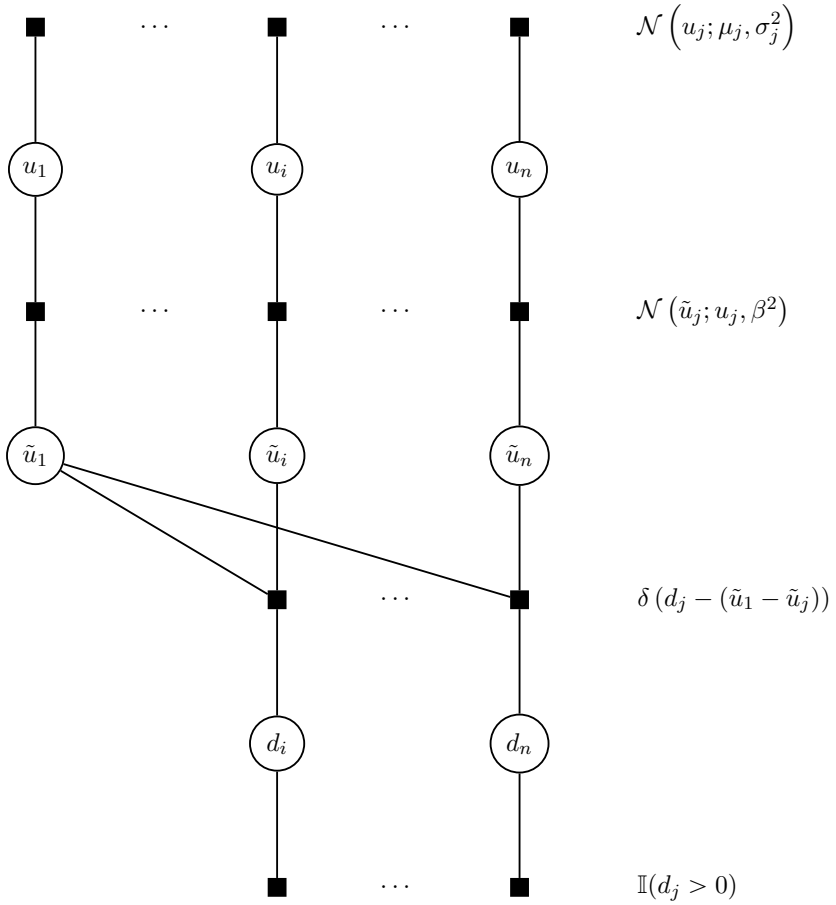


Figure 3.1.: The factor graph for the Urgency Model. The first layer represents the urgencies u_j, with priors modeled as Gaussian distributions. The latent urgencies $\tilde{u}_j$ in the second layer are noisy observations of these urgencies, linked through Gaussian Mean Factors. The third layer captures the differences between the expert's chosen move and other possible moves using Weighted Sum Factors. Greater-Than Factors enforce the expert's chosen move to have a higher latent urgency (indicated by a positive difference).

2. **Latent urgency nodes:** This first layer of urgency nodes u_j is connected to a second layer of *latent urgency nodes* $\tilde{u}_j$ (the second row of variable nodes in Figure 3.1). The key idea is that the expert's perceived urgency is a noisy observation of the true underlying urgency u_j. This relationship is modeled by a **Gaussian Mean Factor** (positioned between each urgency u_j and its corresponding latent urgency node $\tilde{u}_j$ in Figure 3.1). The noise in the model accounts for various factors, such as the expert occasionally choosing a suboptimal move or multiple moves being nearly equivalent in quality, leading the expert to select one at "random". The model incorporates some flexibility in the ranking by urgency, with β as a hyperparameter controlling the level of this noise.

3. **Difference nodes:** The second layer of latent urgency nodes $\tilde{u}_j$ is then connected to a third layer of *difference nodes* d_j (the third and final row of variable nodes Figure 3.1). Each difference node d_j captures the difference between the latent urgency of the expert's chosen move, $\tilde{u}_1$, and that of another move $\tilde{u}_j$. This relationship is formally expressed using a **Weighted Sum Factor**, which generalizes the difference operation. Each difference node is also connected to a **Greater-Than Factor**, which enforces the constraint that the difference must be positive, reflecting the observation that the expert's chosen move must have had a higher perceived urgency than the alternative moves.

3.2.2. BoardVal Model

Consider a board b with a set of legal moves $\boldsymbol{m}$, where $n := |\boldsymbol{m}|$ represents the number of possible moves $m_1, ..., m_n$. We assume that the moves are ordered such that m_1 is the expert's chosen move. Each move m_j leads to a resulting board state b_j, which is represented by a set of features $\mathcal{F}(b_j)$, with $l := |\mathcal{F}(b_j)|$ denoting the number of features $f_{j,1}, ..., f_{j,l}$ on board b_j. Let $\boldsymbol{f}$ represent the set of all unique features across all boards $b_1, ..., b_n$ ($\boldsymbol{b}$), and let $t := |\boldsymbol{f}|$ denote the total number of distinct features. Note that a feature $f_{i_1,j}$ from one board b_{i_1} may refer to the same feature $f_{i_2,k}$ from another board b_{i_2}. Each feature $\boldsymbol{f}_i$ has an associated value $v_{\boldsymbol{f}_i}$, which may be part of a board value v_{b_j}, of which the expert observes a noisy version $\tilde{v}_{\boldsymbol{b}_j}$. The expert's move m_1 is associated with the highest latent board value $\tilde{v}_{\boldsymbol{b}_1}$. The model is formally defined by the joint probability:

$$p(\boldsymbol{v_f}, \boldsymbol{v_b}, \tilde{\boldsymbol{v}}_{\boldsymbol{b}}, \boldsymbol{d}) = \prod_{j=1}^{t} f(v_{\boldsymbol{f}_j}) \cdot \prod_{i=1}^{n} \delta \left(v_{b_i} - \sum_{k=1}^{|\mathcal{F}(b_i)|} v_{f_{i,k}} \right) \cdot \prod_{l=1}^{n} g(v_{b_j}, \tilde{v}_{b_j}) \cdot \prod_{m=2}^{n} h(\tilde{v}_{b_1}, \tilde{v}_{b_m}, d_m)$$

where

$$\begin{aligned} f(v_{\boldsymbol{f}_j}) &= \mathcal{N}\left(v_{\boldsymbol{f}_j}; \mu_{\boldsymbol{f}_j}, \sigma^2_{\boldsymbol{f}_j}\right), \\ g(v_{b_j}, \tilde{v}_{b_j}) &= \mathcal{N}\left(\tilde{v}_{b_j}; v_{b_j}, \beta^2\right), \\ h(\tilde{v}_{b_1}, \tilde{v}_{b_m}, d_m) &= \delta\left(d_m - (\tilde{v}_{b_1} - \tilde{v}_{b_m})\right) \cdot \mathbb{I}(d_m > 0). \end{aligned}$$

Again, note how the expert decision m_1 is modeled indirectly into the joint probability distribution by the correct ordering of the moves (and in turn the boards). The corresponding factor graph for the joint probability is depicted in Figure 3.2.
The model consists of several "node layers", similar to the Urgency Model. Details not explicitly covered here can be referenced from Section 3.2.1 on the Urgency Model.

1. **Feature Value Nodes:** In the first layer, we introduce a *feature value node* $v_{\boldsymbol{f}j}$ for each unique feature in the set $\boldsymbol{f}$. As with the urgency nodes in the Urgency Model, each feature value node is assumed to follow a Normal distribution, with the prior belief formally expressed by the **1D-Gaussian Factor**. These prior initialization factors are shown above each node $v_{f_{j,k}}$ in Figure 3.2.[1]

2. **Board Value Nodes:** The first layer of feature value nodes $v_{\boldsymbol{f}_j}$ connects to a second layer of *board value nodes* v_{b_j}. Each feature value node $v_{\boldsymbol{f}_j}$ is linked to a board value node v_{b_j} through a **Weighted Sum Factor** (a generalization of the pure sum factor) if and only if $\boldsymbol{f}_j \in \mathcal{F}(b_j)$. The idea behind these board value nodes is that they represent the sum of the underlying feature values on the board.

3. **Latent Board Value Nodes:** The second layer of board value nodes v_{b_j} is connected to a layer of *latent board value nodes* $\tilde{v}_{b_j}$. Similar to the Urgency Model, the expert's perceived value of the board is treated as a noisy observation of the true underlying value. This relationship is again modeled using a **Gaussian Mean Factor**.

4. **Difference Nodes:** The structure and connections of the difference nodes follow the same approach to that described in the Urgency Model. Here, we use a **Weighted Sum Factor** to compute the differences between the board value of the chosen move and the other board values. To reflect the expert's preference, we enforce that these differences are greater than zero using a **Greater-Than Factor**.

3.2.3. Inference on Models

Using the sum-product algorithm (see Section 2.4), we can compute the marginals for each u_j (or $v_{\boldsymbol{f}_j}$ in BoardVal Model), i.e the posterior probability distributions of the urgencies u_j (or feature values $v_{\boldsymbol{f}_j}$) after incorporating the expert decision m_1. Efficient inference is achieved through approximate message passing. Specifically, we apply several noteworthy approximations in this model:

- The approximation introduced by the Greater-Than factor is significant. Although the moments are matched (see Theorem 3), the overall shape of the resulting distribution differs substantially (see Section 3.3.4, specifically Figure 3.7).

[1] We use a "plate" notation in Figure 3.2 to iterate over all features of a board. Strictly speaking, the factor graph may appear misleading, but it should be understood that the same feature $\boldsymbol{f}_j$, whether referenced from one board b_{i_1} as $f_{i_1,k}$ or from another board b_{i_2} as $f_{i_2,l}$, is actually a single feature connected to a single prior factor, not multiple ones.

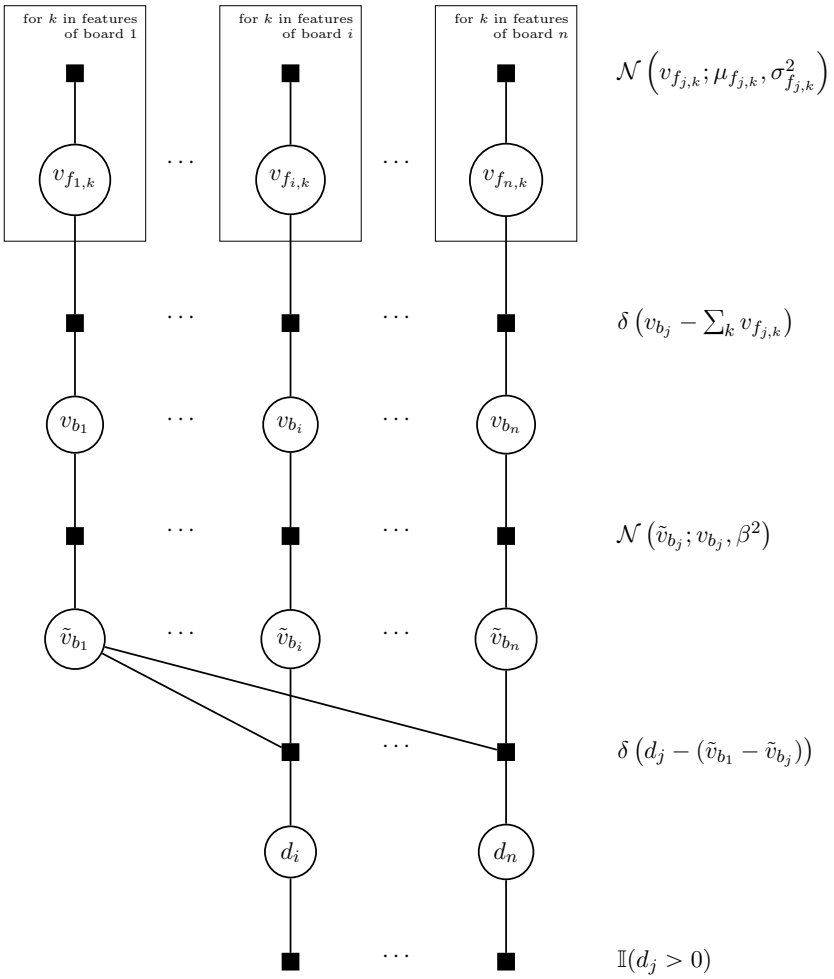


Figure 3.2.: The factor graph for the BoardVal Model. The first layer represents the feature values $v_{\boldsymbol{fi}}$, with priors modeled as Gaussian distributions. Each board value v_{b_j} is computed as the sum of feature values on the corresponding board via Weighted Sum Factors. The latent board values $\tilde{v}_{b_j}$ in the next layer are noisy observations of these board values, linked through Gaussian Mean Factors. The layer of difference nodes enforces that the board resulting from the expert's chosen move has a higher value than the others, using Weighted Sum and Greater-Than Factors.

- We assume that the prior $p(\boldsymbol{u})$ over urgencies fully factorizes into n independent Normal distributions $p(u_i)$ (and similarly that the prior over feature values fully factorizes). This assumes independence, which may (or does) not hold in practice. While this assumption has drawbacks — particularly the loss of potential covariance between urgencies (or feature values) — it offers computational efficiency. Modeling the full covariance matrix with a multivariate Gaussian over a prior vector $\boldsymbol{u}$ (or $\boldsymbol{f}$) would not scale well, given that the number of urgencies or features could reach tens of thousands or even several million.

- Additionally, we will use the marginals $p(u_j)$ conditioned on observed data (or written $p(u_j \mid m_1)$), as the prior $p(\boldsymbol{u})$ for the next match (similarly for feature values). This introduces two assumptions: first, that the posterior continues to factorize similarly to the prior, meaning it can be expressed as independent terms, i.e. $p(\boldsymbol{u} \mid m_1) = p(u_1 \mid m_1) \cdot p(u_2 \mid m_1) \cdot \cdots \cdot p(u_n \mid m_1)$. Second, we would ideally construct a large graph that incorporates all expert decisions across all positions in the dataset, not just for one game. However, this would result in a graph too large to store in memory. To address this, we adopt an "iterative" learning strategy, using the posterior after each game as the new prior. This approximate message passing method, known as *assumed-density filtering* [46], introduces an unintended dependence on the order in which decisions are processed, meaning the ranking outcome can vary depending on the sequence of training data.

The inference process follows a general schedule for both the Urgency Model and Board-Val Model, involving iterative updates, specifically using *expectation propagation*, as discussed in the paragraph on approximate message passing in Section 2.4.5.

Schedule

1. Pass messages "downwards" from the prior factors (1D-Gaussian Factors) and intermediary factors, up to and including the Gaussian Mean Factors, to update the latent variable nodes ($\tilde{u}_j$ in the urgency model or $\tilde{v}_{b_j}$ in the board value model).

2. For every $j \in 2, ..., n$:
   a) Pass a message down to the difference node d_j via the Weighted Sum Factor.
   b) "Apply" the Greater-Than factor, and approximate the marginal at d_j with a Gaussian distribution. Compute the approximate message by dividing the marginals.
   c) Send messages back from the difference node d_j to the involved latent nodes $\tilde{u}_1$ and $\tilde{u}_j$ (or $\tilde{v}_{b_1}$ and $\tilde{v}_{b_j}$ in the board value model).

3. Repeat this iterative process until convergence, i.e., when the difference between consecutive approximated marginals of the difference variables d_j falls below a threshold Δ.

4. Finally, pass messages upwards from the latent variable nodes via all intermediary factors to update the urgencies u_j (Urgency Model) or feature value $v_{\boldsymbol{f}_j}$ (BoardVal Model).

3.3. Factors

In this section, we provide a detailed overview of the factors involved in the models and present the factor-specific message update rules. We focus on deriving the update equations for all **incoming** messages (from factors to variables), as we can obtain the marginals by multiplication of incoming messages (and outgoing messages can be obtained by division, as detailed in the paragraph on storage efficiency in Section 2.4.5). The message and marginal updates always follow the same sequence of steps, as outlined below:

1. Consider a factor f connected to random variables $X_1, \ldots, X_n$. Assume we have marginal distributions $p(x_i)$ and incoming messages $m_{f \to X_i}$ (which, in the case of a freshly initialized state, are uniform (Gaussian) distributions; see paragraph on scheduling in Section 2.4.5 for reference).

2. Compute the (missing) outgoing messages $m_{X_i \to f}$ by dividing the marginals by the incoming messages: $\frac{p(x_i)}{m_{f \to X_i}(x_i)}$.

3. Consider the case where we want to update the marginal and messages involved with x_j. Use the outgoing messages $m_{X_i \to f}$ (in the general case $i \neq j$) to calculate the new (or updated) incoming message $m_{f \to X_j}$, as described in Equation (2.33).

4. Update the marginal $p(x_j)$ by multiplying the new incoming message $m_{f \to X_j}$ with the outgoing message $m_{X_j \to f}$, as described in Equation (2.36)

5. Replace the old marginal and incoming message with the updated values. The outgoing messages are temporary results and can be discarded afterward.

3.3.1. 1D-Gaussian Factor

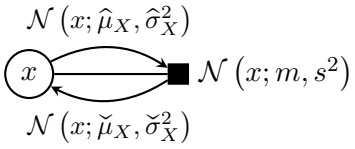


Figure 3.3.: The 1D-Gaussian Factor acts as an "evidence" or "initialization" factor that sends a $\mathcal{N}(x; m, s^2)$ to the associated normally distributed variable X.

If we want to initialize a normally distributed random variable X with fixed mean $m \in \mathbb{R}$ and variance $s^2 \in \mathbb{R}^+$ or have observed m for an unknown normally distributed variable

X with fixed variance s^2, we use the *1D-Gaussian Factor* that is formally specified by (see also Figure 3.3)

$$f(x) = \mathcal{N}\left(x; m, s^2\right) . \tag{3.1}$$

Message $m_{f\to X}$ Since there are no incoming messages from other variables to the factor, (2.33) results in

$$m_{f\to X}(x) = f(x) = \mathcal{N}\left(x; m, s^2\right) . \tag{3.2}$$

Note that this message is also well defined for a Dirac delta or a Gaussian uniform message; in the latter case, one would choose the natural parameters $\widehat{\tau}_X$ and $\widehat{\rho}_X$ for the factor definition.

3.3.2. Gaussian Mean Factor

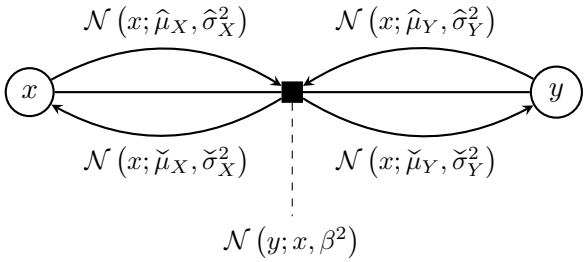


Figure 3.4.: The Gaussian Mean Factor acts as a copy factor with additional variance $\mathcal{N}\left(y; x, \beta^2\right)$ with both X and the Y being normally distributed.

If we want to model a normally distributed random variable Y with a mean X that is also normally distributed with fixed variance $\beta^2 \in \mathbb{R}^+$, we use the *Gaussian Mean Factor* that is formally specified by (see also Figure 3.4)

$$f(x, y) = \mathcal{N}\left(y; x, \beta^2\right) . \tag{3.3}$$

Message $m_{f\to Y}$ Using the definition (2.33), we know that

$$\begin{aligned} m_{f\to Y}(y) &= \int_{-\infty}^{+\infty} f(x, y) \cdot m_{X\to f}(x)\ \mathrm{d}x \\ &= \int_{-\infty}^{+\infty} \mathcal{N}\left(x; y, \beta^2\right) \cdot \mathcal{N}\left(x; \widehat{\mu}_X, \widehat{\sigma}_X^2\right)\ \mathrm{d}x \\ &= \mathcal{N}\left(y; \widehat{\mu}_X, \widehat{\sigma}_X^2 + \beta^2\right) , \end{aligned} \tag{3.4}$$

where we used $\mathcal{N}\left(y; x, \beta^2\right) = \mathcal{N}\left(x; y, \beta^2\right)$ in the second line and Theorem 1 in the final line (noticing that the integral over $\mathcal{G}\left(x; \beta^{-2} \cdot y + \widehat{\sigma}_X^{-2} \cdot \widehat{\mu}_X, \beta^{-2} + \widehat{\sigma}_X^{-2}\right)$ integrates to one over x). Note that the factor becomes a pure copy factor for $\beta^2 \to 0$.

Message $m_{f\to X}$ Noticing that $\mathcal{N}\left(y; x, \beta^2\right) = \mathcal{N}\left(x; y, \beta^2\right)$ we see that the message equation for $m_{f\to X}(\cdot)$ is symmetric to (3.4) and given by

$$m_{f\to X}(x) = \mathcal{N}\left(x; \widehat{\mu}_Y, \widehat{\sigma}_Y^2 + \beta^2\right) . \tag{3.5}$$

Practical Considerations Both message equations (3.4) and (3.5) are well defined for non-uniform messages $\mathcal{N}\left(x; \widehat{\mu}_X, \widehat{\sigma}_X^2\right)$ and $\mathcal{N}\left(y; \widehat{\mu}_Y, \widehat{\sigma}_Y^2\right)$ but deteriorate for uniform incoming messages to the factor f. However, rewriting the message equation in terms of natural parameters solves this because

$$m_{f\to y}(y) = \mathcal{G}\left(y; \frac{\widehat{\mu}_X}{\widehat{\sigma}_X^2 + \beta^2}, \frac{1}{\widehat{\sigma}_X^2 + \beta^2}\right) = \mathcal{G}\left(y; \frac{\widehat{\tau}_X}{1 + \beta^2 \cdot \widehat{\rho}_X}, \frac{\widehat{\rho}_X}{1 + \beta^2 \cdot \widehat{\rho}_X}\right) \tag{3.6}$$

$$m_{f\to x}(x) = \mathcal{G}\left(x; \frac{\widehat{\mu}_Y}{\widehat{\sigma}_Y^2 + \beta^2}, \frac{1}{\widehat{\sigma}_Y^2 + \beta^2}\right) = \mathcal{G}\left(x; \frac{\widehat{\tau}_Y}{1 + \beta^2 \cdot \widehat{\rho}_Y}, \frac{\widehat{\rho}_Y}{1 + \beta^2 \cdot \widehat{\rho}_Y}\right) , \tag{3.7}$$

where we used (2.15) and (2.16) repeatedly. Note that for the case of a Gaussian uniform incoming message, the outgoing message is also a Gaussian uniform.

3.3.3. Weighted Sum Factor

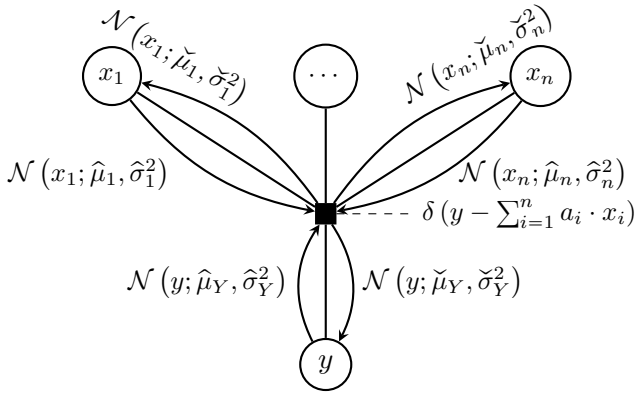


Figure 3.5.: The Weighted Sum Factor which acts as a pure weighted addition factor $\delta\left(y - \sum_{i=1}^n a_i \cdot x_i\right)$ with all the x_i's and the y being normally distributed.

One common factor to use for linear function models is the *Weighted Sum Factor* which maps the output y to the weighted sum $\sum_{i=1}^n a_i \cdot x_i$ of n inputs $x_1, \ldots, x_n$ with the $a_i \neq 0$ as fixed weights. Formally, this factor is specified by (see also Figure 3.5)

$$f(x_1, x_2, \ldots, x_n, y) = \delta\left(y - \sum_{i=1}^n a_i \cdot x_i\right) . \tag{3.8}$$

Message $m_{f\to Y}$ In [33] it is shown that this message is given by:

$$m_{f\to Y}(y) = \mathcal{N}\left(y; \sum_{i=1}^{n} a_i \cdot \widehat{\mu}_i, \sum_{i=1}^{n} a_i^2 \cdot \widehat{\sigma}_i^2\right) . \tag{3.9}$$

Intuitively, the message from the factor to the sum Y is a Gaussian with mean equal to the sum of the weighted means of the incoming messages from all summands and variance equals to the the sum of the variances of the incoming messages from all summands weighted by the square of the a_i's.

Message $m_{f\to X_j}$ In [33] it is shown that this message is given by:

$$m_{f\to X_j}(x_j) = \mathcal{N}\left(x_j; \frac{1}{a_j} \cdot \left(\widehat{\mu}_Y - \sum_{i\neq j} a_i \cdot \widehat{\mu}_i\right), \frac{1}{a_j^2} \cdot \left(\widehat{\sigma}_Y^2 + \sum_{i\neq j} a_i^2 \cdot \widehat{\sigma}_i^2\right)\right) . \tag{3.10}$$

Similarly to $m_{f\to Y}(\cdot)$, the message from the factor to each summand X_j is a Gaussian with mean equal to the difference of the weighted means of the incoming messages from all other summands and the mean of the sum (and the variance equals to the the sum of the variances of the incoming messages from all other summands and the sum weighted by the square of the a_i's).

Practical Considerations A single message computation to any of the n summands $X_1, \ldots, X_n$ takes $O(n)$ computations for the summation of $\widehat{\mu}_i$ and $\widehat{\sigma}_i^2$. Thus, using (3.10) for all the n messages $m_{f\to X_j}(\cdot)$ takes $O\left(n^2\right)$. However, any two updates for $m_{f\to X_i}(\cdot)$ and $m_{f\to X_j}(\cdot)$ have $n-2$ terms in common. Thus, if we precompute in $O(n)$

$$c = \widehat{\mu}_Y - \sum_{i=1}^{n} a_i \cdot \widehat{\mu}_i \tag{3.11}$$

$$d = \widehat{\sigma}_Y^2 + \sum_{i=1}^{n} a_i^2 \cdot \widehat{\sigma}_i^2 , \tag{3.12}$$

and then compute the update n times in $O(1)$

$$m_{f\to X_j}(x_j) = \mathcal{N}\left(x_j; \frac{1}{a_j} \cdot c + \widehat{\mu}_j, \frac{1}{a_j^2} \cdot d - \widehat{\sigma}_j^2\right) , \tag{3.13}$$

we have achieved a simultaneous update to all n summands with an $O(n)$ message update algorithm.

3.3.4. Greater-Than Factor

When we observe a random variable X to be greater than zero, or we want to enforce a constraint such that its distribution has non-zero probability only for positive values, we use the Greater-Than Factor that is formally specified by (see also Figure 3.6)

$$f(x) = \mathbb{I}(x > 0) \tag{3.14}$$

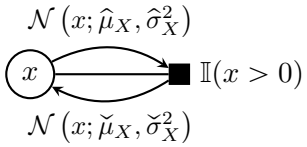


Figure 3.6.: Illustration of the Greater-Than Factor, which imposes a constraint that the random variable X must be greater than 0. This factor is used to incorporate evidence or constraints into the model by restricting the distribution to non-negative values only.

True Message $m_{f\to X}$ Since there are no incoming messages from other variables to the factor, (2.33) results in

$$m_{f\to X} = \mathbb{I}(x>0) \tag{3.15}$$

As we can see, the message $m_{f\to X}$ is a non-Gaussian distribution. Thus, we need to consider the approximation of the posterior $p(x)$ using Moment Matching (see Theorem 3).

True Posterior $p(x)$ Using definition (2.32), we know that

$$p(x) = \mathbb{I}(x>0)\cdot\mathcal{N}\left(x;\widehat{\mu}_X,\widehat{\sigma}_X^2\right)\,. \tag{3.16}$$

Approximate Posterior $\hat{p}(x)$ In [33] it is shown that $\hat{p}(x)$ is given by:

$$\hat{p}(x) = \mathcal{N}\left(x;\widehat{\mu}_X+\widehat{\sigma}_X\cdot v\left(\frac{\widehat{\mu}_X}{\widehat{\sigma}_X}\right),\widehat{\sigma}_X^2\cdot\left(1-w\left(\frac{\widehat{\mu}_X}{\widehat{\sigma}_X}\right)\right)\right)\,, \tag{3.17}$$

where

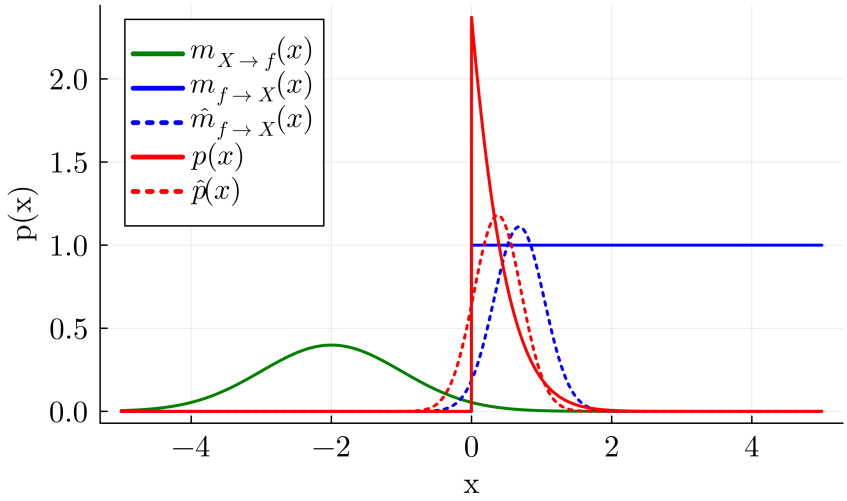
$$v(t) := \frac{\mathcal{N}(t)}{\Phi(t)}\,, \tag{3.18}$$

$$w(t) := v(t)\cdot(v(t)+t)\,. \tag{3.19}$$

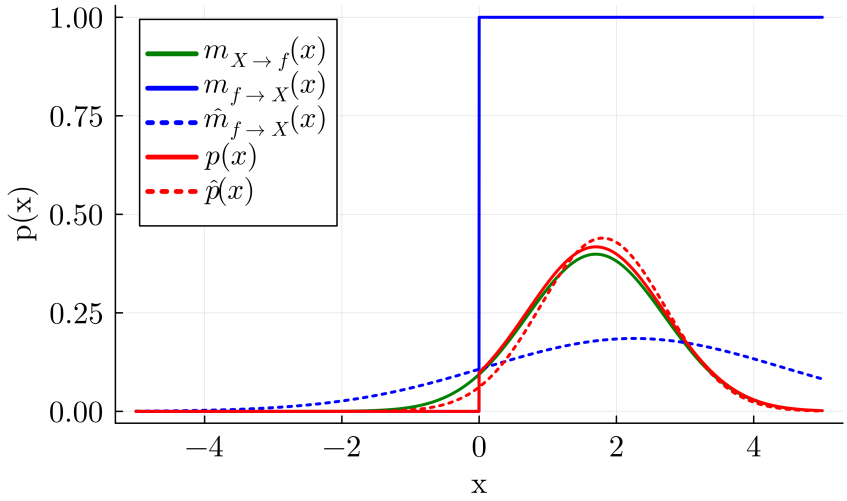
Approximate Message $\hat{m}_{f\to X}$ Using (2.37) we can now approximate the incoming message $m_{f\to X}$ by

$$\hat{m}_{f\to X} = \frac{\hat{p}(x)}{m_{X\to f}(x)}\,. \tag{3.20}$$

The approximation of the posterior works well when the prior belief about the random variable X (outgoing message) suggests that X is positive (see Figure 3.7b). However, when the initial belief indicates that X is likely negative, the approximation of the posterior $p(x)$ becomes quite poor (see Figure 3.7a). Despite this, in comparison to the approximate incoming message, the approximate posterior is still reasonably accurate, while the incoming message's distribution deviates significantly from its true shape.



(a) Approximation of the posterior distribution $p(x)$ and incoming message $m_{f \to X}(x)$ when the prior belief suggests that X is negative ($m_{X \to f}(x) = \mathcal{N}(x; -2.0, 1.0)$). The Gaussian approximation $\hat{p}(x)$ deviates significantly from the true shape of the distribution $p(x)$.



(b) Approximation of the posterior distribution $p(x)$ and incoming message $m_{f \to X}(x)$ when the prior belief indicates that X is positive ($m_{X \to f}(x) = \mathcal{N}(x; 1.7, 1.0)$). The approximation $\hat{p}(x)$ is considerably more accurate in this case.

Figure 3.7.: Approximate distributions in Greater-Than Factor

4. Implementation

4.1. High-Level Overview

```
src/
├── core/
├── engine/
├── parsing/
│   └── clean.jl
│   └── filter_elo.jl
│   └── ...
├── pattern-learning/
│   └── analysis/
│       └── <model analysis tools>
│   ├── gaussian.jl
│   ├── factors.jl
│   ├── ranking.jl
│   ├── train.jl
│   ├── test.jl
│   └── ...
└── <more scripts>
```

Figure 4.1.: The chess engine code is divided into the `core` and `engine` directories. The `core` folder contains the main chess-specific library code, including move generation and other functionality relevant to model training. The `engine` folder extends this with search algorithms and an UCI interface for engine communication, though this part is not directly relevant for this thesis. The `parsing` folder handles the processing of training data. The most significant folder for this thesis is `pattern-learning`, which contains all the files related to the implementation of the model discussed in Chapter 3.

The codebase is publicly available in the GitHub repository `Poppy.jl`, which can be accessed here[1]. The relevant code for this Bachelor thesis is currently located on the branch `bachelor-thesis`.

[1] `https://github.com/aherbrich/Poppy.jl`

The code is organized across several folders: `core`, `engine`, `parsing`, and `pattern-learning`. Of these, the `pattern-learning` folder is the most important for this thesis, as it contains all files related to the implementation of the factor graph model. A more detailed description of the project structure can be found in the caption of Figure 4.1. All code is written in Julia. This decision was made because Julia provides a scripting-like interface, which allows for rapid development and refinement of code. Additionally, Julia offers impressive speed (often comparable to this of a compiled low-level language like C), despite its garbage collection, and has robust libraries such as `Plots.jl` for plotting and data analysis tools.
The file `train.jl` serves as the best entry point into the code (see Listings 4.1 and 4.2 for a modified but conceptually identical file overview; concentrating on the BoardVal Model). Through this, we can gain an understanding of the most relevant code and interfaces used:

Listing 4.1: Julia code for training the BoardVal Model on dataset of chess games

```
function train_model(training_file; with_prediction=false, <other
    ↪ args>)
    # INITIALIZE EMPTY MODEL
    feature_values = Dict{UInt64, Gaussian}()

    # METADATA
    metadata = TrainingMetadata(training_file)

    # TRAIN MODEL
    games = open(training_file, "r")
    while !eof(games)
        game_str = strip(readline(games))

        # TRAIN ON GAME
        train_on_game(game_str, feature_values, metadata,
            ↪ with_prediction=with_prediction)

        # ... code for model dumping (temp saving)
    end

    # code for model saving
end
```

Listing 4.2: Extension of Listing 4.1: Julia code for training the BoardVal Model on a single game

```
function train_on_game(game_str, feature_values, metadata;
    ↪ with_prediction=false)
    # SET BOARD INTO INITIAL STATE
    board = Board()
    set_by_fen!(board, "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/
        ↪ RNBQKBNR w KQkq - 0 1")
```

```
5
6       # TRAIN ON EVERY MOVE IN GAME
7       move_strings = split(game_str)
8       for (i, move_str) in enumerate(move_strings)
9           # SORT THE PLAYED MOVE TO THE FIRST POSITION
10          _, legals = generate_legals(board)
11          move = extract_move_by_san(board, move_str)
12          best_move_idx = findfirst(mv -> mv.src == move.src && mv.
                ↪ dst == move.dst && mv.type == move.type, legals)
13          legals[1], legals[best_move_idx] = legals[best_move_idx],
                ↪ legals[1]
14
15          # MAKE A PREDICTION WITH THE CURRENT MODEL
16          if with_prediction
17              prediction = predict_on(feature_values, board, legals)
18              push!(metadata.predictions, prediction)
19          end
20
21          # ... irrelevant intermediary code
22
23          # UPDATE THE MODEL WITH NEW DATA POINT
24          features_of_all_boards = extract_features_from_all_boards(
                ↪ board, legals)
25          ranking_update!(feature_values, features_of_all_boards)
26          do_move!(board, move)
27      end
28  end
```

The central object is the `feature_values` object. It is defined as a dictionary of type `Dict{UInt64, Gaussian}` in line 3 of Listing 4.1 and holds the most critical part of our model: the Gaussians for our feature values, which are accessible through a unique feature ID represented by a 64-bit unsigned integer.
This object is passed to two functions: the `predict_on` function in line 17 of Listing 4.2, and the `ranking_update!` function in line 25 of the same listing.

- **Model Learning:** The `ranking_update!` function handles all components involved in building the factor graph and inference using the sum-product algorithm. As an additional argument it gets the list of IDs `features_of_all_boards`, which itself is calculated in the line above in the function `extract_features_from_all_boards` to strictly isolate the model learning from the actual features being used. More information on model learning is found in Section 4.4.

- **Model testing:** The `predict_on` function, initially used during training and later in a dedicated testing function, encapsulates the prediction component and, together with the metadata object on line 6 of Listing 4.1, plays a key role in testing the model. Further details on model testing are provided in the Section 4.5.

- **Data Collection:** As indicated in lines 9 and 10 of Listing 4.1 and lines 7 and 8

of Listing 4.2, the algorithm iterates over the file of games, processing each move in a game. Details about the file format, including how these games were selected and processed, are covered in Section 4.3.

- **Chess Engine:** The `board` object and the functions `set_by_fen!`, `generate_legals`, and `do_move!` (located on lines 3, 4, 10, and 26 of Listing 4.2) provide chess-specific functionality. Further details on chess-specific code, particularly the custom-built chess engine *Poppy* are described in the Section 4.2.

4.2. Chess-Specific Functionality

The core of the model is a custom-built chess engine named *Poppy*, designed for simplicity and effective performance. Considerable effort was invested in optimizing move generation speed, as it is crucial for efficient training; slow move generation can significantly hinder the model's performance during game simulations. The final move generator in *Poppy* is only a factor three slower than leading chess engines like Stockfish[2]. Additionally, extensive debugging and validation were performed to ensure that the move generation is accurate and produces all and only legal moves, using a testing suite known as `perft`[3] to compare results against precomputed benchmarks[4].
While an in-depth exploration of the optimization techniques used is beyond the scope of this thesis, in the following are the key design decisions made:

Board Representation

Two primary approaches are typically used for representing a chessboard: square-centric and piece-centric. In the square-centric approach, an array is used to store the piece located on each square. In contrast, the piece-centric approach uses a bitboard for each piece, which utilizes 64-bit registers to represent board positions. Each bit in the bitboard (64-bit number) corresponds to a square on the board, with the bit set to 1 if the square is occupied by a specific piece and 0 otherwise. This piece-centric bitboard method is typically faster than the square-centric because it leverages efficient bitwise operations. For example, moving a piece from a3 to b3 involves shifting the bitboard (64-bit number) 8 bits to the left.

Listing 4.3: Julia struct for a chessboard

```
1 mutable struct Board
2     # bitboards (piece-centric)
3     const bb_for::Vector{UInt64}
4
5     # (redundant) bitboards
```

[2]On the testing system (Apple M1), *Poppy* processes approximately 90 million positions per second, while Stockfish averages about 220 million positions.
[3]`https://www.chessprogramming.org/Perft_Results`
[4]The exact test and validation file, in specific the file `src/test/data/perft_big.txt`, was adapted from the test file used in the Blunder chess engine [47], which is available under the MIT license.

```
    bb_white::UInt64
    bb_black::UInt64
    bb_occ::UInt64

    # (redundant) array representation (square-centric)
    const squares::Vector{UInt8}

    # ... some other fields

    # irreversible information
    history::Vector{IrreversibleInfo}
end

mutable struct IrreversibleInfo
    hash::UInt64            # unique zobrist hash
    # .. some other fields
end
```

As can be seen in Listing 4.3. in *Poppy*, we use a hybrid approach. We maintain a `bb_for::Vector{UInt64}` array that holds bitboards for each of the 12 piece types (with the piece type represented by a number which acts as an index into the array). Additionally, we keep redundant bitboards for all white pieces, black pieces, and occupied squares, along with an array `squares::Vector{UInt8}` that holds the pieces of the board (square-centric). This redundancy, while potentially slowing down move generation due to multiple board updates, facilitates flexible access and analysis of the board from various perspectives.
Additionally, we use a `hash::UInt64` field to store a nearly unique identifier for each board position. This identifier is computed using Zobrist Hashing, a technique known for its efficiency and low collision rate in hashing board structures [48]. This hash facilitates various data analysis tasks, such as determining the number of unique board positions in the dataset.

Move Generator Design

Another key decision is whether to implement a legal or pseudo-legal move generator. A pseudo-legal generator computes all possible moves without initially checking their legality, filtering out illegal moves later[5]. This approach can be faster, especially if search algorithms are optimized, as it avoids the overhead of legality checks for moves which are never searched. However, *Poppy* uses a legal move generator, which directly produces only legal moves[6]. This design choice simplifies the code by ensuring that move

[5] A move is considered illegal if it leaves the king in check, even if it follows the rules of how the pieces move.

[6] The logic with which only legal moves are produced, was inspired by the move generator Surge [49], which somewhat closely follows the approach described in Peter Ellis Jones' blog post "Generating Legal Chess Moves Efficiently" [50], though Poppy's final implementation still differs quite substantially.

legality is handled within the move generation process, enhancing the readability and maintainability of the code for model training.
Regarding the interface provided by *Poppy*, it is designed to expose minimal underlying complexity. The primary functionalities include setting a board using a FEN string (Forsyth-Edwards Notation), extracting the FEN of a board, and performing and undoing moves. Moreover, the `board` and `move` object grant access to specific fields like the bitboards or source and destination squares of a move, if necessary while model training, testing or analysis. These functions, objects and so forth are implemented in the `core/` folder.
While *Poppy* has the potential to play chess, including extending functionalities, this is not the focus of this thesis. The relevant code for such features can be found in the `engine/` folder, which contains functionalities related to chess engine search and chess engine communication. For this thesis, these aspects are not utilized and are considered archived.

4.3. Data Collection and Parsing

The data used to train the model was collected from one of the most popular online chess platforms, Lichess.org. Lichess provides monthly game archives for download here[7]. These games are available in the .pgn (Portable Game Notation) format, an example of which is shown in Figure 4.2

```
[Site "https://lichess.org/Sj83ND8N"]
[BlackElo "2458"]
[WhiteElo "2517"]
[Result "0-1"]

1. g3 d5 2. Bg2 Nf6 3. Nf3 c5 4. O-O Nc6 ...
```

Figure 4.2.: Excerpt of a .pgn file for a chess game

Since the .pgn format is not ideal for clean and efficient processing of moves and games during model training, the files were preprocessed. Specifically, the script `filter_elo.jl` filters games based on a player's Elo rating, allowing for the creation of datasets based on different skill levels. Another script, `clean.jl`, cleans the .pgn files by removing unnecessary metadata, line breaks, spaces, and move numbers between moves.
This preprocessing step is crucial because the raw .pgn files are massive, often tens or even hundreds of gigabytes in size. Filtering by Elo and cleaning the files reduces their size to a few megabytes, making them much easier to handle. The smaller, cleaned datasets fit into memory, preventing memory swapping that would otherwise slow down

[7] `https://database.lichess.org/`

the training process. After preprocessing, each line in the training file represents a single game, with moves separated by a single space.

4.4. Inference - From Theory to Implementation

As outlined in Chapter 2 and 3, our model learning involves three key components: (1) variables and messages, (2) factors, and (3) the actual sum-product algorithm.

Gaussians

All variables and messages in the model are exclusively Gaussian-distributed. The definition of a Gaussian object in our code is straightforward:

Listing 4.4: Julia struct for a Gaussian

```
mutable struct Gaussian
    tau::Float64
    rho::Float64

    Gaussian(tau, rho) = (rho < 0.0 || isnan(rho) || isnan(tau) ||
        ↪  isinf(tau)) ? error("Invalid Gaussian parameters") :
        ↪ new(tau, rho)
end
```

As one can see in Listing 4.4, the natural parameterization of the Gaussian is used, as detailed in Equation (2.12). This choice simplifies multiplication and division, which are performed through the addition and subtraction of parameters (see Theorem 1 and 2). Additionally, in the sum-product algorithm, messages and some marginals need to be initialized as uniform distributions, which means infinite variance. This is more conveniently represented as `rho=0` than setting the variance to `Inf`. When needed, the mean and variance can be extracted via helper functions that handle this conversion. Multiplication and division are implemented through function overloading of base operators, a feature that is natively supported in Julia.
In general, all Gaussian-related code is found in `gaussian.jl`, with extensive error handling built into all operations like multiplication and division. Although this adds a slight performance overhead, it ensures robustness, especially given the numerical instabilities and edge cases encountered during the sum-product algorithm. Without these safeguards, invalid Gaussians (e.g., those with `NaN` or `Inf` parameters) could be generated unnoticed, leading to critical errors. Thus, the error handling is essential for the model's stability.
Additionally, as shown in Listing 4.4, the Gaussian `struct` is defined as `mutable`. Although this is less performant —since it allocates heap memory rather than stack memory — it allows Gaussians to be stored directly within Factor-objects, as demonstrated in Listing 4.5. This enables in-place updates of the Gaussians (i.e. modifying the same object) when applying factor-specific update rules. As a result, any changes to the Gaus-

sians (through the Factor-objects) are automatically reflected in the `feature_values` dictionary (our model).

Factors

The implementation of factors is found in `factors.jl`. The structure of a factor object in our code is as follows:

Listing 4.5: Julia struct for a Factor (Example: Gaussian Mean Factor)

```
struct GaussianMeanFactor <: Factor
    x::Gaussian
    y::Gaussian
    msg_to_x::Gaussian
    msg_to_y::Gaussian
    beta_squared::Float64
end
```

In Listing 4.5, we see the example of a Gaussian Mean Factor. The factor object stores the marginals of the variables it is connected to and the messages from factor to variables. Each factor has corresponding `update_msg_to_variable!` functions (as shown in Listing 4.6 and 4.7) that update the message and marginal for that specific variable. In Listing 4.6, we demonstrate the message update process for variable `x` in the Gaussian Mean Factor:

Listing 4.6: Julia function for updating marginal and messages involved with variable `x` in a Gaussian Mean Factor

```
function update_msg_to_x!(f::GaussianMeanFactor)
    msg_from_x = f.x / f.msg_to_x
    msg_from_y = f.y / f.msg_to_y

    # UPDATE THE MESSAGE TO X
    if (isdirac(msg_from_y))
        update!(f.msg_to_x, GaussianByMeanVariance(mean(msg_from_y
            ↪ ), beta_squared))
    elseif (isuniform(msg_from_y))
        update!(f.msg_to_x, GaussianUniform())
    else
        c = 1.0 / (1.0 + f.beta_squared * msg_from_y.rho)
        update!(f.msg_to_x, GaussianByMeanPrecision(c * msg_from_y
            ↪ .tau, c * msg_from_y.rho))
    end

    # UPDATE THE DISTRIBUTION OF X
    new_marginal_of_x = msg_from_x * f.msg_to_x
    diff = absdiff(f.x, new_marginal_of_x)
    update!(f.x, new_marginal_of_x)

    return diff
```

```
21 end
```

As shown, the (missing) outgoing messages are computed first. The incoming message to `x` is then updated using the factor-specific message update rule (Equation (3.7)). Special cases, such as uniform Gaussians or Dirac delta functions, are handled as needed. After updating the message, the marginal is updated accordingly. In lines 17 and 20, we compute the `absdiff` between the old and new marginal, which measures how similar they are. This difference is tracked for every update (and all factors), allowing us to monitor convergence, especially when a factor is involved in an approximate message-passing update scheme, as discussed in the paragraph on scheduling in Section 3.2.3)

Ranking Model Update via Sum-Product Algorithm

The Listing 4.7 shows the core function for model learning - the function `ranking_update!`. In this case, consistent with Listing 4.1 and 4.2, it implements the model update for the BoardVal Model. The function first constructs the factor graph for a specific game. The relevant board features are provided by a list of IDs in `features_of_all_boards`, which act as keys to the underlying Gaussians stored in the `feature_values` dictionary. Once the factor graph is built (line 2-10), the sum-product algorithm is applied using the scheduling outlined in Section 3.2.3.

Listing 4.7: Julia code for execution of factor graph creation and sum-product algorithm

```
1  function ranking_update!(feature_values, features_of_all_boards)
2      ##############################################
3      # FACTOR GRAPH CREATION
4
5      sum_factors = Vector{SumFactor}()
6      gaussian_mean_factors = Vector{GaussianMeanFactor}()
7      difference_factors = Vector{DifferenceFactor}()
8      greather_than_factors = Vector{GreaterThanFactor}()
9
10     # ... some code to build the actual factor graph
11
12     ##############################################
13     # SUM PRODUCT ALGORITHM
14
15     ### FORWARD-PASS
16     for (i, factor) in enumerate(sum_factors)
17         update_msg_to_sum!(factor)
18     end
19
20     for (i, factor) in enumerate(gaussian_mean_factors)
21         update_msg_to_x!(factor)
22     end
23
24     ### RUN UNTIL LOOP CONVERGES
25     loop_eps = 1e-3
```

```
26
27      eps = 10 * loop_eps
28      while eps > loop_eps
29          eps = 0.0
30          for (i, factor) in enumerate(difference_factors)
31              eps = max(eps, update_msg_to_z!(factor))
32              eps = max(eps, update_msg_to_x!(greather_than_factors[
                    ↪ i]))
33              eps = max(eps, update_msg_to_x!(factor))
34          end
35      end
36
37      ### BACKPASS
38      for (i, factor) in enumerate(difference_factors)
39          update_msg_to_y!(factor)
40      end
41
42      for (i, factor) in enumerate(gaussian_mean_factors)
43          update_msg_to_y!(factor)
44      end
45
46      for (i, factor) in enumerate(sum_factors)
47          # update feature nodes (=summands)
48          update_msg_to_summands!(factor)
49      end
50  end
```

4.5. Model Testing

Testing occurs in two main stages. **First,** we optionally test during training if `with_prediction` is set to `true`, as seen in lines 16-19 of Listing 4.2. In this case, before incorporating each move into the model, a prediction is made using the model's current state. This allows for unbiased testing by ensuring that the model is not evaluated on the same data it is trained on — a crucial aspect of model evaluation [51]. Predicting at every move lets us assess accuracy throughout the training process, helping to identify trends in under-performing models early, without needing to train on the entire dataset. For instance, in the BoardVal model, where a specific feature set must be defined for use in `extract_features_from_all_boards`, this prediction while training helps flag poor feature sets early. Similarly, in the Urgency Model, where moves can generally be represented in different ways, early testing can highlight ineffective move representations without requiring full dataset training. The final feature sets and move representations used and further analysed are listed in Chapter 5 - the Results.
Secondly, we use a dedicated testing function in `test.jl`, as shown in Listing 4.8. The testing employs a simple holdout method, splitting the dataset into training and test sets. Though *k-fold cross-validation* is another common testing method [51], we opted

against it here because of the large dataset (millions of moves/ranking decisions). Cross-validation is more beneficial when datasets are smaller or have high variance, helping to average out inconsistencies. We initially tested with cross-validation and found it unnecessary due to the dataset's size, as the overhead of running multiple tests outweighed the near identical test error as in a simple holdout method.
The model is trained on a subset of games and then saved to a binary file. This file is passed to the testing function via the `filename_model` argument, and the games included in training, which are to be excluded while testing, are specified with `exclude_games` (see Listing 4.8).

Listing 4.8: Julia code for testing the BoardVal Model on a dataset of chess games

```
1  function test_model(filename_model, test_file; exclude_games=[])
2      # LOAD MODEL
3      feature_values = load_model(filename_model)
4      metadata = TestMetadata(test_file, exclude_games)
5
6      # TEST MODEL
7      open(test_file, "r") do games
8          for (idx, game_str) in enumerate(eachline(games))
9              if idx in exclude_games continue end
10
11             # TEST ON GAME
12             test_on_game(game_str, feature_values, feature_set,
                   ↪ metadata)
13         end
14     end
15
16     return metadata
17 end
```

In the `test_on_game` function in line 12 in Listing 4.8, the prediction is performed similarly to how it is done during training (see Listing 4.2) using the `predict_on` function. The accuracy, as calculated by the `predict_on` function within the model, is determined as follows (using the BoardVal Model as an example): We assume that the expert selects the move leading to a board where the sum of all feature values is the highest. These feature values are stored in the `feature_values` dictionary of Gaussians, allowing us to compute the corresponding normally distributed board values. The challenge, then, is how to identify the "winning" move, given that we are dealing with distributions rather than single values. For simplicity and efficiency, we use only the means of the Gaussians, determining that the board with the highest mean value is the predicted choice of the expert. The ranking is strict — our prediction is considered correct only if the chosen move has a strictly greater mean value than all other moves.
A prediction is represented in the code by an object, as shown in Listing 4.9.

Listing 4.9: Julia struct for a prediction

```
1 struct Prediction
```

```
    predicted_values::Vector{Float64}
    predicted_rank::Int
    ply_number::Int
    nr_of_possible_moves::Int
    move_type::UInt8

    # ...potentially more information
end
```

As can be seen in Listing 4.9, the prediction object stores more than just a binary indicator of correctness. It also includes additional information, such as the board (or move urgency) values involved, the specific half-move (ply) in which the move occurred, and the move type (e.g., pawn move, castling). This extra data facilitates more detailed analysis later on.
All relevant data, such as predictions, are stored in structures called `TrainingMetadata` or `TestMetadata`, defined in the `metadata.jl` file (see Listing 4.10).

Listing 4.10: Julia struct for metadata

```
mutable struct TrainingMetadata
    processed::Int
    nr_of_games::Int
    const predictions::Vector{Prediction}
    const model_files::Vector{AbstractString}
    global_start_time::Float64
    # ... potentially more metadata
end

mutable struct TestMetadata
    processed::Int
    nr_of_games::Int
    const predictions::Vector{Prediction}
    global_start_time::Float64
    # ... potentially more metadata
end
```

Both the `prediction` and `metadata` structures are designed to be flexible, allowing easy expansion to include additional fields for further model analysis.
In general, all code related to prediction, accuracy assessment, and various analysis tools, including functions for plotting data, can be found in the `pattern-learning/analysis` sub-folder.

5. Results

In the following, we present the results for both the Urgency Model and BoardVal Model, with the goal of predicting moves. The training and testing of the models were conducted locally on a system with an Apple M1 processor and 16GB of RAM. We used Julia Version 1.10, running in single-threaded mode.

Testing Method: We employed a simple holdout method for testing. Starting with an initial dataset of 16,232,215 games, we filtered the dataset to include only games where both players had an Elo rating of 2300 or higher. This resulted in 69,599 games (approximately 0.43% of the original dataset). These were further split into a training set of 59,599 games and a test set of 10,000 games, yielding around 5,000,000 moves in the training set and 840,000 moves in the test set. As explained in Section 4.5, predictions were made using the μ value (mean) from the Normal distribution of urgencies (or board values), selecting the move corresponding to the highest μ value.

Hyperparameters: The following hyperparameters were used for training:

> **Urgency Model:** The prior distribution over urgencies was set as a standard Normal $\mathcal{N}(\cdot; 0, 1)$, with $\beta = 1$ for the noise introduced in the (latent) Gaussian Mean Factor.
>
> **BoardVal Model:** For the prior over board features, we used $\mathcal{N}\left(\cdot; 0, \frac{1}{c}\right)$, where c corresponds to the number of features on the board where a feature was first encountered. This choice ensures that the prior for the board value (as the sum of its features, derived in Section Weighted Sum Factor) follows a $\mathcal{N}(\cdot; 0, 1)$ distribution. Note that this is an approximation, as a feature can appear on multiple boards, all with a different numbers of features. A β-value of 2 was selected, adding a significant amount of noise — lower values unfortunately caused critical NaN errors during training (further discussion on this issue is provided in Chapter 6, the Discussion).

Feature Sets and Move Representations: Below, we introduce the abbreviations for the models, feature sets, and move representations used in the plots and discussions.

> **Urgency Model:** A move can be represented in different ways. Specifically, we use three representations: *Piece-Type* (PT), *Move-Type* (MT), and *Piece-Type-Move-Type* (Hybrid), all of which encode the source and destination squares along with additional information as described below.

- **Piece-Type (PT):** A move is represented by the combination *src* × *dst* × *piece-type*, where the *piece-type* specifies which piece (pawn, knight, bishop, rook, queen, king) of which color (white or black) is being moved.
- **Move-Type (MT):** A move is represented by *src* × *dst* × *move-type*. The *move-type* can be one of the following: quiet (no special characteristics), double pawn push, king castle, queen castle, capture, en passant, promotion (to knight, bishop, rook, or queen), or promotion capture (to knight, bishop, rook, or queen).
- **Piece-Type-Move-Type (Hybrid):** A move is represented by the combination *src* × *dst* × *piece-type* × *move-type*. This is also referred to as the **Hybrid** move representation, as it combines both the piece and move type.

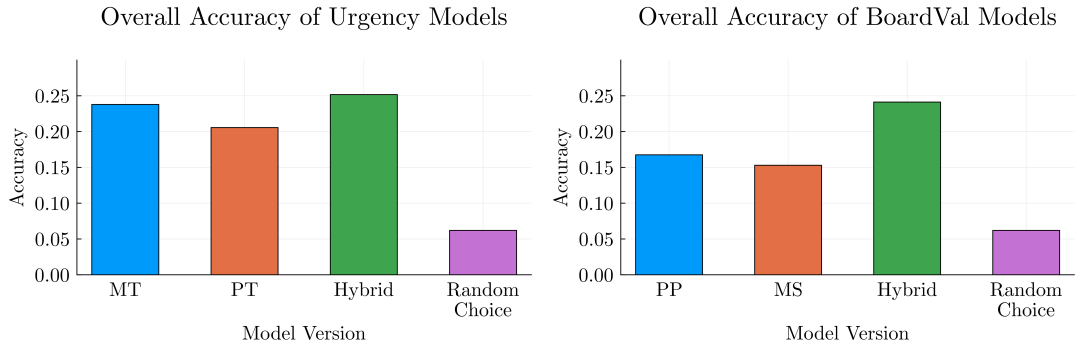
BoardVal Model: This model captures the board state using specific feature sets: *Piece-Position* (PP), *Move-Set* (MS), and *Piece-Position-Move-Set* (Hybrid).

- **Piece-Position (PP):** The board is represented by features that describe which specific pieces are on which squares. Each feature is a combination of *sq* × *piece-type*. For example, if a board has a white pawn, a white king, and a black king on squares *a1*, *a4*, and *c6* respectively, the features would be: *a1-wp*, *a4-wk*, and *c6-bk*.
- **Move-Set (MS):** The board is represented by features that describe which moves are possible for the position. Each move is represented using the same *Hybrid* move representation as defined earlier.
- **Piece-Position-Move-Set (Hybrid):** This representation combines both the positions of pieces on the board and the possible moves. Moves are represented using the *Hybrid* format, while pieces are represented by *sq* × *piece-type*. This is also referred to as the **Hybrid** feature set, as it combines both piece positions and possible moves.

Model Random: As a baseline, we include a model that selects a move at random (from the set of possible moves) to evaluate whether our models have learned anything meaningful.

5.1. Accuracy

Overall Accuracy Figure 5.1a presents the accuracy of the Urgency Model using different move representations (*Piece-Type*, *Move-Type*, and the *Hybrid*), achieving accuracies of 20.6%, 23.8%, and 25.2%, respectively. In comparison, Figure 5.1b shows the accuracy of the BoardVal Model across various feature sets (*Piece-Position*, *Move-Set*, and the *Hybrid*), with corresponding accuracies of 16.8%, 15.3%, and 24.1%. Both models show a clear improvement over the Random model, which achieves only 6.2% accuracy. The Urgency Model demonstrates more consistent performance across its different move representations, while the BoardVal Model shows greater variation between its feature



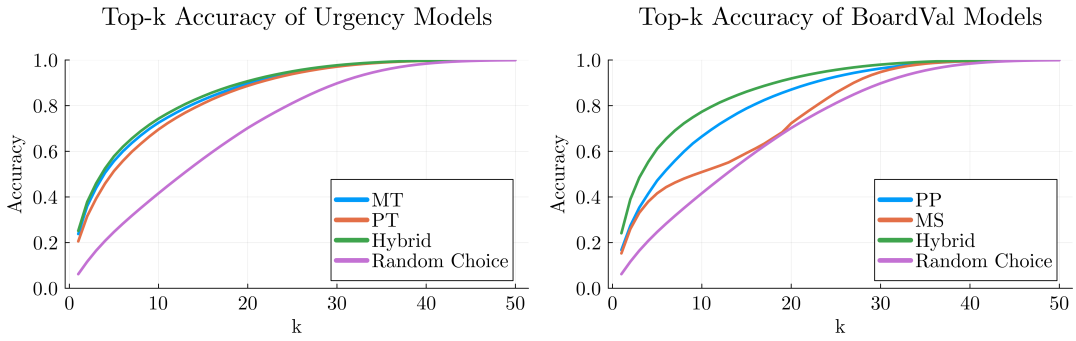
(a) Accuracy of the Urgency Model across different move representations

(b) Accuracy of the BoardVal Model across different feature sets

Figure 5.1.: Accuracy of the Urgency and BoardVal Model. The Urgency Model (Figure 5.1a) achieves accuracies of 20.6%, 23.8%, and 25.2% with the *Piece-Type*, *Move-Type*, and *Hybrid* move representations, respectively. The BoardVal Model (Figure 5.1b) attains accuracies of 16.8%, 15.3%, and 24.1% using the *Piece-Position*, *Move-Set*, and *Hybrid* feature sets, respectively. The random model has an overall accuracy of 6.2%.

sets. In both models, the hybrid approach performs best. Overall, comparing the most accurate versions, the Urgency Model appears to slightly outperform the BoardVal Model.

Top-k Accuracy The top-k accuracy measures how often the expert's move appears within the top k highest-ranked moves predicted by each model.
Figure 5.2a shows the top-k accuracy of the Urgency Model for various move representations. As k increases, the accuracy improves, approaching 100%. Initially, the accuracy of the Urgency Model across all representations increases rapidly, but the rate of improvement slows as k grows. Consistent with overall accuracy, the *Hybrid* representation performs the best, followed by *Move-Type* and then *Piece-Type*. However, the differences in performance between these representations are generally small, only differing by a few percentage points. All models consistently outperform the Random model for different values of k.
Similarly, Figure 5.2a presents the top-k accuracy for the BoardVal Model. While the overall shape of the curve is similar to that of the Urgency Model, the differences between feature sets are more pronounced. For k values between 5 and 15, the *Hybrid* feature set outperforms the second best *Piece-Position* feature set by around 8%-10%. Interestingly, the *Move-Set* feature set performs particularly poorly for k values between 5 and 30, reaching its lowest point at $k = 18$, where the BoardVal Model performs no better than random guessing.
Figure 5.3 compares the best performing move representation with the best performing feature set. In the Top 5, both models rank the expert move with an accuracy of 57.8% (61.1% for the BoardVal Model); in the Top 10, accuracy rises to 74.3% (77.3%); and



(a) Top-k accuracy of the Urgency Model across different move representations

(b) Top-k accuracy of the BoardVal Model across different feature sets

Figure 5.2.: Top-k accuracy of the Urgency and BoardVal Model, which measures how often the expert's move appears in the top k highest-ranked moves by the model. Both models show a substantial improvement over the random baseline across most configurations. However, the BoardVal Model's *Move-Set* feature set performs notably worse for moderate values of k, approaching random performance for certain ranges.

in the Top 20, it reaches 90.7% (91.9%). Although the BoardVal Model's accuracy is approximately 1% lower than that of the Urgency Model at $k = 1$ (perfect prediction), it begins to outperform the Urgency Model as k increases, peaking with a 3.4% accuracy advantage at $k = 8$. Beyond $k = 25$, the performance difference between the two models becomes negligible, with both displaying nearly identical accuracy.

Accuracy vs. Training Data Size Figures 5.4a and 5.4b illustrate how model accuracy improves as the size of the training data increases. For both models, accuracy rises somewhat rapidly within the first few thousand games before gradually leveling off, approaching the performance of a fully (or near fully) trained model. The x-axis is plotted on a log2 scale, so the linear increase—particularly noticeable in the Urgency Model—reflects exponential improvements in accuracy.
In the Urgency Model, by the time the model has been trained on 1024 games (2^{10}), both the *Move-Type* and *Piece-Type* representations have nearly converged. However, the *Hybrid* representation continues to improve, gaining about 2% additional accuracy even after the other representations plateau. It reaches full convergence around 60,000 training games.
The BoardVal Model follows a similar pattern of rapid initial improvement, but the *Piece-Position* and *Move-Set* feature sets show more gradual trends, with accuracy increasing steadily but at a slower rate compared to the Urgency Model. Interestingly, the *Hybrid* feature set maintains a faster learning rate, even after 128 games (2^7), whereas the other feature sets have already begun to plateau. The *Hybrid* set outperforms the others, gaining a notable accuracy advantage, especially after 60,000 training games, where it

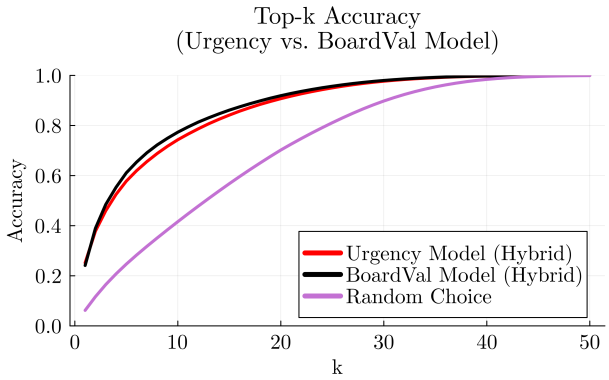
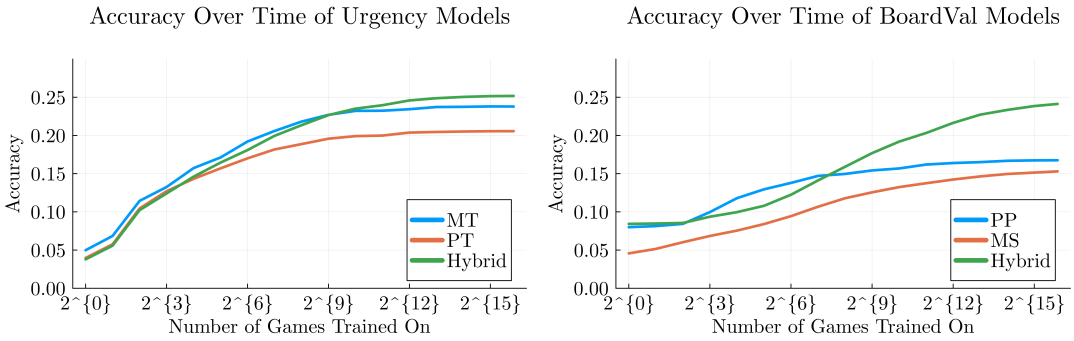


Figure 5.3.: Top-k accuracy comparison between the best performing Urgency Model (Hybrid) and best performing BoardVal Model (Hybrid). The Urgency Model outperforms the BoardVal Model for perfect predictions (k=1) by approximately 1%. However, as k increases, the BoardVal Model begins to perform better, reaching it's peak advantage of 3.4% at $k = 8$. In the Top 5, the better performing BoardVal Model ranks the expert move with an accuracy 61.1%; in the Top 10, accuracy rises to 77.3%; and in the Top 20, it reaches 91.9%.

continues to improve slightly—suggesting that the model has not yet fully converged.

Accuracy vs. Game Stage Figures 5.5a and 5.5b illustrate how model performance varies across different stages of the game, plotting accuracy against the half-move number (ply). Both models, except the *Move-Set* feature set for the BoardVal Model, follow a similar trend: they perform best in the early game, where the Urgency Model achieves up to 40% accuracy and the BoardVal Model around 32%, both far surpassing the Random model's sub-5% accuracy. However, accuracy declines between moves 12 and 25, followed by a gradual improvement as the game progresses into the mid and late stages. Despite this recovery, neither model's mid- and late-game accuracy returns to its early-game peak. Interestingly, this pattern of steady improvement as the game progresses is also observed in the Random model, suggesting that the trend may be caused by the reduced number of possible moves as the game advances, rather than a reflection of model superiority. The *Move-Set* feature set in BoardVal Model stands out as an exception— it does not benefit from the early-game accuracy spike and starts with very low accuracy.

Figure 5.6 compares the best performing Urgency Model (Hybrid) with the best performing BoardVal Model (Hybrid). From ply 20 onwards, there is little to no difference in accuracy between the two models. The main distinction arises in the early game, where the Urgency Model consistently outperforms the BoardVal Model by 5%-10%. This early-game advantage appears to be the primary contributor to the Urgency Model's overall higher accuracy.



(a) Accuracy as a function of training data size for the Urgency Model across different move representations

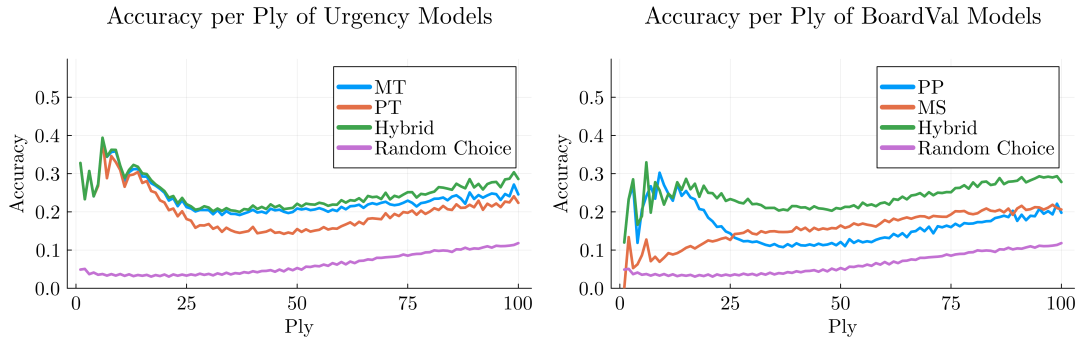
(b) Accuracy as a function of training data size for the BoardVal Model across different feature sets

Figure 5.4.: Accuracy as a function of training data size for the Urgency and BoardVal Models. The x-axis is plotted on a log2 scale, meaning that the early linear increases represent exponential improvements in accuracy as more training data is used. All configurations, except for the *Hybrid* feature set of the BoardVal Model, have plateaued after 60,000 training games, as indicated by the horizontal slope for most models at that point.

Accuracy vs. Move Type Figures 5.7a and 5.7b explore the model's performance across different move types.

The Urgency Model performs exceptionally well on promotion, king-side castling, and en passant moves, achieving accuracies between 90% and 95%. It also yields solid results on capture moves, reaching approximately 60% accuracy with the *Move-Type* and *Hybrid* representations. However, the model struggles with quiet moves, where accuracy drops to between 10% and 15%, representing only a modest improvement over the Random model's 6.5%. Interestingly, the *Piece-Type* move representation performs poorly on captures and promotions, achieving only 31% accuracy on capture moves (compared to 60% for other representations) and 0% on promotions — worse than the Random model. This poor performance on promotions can be explained by a key issue: in the *Piece-Type* move representation, all promotion moves (knight, bishop, rook, and queen) share the same move representation since it reflects the piece type of the moving piece (a pawn). As a result, the model cannot distinguish between these promotion moves. This suggests a straightforward solution: using the piece type on the destination square instead of the source square.

Similarly, the BoardVal Model exhibits variations in accuracy based on move type, though the improvements are less pronounced than those of the Urgency Model. Notably, the BoardVal Model performs poorly on promotion moves, with results barely above random guessing. Overall, the best performing Urgency Model (Hybrid) significantly outperforms the BoardVal Model across most move types. The BoardVal compensates for this by maintaining a slight edge of 5.8% higher accuracy on quiet moves, which constitute the



(a) Accuracy per ply for the Urgency Model across different move representations

(b) Accuracy per ply for the BoardVal Model across different feature sets

Figure 5.5.: Accuracy per ply for the Urgency and BoardVal Models. Both models perform best in the early stages of the game (ply 0-20). Performance declines in the mid and late-game stages, gradually improving over time (as like the random baseline), but never returning to the initial peak. A notable exception is the *Move-Set* feature set in the BoardVal Model, which follows a different pattern.

	Quiet	**Double Push**	**King Castle**	**Queen Castle**	**Capture**	**En Passant**	**Promotion**
Absolute	584281	57155	15101	2221	184376	545	2553
Relative	69.05%	6.75%	1.79%	0.3%	21.79%	0.06%	0.3%

Table 5.1.: The table lists the absolute and relative occurrences of each move type in the test set. Quiet moves are by far the most common, followed by captures and double pawn pushes. King-side castling breaks the 1% mark, while queen-side castling, en passant, and promotion moves are much rarer, each accounting for less than 1% of the total moves.

majority (69.05%) of all moves as listed in Table 5.1.

5.2. Interpretability and Efficiency

When considering the interpretability and efficiency of a predictive model, one of the most fundamental yet important specifications is the actual weights (parameters) of the model. These weights define our trained model and enable us to make predictions. In our case, the parameters of our model correspond to the urgencies (or feature values).

Number of Board Features and Moves Table 5.2 presents the number of board features learned during training—those encountered throughout the training process — alongside the total number of possible board features one could have encountered[1]. Similarly,

[1]The number of possible features for each feature set was calculated as 64×12, $64 \times 64 \times 14$, and $64 \times 12 + 64 \times 64 \times 14$, respectively - using the number of possible values for *src* (64), *dst* (64),

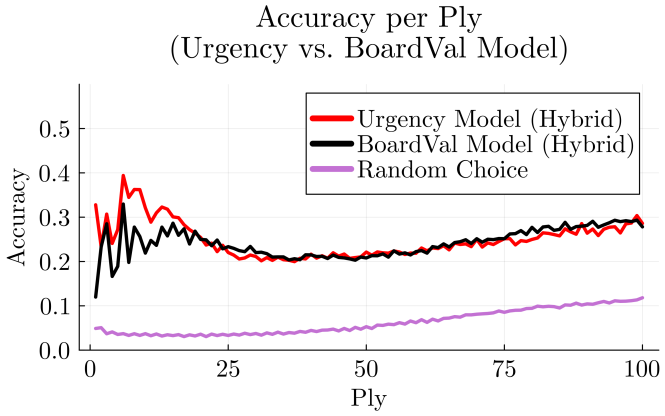


Figure 5.6.: Accuracy per ply comparison between the best-performing Urgency Model (Hybrid) and BoardVal Model (Hybrid). The Urgency Model reaches a peak accuracy of 40%, while the BoardVal Model peaks at 32% - both in early stages of the game. In the later stages, both models show nearly identical performance.

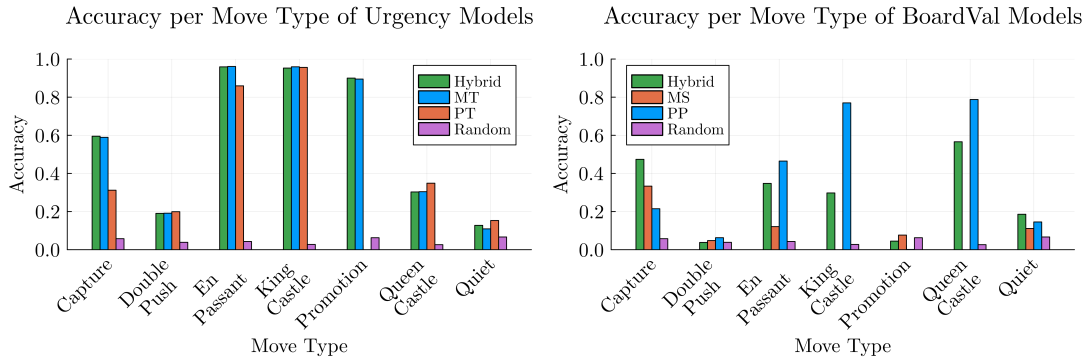
	PP	**MS**	**Hybrid**
Number of features encountered (observed)	732	3808	4541
Number of theoretical features (possible)	768	57344	58112
Ratio observed vs. possible	95.3%	6.6%	7.8%

Table 5.2.: Number of observed and possible board features for each feature set

Table 5.3 provides insight into the number of move urgencies learned versus the number of unique moves possible under each move representation[2]. Notably, for most feature sets and move representations, only a small fraction of the possible board features (or moves) is encountered during training. For example, in the Urgency Model, using the *Hybrid* representation, one encounters only about 2.2% of the possible moves. This suggests that many combinations of source square, destination square, piece type, and move type either do not occur in actual game-play (since chess rules do not allow for them) or are extremely rare at an expert level of chess. A notable exception is the *Piece-Position* feature set of the BoardVal Model, where nearly all possible features are encountered while training (95.3%). On closer inspection, we notice the following: Since pawns cannot legally occupy the first or last ranks of the chessboard, there are in fact only $64 \times 10 + 48 \times 2 = 732$ valid positions, meaning that all 732 (true) possible features are seen at least once during training. This finding is not particularly surprising, given that we trained on 60,000 games; it is reasonable to assume that every piece occupied

move-type (14), and *piece-type* (12).

[2]The number of possible moves for each representation was calculated as $64 \times 64 \times 12$, $64 \times 64 \times 14$, and $64 \times 64 \times 12 \times 14$, respectively.



(a) Accuracy per move type for the Urgency Model across different move representations

(b) Accuracy per move type for the BoardVal Model across different feature sets

Figure 5.7.: Accuracy per move type for the Urgency and BoardVal Models. Both models exhibit move-specific accuracy differences. The Urgency Model excels at promotions, king-side castling, and en passant moves, achieving accuracies above 90%. In contrast, while the BoardVal Model also shows improvements for specific move types, these are less pronounced overall. However, the BoardVal Model maintains a slight edge, with 5.8% higher accuracy on quiet moves (the majority, at 69.05%, as noted in Table 5.1), for the best performing *Hybrid* configuration.

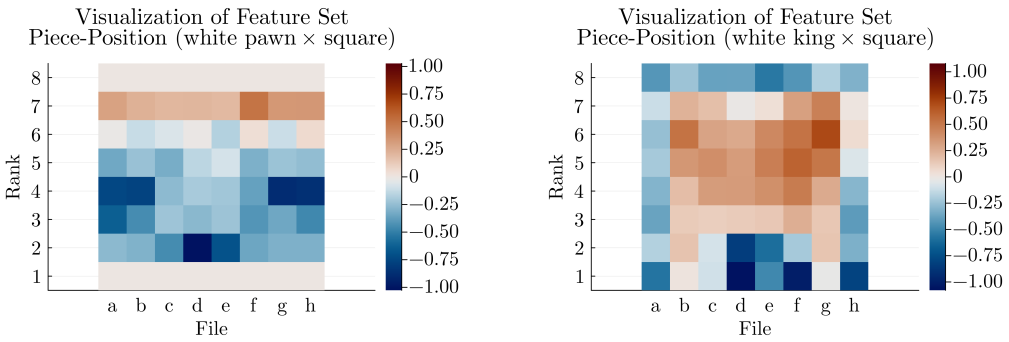
	PT	**MT**	**Hybrid**
Number of moves encountered (observed)	7620	3808	15107
Number of theoretical moves (possible)	49152	57344	688128
Ratio observed vs. possible	15.5%	6.6%	2.2%

Table 5.3.: Number of observed and possible moves for each move representation

every square at some point within such a large dataset.

Visualization of Board Features and Moves Next, we visualize the features and move representations for the example move representation *Piece-Type* and the feature set *Piece-Position*.

Figures 5.8a and 5.8b visualize the *Piece-Position* feature set, specifically showing heat maps for the white pawn and the white king. For each square *sq*, we output the mean value of the feature *piece-type* × *sq*, using colors that range from blue to beige. In these heat maps, blue indicates negative mean values, while beige represents positive mean values. Similarly, Figures 5.9a and 5.9b visualize the *Piece-Type* move representation, displaying heat maps for the black pawn and black queen. For each combination of source (*src*) and destination (*dst*) squares, we output the mean value of the move urgencies *src* × *dst* × *piece-type*, with colors ranging from red to blue. Blue indicates negative values,



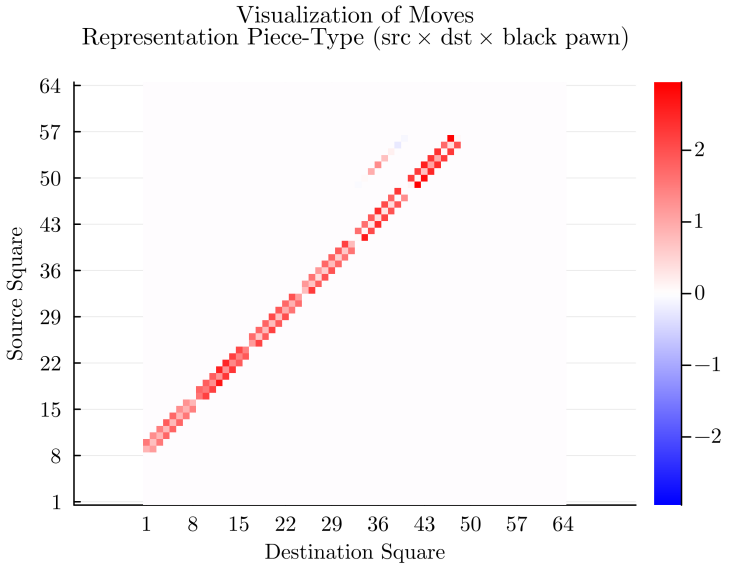
(a) Visualization of features involving the white pawn

(b) Visualization of features involving the white king

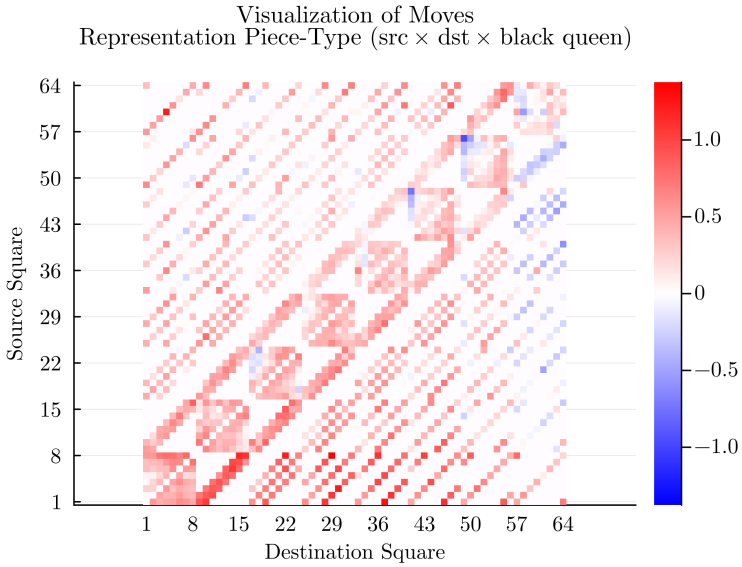
Figure 5.8.: Visualizations of the *Piece-Position* feature set. Figures 5.8a and 5.8b display heat maps for the white pawn and the white king, respectively. Each tile represents the mean value of the feature *piece-type* × *sq*, with color gradients ranging from blue (indicating negative values) to beige (indicating positive values).

and red represents positive values. It is noteworthy that there is a significant proportion of white area in the heat map. White signifies a zero mean, primarily caused by many *src* to *dst* combinations simply being impossible. The value for these moves is assumed to be zero (i.e., a mean of 0). Overall, the fascinating patterns that emerge here possess a unique structure dictated by the specific rules of chess, which restrict how pieces can move. These visualizations allow for intriguing interpretations that we will explore in the next chapter, the Discussion.

Training Time and Resource Allocation Regarding computational requirements, as briefly mentioned at the beginning of this chapter, the models were trained on a system equipped with an Apple M1 processor and 16 GB of RAM. Training the Urgency Model with 60,000 games took approximately 10 minutes, while the BoardVal Model required slightly more time, ranging from 20 to 30 minutes on the same system. In general, training the BoardVal Model takes longer for larger feature sets because the sum-factor involves more message updates (to every board feature). In terms of memory consumption, there is nothing significant to report; the number of features (or urgencies) is, at least for the current feature sets and move representations, small enough that memory poses no restriction. Additionally, at no point during training not even during the construction of larger factor graphs (60+ boards, each with 80+ features) — did memory impose any constraints. We also want to note that the code is optimized for readability and reproducibility and includes model dumping and debugging tools. Moreover, as noted in the Implementation chapter, the factor graph updates and Gaussian object incorporate many built-in error handlers. We believe that an optimized version of the code could run at least an order of magnitude faster than the current version.



(a) Visualization of moves involving the black pawn



(b) Visualization of moves involving the black queen

Figure 5.9.: Visualization of moves under the *Piece-Type* move representation. Figures 5.9a and 5.9b display heat maps for the black pawn and the black queen, respectively. Each tile represents the mean value for the combination *src* × *dst* × *piece-type*, with color gradients ranging from blue (indicating negative values) to red (indicating positive values). Many tiles show a mean of zero, indicating that certain moves from one square to another do not exist.

6. Discussion

In this chapter, we conduct a critical analysis of the results from our explainable chess move prediction approach using inference over a ranking model to learn this function. We will interpret the findings and contextualize them within related research. Furthermore, we will discuss the implications of our results, delve into potential improvements and areas for future research, and highlight the limitations of our work.

Both the Urgency and BoardVal Models (in there best versions) achieve an accuracy of approximately 25%, which is four times higher than that of a random model (see Figures 5.1a and 5.1b). While 25% may not seem particularly impressive at first glance, it is important to note that even strong human players often disagree on the best move, as pointed out by Stern et al. in their 2006 paper "Bayesian Pattern Ranking for Move Prediction in the Game of Go" [45]. For comparison, the model applied to Go achieved an accuracy of 34%, which was considered excellent. Although Go presents significantly more possible moves — especially in the early stages, where hundreds of options exist compared to the typical 10-50 moves in a chess position — we believe that our prediction accuracy of 25% is still reasonably strong. It essentially increases the odds from 1 in 20 to 1 in 4.

Upon further inspection, the difference in overall accuracy between our models and the Go model likely stems from how the moves are represented, which significantly enhances the Go model's early game prediction performance. In the Go model, moves were represented by the resulting board state, specifically the largest possible sub-board found in the pattern base. This allowed for near-perfect pattern matching and highly accurate predictions in the early stages, as evidenced by the statement that "in almost all cases where [the model] match[ed] a full board pattern, the resulting ranking g[ave] a perfect prediction of expert play" [45]. In contrast, our models do not directly incorporate the board state, which limits their ability to achieve perfect pattern matching. This limitation is reflected in the early phases of the game, where we observed accuracies between 30% and 40% (see Figures 5.5a and 5.5b). Although these figures are relatively strong, they fall short of the near-perfect predictions seen in the Go model during the same game stage. While this limits peak accuracy, we view it as a trade-off for better generalization, especially since our model was designed with interpretability as a key focus.

When comparing the Urgency Model and the BoardVal Model, several interesting observations arise. The motivation for introducing the alternative BoardVal Model stemmed from a potential limitation of the Urgency Model: it ranks moves somewhat independently of the board context in which they occur. For example, consider the move *e2-e3*, which can occur in various situations. The Urgency Model learns to rank this move based on its urgency across all instances where it is a possible option, with each instance contributing to the model's learning. This results in an average urgency value that applies to

all cases. However, we thought this approach might overlook important context-specific differences, such as when *e2-e3* might be optimal in one position but suboptimal in another. To address this, we introduced the BoardVal Model, which aimed to capture these context-dependent nuances. Instead of ranking moves in isolation, the BoardVal Model ranks the resulting board positions based on their board value.

To our surprise, the results show that the BoardVal Model does not perform better than the Urgency Model. When comparing the best-performing versions of each model (the *Hybrid* feature set for BoardVal and the *Hybrid* move representation for Urgency), the Urgency Model still has a 1% higher overall accuracy (see Figures 5.1a and 5.1b). This performance gap widens even further when comparing models using less effective feature sets and move representations.

Interestingly, when we examine top-k accuracy, the BoardVal Model, while having worse perfect prediction accuracy at $k = 1$, gains additional accuracy as k increases. At $k = 8$, the BoardVal Model outperforms the Urgency Model by up to 3.4% (see Figure 5.3). This suggests that the BoardVal Model may indeed be capturing some of the context-specific improvements we aimed for. The notion of stronger and weaker moves seems to integrate better in this model than in the Urgency Model. Supporting this theory, the BoardVal Model shows a 5.8% higher accuracy when ranking quiet moves (see Figures 5.7a and 5.7b), which often have a more subtle, strategic and context-dependent nature[1]. In contrast, the Urgency Model excels in more decisive moves, such as capture, promotion and castling moves. We believe this is due to the way such moves are encoded. Consider the following example: if a capture is the expert move, the Urgency Model can easily detect it (through the capture flag encoded into the move representation), while the BoardVal Model would need to infer this through features of the resulting board, making it harder to "recognize". Similarly, in positions where promotion or castling is the optimal move, the Urgency Model performs better, likely due to its straightforward approach to move-specific tasks.

In terms of accuracy over different stages of the game, both models perform similarly, with the Urgency Model having a slight edge in the early game (see Figure 5.6). The reason for this is unclear, and any explanation would be speculative. Overall, the two models achieve comparable accuracy but through different mechanisms. The Urgency Model performs particularly well in the early game and in move-specific scenarios like promotions and captures, while the BoardVal Model exhibits more consistent performance across all game stages and move types, potentially indicating better generalization. When analyzing accuracy in relation to training data size, we noticed that the BoardVal Model may not have fully converged yet (see Figure 5.4b), suggesting that both models could ultimately reach similar prediction performance. One additional, and notable advantage of the BoardVal Model is that it can not only be used for move prediction — such as ordering moves for exploration in search — but also for board evaluation.

When moving from interpreting results to suggesting potential improvements for the

[1] While chess-specific knowledge is typically passed down through word of mouth, blogs such as those by Grandmaster Pawel Eljanow and high-Elo player ThePianoManRyan provide insights into the subtlety and strategic significance of quiet moves [52, 53]

models, we identify four key observations that seem particularly promising:

Firstly, when we examine top-k accuracy, the performance curves for the Urgency and BoardVal models closely resemble those of the ranking model for the game of Go [45]. Although our models have slightly lower accuracy in the early stages, the general trend remains similar. For instance, at a relatively small value of $k = 5$, our models achieve approximately 61% accuracy (see Figure 5.3), compared to 66% in the Go model. At $k = 10$, our model reaches 77.3%, slightly ahead of the Go model's 76%. Notably, for larger k values, such as $k = 20$, our model shows approximately 6% higher accuracy than the Go model [45]. This suggests that we have somewhat replicated the top-k accuracy results, or at least the trend and shape of the curve, which indicates that we have implemented key aspects correctly. The fact that Go typically has more possible moves may explain the better performance of our model as k increases, allowing our model to converge toward 100% accuracy more quickly due to the lower average number of possible moves in chess.

More importantly though, unlike the linear top-k curve seen in the random model, our models show a sharp increase in accuracy at smaller k values (see Figure 5.3). This indicates that they are capturing the idea of stronger and weaker moves, even if they do not always predict the best move perfectly. We believe that improvements in overall accuracy, such as through better performing feature sets and move representations, should also significantly enhance top-k accuracy, particularly at the lower k values. Notably, improving top-k performance, especially in the range of top-3 to top-5, is crucial for search-based applications. In heuristic chess play (whether by humans or chess engines), identifying a small set of "best" move candidates is often equally (if not, more) valuable than finding the single optimal move. In fact, our top-k accuracy suggests that our models could already be used for move ordering.

Furthermore, our models have a significant advantage over typical move-ordering techniques, as they store values as Gaussians. This allows for an uncertainty measure in move ranking. For example, in Monte Carlo Tree Search, we could sample the Gaussians for all move urgencies (or resulting board values) and select the one with the highest value. This would provide a probabilistic move ordering, favoring those with higher means while still allowing for exploration of lower ranked moves through a measure of learnt variability.

Secondly, our analysis of model performance over time reveals promising results as training progresses. The models reach their final accuracy after training on just a few thousand games (see Figures 5.4a and 5.4b). For context, a chess expert might play around 600,000 games over their career (16 hours/day $\times$ 365 days/year $\times$ 20 years $\times$ 5 games/hour), though this is likely an overestimate, with the true figure closer to 10,000 to 100,000 games. The fact that our model converges so quickly suggests it can infer key aspects effectively, much like human experts, reaching a "high" level of understanding from a relatively small number of games.

An interesting observation is that simpler move representations and feature sets tend to achieve their final accuracy more quickly, whereas more complex representations take longer to converge but ultimately yield higher accuracy (see Figures 5.4a and 5.4b). Specifically, the *Hybrid* feature set of the BoardVal model has not fully converged yet,

suggesting that additional training could further improve its performance.

The convergence time appears related to the complexity of the move representation or feature set, which influences the number of unique features or moves encountered during training. For example, the simplest *Piece-Position* feature set in the BoardVal model encountered only 732 unique features, whereas the more complex *Hybrid* set encountered 4,541 (see Table 5.2). This suggests that the limited number of features in simpler sets (or moves in the Urgency Model) may constrain the model's predictive performance. Essentially, the simpler model attempts to predict expert moves with just a few values, while more complex feature sets offer a richer array of values, enabling finer-grained predictions. In other words, more complex feature sets (or move representations) provide the model with more "knobs" to fine-tune its predictions.

Interestingly, the *Piece-Position* feature set, despite having fewer features (732) than the *Move-Set* with 3,803 (see Table 5.2), achieves a slightly higher overall accuracy by 1.5% (see Figure 5.1b). Moreover, the *Move-Set* shows worse early game performance (see Figure 5.5b) and detrimental top-k performance for moderate k values (see 5.2b). This indicates that performance relies not only on feature complexity but also on the quality of the feature design. A well-constructed feature set appears to capture more relevant information, enhancing model performance. Consequently, future research could focus on developing more complex **and** well-designed yet interpretable move representations and feature sets. Overall, the differences in accuracy across various move representations and feature sets — particularly pronounced in the BoardVal model — suggest that the choice of feature set (or move representation) is critical, leaving hope for significantly stronger models, considering the simplicity of those explored in this work.

Despite the increasing complexity of such feature sets and move representations, we believe that training should remain feasible. As shown in Tables 5.2 and 5.3, while complex feature sets introduce many theoretical features, only a fraction of them are encountered during training. Importantly, since the model only needs to store a Gaussian distribution when a new feature or move is encountered, memory limitations are unlikely to become a concern. If the number of values does indeed grow too large, we propose a "garbage collection" strategy as a potential solution. The model could periodically review all urgency or feature values and remove those that are most similar to their prior values, retaining only a fixed number of significant values. This approach seems promising because if a value closely matches its prior, it either represents a rare move or, even if it is common, its removal would have little impact on performance, as the learned value was already near the prior.

Thirdly, we observed an intriguing pattern: the model performs better during the opening stages of the game compared to the mid and late game (see Figures 5.5a and 5.5b). While the exact cause is unclear, we speculate that this is because players often follow well-established opening lines, leading to more consistent expert behavior. In contrast, accuracy noticeably drops in the mid-game, likely due to the increased variability in expert decisions as the game progresses and diverges from known theory.

We believe that training the model on even higher-Elo games could improve its accuracy, as stronger players tend to exhibit more consistent and precise play. Unfortunately, the

proportion of even higher-Elo games in our dataset was too small to conduct meaningful analysis, and downloading additional data was not feasible on our current system[2]. However, with access to more memory or pre-Elo-filtered datasets, it would be possible to collect and train the model on a larger set of high-Elo games.

Given that training 60,000 games takes only 10 minutes on an Apple M1 chip, scaling up to train on millions of games is certainly feasible. Considering that more complex feature sets (or move representations) will likely require additional training to fully converge, it's encouraging to know that expanding the training process is both practical and achievable.

Finally, we noticed varying accuracy across different move types (see Figures 5.7a and 5.7b). For example, the model performs exceptionally well on promotion moves, likely because these moves are decisive and relatively predictable when they become available. Similarly, castling moves are predicted with high accuracy, as they are standard in certain situations. Notably, castling is only available during specific stages of the game and becomes impossible once played. It seems as if the model learns the typical timing and conditions under which experts castle. Capturing moves also show relatively strong accuracy, likely due to the consistent behavior associated with material gain or forced sequences. In contrast, quiet moves — those that do not immediately threaten the opponent— prove more challenging for the model, exhibiting poor accuracy. This difficulty may arise from the subtlety of such moves[3], which are harder to predict.

Interestingly, we found that move types with higher accuracy tend to be less frequent in the dataset (see Table 5.1). For example, while promotion moves are rare (0.3% of moves), they are almost always predicted correctly. In contrast, quiet moves with low accuracy occur far more frequently (69.05%).

We believe that this connects to our earlier observation that more complex feature sets and move representations seem to lead to higher overall accuracy, likely by allowing the model to better distinguish between different (type of) moves (on different boards). Specifically, the strong results in move-specific prediction tasks suggest that "subtyping" (e.g., into quiet, castling, and promotion as we did) is an effective what to introduce complexity into feature sets and move representations — an area we find intriguing for future research.

In particular, we propose an enhancement for the Urgency Model, although the exact implementation remains unclear: quiet moves are particularly challenging, likely due to their subtlety and context-dependent nature[4]. It therefore seems logical to include some kind of context flag in the move representation and further subtype this group. Specifically, if we could cluster boards into n groups based on some measure of similarity, we could add this cluster index to the move representation, thereby further subtyping moves by the context in which a move was played. This could split the quiet move group into n smaller categories, each potentially benefiting from subtype-specific accuracy improvements.

[2]Though the training file, after Elo-filtering and cleaning, is rather small (a few hundred Megabyte in size), the size of even larger raw data files was too large to unpack on the training and testing system.

[3]See footnote 1

[4]See footnote 1

Overall, we believe our model demonstrates good performance with significant potential for improvement, although we were unable to fully replicate the perfect prediction accuracy levels achieved in Stern's "Bayesian Pattern Ranking for Move Prediction in the Game of Go" [45].

To conclude, we would like to propose an additional adjustment that could further enhance model accuracy. As discussed in the Results chapter, we used a beta value of 1 (or 2 for the BoardVal model) during training. This is relatively high, particularly since the prior variance of features is also set to 1, essentially implying that the model accounts for as much (or more) noise as there is uncertainty in the target values. We opted for this higher beta to address NaN errors that emerged from the greater-than factor during training. Upon investigation, these errors arose when the model had already learned about a move but then encountered a situation where the expert move had a significantly lower urgency (or board value) compared to other options. In these instances, the model struggled to "explain" why the expert selected that move, leading to NaN errors because it could not assign a sensible probability to the outcome. We suspect this occurs when experts make substantial mistakes or "blunders." One possible solution would be to skip examples that result in NaN errors, which would allow us to use a smaller beta since we would not need to assume as much noise to account for these "unexplainable" situations. Additionally, we considered an alternative approach: modifying the greater-than factor to a probabilistic one, which would enforce the greater-than condition in p% of cases while allowing for exceptions in (1-p)% of cases. Although this adjustment was not included in the final model, initial tests suggest it could help prevent NaN errors and reduce the model's tendency to assign extreme values in nearly "unexplainable" cases, thereby minimizing the risk of extreme updates in the model.

Beyond accuracy, interpretability was a central focus of this research, even at the potential cost of some predictive performance. Fortunately, our model's design facilitates inherent interpretability, primarily due to the structured nature of the factor graph, in which each "layer" conveys a specific meaning. This stands in contrast to neural networks, where the interpretability of individual layers can be more opaque. Furthermore, the use of relatively straightforward feature sets and move representations enables effective visualization of the model's learned patterns. These visualizations provide valuable insights, showing that the model's results align with human understanding—a primary goal of this thesis.

For instance, in the *Piece-Position* feature set, the heatmap for the white pawn (see Figure 5.8a) indicates that advancing toward the promotion squares (the second-to-last rank) is beneficial, as reflected by the positive mean values for those squares. Conversely, leaving the central d- and e-pawns on their initial squares is detrimental, as shown by the negative values. This observation aligns with standard chess theory [54], which emphasizes central development and pawn advancement as advantageous strategies. For the king (see Figure 5.8b), the model accurately infers that castling— whether kingside or queenside — is generally beneficial [54], evidenced by the relatively high mean values for those squares and lower values for adjacent ones. It also recognizes that positioning the king at the edge of the board is usually poor, likely due to the increased risk of checkmate,

while keeping it near the center is supposedly better, likely due to greater freedom of movement. Additionally, the high mean values on the third-to-last rank suggest that the model understands the king's role in assisting pawns during promotions in the endgame [54].

The move representation is also interpretable. For each piece type, we can analyze the performance of moves from one square to another. In the heatmaps (see Figures 5.9a and 5.9b), many fields are white (indicating a zero mean) because certain moves are impossible for specific source-to-destination square combinations. However, when we focus on relevant fields, clear trends emerge. For example, the heatmap for the black pawn (see Figure 5.9a) reveals that central double-pawn pushes seem somewhat favorable, while pushes on the outer files are less advantageous. Similarly, the heatmap for the black queen (see Figure 5.9b) shows that moving the queen to the last rank (fields 57-64) is generally not very good, as indicated by lower mean values. The only exceptions occur when the queen is already on the last rank (fields 57-64) and simply re-positions within those squares.

Moreover, the interpretability of the model offers an additional benefit: it facilitates the analysis and further optimization of feature sets. One of our key observations was that the model intuitively promotes or demotes certain moves and the positioning of pieces on specific squares in ways that align with human intuition. These learned values apply consistently across all stages of the game. A straightforward extension to enhance the model would involve introducing a more complex feature by combining existing feature sets with the ply number (or some other form of game stage). This adjustment would allow the model to learn game stage-dependent piece-square values; for instance, it might then assign an even higher bonus for advancing the king in the late game while promoting a more defensive strategy in the early game.

7. Conclusion

This thesis explored a novel approach to chess move prediction, prioritizing interpretability while aiming for relatively high accuracy and efficient training. We achieved a prediction accuracy of 25%, which, while good, falls slightly short compared to the excellent 34% prediction accuracy in the model introduced in "Bayesian Pattern Ranking for Move Prediction in the Game of Go" [45]. We noticed that the accuracy gap is particularly pronounced in the early game, where the Go model demonstrates near-perfect prediction performance, thereby gaining an advantage in overall accuracy. However, we view this as a trade-off for better generalization at the expense of peak accuracy. Regarding top-k accuracy, we successfully replicated the results and observed significant improvements over the random model. This indicates that our model effectively distinguishes between stronger and weaker moves, even if its perfect prediction performance may not be exceptional. In fact, the strong top-k accuracy suggests that our model could already be used as a move exploration or ordering technique in search algorithms. Our results also demonstrate that the model learns quickly, reaching final accuracy faster when using simpler feature sets and move representations. Additionally, more complex feature sets and move representations seem to lead to higher accuracy, although they require more training — an area worthy of further investigation. The model also performed exceptionally well in specific prediction tasks (such as move type-specific predictions), highlighting the potential for even greater accuracy through more expressive feature sets and move representations. Overall, the model shows considerable promise for further improvement. In terms of interpretability and energy efficiency, we successfully met our goals. The model is highly interpretable — both in terms of the learned values and its overall design. It captures patterns that align with human intuition in chess. Initial tests suggest that the model is also energy-efficient during training, as in terms of its ability to converge to a "trained" state after a relatively small number of games without requiring extensive training. However, we have not extended this analysis to more complex model designs, which might demand significantly more training but could offer higher accuracy. More importantly though, this study provides an introductory exploration of Bayesian-based methods (as applied to chess), which could be extended to other fields of AI that seek more interpretable models and results. This approach may be particularly useful in domains less complex than chess, where neural networks might be unnecessarily complex. In future studies, we plan to incorporate this move prediction function into search-based algorithms, such as Monte Carlo methods, and evaluate its performance in actual gameplay. We also aim to then test the engine against human players to further understand its strengths and weaknesses. Additionally, we intend to continue exploring meaningful and expressive feature sets to improve accuracy, as well as consider refinements or a complete redesign of the model structure for future research. In conclusion, we believe this work

contributes to narrowing the gap between raw computational power and human-like strategic thinking in chess, while also offering valuable insights into the broader field of artificial intelligence, with a focus on interpretability.

A. Proofs

Proof of Theorem 1. In order to prove this theorem, we show that

$$\frac{\mathcal{G}\left(x;\tau_1,\rho_1\right)\cdot\mathcal{G}\left(x;\tau_2,\rho_2\right)}{\mathcal{G}\left(x;\tau_1+\tau_2,\rho_1+\rho_2\right)}=\mathcal{N}\left(\mu_1;\mu_2,\sigma_1^2+\sigma_2^2\right). \tag{A.1}$$

Using (2.12), we see that the left-hand side of (A.1) equals

$$\sqrt{\frac{\rho_1\rho_2}{2\pi\cdot\left(\rho_1+\rho_2\right)}}\cdot\exp\left(-\frac{1}{2}\cdot\left[\frac{\tau_1^2}{\rho_1}+\frac{\tau_2^2}{\rho_2}-\frac{\left(\tau_1+\tau_2\right)^2}{\rho_1+\rho_2}\right]\right). \tag{A.2}$$

At the same time, using (2.11), we see that the right-hand side of (A.1) equals

$$\sqrt{\frac{1}{2\pi\cdot\left(\sigma_1^2+\sigma_2^2\right)}}\cdot\exp\left(-\frac{1}{2}\cdot\frac{\left(\mu_1-\mu_2\right)^2}{\sigma_1^2+\sigma_2^2}\right). \tag{A.3}$$

It remains to show that (A.2) equals (A.3) using the identities (2.16). First, notice that

$$\sigma_1^2+\sigma_2^2=\rho_1^{-1}+\rho_2^{-1}=\rho_1^{-1}\rho_2^{-1}\cdot\left(\rho_1+\rho_2\right)=\frac{\rho_1+\rho_2}{\rho_1\rho_2},$$

which shows the equivalence of the square-root term of (A.2) and (A.3). At the same time, we have

$$\begin{aligned}\left(\mu_1-\mu_2\right)^2&=\left(\tau_1\rho_1^{-1}-\tau_2\rho_2^{-1}\right)^2\\&=\left(\rho_1^{-1}\rho_2^{-1}\cdot\left(\tau_1\rho_2-\tau_2\rho_1\right)\right)^2\qquad\Leftrightarrow\frac{\left(\mu_1-\mu_2\right)^2}{\sigma_1^2+\sigma_2^2}=\frac{\left(\tau_1\rho_2-\tau_2\rho_1\right)^2}{\rho_1\rho_2\cdot\left(\rho_1+\rho_2\right)}.\end{aligned}$$

To complete the proof, we see that

$$\begin{aligned}\frac{\tau_1^2}{\rho_1}+\frac{\tau_2^2}{\rho_2}-\frac{\left(\tau_1+\tau_2\right)^2}{\rho_1+\rho_2}&=\frac{\tau_1^2\rho_2\cdot\left(\rho_1+\rho_2\right)+\tau_2^2\rho_1\cdot\left(\rho_1+\rho_2\right)-\rho_1\rho_2\cdot\left(\tau_1+\tau_2\right)^2}{\rho_1\rho_2\cdot\left(\rho_1+\rho_2\right)}\\&=\frac{\left(\tau_1\rho_2\right)^2+\left(\tau_2\rho_1\right)^2-2\left(\tau_1\rho_2\right)\left(\tau_2\rho_1\right)}{\rho_1\rho_2\cdot\left(\rho_1+\rho_2\right)}\\&=\frac{\left(\tau_1\rho_2-\tau_2\rho_1\right)^2}{\rho_1\rho_2\cdot\left(\rho_1+\rho_2\right)},\end{aligned}$$

which shows the equivalence of the exponential term of (A.2) and (A.3). □

Proof of Theorem 2. In order to prove this theorem, we use Theorem 1 in the following form:

$$\begin{aligned}\mathcal{G}\left(x;\tau_3,\rho_3\right)\cdot\mathcal{G}\left(x;\tau_2,\rho_2\right) &= \mathcal{G}\left(x;\tau_3+\tau_2,\rho_3+\rho_2\right)\cdot\mathcal{N}\left(\mu_3;\mu_2,\sigma_3^2+\sigma_2^2\right)\,,\\ \frac{\mathcal{G}\left(x;\tau_3+\tau_2,\rho_3+\rho_2\right)}{\mathcal{G}\left(x;\tau_2,\rho_2\right)} &= \frac{\mathcal{G}\left(x;\tau_3,\rho_3\right)}{\mathcal{N}\left(\mu_3;\mu_2,\sigma_3^2+\sigma_2^2\right)}\,,\end{aligned}$$

where the second line follows by division with $\mathcal{G}\left(x;\tau_2,\rho_2\right)\cdot\mathcal{N}\left(\mu_3;\mu_2,\sigma_3^2+\sigma_2^2\right)$. All that is left is to set $\tau_1=\tau_3+\tau_2$ and $\rho_1=\rho_3+\rho_2$, substitute on the left side, and rewrite the right side in terms of μ_1,μ_2,σ_1^2, and σ_2^2. It is easy to see that $\tau_3=\tau_1-\tau_2$ and $\rho_3=\rho_1-\rho_2$. This implies, using (2.16), that

$$\begin{aligned}\sigma_3^2+\sigma_2^2 &= \frac{1}{\rho_3}+\frac{1}{\rho_2}=\frac{1}{\rho_1-\rho_2}+\frac{1}{\rho_2}\,,\\ \mu_3 &= \frac{\tau_3}{\rho_3}=\frac{\tau_1-\tau_2}{\rho_1-\rho_2}\,,\end{aligned}$$

thus, $\mathcal{N}\left(\mu_3;\mu_2,\sigma_3^2+\sigma_2^2\right)=\mathcal{N}\left(\frac{\tau_1-\tau_2}{\rho_1-\rho_2};\frac{\tau_2}{\rho_2},\frac{1}{\rho_1-\rho_2}+\frac{1}{\rho_2}\right)$. Putting it all together

$$\frac{\mathcal{G}\left(x;\tau_1,\rho_1\right)}{\mathcal{G}\left(x;\tau_2,\rho_2\right)}=\frac{1}{\mathcal{N}\left(\frac{\tau_1-\tau_2}{\rho_1-\rho_2};\frac{\tau_2}{\rho_2},\frac{1}{\rho_1-\rho_2}+\frac{1}{\rho_2}\right)}\cdot\mathcal{G}\left(x;\tau_1-\tau_2,\rho_1-\rho_2\right)\,.$$

□

Bibliography

[1] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[2] Mike Klein. Stockfish outlasts "rybkamura". `https://www.chess.com/news/view/stockfish-outlasts-nakamura-3634`, 2014. [Online; last accessed Oct. 6, 2024].

[3] Steven Strogatz. One giant step for a chess-playing machine. *New York Times*, 26, 2018.

[4] Tcec tournament results. `https://en.wikipedia.org/wiki/Top_Chess_Engine_Championship`. [Online; last accessed Sep. 10, 2024].

[5] Stockfish Team. Stockfish github. `https://github.com/official-stockfish/Stockfish`, 2024. [Online; last accessed Oct. 6, 2024].

[6] Dominik Klein. Neural networks for chess. *arXiv preprint arXiv:2209.01506*, pages 174–182, 2022.

[7] Yuj Nasu. Efficiently updatable neural-network-based evaluation functions for computer shogi. *The 28th World Computer Shogi Championship Appeal Document*, 185, 2018.

[8] Stockfish Team. Introducing nnue evaluation. `https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/`, 2020. [Online; last accessed Sep. 10, 2024].

[9] Stockfish Team. Regression tests. `https://github.com/official-stockfish/Stockfish/wiki/Regression-Tests`, 2024. [Online; last accessed Sep. 10, 2024].

[10] Stockfish Development Team. Stockfish nnue implementation. `https://github.com/official-stockfish/Stockfish`, 2020. Image file: HalfKAv2-45056-256x2P1x2-32-32-1.svg. [Online; last accessed Sep. 10, 2024].

[11] Stockfish Team. Nnue documentation. `https://github.com/official-stockfish/nnue-pytorch/blob/master/docs/nnue.md`, 2023. [Online; last accessed Sep. 10, 2024].

[12] Stockfish Team. Stockfish github. `https://github.com/official-stockfish/nnue-pytorch/wiki/Training-datasets`, 2024. [Online; last accessed Oct. 11, 2024].

[13] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[14] Robert Moss. Mcts-steps. `https://commons.wikimedia.org/wiki/File:MCTS-steps.svg`, 2020. [Online; last accessed Sep. 10, 2024].

[15] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[16] Leela Chess Zero Team. Leela chess zero github. `https://github.com/LeelaChessZero/lc0/wiki/Getting-Started`, 2024. [Online; last accessed Oct. 11, 2024].

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[18] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13693–13696, 2020.

[20] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[21] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. Estimating the carbon footprint of bloom, a 176b parameter language model. *Journal of Machine Learning Research*, 24(253):1–15, 2023.

[22] Statistisches Bundesamt. Electricity consumption of private households by household size. `https://www.destatis.de/EN/Themes/Society-Environment/Environment/Material-Energy-Flows/Tables/electricity-consumption-households.html`, 2018. [Online; last accessed Sep. 10, 2024].

[23] Dr. Harald Bögeholz. Künstliche intelligenz: Architektur und performance von googles ki-chip tpu. Heise Online, 2017. [Online; last accessed Sep. 10, 2024].

[24] Leela chess zero's official training web interface. `https://training.lczero.org/`. [Online; last accessed Sep. 10, 2024].

[25] Stockfish's official testing web interface (fishtest). `https://tests.stockfishchess.org/contributors`. [Online; last accessed Sep. 10, 2024].

[26] Mohit Kumar, Xingzhou Zhang, Liangkai Liu, Yifan Wang, and Weisong Shi. Energy-efficient machine learning on the edges. In *2020 IEEE international parallel and distributed processing symposium Workshops (IPDPSW)*, pages 912–921. IEEE, 2020.

[27] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[28] Grant Sanderson (3Blue1Brown). Gradient descent, how neural networks learn | chapter 2, deep learning. `https://www.youtube.com/watch?v=IHZwWFHWa-w&t=842s`, 2017. [Online; last accessed Oct. 10, 2024].

[29] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[30] Christopher M Bishop. Pattern recognition and machine learning. *Springer google schola*, 2:1122–1128, 2006.

[31] Ralf Herbrich. Introduction to probabilistic machine learning. Hasso Plattner Institute, Potsdam, Germany (tele-TASK HPI, `https://www.tele-task.de/series/1499/`), 2024. [Online; last accessed Oct. 11, 2024].

[32] Christian P Robert et al. *The Bayesian choice: from decision-theoretic foundations to computational implementation*, volume 2. Springer, 2007.

[33] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill™: a bayesian skill rating system. *Advances in neural information processing systems*, 19, 2006.

[34] Brendan J Frey, Frank R Kschischang, Hans-Andrea Loeliger, and Niclas Wiberg. Factor graphs and algorithms. In *Proceedings of the Annual Allerton Conference on Communication Control and Computing*, volume 35, pages 666–680. Citeseer, 1997.

[35] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

[36] Nabila Aghanim, Yashar Akrami, Mark Ashdown, Jonathan Aumont, Carlo Baccigalupi, Mario Ballardini, Anthony J Banday, RB Barreiro, Nicola Bartolo, S Basak, et al. Planck 2018 results-vi. cosmological parameters. *Astronomy & Astrophysics*, 641:A6, 2020.

[37] Charles L Bennett, Davin Larson, Janet L Weiland, N Jarosik, G Hinshaw, N Odegard, KM Smith, RS Hill, B Gold, M Halpern, et al. Nine-year wilkinson microwave anisotropy probe (wmap) observations: final maps and results. *The Astrophysical Journal Supplement Series*, 208(2):20, 2013.

[38] Srinivas M Aji and Robert J McEliece. The generalized distributive law. *IEEE transactions on Information Theory*, 46(2):325–343, 2000.

[39] Tom Minka et al. Divergence measures and message passing. Technical report, Technical report, Microsoft Research, 2005.

[40] Yuan Qi and Tom Minka. Tree-structured approximations by expectation propagation. *Advances in Neural Information Processing Systems*, 16, 2003.

[41] Philip E Ross. The expert mind. *Scientific American*, 295(2):64–71, 2006.

[42] Merim Bilalić, Robert Langner, Michael Erb, and Wolfgang Grodd. Mechanisms and neural basis of object and pattern recognition: a study with chess experts. *Journal of Experimental Psychology: General*, 139(4):728, 2010.

[43] Fernand Gobet. Expertise in chess. In *Cambridge Handbook of Expertise and Expert Performance*. Cambridge University Press, 2006.

[44] Richard Réti. *Modern Ideas in Chess: 21st Century Edition*. Russell Enterprises, Inc., Milford, CT, USA, 2009. Edited by Bruce Alberston.

[45] David Stern, Ralf Herbrich, and Thore Graepel. Bayesian pattern ranking for move prediction in the game of go. In *Proceedings of the 23rd international conference on Machine learning*, pages 873–880, 2006.

[46] Thomas Peter Minka. *A family of algorithms for approximate Bayesian inference*. PhD thesis, Massachusetts Institute of Technology, 2001.

[47] Christian Dean. Blunder. `https://github.com/deanmchris/blunder`, 2024. [Online; last accessed Oct. 14, 2024].

[48] Albert L Zobrist. A new hashing method with application for game playing. *ICGA Journal*, 13(2):69–73, 1990.

[49] nkarve. Surge. `https://github.com/nkarve/surge`, 2024. [Online; last accessed Oct. 14, 2024].

[50] Peter Ellis Jones. Generating legal chess moves efficiently. `https://peterellisjones.com/posts/generating-legal-chess-moves-efficiently/`, 2024. [Online; last accessed Oct. 15, 2024].

[51] Tadayoshi Fushiki. Estimation of prediction error by using k-fold cross-validation. *Statistics and Computing*, 21:137–146, 2011.

[52] How quiet moves can turn around a game. `https://chessmood.com/blog/quiet-moves-in-chess`, 2022. [Online; last accessed Oct. 6, 2024].

[53] The importance of quiet moves. `https://www.chess.com/blog/ThePianoManRyan/the-importance-of-quiet-moves`, 2022. [Online; last accessed Oct. 6, 2024].

[54] Siegbert Tarrasch, G. E. Smith, and T. G. Bone. *The Game of Chess: A Systematic Textbook for Beginners and More Experienced Players*. Chatto and Windus, London, 1935.