TU WIEN Informatics

Strength Estimation in the Game of Go

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Peter Neubauer, BSc
Matrikelnummer 00725263

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Günther Raidl

Wien, 21. Oktober 2024

Peter Neubauer	Günther Raidl

TU WIEN Informatics

Strength Estimation in the Game of Go

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Peter Neubauer, BSc
Registration Number 00725263

to the Faculty of Informatics

at the TU Wien

Advisor: Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Günther Raidl

Vienna, 21st October, 2024

Peter Neubauer	Günther Raidl

Erklärung zur Verfassung der Arbeit

Peter Neubauer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 21. Oktober 2024

Peter Neubauer

Acknowledgements

I sincerely thank my supervisor Günther Raidl for providing me not just with regular feedback and guidance, but also with a designated place in the office among his PhD students. Ever calm and patient, he allowed me the freedom I needed to hold my own pace and finish this thesis.

Much well-deserved praise goes to my good friend Simon Naarmann for his meticulous proofreading of my thesis. If there is not one misplaced comma in it, it is in great part to his credit.

I truly appreciate my friend Christian Polster, who stepped up to host the companion web application of this thesis on his personal server.

Thanks are also owed to KataGo author David Wu for reading my initial thesis proposal and for giving me a bunch of questions and suggestions to consider.

Further, I shall very honorably mention my fellow Go players and users of the Online Go Forum who quickly showed interest in my project, tried out the web application and responded with their own valuable insights.

I love my girlfriend Simone, who had to suffer my intense commitment to this work. She has put up with it for all this time.

Finally, I wholeheartedly appreciate my ever-understanding friends and family, who barely got to see my face for months and years while I dug in to finish my degree. I eagerly look forward to spending more time with all of them in the future.

Kurzfassung

Um die Stärke von Spielern im Go-Spiel und anderen Bereichen des Wettkampfes zu beziffern, bedienen wir uns üblicherweise eines paarweisen Wertungssystems wie Elo oder Glicko-2. Derartige Systeme ordnen dem Spieler anhand seiner errungenen Siege und Verluste eine Wertungszahl zu.

In dieser Arbeit vergleichen wir eine Implementierung des klassischen Wertungssystems Glicko-2 mit einem neu entwickelten, verhaltensbasierten Ansatz, welcher auf einem gelernten Stärkemodell basiert. Dieses Modell verarbeitet die Information über alle Züge aus verfügbaren Spielen eines Spielers in der jüngeren Vergangenheit, wodurch es mit erheblich weniger Partien ein klares Bild zur akkuraten Wertung erhält.

Wir konstruieren unser Stärkemodell in Form eines neuronalen Netzwerks. Da wir uns zur Auswertung der Brettstellungen das fachspezifische neuronale Netzwerk des bestehenden Go-spielenden Programms KataGo durch Transfer Learning zunutze machen, müssen wir das Wissen über das Spiel nicht von vorne lernen. Unser Stärkemodell verwendet die interne Wissensdarstellung aus dem neuronalen Netzwerk von KataGo als Eingabe weiter und benötigt somit keine eigenhändig ausgearbeiteten Features.

Die Architektur unseres Stärkemodells basiert auf einem Set Transformer. Diese Architektur wendet wirkungsvolle Elemente aus der leistungsfähigen Transformer-Architektur, wie Residual Blocks und Attention, auf eine ungeordnete Menge von Eingabe-Elementen an. Die Anzahl der Rechenoperationen wächst nur linear mit der Größe der Menge, da die Attention-Aufrufe auf eine konstante Zahl von Induktionselementen beschränkt sind. Diese Induktionselemente richten die Aufmerksamkeit des Modells auf wenige aussagekräftige Spieler-Züge unter Hunderten.

Um eine geeignete Konfiguration für das Training unter dieser Architektur zu bestimmen, führen wir eine randomisierte Suche in fünf Etappen über die Hyperparameter durch. Das Modell aus jenem Trainingslauf, welcher die genauesten Vorhersagen über den Ausgang der Test-Partien hervorbringt, wird als unser Stärkemodell im Endergebnis ausgewählt.

In Experimenten zeigt sich, dass unser Modell mit -0.58 durchschnittlicher log-Wahrscheinlichkeit generell etwa gleich genau wie Glicko-2 den Partie-Ausgang abschätzen kann. Im Vergleich über eine Auswahl von Partien zwischen Spielern mit wenig Vorgeschichte kommt unserem Modell der verhaltensbasierte Aspekt zugute, dass es jeden einzelnen Zug aus den vorherigen Spielen heranziehen kann. Dadurch kann es eine durchschnittliche

log-Wahrscheinlichkeit von -0.55 erreichen und Glicko-2 mit -0.64 in diesem Szenario übertreffen.

Wir demonstrieren unser Stärkemodell an zwei praktischen Anwendungsfällen. Zum einen stellen wir eine öffentliche Website zur Verfügung, wo interessierte Spieler das Stärkemodell aufrufen können. Zum anderen führen wir eine vollständige Auswertung einer Problemsammlung von Trickzügen durch.

Abstract

To quantify the performance of players in the game of Go and other competitive domains, we usually employ a paired comparison rating system like Elo or Glicko-2. Such systems assign a rating to a player based solely on their win/loss record.

In this work, we compare an implementation of the classical rating system Glicko-2 with a newly developed approach based on a learned strength model. This model uses information about all of a player's moves played in available recent games, providing a clear picture with significantly fewer games to accurately determine the rating.

We construct our strength model as a neural network. We take advantage of the domain neural network of the existing expert Go-playing program KataGo by transfer learning for board position evaluation. Therefore we avoid learning game knowledge from scratch. Our strength model uses the internal knowledge representation from the domain network's trunk as input and does not require us to manually engineer features.

The architecture of our strength model is based on a Set Transformer. This architecture employs effective elements from the trendy Transformer architecture, such as residual blocks and attention, on an unordered set of input elements. Its number of operations grows just linearly in the size of the set because the attention queries are limited to a fixed number of inducing points. These inducing points focus the attention of the model on a few significant telling game moves among hundreds.

We determine a suitable configuration for training this architecture by executing a random search in five iterations over all hyperparameters. The resultant model from the training run that produces the most accurate predictions over the games in the test set becomes our final strength model.

Our experiments show that our model can, with a mean log-likelihood of -0.58, estimate the outcome of a game approximately as well as Glicko-2. In comparison over a selection of games between players with scarce history, our model leverages the behavioral aspect, being able to draw on every move of the players' preceding games. Through this it can achieve a mean log-likelihood of -0.55, surpassing Glicko-2 with -0.64 in this scenario.

We demonstrate our strength model in two practical applications. For one, we provide a public website where interested players can query the strength model. For the other, we perform a complete strength evaluation on a collection of trick play problems.

Contents

CHAPTER 1

Introduction

Go is an ancient two-player strategy game that is popular all over the world, especially in its Far Eastern origin countries. It used to be known for its intractability by traditional game-playing algorithms like alpha-beta tree search, which could never reach the level of expert human players. With the breakthrough development of AlphaGo by DeepMind [Sil+16; Sil+17], Go-playing programs (*bots*) based on deep neural networks have achieved the playing strength to defeat any human professional. Combining a learned positional value and policy function with tree search, AlphaGo can accurately predict the winrate of both the black and white player for many positions. DeepMind's publication demonstrated the effectiveness of its deep reinforcement learning approach in a domain that long resisted classical approaches.

Following in the footsteps of AlphaGo, the open-source bot KataGo [Wu20] cannot only estimate who is winning, but also by how many points. In addition to this judgement, a human expert can read the strength of players from their moves. Although this task has received comparatively little scientific attention so far, we recognize its practical application. When a newcomer enters an established circle of Go players, such as a Go club, there are only two realistic methods to ascertain the newcomer's skill: They either self-declare their "rank", or they have to provide samples, i.e. play games against established folk.

The rank is a traditional, globally recognized scale, on which "20-kyu" describes a beginner, "1-kyu" describes a decent amateur, and "1-dan" to "9-dan" describe formidable amateur enthusiasts whose knowledge of the game is not entirely misguided. Beyond that, there are separate professional leagues in China, South Korea, Japan, Taiwan, Europe, and North America. The expert Go players who often make their career competing in these leagues are ranked from "1-dan" to "9-dan". We can disambiguate between "1-dan amateur" and "1-dan professional" where necessary. The drawbacks of this rank system are that it is coarse, sometimes ambiguous, not well-regulated, and inconsistent between distinct communities.

Hosts of tournaments and other organized competition, such as various Go associations around the world and web-based Go platforms, mostly use rating systems to estimate players' strengths from wins and losses in the form of a rating. These classical rating systems like Elo and Glicko-2 apply one objective algorithm equally and consistently to all participants within the same pool. Rating numbers have a much finer resolution than traditional ranks.

While a classical rating system needs quite a few samples—usually around 10 games—to produce a decently accurate rating, a human expert can often make this judgement by simply observing the moves played in just one or two games. As such, one traditional method of estimating the strength of a newcomer is by having the person play an evaluation game against the local master, who declares their rank. The "binary" rating system based on wins and losses is distinguished from the "behavioral" strength estimation. The term refers to observing the behavior of an agent in an environment, a common perspective in the field of artificial intelligence.

Contemporary bots contain sufficient domain expertise within their neural network models to perform the strength estimation task, but they have not yet been properly employed for this specific purpose. The occasional previous research in this direction either was too early to have access to this current generation of expert algorithms, or did not exploit the technology to its full potential.

Our contribution in this thesis is the specification and development of a new, rigorous behavioral solution to the strength estimation problem and its rigorous experimental evaluation. It targets ratings and compares favorably to established binary rating systems. We propose a trained neural network model that learns the task from public competitive Go game data while leveraging the domain expertise of the KataGo network by transfer learning.

In Chapter 2, we introduce the basic terminology and formal definitions, covering the problem domain and techniques of machine learning. Chapter 3 delves into the literature relating to the different groundwork topics for this work: classical rating systems including aspects of Glicko-2 in a way that is tailored towards comparison with our approach, Go-related algorithms and history, and previous strength estimation approaches comparable to ours. Chapter 4 describes the dataset on which we train, the strength models that we train and the training itself. Two of the models in this chapter serve as baselines for our performance comparison. The third is our core work, the full strength model. In Chapter 5, we show the results of the training process and evaluate model performance experimentally by comparison. The concluding remarks in Chapter 6 summarize our contribution and discuss notable aspects and caveats. Chapter 7 presents two practical applications of our strength model: We offer a website as a public interface to perform strength estimation and we examine a set of trick play challenges from a book.

CHAPTER 2

Preliminaries

We begin by introducing the fundamental concepts and definitions used in this work.

2.1 The Game of Go

The basic rules of the game are relatively simple: The game starts on an empty board, a grid of 19 by 19 intersections. The player holding the black stones moves first by placing a black stone on an unoccupied intersection, followed by the white player placing a white stone. This alternating move sequence continues until the game is finished, the board being unambiguously divided by closed borders of stones into black and white territories. As as side effect of a move, one may capture the opponent's stones, or even entire "strings" of multiple stones linked by grid adjacency, by occupying all their neighboring locations

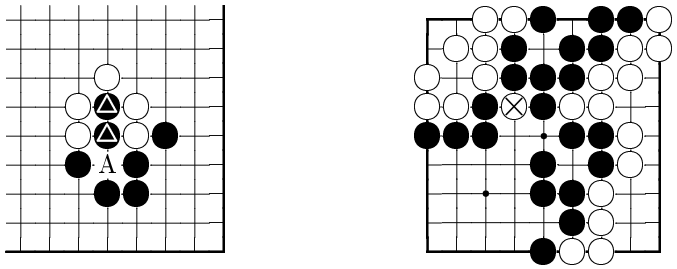


Figure 2.1: The capture rule: If white plays a stone at A in the situation on the left, the two surrounded black stones marked △ are taken off the board as prisoners for white. Counting: On the small (9×9) finished example board on the right, black has 23 points of territory and white has 4 in the top left plus 11 on the right. The crossed-out white stone in black's territory counts as black's prisoner for one more point. Even adding a *komi* of 6.5 to the white score is not enough to overcome black's total of 24 points, thus black wins the game.

such that there are no free grid intersections ("liberties") reachable from any stone in the string. Captured stones are removed from the board and kept by the capturing player as "prisoners". It is forbidden to repeat board positions. At the end of the game, any stones inside opponent territory are also taken prisoner. The total points of each player are determined as the number of free intersections inside their own territory plus the number of prisoners taken. Additionally, the white player receives 6.5 points called *komi* to compensate for black's first-move advantage and to exclude a draw result. Whoever has more points is the winner.

This brief summary of the rules, illustrated in Figure 2.1, shall suffice for our purposes. We have glossed over details such as the rigorous definition of territory and different rulesets. For more details, refer to the translated Japanese rules of Go [NK].

Go is a game of perfect information. Consequently, there exists a perfect strategy, i.e. a total mapping from the set of legal board positions to the respective best move in that position. The best move (under arbitrary resolution of ambiguity) preserves the possibility of a forced win, if it exists, or otherwise preserves the possibility of a forced draw by quirks of the rules, if it exists. Further, the best move preserves the maximum forcibly achievable final score for the player to move, determined by their points minus the opponent's points. We refer to a sequence of best moves from both sides as *optimal play*, possibly from a context position. Properties of a board position can be asserted (implicitly) *assuming optimal play*, which means that the property holds if the game were to be concluded using only optimal play from that position forward. For example, the statements that "the white group of stones is alive" (immune to capture) or that "black is 10 points ahead" are understood to hold assuming optimal play.

Which moves constitute optimal play is unknown in most positions and thus usually hypothetical. Instead, we approximate the expected game outcome by estimating the players' *winrate* [ET20]. Unless stated otherwise, the winrate in a board position means the winrate of the player to move. When we view (the remainder of) a game as a Bernoulli experiment between equally matched opponents with leeway for mistakes, the winrate allows us to quantify the advantage of one side in the current position. For example, if the winrate for white is 60%, then the game outcome will be "white wins" 60% of the time. Through the idea of equally matched—not necessarily perfect—opponents, we separate the positional winrate concept from the particular player's skill.

Especially at the level of human players, the extreme rate and scale of blunders can quickly put one side at such a decisive advantage that even mediocre or bad moves do not significantly impact the winrate. To retain a meaningful distinction between good and bad moves in such scenarios, we need to measure the distance between the current state and the tipping point of equal chances. This measure is the *lead* in points: the number of additional points that would hypothetically have to be gifted to the other player in the position to equalize the winning chances.

A Go-playing program can estimate both the winrate and the points lead. In this work, we call a Go-playing program a *bot* for short. Derived from these estimates, the *winrate*

loss and *points loss* of a single move are measures of move quality. They are determined as the winrate or points lead in the board position before the move in question minus the winrate or points lead in the resultant position after the move, viewed from the perspective of the moving player. As a notable caveat, there may be a tradeoff between winrate and lead. Since both are only ever estimates to substitute for the unknown game outcome assuming optimal play, we can never absolutely state that one should be preferred over the other. Strong bots maximize winrate, while humans prefer to assess the more tangible points.

2.2 Neural Networks

In this section, we cover the relevant basics on neural networks as found in textbooks, such as [GBC16; RN21].

A *neural network* f is a model for approximating a function mapping input $\mathbf{x}$ to output $\mathbf{y}$ by tuning the model parameters θ. Formally,

$$f(\mathbf{x};\theta) = \mathbf{y}.$$

The *input features* in $\mathbf{x}$ and the *output features* in $\mathbf{y}$ are real-valued random variables. We consider $\mathbf{x}$ to stem from an unknown *data generating distribution*. The parameters in θ are real numbers.

The structure of $\mathbf{x}$ depends on the application. It consists of one or multiple *feature vectors*. Multiple related feature vectors are compounded into multi-dimensional *tensors* as a convention to simplify definitions of operations on them.

Networks are structured into *layers*, which are optionally compounded into *blocks*. Blocks and layers as nodes form a directed acyclic graph, in which every source node represents an input feature $\mathbf{x}_{(\cdot)}$ and every sink node represents an output feature $\mathbf{y}_{(\cdot)}$. Layers receive inputs only from higher layers and pass output only to deeper layers.[1] The most basic structure is the *chain*, formally

$$f(\mathbf{x};\theta) = f_l(\dots f_1(\mathbf{x};\theta_1)\dots;\theta_l), \quad \theta = \theta_1 \cup \dots \cup \theta_l,$$

where f_i, $i = 1,\dots,l$ are blocks or layers. Blocks can have a chain structure as well, following the same definition. If f is a block and f_1 and f_2 are layers or blocks, they are *nested* inside f.

A layer models a clearly described function or class of functions using a few basic operations.

A block models a higher-level abstraction as a composite of as many different functions as the *hidden dimensions*, the dimensionality of its intermediate features between layers, will allow.

[1] This specifically describes *feed-forward networks*, the only type of network used in this work.

2.2.1 Basic Layers

The most essential layer is the *fully-connected layer* with parameters $W \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^n$, $n, m \in \mathbb{N}$, which models an affine transformation of the input features. It may further include an *activation function* $\sigma \colon \mathbb{R} \to \mathbb{R}$ that applies element-wise to feature vectors, such as the *rectified linear activation function* $\mathrm{ReLu}(x) = \max(0, x)$. The classical definition of the fully-connected layer is

$$\mathrm{FC}(\mathbf{x}; W, \mathbf{b}) = \sigma(W\mathbf{x} + \mathbf{b}).$$

Fully-connected layers can be varied based on context, needs, and conventions. They may be defined to include normalization of input features. They may use W as the only parameter, implying $\mathbf{b} = 0$. In a *preactivation architecture*, σ applies before the transformation: $W\sigma(\mathbf{x}) + \mathbf{b}$.

Originally inspired by the way biological neurons propagate information in the brain, the fully-connected layer's interpretation for our purposes is that it can recognize features in the input that are more abstract. The role of the affine transformation in achieving this is like aligning paper in a cutter, which is then metaphorically "cut" by the nonlinear activation function.

Dimensionality reduction is the linear transformation into a lower-dimensional space.

$$f(\mathbf{x}; W) = W\mathbf{x}, \quad \text{where } W \in \mathbb{R}^{m \times n},\ m < n$$

This can be done as part of the first layer of a *bottleneck residual block* to prepare for computing expensive layer operations more efficiently in fewer dimensions. The features are projected back into the high-dimensional space at the end of the block.

The *softmax* layer implements weighted multiple-choice among any number of options, such as output classes in a classification task. Each option i is weighted proportional to its scalar *logit* z_i.

$$\mathrm{softmax}(z_1, \ldots, z_n)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

Another interpretation is that softmax generalizes the *sigmoid activation function* $\varsigma(z) = \frac{e^z}{1+e^z}$ to multiple signals, only the strongest of which "fire".

The *batch normalization* [IS15] layer normalizes all elements of its d-dimensional input $\mathbf{x}$ to a mean of 0 and to a variance of 1, then applies a linear transformation according to the learned parameters $\mathbf{g} \in \mathbb{R}^d$ and $\mathbf{b} \in \mathbb{R}^d$:

$$\mathrm{BatchNorm}(\mathbf{x}; \mathbf{g}, \mathbf{b})_i = g_i \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\mathrm{Var}[x_i]}} + b_i,$$

where x_i, g_i, and b_i denote the i-th element of $\mathbf{x}$, $\mathbf{g}$, and $\mathbf{b}$, respectively.

To find the mean and variance of the input distribution, the batch normalization layer estimates them from a moving average of every input value in the current batch during training, and again from a larger sample size to finalize the layer after training. The effect of BatchNorm is to reduce amplification of changes as they propagate through many layers, leading to more reliable training of deep layers. Among the downsides to this approach are that the output of one sample now depends on all other samples in the batch, and that it assumes a feed-forward network architecture where every input is used exactly once.

The *layer normalization* [BKH16] layer LayerNorm$(\mathbf{x}; \mathbf{g}, \mathbf{b})$ works like batch normalization, except that it estimates the input mean and variance from all the real-valued components of the input sample. Batch elements are thus treated separately. This resolves some downsides of batch normalization.

2.2.2 Residual Connection

A *residual connection* is a neural network component in which we train a *residual function* $\mathcal{F}(\mathbf{x}; \theta)$ that adds to the input.

$$f(\mathbf{x}; \theta) = \mathbf{x} + \mathcal{F}(\mathbf{x}; \theta)$$

This centering of the model on the identity function mitigates a phenomenon of accuracy degradation in deep networks during training [He+16]. A network built of components with residual connections is a *residual network* or *ResNet* for short.

2.2.3 Pooling

For an input that consists of multiple elements that are same-size feature vectors, the *(global) pooling* operation determines the occurrence or prevalence of a feature among all elements. It spreads this global information to multiple output elements in the form of a residual function added to them.

Given input features $\mathbf{x}(i)$ at location i, pooling inputs $\mathbf{g}$ which may be separate inputs distinct from $\mathbf{x}$, an aggregation function s, and weights W, pooling is defined as

$$\mathbf{y}(i) = \mathbf{x}(i) + Ws(\mathbf{g}).$$

The function s computes one feature vector from statistics, for eaxmple, the mean or maximum or both, of each channel.

2.2.4 Attention

Like pooling, *attention* can compute aggregated information from multiple inputs. Unlike simple pooling, attention scores the elements according to their *keys*, which are feature vectors derived from the inputs. The attention mechanism evaluates each key according

to multiple *queries*, which are also feature vectors. The resultant *attention coefficients* apply as weights to the *values* derived from the inputs. Each attention query thus results in a differently-weighed sum of input values.

In this work, we use *dot-product attention*. Attention coefficients scale with the dot product between the respective query and key feature vectors. Keys that are similar to the query weigh more under this measure.

The attention operation for a singular query is defined as

$$\mathrm{AT}(\mathbf{q}, \mathbf{k}, \mathbf{v}) = \mathrm{softmax}\left(\frac{\mathbf{q}^\intercal \mathbf{k}}{\sqrt{d_\mathrm{q}}}\right)\mathbf{v}$$

where

$\mathbf{q} \in \mathbb{R}^{d_\mathrm{q}}$	is a query feature vector,
$\mathbf{k} \in \mathbb{R}^{d_\mathrm{q} \times n}$	are n element keys and
$\mathbf{v} \in \mathbb{R}^{d \times n}$	are n element values.

We extend this definition to m queries $\mathbf{q}' \in \mathbb{R}^{d_\mathrm{q} \times m}$ by concatenating the results, yielding $\mathrm{AT}(\mathbf{q}', \mathbf{k}, \mathbf{v}) \in \mathbb{R}^{d \times m}$.

2.2.5 Convolution

If the input features are arranged in a grid-like topology, like pixels in an image or intersections on a Go board, we can apply the *convolution* operation defined as

$$\mathbf{y}(i)_j = \sum_{a \in A} \mathbf{x}(i-a)^\intercal f_j(a)$$

where

$\mathbf{x}(i) \in \mathbb{R}^d$	is the input feature vector associated with grid location i,
$\mathbf{y}(i)_j \in \mathbb{R}$	is the j-th component of the output feature vector at i,
$f_j(k) \in \mathbb{R}^d$	is the weight vector defined by the j-th *filter* at offset k, and
A	is a set of offsets defining an area the size of the filters.

For example, a "3×3 convolution" models each output *channel* (feature component) as a linear combination of the input features at the same location and in its 8-neighborhood. A *convolution layer* is a layer that uses convolution as its basic operation in the same way the fully-connected layer uses the affine transformation. A *convolutional network* uses convolution layers as its core component.

The purpose of filters is to recognize local patterns. Since they are *equivariant to translation*—the filter detects the pattern no matter where it occurs—convolution separates the concept of the pattern from its location in training and in application.

2.2.6 Training

Let $\mathcal{T} \dot\cup \mathcal{V} \dot\cup \mathcal{E} \subseteq \mathcal{O} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots\}$ be a set of observation data. We want to approximate it by a function with a model $f(\mathbf{x}; \theta)$. We do this by repeatedly sampling *minibatches* $\mathcal{B} = \{(\mathbf{x}_1^{\mathcal{B}}, \mathbf{y}_1^{\mathcal{B}}), (\mathbf{x}_2^{\mathcal{B}}, \mathbf{y}_2^{\mathcal{B}}), \ldots\}$ from the *training set* $\mathcal{T}$.

The *parameter update* based on $\mathcal{B}$ with *learning rate* η towards minimal *loss* L, defined as

$$\theta \leftarrow \theta + \Delta\theta, \quad \Delta\theta \propto -\eta \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{B}} \frac{\partial L(\mathbf{y}, f(\mathbf{x};\theta))}{\partial\theta},$$

implements *stochastic gradient descent*. The learning rate starts at its maximum value and decays over the training epochs, controlled by the *(exponential) learning rate decay* parameter β.

$$\eta_{i+1} \leftarrow (1-\beta)\,\eta$$

This allows early training to reach a promising region in parameter space and later training to be sensitive to smaller improvements.

We choose the training loss function $L = L_t$ to be a measure of distance from the training target $\mathbf{y}$. Minimal loss corresponds to desirable model behavior. The popular *Adam* optimization algorithm [KB17] performs stochastic gradient descent and normalizes every gradient update component according to an exponential moving estimate of its mean and variance.

The variables that define the model architecture and training mode, such as the learning rate η, number of network layers, and feature dimensions, are *hyperparameters*.

A well-chosen set of hyperparameters regulates the effective *capacity* of the model such that it fits the training data and generalizes well. If the model is too small, it underfits, and if it is too large, overfits the training data.

We use the *validation error*

$$\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{V}} L_p\left(\mathbf{y}, f(\mathbf{x};\theta)\right)$$

to evaluate model performance during training and guide our search for the ideal hyperparameters. While the training loss function L_t includes aspects like secondary targets and regularization to give the best learning experience to the model, the performance loss function L_p specifies our true performance measure for model validation and testing.

If the validation error stops improving for a chosen fixed number of training iterations (the *patience*), we omit the remainder of the training run and end with the model that has the lowest validation error to that point. This *early stopping* technique implicitly optimizes the training duration as a hyperparameter.

The *iterated random search* algorithm for hyperparameter optimization [BB12] proposes a fixed number of n random points in the space of hyperparameters. For each point, it assigns a random value to each hyperparameter according to an individually defined

marginal distribution. The best point is determined as the one that minimizes the validation error of the correspondingly trained model. The process iterates k times by shrinking the search space to the lower-magnitude vicinity of the previous best point. Training completes after $n \cdot k$ runs.

After we train the final model on the best of the found hyperparameters, the *test error* based on $\mathcal{E}$ gives us an unbiased estimate of the expected trained model performance in the real world.

2.3 Rating Systems

A *rating system* in games like Go is an algorithm that quantifies the fitness, called *rating*, of *players* from a *pool* who participate in some competitive endeavour. Rating systems are renowned for their application to sports and games, but they similarly apply to other domains with competitive elements such as wine tasting, online dating and many more. The central fitness quantity is the *rating number*: the higher, the fitter. Under more detailed systems like Glicko-2, the rating includes more constituent components.

The two main purposes of rating systems are evaluation and matchmaking. Evaluation is the determination of the competitors' fitness—*playing strength* or just *strength* in our setting—which can be taken as personal feedback, status symbol, or for ranking purposes, for example to decide who qualifies to participate in a high-stakes championship tournament. Matchmaking is the problem of brokering matches between worthy opponents among willing would-be competitors. An opponent is worthy if their strength—a hidden, time-varying value that is estimated by the rating number—is similar to that of the matched player, giving both a chance at victory.

We restrict this work to *paired* rating systems, in which each in a series of *games* constitutes a confrontation between a pair of players, exactly two *opponents*. One of them is designated as the *black player*, the other as the *white player*. The outcome of the game is determined by the opponents playing a series of *moves*. The black player makes every odd move and the white player makes every even move in the game. As a result, one opponent is the *winner* and the other is the *loser*. We represent this fact numerically as the *score* s of the match: 1 if black is the winner, 0 if white is the winner. We further do not take draws and other sorts of undecided games into consideration. All definitions in this work can be extended to allow undecided outcomes by treating them as either having never happened, or as half a win—with score 0.5—for both.

When we talk of "score", unless stated otherwise, we are referring to the score of a game and not to the—largely irrelevant—points lead in the end position of a Go game defined in Section 2.1.

Definition 1 (Pool) *A* game *is a tuple* $\langle b, w, s \rangle$ *that includes the opponents* b *and* w*,* $b \neq w$*, and the score* $s \in \{0, 1\}$*.*

We define the pool G *as a time series of* $t_{\max}$ *games.*

$$G = (g_t), \quad 1 \leq t \leq t_{\max}$$

The set

$$P = \{p \mid \exists \langle b, w, \cdot \rangle \in G : p = b \vee p = w\}$$

is the set of all players who appear in a game in G.

Every game g_t has an associated move set $M(g_t) = \{m_{t,1}, \ldots, m_{t,\max}\}$. For every move in the set, we know all the necessary information to extract features, such as the board state before the move, the move coordinates, and all previous moves. Every move $m_{t,i}$ has an associated *color* $c(m_{t,i})$ such that, for a move set $M(g_t)$ as just defined,

$$c(m_{t,i}) = \begin{cases} \textit{black} & \text{if } i \text{ is odd,} \\ \textit{white} & \text{if } i \text{ is even.} \end{cases}$$

Additionally, we define the color $c(p, g)$ of player p in the game g as

$$c(b, \langle b, w, \cdot \rangle) = \textit{black},$$
$$c(w, \langle b, w, \cdot \rangle) = \textit{white}.$$

Definition 2 (Recent Moves) *The* history H *of a player* p *at time* t_{now} *is the set of all their games until and including time* t_{now}*.*

$$H(p \mid t_{\text{now}}) = \{g_t = \langle b, w, s \rangle \in G \mid t \leq t_{\text{now}} \textit{ and either } b = p \textit{ or } w = p\}$$

The past move set *of* p *at time* t_{now} *is*

$$M^{\text{past}}(p \mid t_{\text{now}}) = \{m_{t,i} \in M(g_t) \textit{ for all } g_t \in H(p \mid t_{\text{now}}) \mid c(m) = c(p, g_t)\}, \textit{ except passes.}$$

Let the window size N *be a predetermined desired number of recent moves. Then the* recent move set

$$M^{-}(p \mid t_{\text{now}}, N) = \text{Top}_N(M^{\text{past}}(p \mid t_{\text{now}}))$$

is the set of the latest up to N *past moves* $m_{t,i}$ *according to the order*

$$m_{t_0,i} < m_{t_1,j} \quad \textit{if } t_0 < t_1 \textit{ and}$$
$$m_{t,i} < m_{t,j} \quad \textit{if } i < j.$$

A rating system defines an estimator for the hidden true strength of the players participating in G by mapping their past performance to a *rating*. Let R be the domain of ratings in a rating system. We formally define the *rating estimator* as

$$\hat{r}(p \mid t)\colon P \times \{0, \ldots, t_{\max}\} \to R. \tag{2.1}$$

The rating number $\mu(r)\colon R \to \mathbb{R}$ is the key element in (or even the entirety of) a rating. If p_1 is a weaker player than p_2, then we expect that $\mu(\hat{r}(p_1 \mid t_{\max})) < \mu(\hat{r}(p_2 \mid t_{\max}))$.

A classical *binary rating system* like Elo or Glicko-2 infers the player ratings from the binary score s alone, while a *behavior rating system* based on models like that of Moudřík et al. [MBN15], or the ones presented in this work, infers ratings from the moves M of previous games instead of s.

One design goal of rating systems is to predict game outcomes accurately. As a statistic to measure the quality of a rating system, we can calculate its *predictive accuracy* α as the rate of outcomes in which the system correctly predicted the winner. Let $g_t = \langle b_t, w_t, s_t \rangle$.

$$\alpha = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} \left| s_t - I_{\leq 0}(\mu(\hat{r}(b_t \mid t-1)) - \mu(\hat{r}(w_t \mid t-1))) \right| \tag{2.2}$$

This measure, although occasionally used in literature, is crude because it fails to distinguish the degree of certainty with which the score was predicted. We therefore use $\hat{s}(t)$, an estimate of the outcome s_t of game g_t based on the ratings $\hat{r}(b_t \mid t-1)$ and $\hat{r}(w_t \mid t-1)$. We formally define the *score estimator* as

$$\hat{s}(t)\colon \{1, \ldots, t_{\max}\} \to [0, 1]. \tag{2.3}$$

For rating and score estimators, $\mu(\hat{r}(b_t \mid t-1)) > \mu(\hat{r}(w_t \mid t-1))$ holds if and only if $\hat{s}(t) > 0.5$. The accuracy α is equivalently defined via $\hat{s}$.

$$\alpha = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} |s_t - I_{\leq 0}(\hat{s}(t) - 0.5)| \tag{2.4}$$

When the chances are dead even, our definition treats this as a prediction of white victory. This is because white has a slightly higher empirical winning rate in Go and, absent other information, the system should "guess" white.

Using a score estimator, we strive to maximize the *log-likelihood* statistic λ, the (normalized) natural logarithm of the estimated probability that, for each game, the result will be the observed result.

$$\lambda = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} \ln\left(1 - |s_t - \hat{s}(t)|\right) \tag{2.5}$$

CHAPTER 3

Related Work

In this chapter, we review relevant literature on the general topics of rating systems, Go-playing programs, and prediction of player attributes.

3.1 Rating Systems

All rating systems used in competitive Go environments, namely the Go associations of the world with their amateur and professional leagues as well as online platforms such as OGS, are binary rating systems: They determine rating numbers based entirely on game outcomes (wins and losses). One major advantage is transparency through simplicity, contributing to the perceived fairness that is necessary for their broad acceptance. The determination of the winner becomes the bottleneck of any contention from the competitors, systematically addressed through rigid game rules. Moreover, the resulting numbers need only to be subjectively accurate enough to be fair and entertaining. A hypothetical system that correctly predicted absolutely every game outcome would only harm the general interest in the competition.

The Elo system [Elo78], invented by Arpad Elo to rank Chess players, is the most widely recognized paired comparison system founded in statistics. Glickman [Gli99; Gli01] formulated the Glicko and Glicko-2 rating systems, which expanded the representation of a player's skill from just the rating number to also include the rating deviation and volatility parameters. We describe the Elo and Glicko-2 systems in in this section. For further information on Glicko-2, refer to the example on Glickman's website [Gli22] or to the particular implementation of the Go platform OGS [NC]. The whole-history rating system by Coulom [Cou08] improves accuracy by investing computation time into an iterative approximation algorithm over the whole pool of games. As a welcome side effect, match outcomes added over time can cause retroactive corrections on the ratings of all participants, even of those who are not directly involved in the new paired comparisons.

This makes whole-history rating a suitable basis for the international comparison of professional Go players[1], even if most games occur between players of the same country.

3.1.1 Bradley-Terry Model

The Bradley-Terry model [BT52] allows us to draw the connection between (scaled) rating numbers and winning chances. We view the game outcome as a probabilistic event based on the rating numbers which correspond to the real, hidden playing strengths of the opponents. Let μ_1 and μ_2 be the rating numbers of opponents p_1 and p_2. Then the probability that p_1 defeats p_2 in a game between the two is

$$\hat{s}^{\text{B-T}}(t) = \frac{e^{\gamma\mu_1}}{e^{\gamma\mu_1} + e^{\gamma\mu_2}}. \tag{3.1}$$

The scaling factor γ is an arbitrary positive real number that we can adjust to make our ratings "look nice". We may choose γ to let beginners have rating numbers below 1000, strong club players perform around 2000, and world-class players top out around 2800. This fits established expectations in Chess and other disciplines.

To illustrate, imagine that player p_1 puts $e^{\gamma\mu_1}$ red balls into an empty urn. Player p_2 adds $e^{\gamma\mu_2}$ blue balls into the urn. Then the color of a single ball drawn from the urn determines the winner of the match.

3.1.2 Elo System

In the binary rating system described by Elo [Elo78], the rating number is the sole quantity that models the strength of a competitor. Players' hidden strengths and rating numbers are now defined as time-varying, i.e. as functions in t. The system keeps track of the rating number $\mu_i(t)$ of every player p_i over time, with the reference point $\mu_i(\cdot) = 2000$ describing a strong amateur club player based on tradition.

In the original application of the Elo system, the winning probability of p_1 with rating μ_1 over p_2 with rating μ_2 at time t is modeled by the standard normal distribution. Denoting its cumulative function by Φ, we have

$$\hat{s}^{\text{Elo}}(t+1) = \Phi\left(\frac{1}{200\sqrt{2}}(\mu_1(t) - \mu_2(t))\right). \tag{3.2}$$

In the book [Elo78], Elo discusses a logistic model like Bradley-Terry as an alternative in his later chapter on rating system theory.

Furthermore, every game outcome s_{t+1} at time $t+1$ works as an incremental piece of information to the system about the hidden strengths of the players involved. It adjusts, for both players, the rating number $\mu_{(\cdot)}(t+1)$ in proportion to the difference between the expected score $\hat{s}^{\text{Elo}}(t+1)$ and the actual score s_{t+1}.

[1] Current whole-history ratings of top players: `https://www.goratings.org`, visited: 2024-10-01

$$\mu_i(t+1) \leftarrow \mu_i(t) + K \cdot (s_{t+1} - \hat{s}^{\text{Elo}}(t+1)) \tag{3.3}$$

All other ratings are unaffected: $\mu_i(t+1) \leftarrow \mu_i(t)$.

The scaling factor K is a positive global constant. A high K allows faster adjustment to change, while a low K keeps the rating estimate closer to the target.

The Elo system can be interpreted as performing stochastic gradient descent on rating numbers towards the moving target of hidden strengths. Then K is the learning rate.

In terms of our rating system definition, the rating estimator of Equation 2.1 is defined under the Elo system as

$$\hat{r}(p_i \mid t) = \mu_i(t).$$

The original definition by Elo includes more details which we omit here. For example, it extends to multiple matches, akin to minibatches, happening at the same time. This applies when rating all games from a tournament at the end.

Since the inception of the Elo system, numerous extensions and improvements have been suggested: more accurate representations of ratings, more accurate models of winning probability, better handling of drop-outs and newcomers to the pool, team ratings, and so on. Thanks to the Elo system's simplicity, it enjoys widespread use as a practical rating system and as a reference point for other systems.

3.1.3 Glicko-2

The Glicko rating system was introduced by Glickman [Gli99; Gli16] as an improvement over the Elo system. It represents a player's rating with a *rating deviation* value on top of the rating mean. The rating deviation models the system's confidence in the rating estimate. The deviation decreases with each rating update and increases over time.

The newer revision Glicko-2 [Gli01; Gli22] adds a third player property, the *rating volatility*. This volatility discriminates established, stable players from those who are fast-improving—or, less realistically, getting worse fast. In Glicko-2, the ratings deviation increases over time proportional to the volatility.

Let $\langle \mu_i(t), \phi_i(t), \sigma_i(t) \rangle$ be the estimated rating $\hat{r}(p_i \mid t)$ of player p_i at time t, constituted of the rating mean $\mu_i(t)$, rating deviation $\phi_i(t)$ and rating volatility $\sigma_i(t)$. The calculations for Glicko-2 operate on a smaller scale than Elo for a more natural formulation, but the rating and deviation are always scaled to Elo-like numbers in the thousands of points for presentation. The relationship is

$$\begin{aligned} \text{rating number } \mu(\hat{r}(p_i \mid t)) &= 1500 + \frac{400}{\ln 10}\mu_i(t), \\ \text{presented rating deviation} &= \frac{400}{\ln 10}\phi_i(t). \end{aligned}$$

In this section, we use the Glicko-2 scale for $\mu_i(t), \phi_i(t)$ and $\sigma_i(t)$ to match Glickman's definitions. In all other parts of this work, we always use the large-scale presentation numbers; we even use the term "rating deviation" to mean the presented rating deviation and not ϕ.

Consider a match $\langle b, w, s\rangle$ at time $t+1$ against an opponent p_o with an estimated rating specified by $\hat{r}(p_o \mid t) = \langle \mu_o(t), \phi_o(t), \sigma_o(t)\rangle$. The Glicko-2 system takes the players' rating deviations into account for the winning probability.

$$\begin{aligned} g(\phi) &= \frac{1}{\sqrt{1+3\phi^2/\pi^2}} \\ \gamma &= -g\left(\sqrt{\phi_i^2+\phi_o^2}\right) \\ \hat{s}^{\text{Glicko-2}}(t+1) &= \frac{e^{\gamma\mu_i(t)}}{e^{\gamma\mu_i(t)}+e^{\gamma\mu_o(t)}} \end{aligned} \tag{3.4}$$

Let ξ be the score from the perspective of p_i: 0 for a loss and 1 for a win.

$$\xi = \begin{cases} s & \text{if } b = p_i \\ 1-s & \text{if } w = p_i \end{cases}$$

The (simplified) process to determine the given player's successive rating $\langle \mu_i(t+1), \phi_i(t+1), \sigma_i(t+1)\rangle$ based on the actual match score s_{t+1} consists of the following steps.

1. $g \leftarrow \frac{1}{\sqrt{1+3\phi_o(t)^2/\pi^2}}$
2. $E \leftarrow \frac{1}{1+e^{-g\cdot(\mu_i(t)-\mu_o(t))}}$
3. estimated rating variance $v \leftarrow [g \cdot E \cdot (1-E)]^{-1}$
4. estimated rating improvement $\Delta \leftarrow v \cdot g \cdot (\xi - E)$
5. determine $\sigma_i(t+1)$ from all of the above by a certain iterative process
6. volatility-adherent deviation $\phi^* \leftarrow \sqrt{\phi_i^2 + \sigma(t)^2}$
7. $\phi_i(t+1) \leftarrow 1/\sqrt{\frac{1}{\phi^{*2}} + \frac{1}{v}}$
8. $\mu_i(t+1) \leftarrow \mu_i(t) + \phi_i(t+1)^2 \cdot g \cdot (\xi - E)$

All players p_j in the pool who are not involved in a match at time $t+1$ increase their rating deviation based on their unchanged volatility $\sigma_j(t+1) \leftarrow \sigma_j(t)$ by $\phi_j(t+1) \leftarrow \phi^*$ as defined in Step 6. This models the increasing uncertainty of the system as time passes without new knowledge on inactive players.

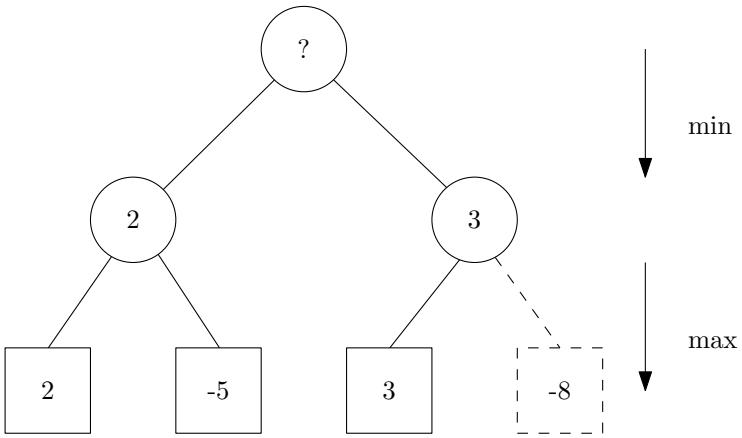


Figure 3.1: For the *minimax algorithm*, the bot builds a game tree from the current position at the top. It labels positions with their estimated utility towards winning the game, which it strives to maximize. When it is the opponent's turn, the opponent wants to minimize the same value. When we discover an option for the opponent to refute our initial move, the resultant low value propagates up the tree and we avoid a blunder.

Glicko-2 fulfills our rating system definition in Equation 2.1 and Equation 2.3 using the rating estimator and score estimator

$$\mu(\hat{r}(p_i \mid t)) = 1500 + \frac{400}{\ln 10}\mu_i(t),$$
$$\hat{s}(t) = \hat{s}^{\text{Glicko-2}}(t).$$

The guide provided by Glickman [Gli22] includes details on the volatility update and allows for multiple matches to be ranked in a single "rating period", just like the original definition of the Elo system. In this work, we simplify to one match at a time. This agrees with the reference implementation of OGS, which is also the source of our dataset. The implementation choice is motivated by the online platform context, where new matches continuously add to the pool. Users expect to see their results immediately reflected in their rating rather than to wait for an "update window".

3.2 Go-Playing Programs

In this section, we summarize the history and methods of Go bots. Figure 3.2 illustrates their progression.

Throughout the 1960s, 1970s and 1980s, the first Go programs were developed. They were mostly restricted to smaller board sizes like the beginner-friendly, popular 9×9 [Ass].

The basic design of a Go bot is the same as that for many turn-based perfect information games: the minimax algorithm with alpha-beta pruning [RN21], illustrated in Figure 3.1.

We build a tree of game positions with the current position as the root. The edges represent legal moves. This tree reaches the entire search space of the problem. The hard part is knowing which parts of the tree we should explore by expanding the nodes. Expansion generates the successors of a chosen node for further consideration. Only expanded nodes can inform the bot's next move decision. The bot decides on the next node to expand using its *policy function* and it evaluates the discovered positions using its *value function*.

Compared to Chess, in which the breakthrough of a computer defeating a world champion human player happened with the victory of Deep Blue over Garry Kasparov in 1997 [CHH02], Go remained the more challenging computer discipline for several reasons [McD+01; BW95]. First, the search space of legal moves is substantially larger. In Chess, we expect around 35 legal moves in an average position (the *branching factor*) and around 80 moves in an average game. The Go board allows 361 legal moves in its empty initial state, gradually declining over a game length of more than 200 moves in most cases. These numbers may be just rough upper bounds, but the difference is clear. Second, the discovery of promising moves is only viable using pattern recognition. These patterns are too numerous and vague to be defined as hand-crafted heuristics. Third, tactical choices often depend on a *late payoff*. The difference between a particular move being either very beneficial or actively detrimental might only be revealed at the end of a long hypothetical line of play. In addition to the calculation of potentially viable moves, the player is burdened with reading out these sequences that will never realistically happen on the board. At its extreme, the ownership of every point on the entire board can only be definitively determined at the latest time, the very end of the game.

Around 2007, the adoption of Monte Carlo Tree Search (MCTS) [Cou06] algorithms with Upper Confidence Bounds applied to Trees (UCT) [KS06] for Go bots established a new state of the art, capable of challenging stronger, about 2-dan, human amateur players. These algorithms address the above difficulties in the following manner.

We start with the general assumption that the expected game outcome from a starting position between equally matched very strong opponents does not change if we substitute equally matched weak opponents. Monte Carlo tree search gathers information about candidate moves following this idea by sampling "playouts", entire games played to the end using fast heuristics. The finished sample games can be scored and aggregated into an expected outcome. This algorithm still grows a tree of positions, one node at a time, to hold statistics about the new variation it played out, but the size of this tree is no longer a limiting horizon beyond which we cannot see. Figure 3.3 shows this idea. Furthermore, within this grown tree of known positions, the UCT rule guides the selection of the next playout move. This rule chooses the next move to maximize the sum of an *exploration term* and an *exploitation term*. The exploration term encourages moves that have a low playout count, while the exploitation term encourages moves with high expected value. Finally, the programs of this generation recognize patterns learned from human games. The approach is generally to simply remember prevalent patterns and prefer these moves when they are an option.

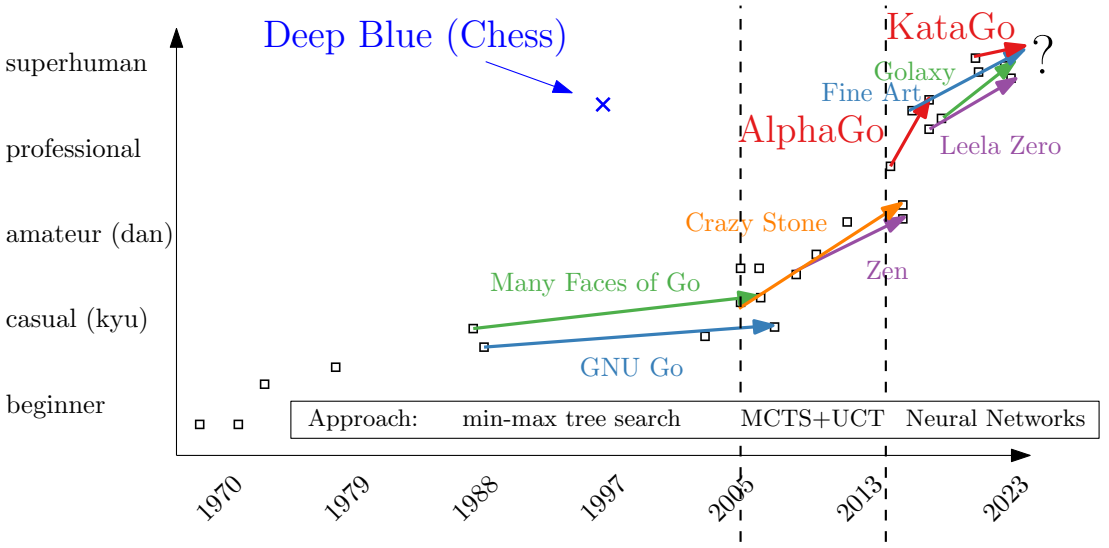


Figure 3.2: The timeline of Go bots and their respective strengths can be divided into three phases based on the dominant technology at the time. Despite their assorted creative ways to mitigate the difficulties of Go on top of traditional search methods, early programs were very restricted in their results. MCTS with UCT constituted one notable breakthrough. With MCTS in combination with neural networks, Go bots have exceeded all the strongest players. Arrows are drawn from the earliest appearance of the respective bot to its latest released version.

Examples of MCTS-based programs are Crazy Stone [Cou07b; Cou07a], Fuego [Enz+10] and Pachi [HP11].

The debut of AlphaGo [Sil+16] proved the viability of the deep learning approach for positional evaluation, resulting in superhuman performance. One year later, the follow-up paper [Sil+17] documented that the bot could be trained without training data from human games, setting it free to discover its own style and break entrenched notions.

Since the source code and network data of AlphaGo were never released to the public, several projects emerged to replicate its training process. The open-source bots Leela Zero [THW22; Pas] and KataGo crowdsourced the computation to community volunteers. KataGo adds new techniques which accelerate the training process [Wu20] and new network heads. These heads provide a variety of information about the evaluated positions, which makes the program stronger and also more versatile.

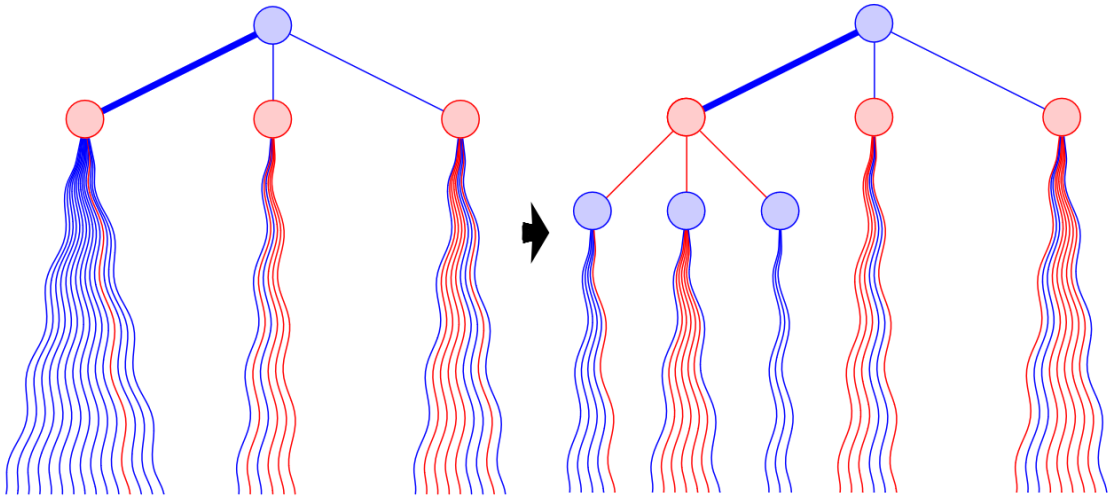


Figure 3.3: This visualization of MCTS is taken from two slides of Coulom's presentation on Crazy Stone [Cou07b]. New playouts are allocated to the preferred leaf in the tree according to the UCT formula, the first of three in this example. As more playouts pass through each node, it accumulates its value as the mean of playout values instead of the traditional min-max backup. This value is proven to converge to the theoretical min-max values when the most promising nodes are expanded first.

3.3 KataGo Architecture

KataGo is the dominant state-of-the-art Go bot[2]. Inspired by the AlphaGo Zero architecture [Sil+17], it consists of a search algorithm guided by a deep neural network. This network is relevant for us because it holds the domain knowledge to support our strength model. In this section, we compile the most important bits from the original KataGo paper [Wu20] and the documentation page "Other Methods Implemented in KataGo" [Wub] as an overview.

The search tree grows by MCTS playouts and it is guided by UCT. However, its playouts only reach the next new node, where its trained neural network performs the positional evaluation in place of the full sample games of previous-generation bots. The next node u at any step in a playout is the one that maximizes

$$\mathrm{PUCT}(u) = V(u) + c_{\mathrm{PUCT}} P(u) \frac{\sqrt{\sum_{u'} N(u')}}{1 + N(u)}$$

where $V(u)$ is the expected value determined as the average *utility* of all sub-nodes, c_{PUCT} is a weighing coefficient, $P(u)$ is the policy value of u determined by the neural net, and $N(u)$ is the number of previous playouts involving u.

In other words, $V(u)$ is the exploitation term which prioritizes the best known moves so far, while $P(u)\frac{\sqrt{\sum_{u'} N(u')}}{1+N(u)}$ is the exploration term with neural net guidance. It generally grows with the proportion between total number of playouts $\sqrt{\sum_{u'} N(u')}$ and node playouts like the traditional UCT exploration term. The policy $P(u)$ contributes bias towards exploring moves that "look interesting". Utility is different from winrate in that KataGo adds a slight preference for more points. Also, KataGo can estimate its own uncertainty and counts broad utility estimates as fractional playouts.

The neural network of KataGo is a convolutional residual network with a deep trunk and several output heads as depicted in Figure 3.4. Its main hyperparameters are the number of *blocks* b and the number of *channels* c. Through public-distributed training efforts, the KataGo project produced different networks with up to 60 blocks and 320 channels. The strongest networks as of April 2024 use just 18 blocks with 384 channels because a faster network allows deeper searches. The KataGo program can load any of these different networks at run time.

The network takes the board state as its input, including more algorithmically determined higher-level input features used by previous-generation bots, but not by AlphaGo. Its

[2]Current competitive bot ratings can be observed in the ratings on CGOS (Computer Go Server): `http://www.yss-aya.com/cgos/19x19/bayes.html`, visited: 2024-10-01. Several strong commercial bots with no presence on CGOS are not represented in this list. There is currently no comprehensive and reliable comparison between bots. In a September 24, 2024 interview (`https://www.youtube.com/watch?v=ix-OE1rsAYE`, visited: 2024-10-01), top professional Shin Jinseo stated that he personally uses KataGo and that Fine Art is currently the strongest bot.

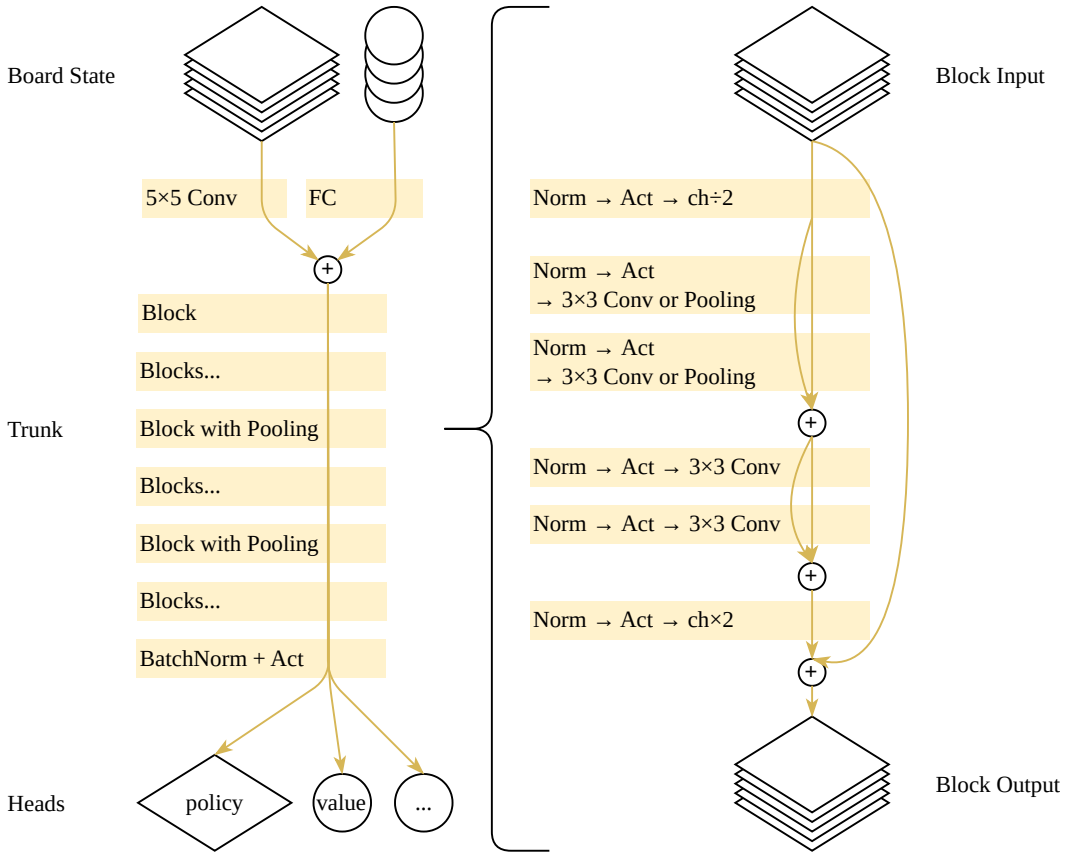


Figure 3.4: The KataGo network outlined on the left consists of a long chain of blocks forming the trunk. The game state is split into spatial and global features, both of which are raised to the full number of channels and combined to be input to the first block. Various heads transform the output of the final block into purposeful results. The *nested bottleneck* block structure, shown on the right, allows these blocks to process features efficiently through the expensive convolutions by projecting them onto fewer dimensions. The inspiration for this idea is attributed to the appendix of [Dan+22], which references [Kai+16]. Additional curved arrows indicate residual connections. A select few blocks, spread out evenly in the trunk, include an extra *global pooling structure*, an adaptation from other domains into KataGo, by which the network can recognize and process whole-board features.

output heads include the obligatory *policy head* $\hat{\pi}(m)$, which predicts the probability that m is the best next move, and *value head* $\hat{z}$, which predicts the winrate. Whereas the initial version of AlphaGo used two separate networks for these predictions, AlphaGo Zero and newer bots train both functions into one network by summing the loss functions of all the heads. As such, the single network benefits from shared information learned via both training targets.

The KataGo paper conjectures that additional, even auxiliary, training targets can improve performance if the added target correlates with the desired target. KataGo includes many more heads for auxiliary targets. Among the more important ones, the *opponent policy head* $\hat{\pi}_{\text{opp}}(m)$ predicts likely response moves, the *ownership head* $\hat{o}(l, p)$ predicts whether location l will become territory[3] for player p and the *score belief head* $\hat{p}_s(x)$ predicts whether the final points difference will be x.

Compared to previous AlphaGo-like bots, KataGo also improves self-play training by *playout cap randomization*. Most moves receive few playouts so that more finished games may be used to train the value estimation, while a few randomly selected moves receive many playouts to gather data for training policy estimation. Another improvement, a minimum number of *forced playouts*, increases exploration during training.

All these improvements and several more [Wub] drastically shorten the required training time and resources of KataGo compared to earlier bots. Now, a network near top human strength can be self-trained on just a single GPU in several days.

3.4 Prediction from Moves

Professional Go player Alexander Dinerchtein authored two interactive web quizzes which automate the assessment by move choice of the user's rank [Dina] and playing style [Dinb]. Over 15-20 questions, each consisting of a board position and a choice for the next point to play, this site estimates its result based on a simple tally of points.

The idea that we can infer information about the players of the game from the moves using trained models is not entirely novel. It dates back to before the advent of strong Go bots and modern neural network architectures. Past approaches relied on hand-picked features taken from the output of the bots available at the time.

Ghoneim et al. [GEA11] aimed to assess the "competency" of actors given their actions in situations in a Go game context, i.e. to evaluate players' strengths based on their moves. They accomplished this by analyzing the reason for the action/move with the bot GNU Go and classifying the player as beginner, intermediate or advanced using trained random forests.

Moudřík et al. [MBN15] also used a bot, Pachi [HP11], to extract features like patterns from game records. Using a bag of 20 feed-forward neural networks, each with one hidden

[3]To be precise, ownership is not just territory. The KataGo paper defines ownership in accordance with *area scoring*, another counting method used under e.g. Chinese rules, which yields the same result as territory scoring in almost all cases.

layer of 20 units, they predicted the player's rank with an estimated standard deviation of 2.66 ranks. Additionally, the authors applied the same model to classify the playing style. Complementing this, Moudřík and Neruda [MN16] turned to deep convolutional networks without involving a Go bot. This time, the authors simply trained such a network on a large number of Go positions with three output classes: weak, intermediate and strong.

A newer conference paper by Kosaka and Ito [KI18] examines game statistics derived from the evaluation of the moves by the open-source bot Ray as indicators of strength. They concluded that the bot is too weak—about 2-dan amateur strength—to judge player move strength with confidence.

Gao et al. [GZL23] present a dataset of professional Go games, annotated with statistics. The authors applied different models to this data for prediction of game outcomes. For input features, they used various player statistics, including pre-existing ratings computed using the whole-history rating system [Cou08] and aggregate move quality indicators determined by KataGo. The authors increased predictive accuracy above the level given by just the input ratings.

The KataGo program supports the supplementation of a separate *human policy network* trained on human games [Wua]. This network takes game metadata as additional inputs. Among these is the rank of the players, which allows the network to imitate the board perception of humans at a specific strength level and adjust its policy accordingly. In contrast to our approach, the human network only operates on one position at a time. The author mentioned that one could use this feature for strength estimation by evaluating it on a player's game positions across a variety of ranks. The player's rank might be the one that, when input to the human network, gives the highest policy values for the player's actual move locations.

CHAPTER 4

Strength Model

As the core of this work, we design and train a model that predicts player ratings. This strength model combines with the Bradley-Terry model from Section 3.1.1 for expected scores into a behavior rating system. It distinguishes itself from our reference systems presented in Section 3.1 first and foremost by taking more input information, namely the moves contained in evaluated game records, into account. From this basic idea, we expect improved prediction accuracy.

This chapter begins with our training data and its preparation. Then we introduce the different models that we use as baselines and for our core experiment. The measured goal for all our models is high prediction performance as a rating system.

All our models, the baselines and the full model, draw on ingame player behavior via preprocessed move information through the KataGo bot. We start by introducing our *stochastic model* as the first baseline, which provides us with a realistic expectation for predictive performance when using ingame features. Then we describe the common architecture and training process of our neural network models: the *basic model* built on normal KataGo output features and the *full strength model* using features taken directly from the internal representation of the KataGo network trunk.

The chapter concludes with our training targets and methods.

4.1 Dataset

Our dataset is the OGS 2021 collection [Van], a public set of games played on the OGS platform. This dataset is a representative sample of online amateur Go over almost two decades. We present the preparatory steps by which we select the games that we need for model training. Then we present some statistics and insights into the data.

4.1.1 Dataset Preparation

This dataset is not only very large, it also contains many records of insufficient quality. We must aggressively prune records that we deem unfit to be used and fill in missing data.

We instantly dismiss any games on alternative board sizes, i.e. not the standard 19-by-19 lines. These are not the focus of our experiment. The same applies to games with handicap stones, which are defined by a starting disadvantage for the white stones and the incentive to turn the position around with unconventional moves. We filter out games with time allowance of five seconds per move or less. They contain noise (moves below the players' true ability) introduced by internet connection delays. If a game ends in less than 20 moves, it can hardly be called a competition of skill and is filtered out. Games on OGS are either "ranked" or "unranked", the latter meaning that the players can enjoy the game without putting their rating points at stake. We respect this designation by retaining ranked games only.

Games can be decided not just by counting at the end of the game, but also by resignation, timeout, cancellation, or forfeit. In the interest of providing accurate learning data, we consider only games that result in counting, resignation or timeout. The results by resignation and timeout may be inaccurate. Resignation may be based on the resigning player's emotional state or external circumstances. Timeout can happen even to the player who holds the positional advantage. We overwrite the winner of these games with the winner as designated by KataGo evaluation and throw out games where KataGo is less than 60% sure either way.

Some records contain illegal moves or early passes before move 50, lack mandatory properties like the result, or are otherwise malformed. We dismiss all those records as well.

All the records that we have not excluded form our pool. We need to select a *training set* $\mathcal{T}$, a *validation set* $\mathcal{V}$ and a *test set* $\mathcal{E}$ from the sequence of games G. However, if we just form three distinct pools from the original one, we tear apart players' rating histories, depriving our algorithms of the data from which they derive their predictions. Instead, we assign 10,000 games to the training set and 5000 each to the validation and test sets, while keeping them in the same pool. An additional 5000 games form the *exhibition set* $\mathcal{X}$. This is a collection of games between players with short histories: no more than 4 recent games each. While the test set is representative of the entire pool, the exhibition set is representative of conditions where our model outperforms classical paired comparison systems. The remaining games go unused due to constraints on computation resources. In the training process, we train only on training games and test only on validation games, while the combined game data is available in the rating history. This technique stems from link prediction problems in social networks, where random test edges are removed from the full graph and later predicted by the model trained on the remaining edges.

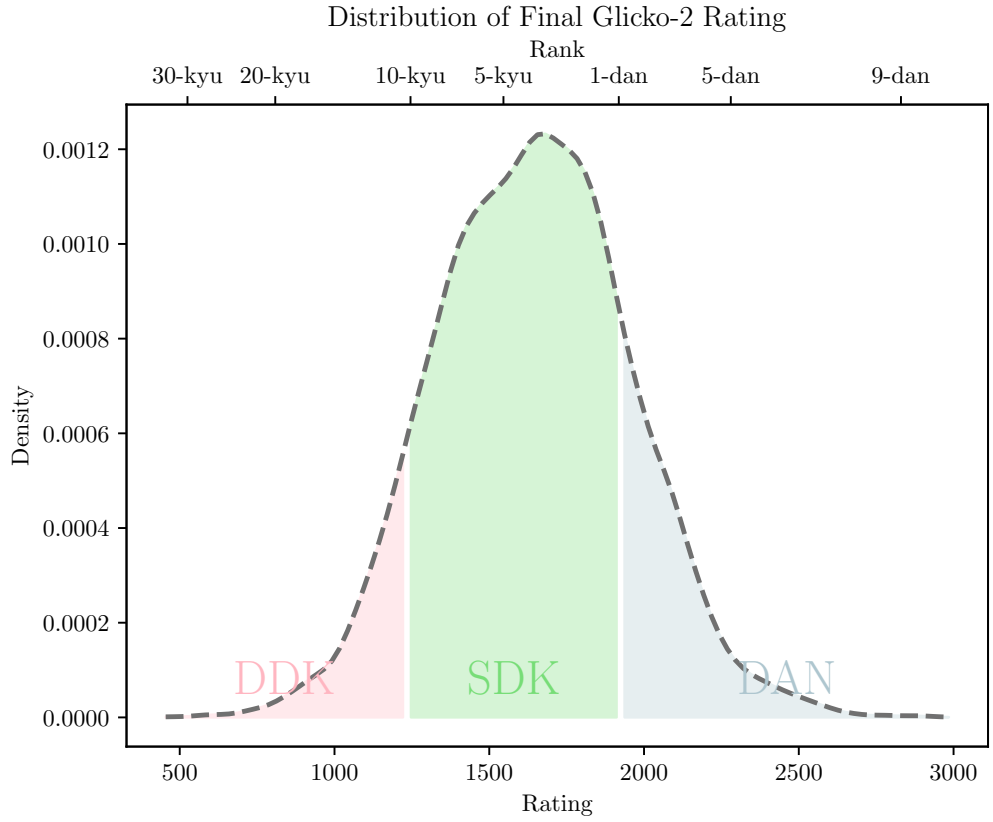


Figure 4.1: This figure shows the density of rating labels in our training set. The bulk of samples have ratings in the SDK (single-digit kyu: 1-kyu to 9-kyu) range. Fewer samples are labeled with DDK (double-digit kyu: 10-kyu and below) ratings, shown in red, or labeled with dan (1-dan and above) ratings.

Even more than the players' strength, match outcomes are very noisy. To improve the quality of training information, we withhold noisy records from the training set, but not from the other sets. We admit a game to the training set only if its rating labels, defined in Section 4.7.1, agree with the score. This means that the stronger player beats the weaker player. Training games must additionally involve players with at least an amount $A = 10$ of future game history. This ensures that more information flows into the rating labels in the training set.

For all recent move sets $M^-(p \mid t-1, N)$—see Definition 2—and for each t indicating a game in our drafted subsets $\mathcal{T}, \mathcal{V}, \mathcal{E},$ and $\mathcal{X}$, we prepare precomputed move features, to be described in Section 4.2. We choose the prepared maximum window size $N = 500$ within our computational constraints. This is the extent of data that we make available for our training and evaluation.

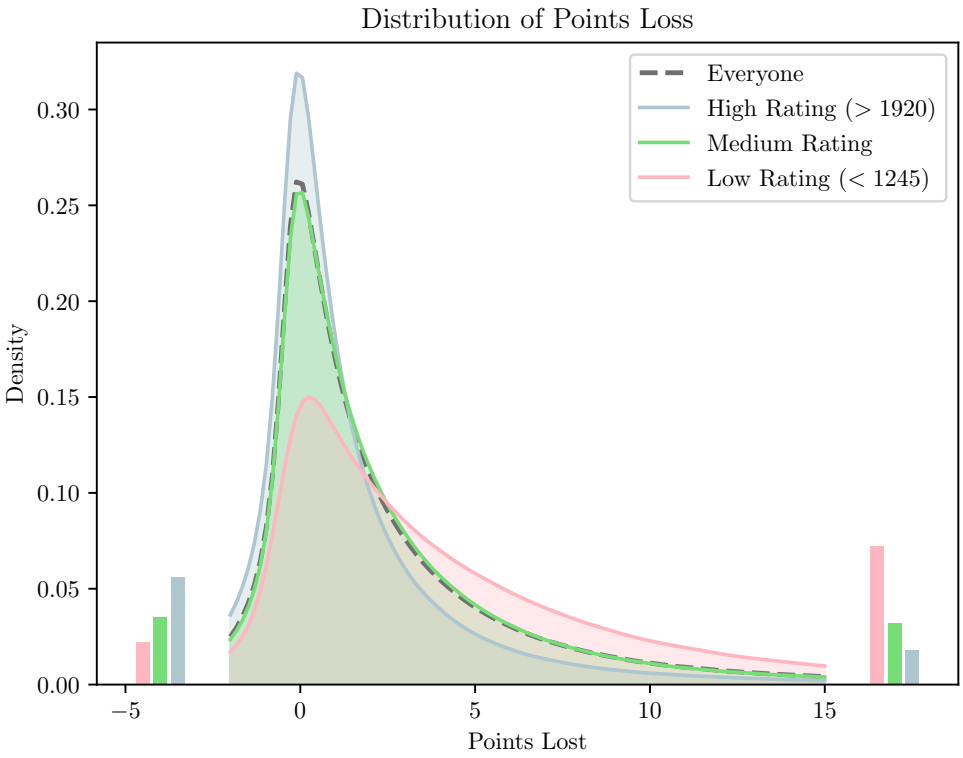


Figure 4.2: This figure shows the density of points loss in the recent moves of our training set. Stronger players lose fewer points per move. We can expect dan-level players to lose less than one point in 25% of moves. While the modal loss is close to 0 in all classes, weaker players have a much wider distribution into blunder territory. The bars on the left and right summarize the outlier surprising gains above 2 points and the huge blunders beyond 15 points. Beginners make such mistakes about 7% of the time.

4.1.2 Dataset Analysis

In this sample of 6,986,379 games, we adjudicated 60.53% of them as won by white and 39.47% by black.

Figure 4.2 illustrates the a-priori distribution of points loss values in our data, broadly categorized by rating. Our measurement of points loss here is the same as defined for the stochastic model in Section 4.3, using the raw KataGo network estimate without search. This explains the frequent "surprising gains" on the negative side of the axis. With some search effort, making gains against KataGo's positional evaluation (anything left of 0) becomes much less likely.

Figure 4.3 illustrates the a-priori distribution of winrate loss values by rating, again determined by raw KataGo evaluation with the same caveats.

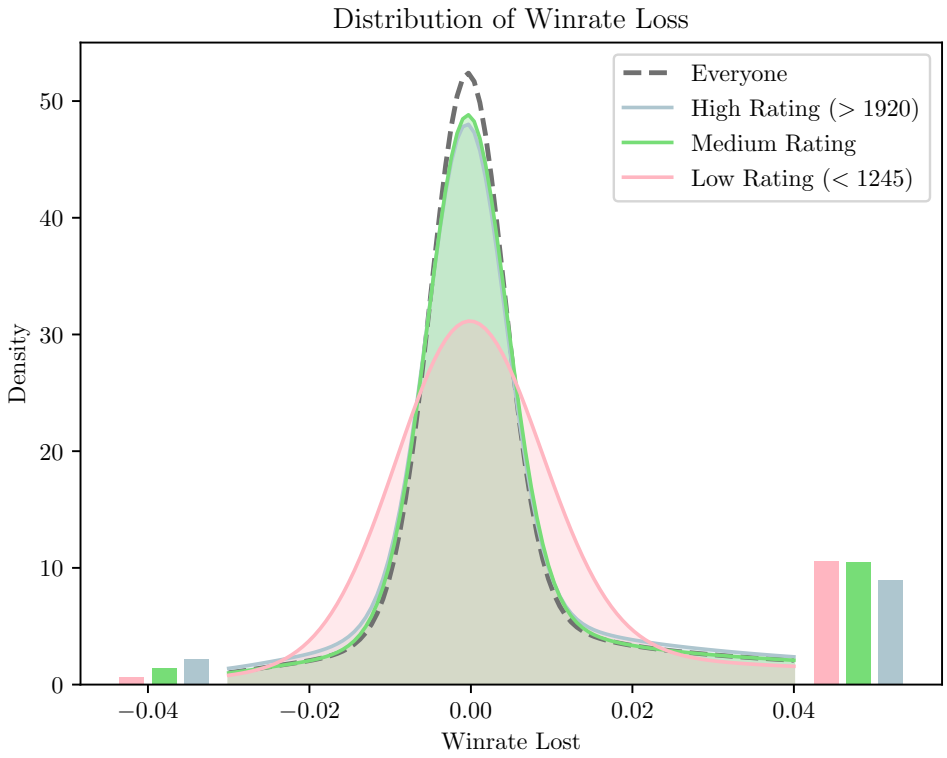


Figure 4.3: This figure shows the estimated density of winrate loss data in our dataset. The difference between players of different skill is not as clear as in the case of points loss. This is because, at a human level, much of the game takes place in very one-sided positions where frequent winrate shifts of less than 1 percentage point do not produce a reliable signal.

In Figure 4.4 and Figure 4.5, we observe how long it takes for Glicko-2 to narrow down its rating estimate. It settles at a rating deviation of around 73.

Among the match outcomes in the dataset, 2,069,239 ended by counting. Their number of moves is shown in Figure 4.6. Unless one loses due to some sudden circumstance like resignation, timeout or forfeit, the players normally continue until there are no more gains to be made on the board. Only then does it make sense to agree to stop and count the points. Yet we observe a wide spread of moves until counting. Hundreds of games have been counted with as little as 100 moves. This is usually too few to properly play out the board. In practice, the end of the game and counting phase is triggered by agreement of the players and can happen at any time. The longest game lasted for 1087 moves.

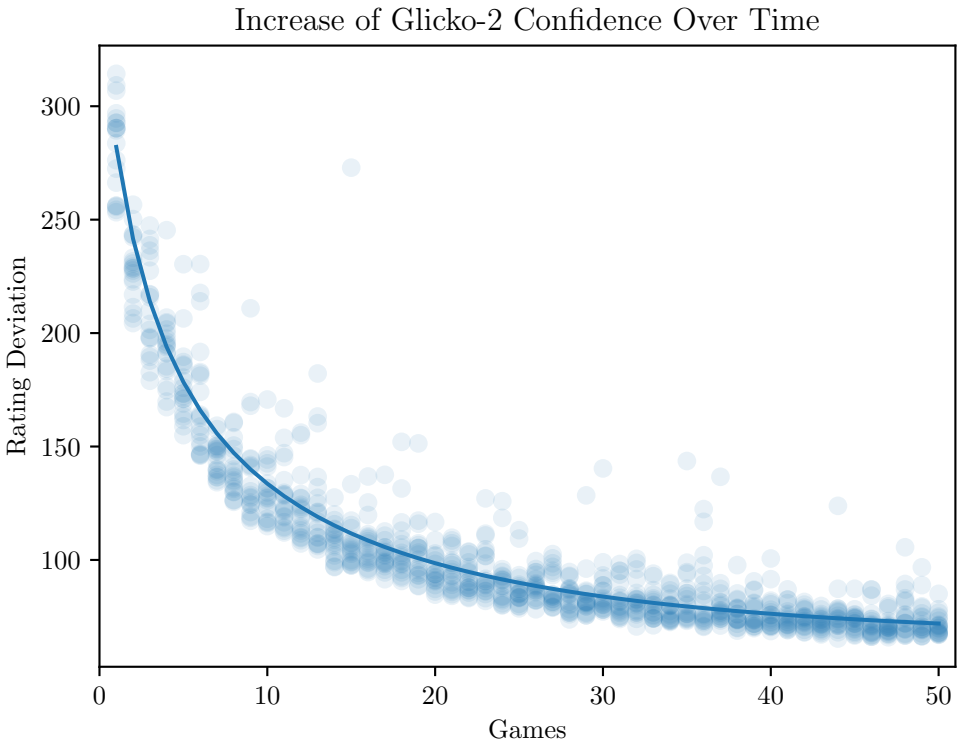


Figure 4.4: As players play more games under the Glicko-2 system, their rating deviation decreases from its initial value of 350. Thus, the confidence of the rating system in its rating estimate increases. For this plot, we use the rating deviations computed by Glicko-2 over our entire dataset. This plot shows a representative sample of 20 deviation values after 1-50 games each. The line shows the mean deviation. It settles around 73 rating points after 50 games played.

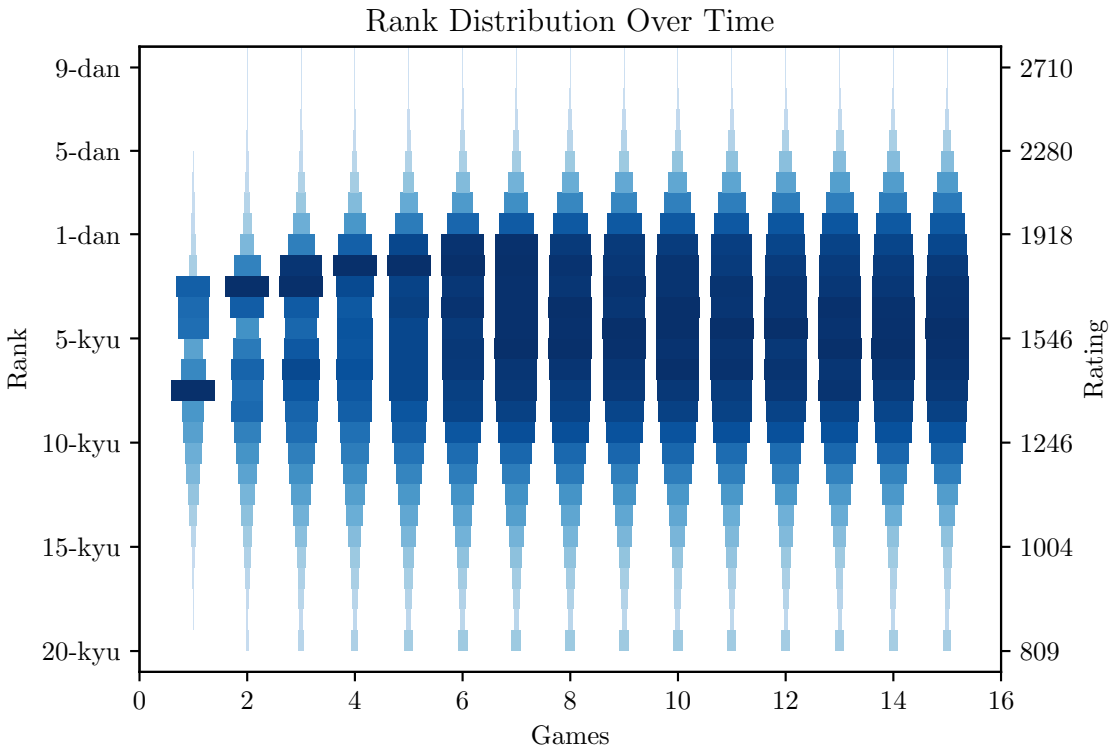


Figure 4.5: Over our entire dataset, the rating system settles new participants into their place in the rating distribution. Each column by itself is a histogram of the amount of players at the various traditional ranks, clipped to the range from 20-kyu through 9-dan, after the specified amount of games on record. The distribution stabilizes after about 7 games. We see the bulk of players in the single-digit kyu range.

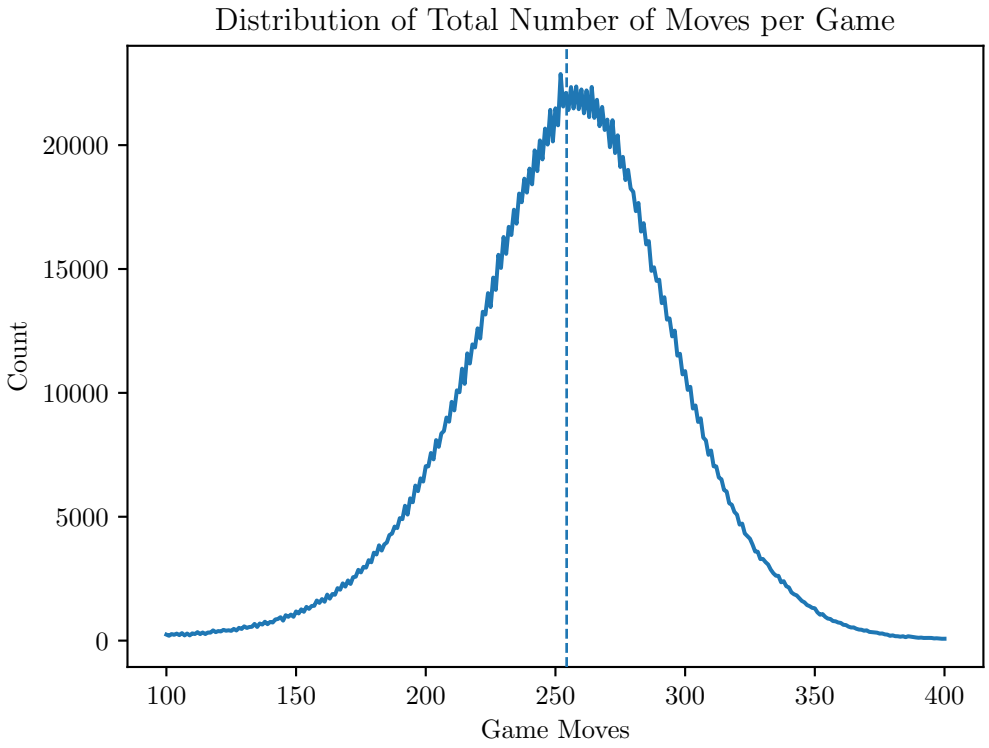


Figure 4.6: This figure shows the distribution of game lengths across our entire dataset. The graph shows the absolute number among ≈2 million counted games. The dashed line indicates the mean length at ≈254 moves.

4.2 Preprocessing by KataGo

All our models are built entirely without exploiting any knowledge of the game of Go. Instead, they depend on the existing KataGo network to understand board positions. More accurately, KataGo networks are available online[1] as weight files, which are continually released when they achieve good performance in ongoing training. These networks differ in their weight values, number of layers, and even layer composition. The specific network that we use in this thesis is `kata1-b18c384nbt-s9131461376-d4087399203`.

# Channels	Feature
1	Location is on board
2	Location has {own,opponent} stone
3	Location has stone with {1,2,3} liberties
1	Moving here illegal due to ko/superko
5	The last 5 move locations, one-hot
3	Ladderable stones {0,1,2} turns ago
1	Moving here catches opponent in ladder
2	Pass-alive area for {self,opponent}

Table 4.1: These are the 18 binary spatial-varying features of the board state passed as input to the KataGo network. They are represented as a $b \times b \times 18$ tensor, where $b = 19$ is the board size. A "ladder" is a direct threat to capture stones on the next move, in a position where this threat could be immediately renewed again and again if the opponent were to attempt an escape. This table is taken from appendix A.1 of [Wu20].

# Channels	Feature
5	Which of the previous 5 moves were pass?
1	Komi / 15.0 (current player's perspective)
2	Ko rules (simple,positional,situational)
1	Suicide allowed?
1	Komi + board size parity

Table 4.2: These are the 10 overall features of the board state passed as input to the KataGo network. Only the first 5 are relevant to our use case because we restrict all evaluations to the same rule set, regardless of the rules under which the games were actually played. This table is taken from appendix A.1 of [Wu20].

[1] `https://katagotraining.org/networks/`, visited: 2024-10-01

As the basis for move evaluation in our models, we draw from the KataGo network preprocessing the *pre-move* and the *post-move* board state. A breakdown of the board state input representation is listed in Tables 4.1 and 4.2.

Our baseline models to be introduced in Sections 4.3 and 4.5 use the following usual KataGo outputs:

- the winrate in both the pre-move and the post-move board states,
- the points lead in both the pre-move and the post-move board states,
- the policy value of the move, i.e. the bot's estimated likelihood that this is the best move, in the pre-move board state,
- the maximum policy value anywhere in the pre-move board state.

These outputs are typical for Go bots. Building other models on them is the natural choice for comparable topical works such as Moudřík et al. [MBN15]

Our full strength model to be introduced in Section 4.6 uses only the trunk output of the KataGo network in the pre-move board state. It does not use the network's heads. The trunk output is a $b \times b \times c$ tensor, where $b = 19$ is the board size and $c = 384$ is the number of channels representing the knowledge associated with each board location. We determine the output feature vector $\in \mathbb{R}^c$ at the location of the move under evaluation as its associated preprocessed feature vector.

4.3 Stochastic Model

Our first ingame-baseline model is the *stochastic model*. This approach foregoes the estimate of rating numbers for the players and the Bradley-Terry formula. This is not a rating system according to our definition in Section 2.3, but we can still determine its accuracy and log-likelihood measures. All that we really need is the estimated score $\hat{s}(t)$, which we estimate in the simplest possible way from the recent move sets of both contestants.

The single move feature that we use in this model is the *points loss*, determined as the difference between the pre-move lead and the post-move lead of the moving player. We model the points loss of a move as an independent stochastic experiment under a normal probability distribution with mean μ and variance σ^2. The parameter values depend only on the player. Naturally, players of higher skill lose fewer points on average.

Let X_i be a specific player's random total amount of points lost over i moves by this player. Its mean is $\mu_i = i\mu$ and its variance is $\sigma_i^2 = i\sigma^2$. Due to the central limit theorem, as i increases, the distribution of X_i ever more closely approximates the cumulative normal distribution function Φ:

$$\lim_{i\to\infty} P\left(\frac{X_i - \mu_i}{\sigma_i} \leq x\right) = \Phi(x). \tag{4.1}$$

This motivates our choice to model even single moves with Φ instead of with a more complicated model that better fits Figure 4.2.

To estimate the score $\hat{s}(t)$ under the stochastic model, we use all available information from the points loss feature to estimate the distribution parameters μ_B, σ_B^2, μ_W and σ_W^2 for the black player and the white player. Knowing the distributions, we can use them to determine the probability that black is ahead or behind on points after C moves. The score estimate concerns the end of the game, assuming that it finishes by counting and not by other outcomes such as resignation. As we can not know the total number of moves in the game beforehand, we simply assume the average that we can observe from our training set. We determine that this average game length rounds to 254 and thus choose $C = 127$. Whoever has blundered fewer points up to that point is the winner.

Let $B_C \sim \mathcal{N}(C\mu_B, C\sigma_B^2)$ and $W_C \sim \mathcal{N}(C\mu_W, C\sigma_W^2)$ with $\sigma_B^2, \sigma_W^2 > 0$ be the approximate random points losses of the black and white player respectively. Then the approximate black win probability is

$$P\left(B_C < W_C\right) = \Phi\left(\frac{C\mu_W - C\mu_B}{\sqrt{C\sigma_B^2 + C\sigma_W^2}}\right). \tag{4.2}$$

To estimate this probability for a game $g_t = \langle p_b, p_w, s\rangle$ at time t, we obtain the recent move sets $M^-(p_b \mid t-1, N_B)$ and $M^-(p_w \mid t-1, N_W)$ of the black and white player. In our comparative evaluation, the window sizes $N_B, N_W \leq 500$ are constrained by what we have prepared from our dataset as specified in Section 4.1.1. Let $B = |M^-(p_b \mid t-1, N_B)|$ and $W = |M^-(p_w \mid t-1, N_W)|$. We derive the observed points losses $b_1, \ldots, b_B$ and $w_1, \ldots, w_W$ from the respective recent move sets. To "observe" points loss in our context always means to preprocess the move data through a strong engine, KataGo in our case, as detailed in Section 4.2. The inaccuracy of the engine becomes noise in our observations. From this data, and if $B, W > 1$, we straightforwardly estimate the win probability:

$$\begin{aligned}
\hat{\mu}_B &= \frac{1}{B}\sum_i b_i, \quad \hat{\sigma}_B^2 = \frac{1}{B-1}\sum_i (b_i - \hat{\mu}_B)^2, \\
\hat{\mu}_W &= \frac{1}{W}\sum_i w_i, \quad \hat{\sigma}_W^2 = \frac{1}{W-1}\sum_i (w_i - \hat{\mu}_W)^2, \\
\hat{s}(t) &= \Phi\left(\frac{C\hat{\mu}_W - C\hat{\mu}_B}{\sqrt{C\hat{\sigma}_B^2 + C\hat{\sigma}_W^2} + \varepsilon}\right).
\end{aligned} \tag{4.3}$$

The arbitrary small value $\varepsilon > 0$ prevents division by 0.

4.4 Network Architecture

Our neural network models are based on the Set Transformer [Lee+19] architecture. In this architecture, the inputs $\mathbf{x}$ represent n elements of an unordered set—in our case, the set of recent moves. Unlike the popular Transformer [Vas+17], which operates on an ordered sequence, its inputs are position-invariant. Any permutation of the input elements yields the same output.

The network consists of a multi-block *encoder*, which uses an attention mechanism to find and distribute knowledge among the set elements, and a single *decoder* layer, which pools the encoded feature vectors into a single output using a final attention query. Residual connections improve training at every step.

The encoder blocks are *induced set attention blocks* or ISABs, characterized by their fixed number $m = |\mathbf{i}|$ of learned *inducing points*. In a first attention query, the inducing points act as query inputs to the attention mechanism, attending to all set elements. The second attention query then attends from every set element to the results of the first query. In contrast to the popular self-attention from every element to every element with its expensive $O(n^2)$ parallelizable operations for $n = |\mathbf{x}|$ elements, induced set attention requires $O(mn)$ parallelizable operations. It suits our use case with up to hundreds of moves, only a few of which are probably interesting.

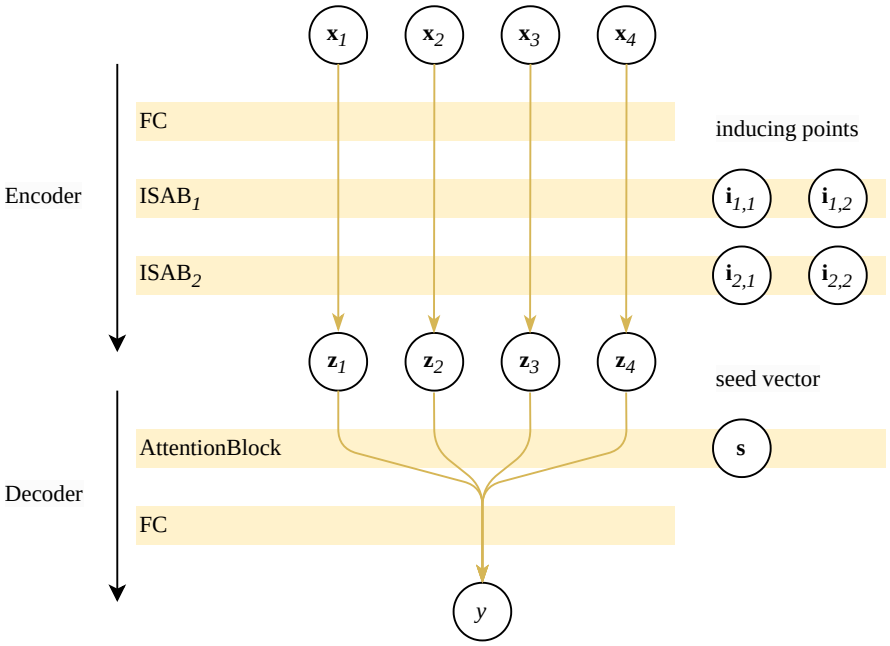


Figure 4.7: Our architecture is a basic implementation of the Set Transformer. The encoder transforms the input set using a stack of efficient attention blocks. The decoder pools the information into the output rating.

We formally define our network as follows.

$$\hat{\mathbf{h}} = \text{LayerNorm}(\mathbf{q} + \text{AT}(W^{\text{Q}}\mathbf{q}, W^{\text{K}}\mathbf{h}, W^{\text{V}}\mathbf{h})), \tag{4.4}$$

$$\text{AttentionBlock}(\mathbf{q}, \mathbf{h}) = \text{LayerNorm}(\hat{\mathbf{h}} + \text{FC}^{\text{b}}(\hat{\mathbf{h}}; d, \text{ReLu})), \tag{4.5}$$

$$\text{ISAB}(\mathbf{h}) = \text{AttentionBlock}(\mathbf{h}, \text{AttentionBlock}(\mathbf{i}, \mathbf{h})), \tag{4.6}$$

$$\mathbf{z} = \text{Encoder}(\mathbf{x}) = \text{ISAB}^{l}(\text{FC}(\mathbf{x}; d, \text{id})), \tag{4.7}$$

$$y = \text{Decoder}(\mathbf{z}) = \text{FC}(\text{AttentionBlock}(\mathbf{s}, \mathbf{z}); 1, \text{id}), \tag{4.8}$$

where

$\text{AT}(\mathbf{q}, \mathbf{k}, \mathbf{v})$	is dot-product attention as defined in Section 2.2.4,
$\text{LayerNorm}(\mathbf{x})$	is layer normalization as defined in Section 2.2.1,
$\text{FC}(\cdot; d, \sigma)$	is an unbiased fully-connected layer with output dimension d and activation function σ (ReLu or the identity function id),
$\text{FC}^{\text{b}}(\cdot; d, \sigma)$	is a fully-connected layer with a bias parameter,
$\mathbf{x} \in \mathbb{R}^{d_x \times n}$	is the input set,
$y \in \mathbb{R}$	is the predicted rating number (not to Glicko-2 scale),
$\mathbf{i} \in \mathbb{R}^{d \times m}$	are the learnable inducing points,
$\mathbf{s} \in \mathbb{R}^{d}$	is a learnable *seed vector*,
$W^{\text{Q}} \in \mathbb{R}^{d_{\text{q}} \times d}$	is a learnable transformation yielding queries,
$W^{\text{K}} \in \mathbb{R}^{d_{\text{q}} \times d}$	is a learnable transformation yielding keys,
$W^{\text{V}} \in \mathbb{R}^{d \times d}$	is a learnable transformation yielding values, and
d, d_{q}, m, l	are hyperparameters.

Refer to Figure 4.7 for the overall architecture, and to Figure 4.8 for a diagram of the attention blocks.

We initialize every element of all weights and biases $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ from a continuous uniform distribution $\mathcal{U}(-\frac{1}{d_{\text{in}}}, \frac{1}{d_{\text{in}}})$. This is a popular heuristic that aims to give the distribution of initial layer outputs the same standard deviation as the inputs. Inducing points $\mathbf{i}$ and the seed vector $\mathbf{s}$ are simply initialized from $\mathcal{U}(0, 1)$.

4.4.1 Motivation

Attention, normalization layers, and residual connections are popular staples of modern neural networks that can be adopted almost by default. The Set Transformer is a solid choice for our application that includes all of these.

We intuitively speculate that the attention mechanism affords our network a sharp focus on the few interesting moves among the input from which it must discern player strength.

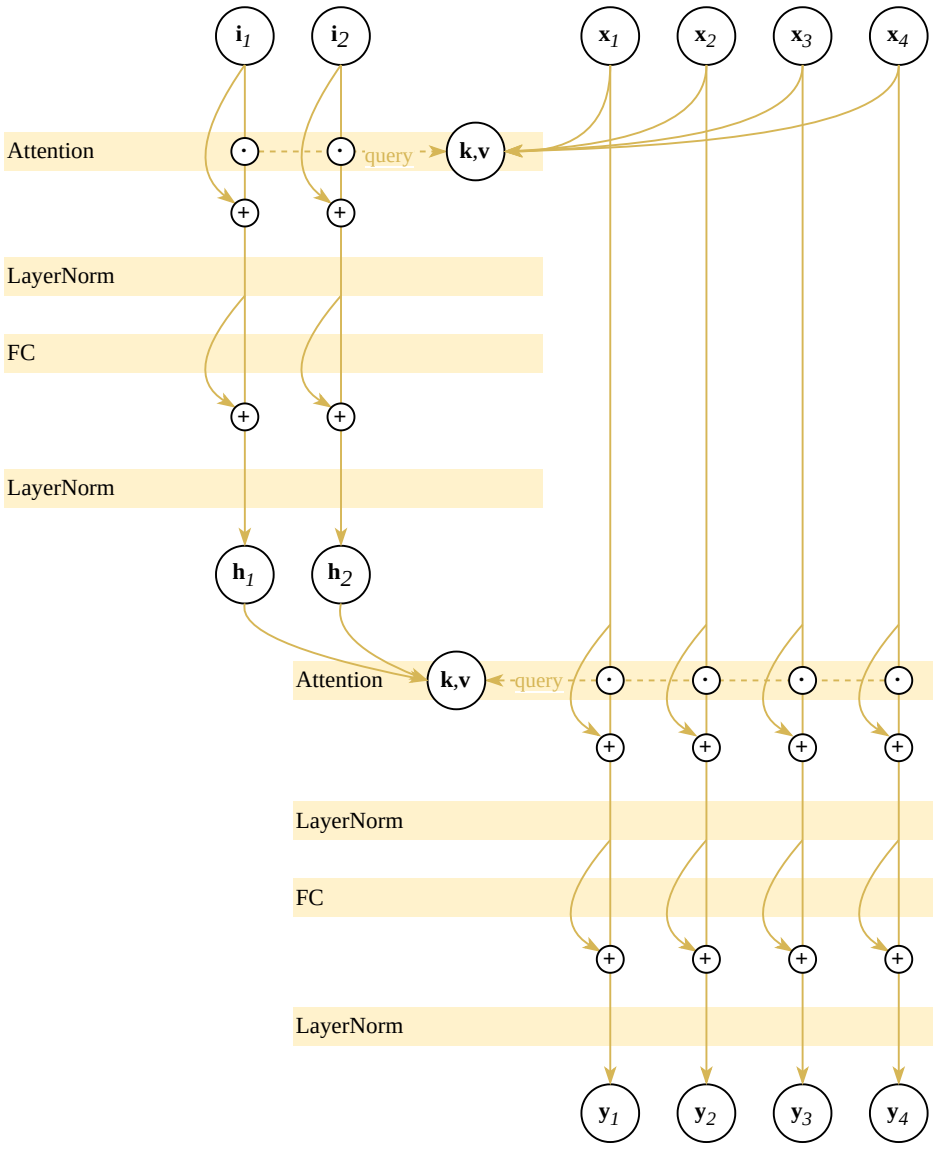


Figure 4.8: The Induced Set Attention Block (ISAB) is an attention mechanism with two phases. First, the inducing points $\mathbf{i}$, which are part of the ISAB as learnable parameters, attend to the inputs $\mathbf{x}$. Through a fully-connected layer with ReLu activation, the query results become the hidden features $\mathbf{h}$. The input elements attend to the hidden values in the second phase and then pass through another fully-connected layer to yield the block outputs $\mathbf{y}$. All attention and fully-connected operations have residual connections.

This improves on the previous work by Moudřík and Neruda [MN16], in which the authors aggregate per-move strength estimates with basic functions like sum and mode.

Considering how a Go game record holds a sequence of moves, one might wonder why a sequence-based architecture like the original Transformer is not our first choice. However, a recent move set as we define it does not fit the sequence structure well for several reasons.

First, recent moves can stem from multiple past games. It would be dubious to simply concatenate these games' moves as if the last move of the previous game was the sequential predecessor to the first move of the next game. Second, we only include every second move according to the perspective of the player under evaluation. This takes out half the sequence elements. Finally, pertaining to our full model of Section 4.6, the board state inputs to the KataGo network include information on five preceding moves, including the opponent's. We can assume that, insofar this information is relevant to KataGo's opinion of the position, the knowledge gained from it is also encoded in the trunk feature vector that we sample for our inputs. This alleviates the need for an explicit sequence structure.

4.4.2 Differences to Original Set Transformer

The authors of [Lee+19] leave room in their Set Transformer definition to choose any kind of feed-forward layer, any number of decoder outputs, and multi-head attention. In this work, we specifically opt for the simplest composition for our purpose. We have also adapted the notation and superficial conventions, e.g. we use column vectors instead of row vectors.

4.5 Basic Model

We build the *basic model* as a Set Transformer model that uses regular bot outputs as input features. It is this thesis's analogue to existing neural-network approaches from literature, such as that of Moudřík et al. [MBN15] The main difference to our full strength model is that by building on the final output of KataGo on every move, the basic model only sees a condensed summary of events rather than the full internal representation of the KataGo network. As such, it is an ablation model with regards to the transfer learning aspect of the full model.

Let M^- be a set of recent moves by the player whose rating we want to predict. We determine the inputs $\mathbf{x} \in \mathbb{R}^{6\times n}$ to the basic model for every move based on the bot outputs listed in Section 4.2. The input features for the move of player p are

1. the post-move winrate from p's perspective, normalized to $[-0.5, 0.5]$,
2. the post-move points lead from p's perspective, scaled by 0.1,

3. the policy value of the move, i.e. the bot's estimated likelihood that this is the best move, normalized to $[-0.5, 0.5]$,
4. the maximum policy value anywhere in the pre-move position, normalized to $[-0.5, 0.5]$,
5. the winrate loss, i.e. the percentage points difference between the pre-move and post-move position,
6. the points loss, i.e. the lead difference between the pre-move and post-move position, scaled by 0.1.

4.6 Full Strength Model

The *full strength model* is our main contribution in this thesis. It uses the existing expert knowledge in the domain network—for which we use the KataGo network—to its full potential while training the same architecture towards the same targets as the basic model.

The input is a set of n recent moves M^-, preprocessed into feature vectors $\mathbf{x} \in \mathbb{R}^{c \times n}$ taken from the trunk output of KataGo as specified in Section 4.2.

Again, these inputs are fed into our Set-Transformer-based architecture from Section 4.4.

The trunk output features represent refined insights that form the basis of KataGo's decisionmaking. Our hypothesis is that they contain more pertinent information for strength estimation than the final bot outputs used in other neural network models, specifically our basic model and the model of Moudřík et al [MBN15].

Figure 4.9 illustrates the feature pipeline for the full strength model.

4.7 Training Regime

We train both the basic model and the full strength model using the same process, described here. The only difference between the two models is the nature and dimensionality of their input features.

Our optimizer algorithm of choice is Adam [KB17]. We first determine our opimization targets. Then we delve into the details of the algorithm. We conclude with the orchestration of multiple training runs to find good hyperparameters.

4.7.1 Targets

Our experiment covers three kinds of targets: the *future Glicko-2 target*, the *Bradley-Terry target* and the *regularization target*. For the future Glicko-2 target, we precompute the full Glicko-2 rating history over the training pool. Every game provides two *rating labels*, one for each opponent. They are determined as the normalized calculated Glicko-2 rating

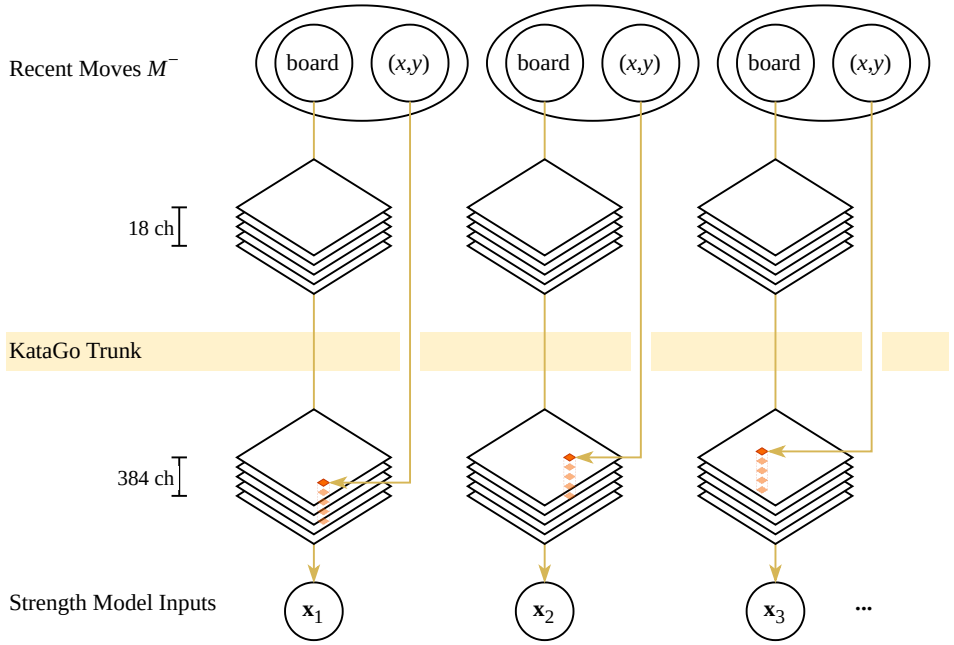


Figure 4.9: The full strength model works on trunk output features of the KataGo network, which extracts expert domain knowledge from the raw input. To that end, the board state before every input move is preprocessed by KataGo, and the feature vector from the internal representation at the move location forms one element of the strength model input set.

number of the particular player, $A = 10$ of their games down the line. The arbitrary value of A is justified by Figure 4.4: An A of 10 roughly halves uncertainty towards its asymptotic value.

We apply standard normalization to all labels, estimated from the ratings in the training set. This allows us to train our neural networks at conventional scales for their output and error values.

For the Bradley-Terry target, every game provides its outcome (score) as the only label. The model under training predicts the rating numbers of both opponents, which feed into the Bradley-Terry model to estimate the expected score.

Let

$G = (g_t), 1 \leq t \leq t_{\max}$	be the pool as a sequence of games,
$\mathcal{T} \subseteq G$	be the training set,
$\mu_p(t)$	be the Glicko-2 rating number of player i at time t,
$H(p) = H(p \mid t_{\max})$	be the game history of player p (see Definition 2),
$\{h_1^p, \ldots, h_{\lvert H(p)\rvert}^p\}$	be the elements of $H(p)$, and
$A = 10$	be how many games in $H(p)$ we *advance*.

The rating label $r(p \mid t)$ for player p at time t is based on their Glicko-2 rating either A history elements ahead, or, if there are not that many future games in $H(p)$, their latest known rating.

$$
\begin{aligned}
r'(p \mid t_0) &= \mu_p(t_1) \quad \text{with } j = \min(i + A, |H(p)|),\ h_i^p = g_{t_0},\ h_j^p = g_{t_1} \\
\overline{r'} &= \frac{1}{2|\mathcal{T}|} \sum_{\substack{g_t \in \mathcal{T}, \\ g_t = \langle b, w, \cdot \rangle}} \left(r'(b \mid t) + r'(w \mid t)\right) \approx 1623.19 \\
s_r^2 &= \frac{1}{2|\mathcal{T}| - 1} \sum_{\substack{g_t \in \mathcal{T}, \\ g_t = \langle b, w, \cdot \rangle}} \left(\left(r'(b \mid t) - \overline{r'}\right)^2 + \left(r'(w \mid t) - \overline{r'}\right)^2\right) \approx (315.81)^2 \\
r(p \mid t) &= \frac{r'(p \mid t) - \overline{r'}}{s_r}
\end{aligned}
$$

Let $M^-(p \mid t, N)$ be the set of recent moves of player p at time t (see Definition 2), and f the model under training. The estimated rating produced by the model is

$$\hat{r}(p \mid t) = f\left(M^-(b \mid t, N)\right).$$

When we learn the Glicko-2 target from a game $g_t = \langle b, w, s \rangle$, we minimize the two squared errors as losses:

$$
\begin{aligned}
L_b &= \left(r(b \mid t) - \hat{r}(b \mid t)\right)^2 \text{ and} \\
L_w &= \left(r(w \mid t) - \hat{r}(w \mid t)\right)^2.
\end{aligned}
$$

Our trained models are rating systems conforming to Equation 2.1 and Equation 2.3 using the rating estimator $\hat{r}$ and score estimator $\hat{s}$:

$$
\begin{aligned}
\mu(\hat{r}(p \mid t)) &= \overline{r'} + s_r \hat{r}(p \mid t), \\
\hat{s}(t) = \hat{s}^{\text{B-T}}(t) &= \frac{e^{\gamma \hat{r}(b|t)}}{e^{\gamma \hat{r}(b|t)} + e^{\gamma \hat{r}(w|t)}}, \quad \gamma = s_r \frac{\ln 10}{400}.
\end{aligned}
$$

The Bradley-Terry target of the same game g_t is its score s. When we learn the Bradley-Terry target from game $g_t = \langle b, w, s \rangle$, we minimize the negative log-likelihood

$$L_s = -\ln(1 - |s - \hat{s}(t)|).$$

Let θ be the set of model parameters. For the regularization target, we simply minimize the mean of squares

$$L_\theta = \frac{1}{|\theta|} \sum_i \theta_i^2.$$

Our training loss function L_t uses all three targets. The balance between them is given by two hyperparameters: the *rating loss scaling factor* τ_r and the *regularization factor* τ_θ.

$$L_t = L_s + \tau_r(L_b + L_w) + \tau_\theta L_\theta$$

The performance loss function for validation and testing is just the score loss.

$$L_p = L_s$$

4.7.2 Training Algorithm

Let $f(\mathcal{B}; \theta)$ be a strength model function on a batch of recent moves $\mathcal{B}$ with parameters θ. Let f be extended here to apply the Bradley-Terry model from Section 3.1.1 to the black and white estimated ratings $\hat{r}_b$ and $\hat{r}_w$, producing the estimated score $\hat{s}$.

All parameters of all weight tensors W in the initial set θ_0 are initialized with uniform probability over $\left[-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}\right]$ where d is the input dimension of W. This initialization preserves the variance of the input data to stabilize the first steps of training.

We train the models for $E = 100$ epochs. Each epoch consists of $T = 100$ steps. In each step, we sample a random minibatch of $B = 100$ from the training games in the pool. After every epoch, we test against the set of validation games. The test calculates the mean squared error of all targets as well as the predictive accuracy α and log-likelihood λ on the validation games. We stop the training early after we have seen no improvement for $p = 10$ consecutive epochs.

Algorithm 4.1 describes the process in pseudocode.

4.7.3 Hyperparameter Optimization

This section describes how we use search to determine the best configuration of hyperparameters for our full strength model. In contrast, we only train one basic model using the same set of hyperparameters as the best-performing full model at the end of the search.

We use random search in 5 iterations of 15 training runs each to locate a good point in hyperparameter space according to the trained model's validation error. The scope of

Algorithm 4.1: Our training algorithm, a classical gradient descent optimization algorithm.

Function `training`$(f, \mathcal{T}, \mathcal{V}, E, T, B)$
  $\theta \leftarrow$ initial values $\sim U(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}})$;
  $\theta^* \leftarrow \theta$;
  $\eta \leftarrow \eta_0$;
  $s \leftarrow 0$;
  $\lambda^* \leftarrow -\infty$;
  for *epoch* $= 1$ ***to*** E **do**
    for *step* $= 1$ ***to*** T **do**
      $\mathcal{B} \leftarrow B$ random games from $\mathcal{T}$;
      grads $\leftarrow 0$;
      for $g_t = \langle b, w, s \rangle \in \mathcal{B}$ **do**
        $\hat{r}_b, \hat{r}_w, \hat{s} \leftarrow f(\mathcal{B}; \theta)$;
        $L_b \leftarrow (r(b \mid t) - \hat{r}_b)^2$; `// black rating target`
        $L_w \leftarrow (r(w \mid t) - \hat{r}_w)^2$; `// white rating target`
        $L_s \leftarrow -\ln(1 - |s - \hat{s}|)$; `// Bradley-Terry target`
        $L_\theta \leftarrow \frac{1}{|\theta|} \sum_i \theta_i^2$; `// regularization target`
        grads $\leftarrow$ grads $+ \eta \nabla_\theta \left(L_s + \tau_r(L_b + L_w) + \tau_\theta L_\theta\right)$;
      end
      $\theta \leftarrow \mathrm{Adam}(\theta, \mathrm{grads})$;
    end
    $\eta \leftarrow (1 - \beta)\, \eta$; `// learning rate decay`
    $\lambda \leftarrow 0$;
    for $g_t = \langle b, w, s \rangle \in \mathcal{V}$ **do**
      $\hat{r}_b, \hat{r}_w, \hat{s} \leftarrow f(\mathcal{B}; \theta)$;
      $\lambda \leftarrow \lambda + \ln(1 - |s - \hat{s}|)$;
    end
    if $\lambda > \lambda^*$ **then**
      $\theta^* \leftarrow \theta$;
      $\lambda^* \leftarrow \lambda$;
      $s \leftarrow 0$;
    else
      $s \leftarrow s + 1$;
      if $s \geq p$ **then** `// early stopping with patience` $p = 10$
        break;
      end
    end
  end
  return θ^*;

the random search scales with a factor γ. It starts with $\gamma = 1$ in the first and broadest iteration, allowing for minimalist to million-parameter models. The scale decays by a factor of 0.6 in each iteration, yielding $\gamma = 1$, 0.6, 0.36, 0.216, and 0.1296. The later iterations refine the selection in the local space around the most promising point.

The space of hyperparameters that we consider is the following.

- initial learning rate $\eta_0 \in [10^{-5}, 1]$
- learning rate decay $\beta \in [0.9, 1]$
- rating loss scaling factor $\tau_r \in [0.001, 10]$
- regularization factor $\tau_\theta \in [0.001, 100]$
- model depth $l \in \{1, 2, 3, 4, 5\}$
- hidden feature dimensions $d \in [8, 256]$
- attention query feature dimensions $d_q \in [8, 256]$
- number of inducing points $m \in [1, 64]$

This does not include the number of training epochs, which is implicitly optimized using early stopping with patience 10.

In the extreme case of the largest hyperparameters, the full strength model with $d_x = 384$ contains 2,821,376 trainable parameters.

Let θ^- be the assignment to hyperparameter θ from the best-performing hyperparameter set of the previous iteration, or the specific values

$$\eta_0^- = 10^{-3},\ \beta^- = 0.95,\ \tau_r^- = 1.0,\ \tau_\theta^- = 10.0,\ l^- = 3,\ d^- = 64,\ d_q^- = 64,\ m^- = 32$$

in the first iteration. Let

$$\text{clamp}(\theta) = \min(\theta_{\max}, \max(\theta_{\min}, \theta))$$

be the clamping function that limits θ within its individual domain $\mathcal{D}(\theta) = [\theta_{\min}, \theta_{\max}]$ as specified above. Let $\mathcal{U}(a, b)$ be the continuous uniform distribution with bounds a and b and let $\mathcal{U}_{\mathbb{Z}}(a, b)$ be the discrete uniform distribution on $[a, b] \cap \mathbb{Z}$. In each iteration, we

randomly adjust the hyperparameters under individual marginal distributions as follows.

$$
\begin{aligned}
\eta_0 &\sim \text{clamp}(\eta_0^- \cdot 10^X), X \sim \mathcal{U}(-3\gamma, 3\gamma) \\
\beta &\sim \text{clamp}(\beta^- + X), X \sim \mathcal{U}(-0.05\gamma, 0.05\gamma) \\
\tau_r &\sim \text{clamp}(\tau_r^- \cdot 2^X), X \sim \mathcal{U}(-3\gamma, 3\gamma) \\
\tau_\theta &\sim \text{clamp}(\tau_\theta^- \cdot 2^X), X \sim \mathcal{U}(-3\gamma, 3\gamma) \\
l &\sim \mathcal{U}(\mathcal{D}(l)) \\
d &\sim \text{clamp}(\lceil d^- \cdot 2^X \rceil), X \sim \mathcal{U}(-2\gamma, 2\gamma) \\
d_q &\sim \text{clamp}(\lceil d_q^- \cdot 2^X \rceil), X \sim \mathcal{U}(-2\gamma, 2\gamma) \\
m &\sim \text{clamp}(m^- + X), X \sim \mathcal{U}_{\mathbb{Z}}(\lfloor -32\gamma + 0.5 \rfloor, \lfloor 32\gamma + 0.5 \rfloor)
\end{aligned}
$$

Finally, we designate the network with the smallest validation error over all training runs over all iterations as the single resultant trained network.

CHAPTER 5

Results

Here we describe the results of the training process and compare the reference system Glicko-2, discussed in Section 3.1.3, with our strength models presented in Chapter 4.

5.1 Training of the Basic Model

The basic model was trained on the same data with the same hyperparameters as the best-performing full model. The hyperparameters are listed in Table 5.1. Figure 5.1 shows the training progress, which is by itself unremarkable. As expected, its performance turned out slightly worse than that of the full strength model.

5.2 Full Model Training

We determined the approximate space of hyperparameters to search for through informal experimentation and consideration of computational resource limits. A preliminary hyperparameter search is illustrated in Figure 5.2. This search used only $T = 10$ steps per epoch, it used early stopping after $p = 3$ non-improvements, and it did not include a few rigorous training features like the regularization target. The window size N examined here is the maximum number of recent moves from the player's history that the model should use for training. The maximum available window size is $N = 500$ from our feature precomputation. Due to the preliminary results, we discarded N as a hyperparameter and fixed it to the maximum.

The proper hyperparameter search is shown twice: once in Figure 5.3 and again, zoomed-in to the more interesting later iterations, in Figure 5.4. Table 5.1 lists the resultant hyperparameters of the best-performing model. We illustrate the training progress of the best model out of the search in Figure 5.5. With 2,469,388 parameters in total, this best

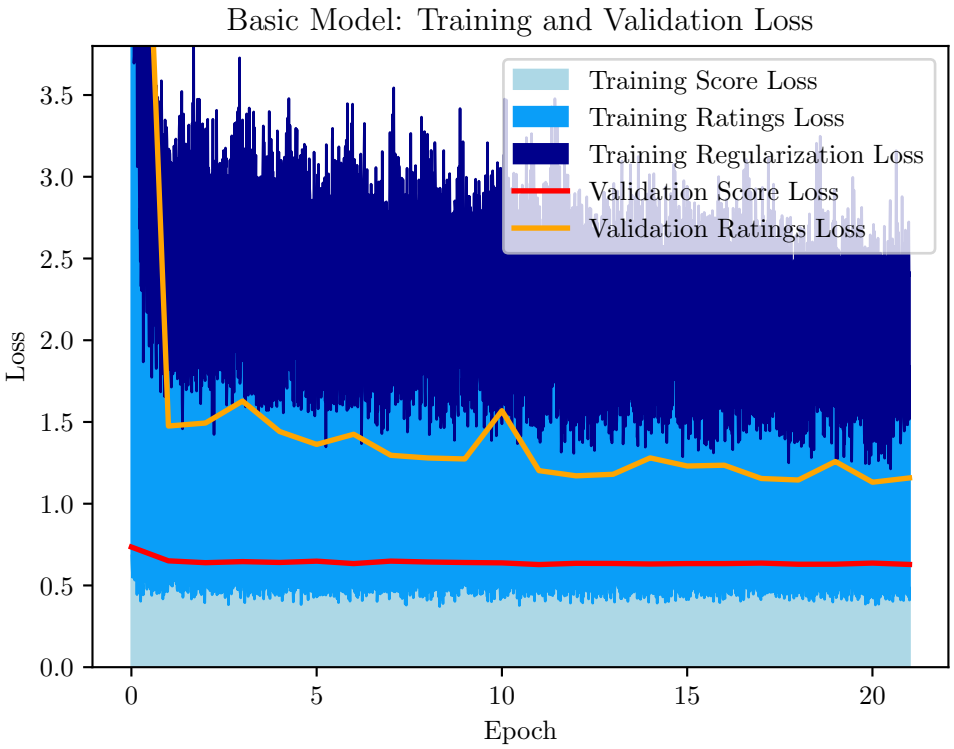


Figure 5.1: Validation performance of the basic model peaked after 11 epochs and training stopped early after 21 epochs. The three components of the training error are sampled at every training step and stacked in blue.

Hyperparameter	**Best Value**
initial learning rate η_0	$\approx 2.14 \cdot 10^{-4}$
learning rate decay β	≈ 0.995
rating loss scaling factor τ_r	≈ 2.05
regularization factor τ_θ	≈ 73.9
model depth l	5
hidden feature dimensions d	254
attention query feature dimensions d_q	154
number of inducing points m	61

Table 5.1: Our final hyperparameter search settled on these parameter values as the highest-performing configuration, achieving a mean log-likelihood of $\lambda \approx -0.578$ on the validation set.

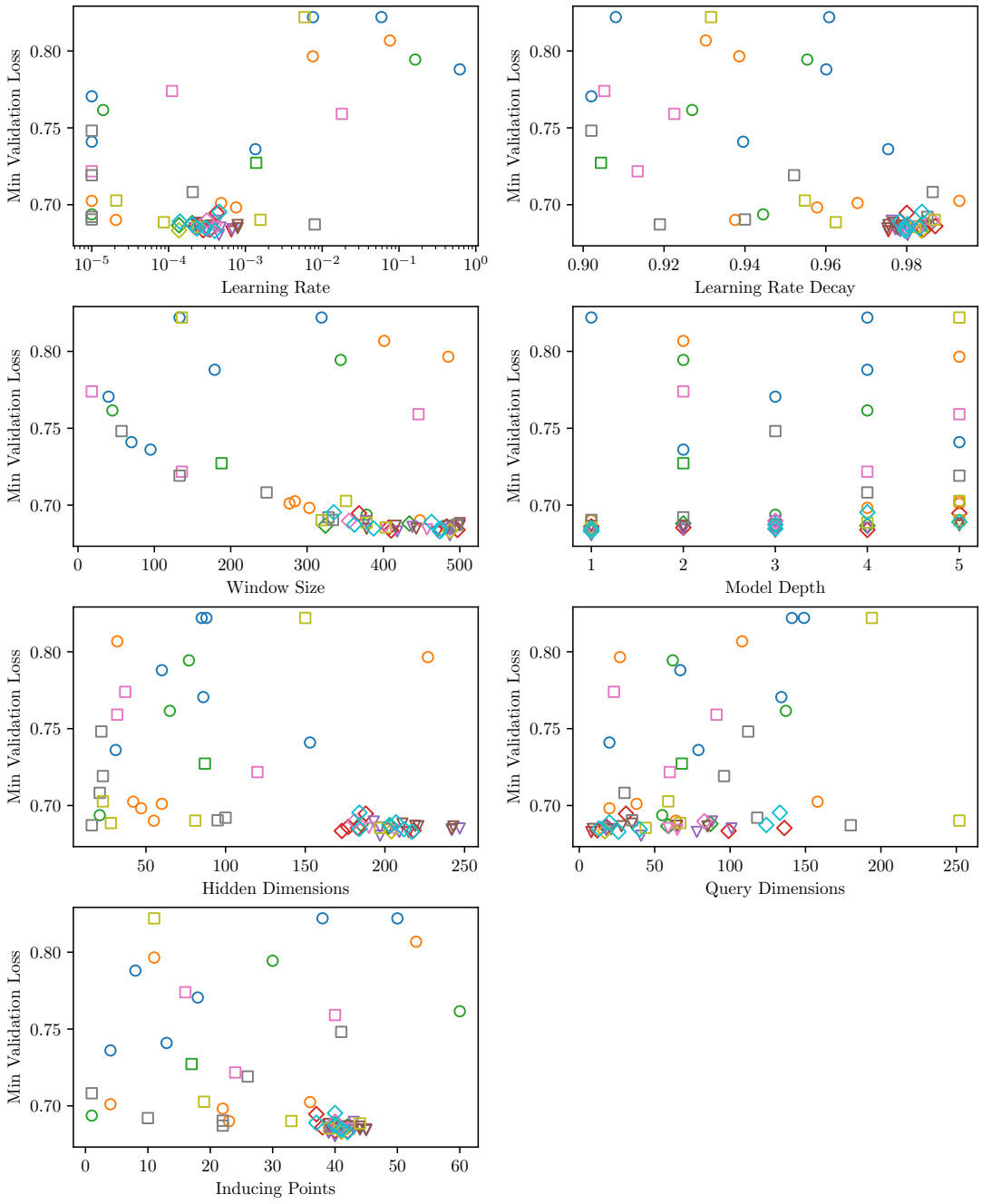


Figure 5.2: A preliminary hyperparameter search showed that the window size N should be as high as possible within our computational constraints. We fixed $N = 500$ in the proper hyperparameter search.

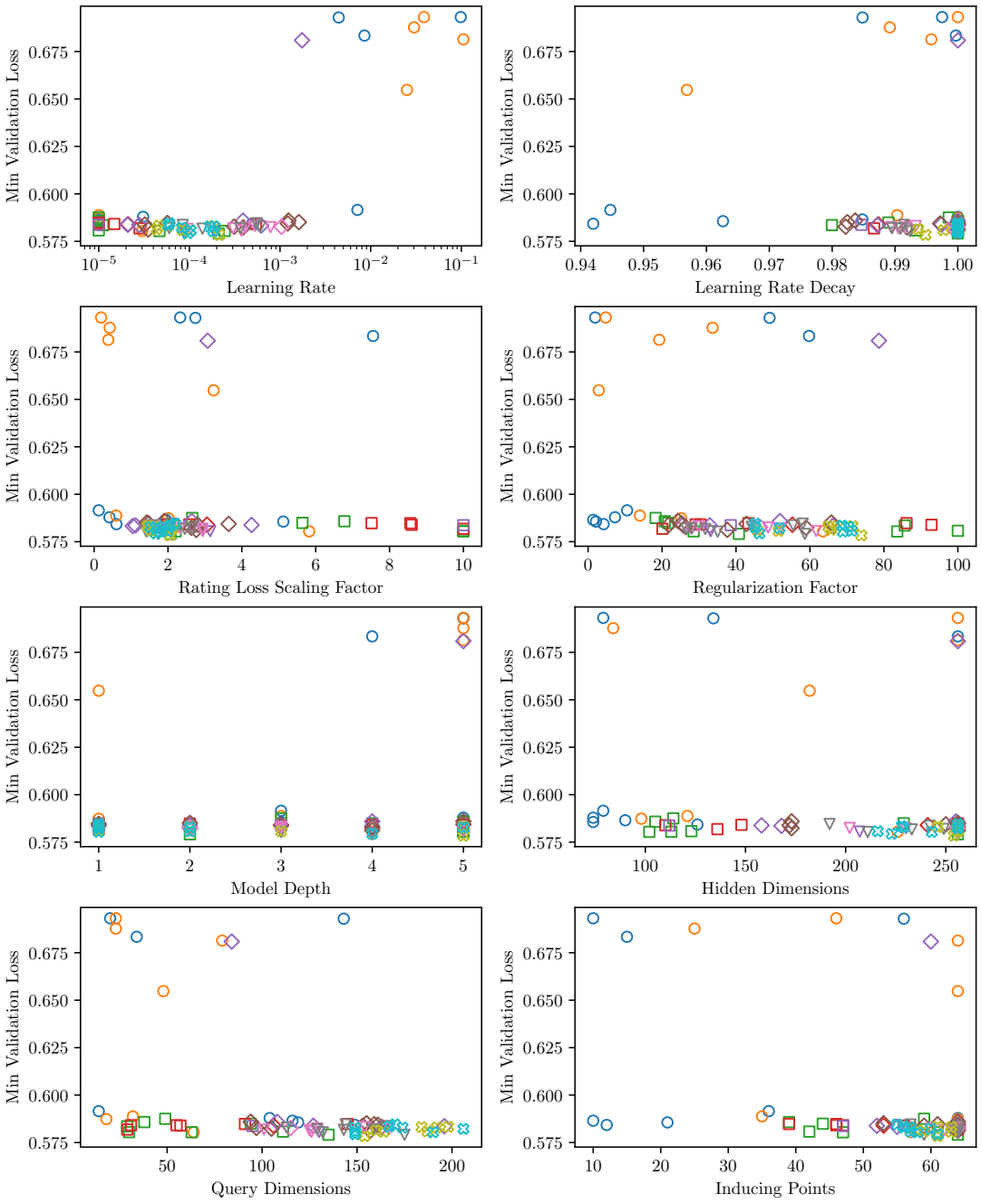


Figure 5.3: Each mark in the scatter plot represents one training run in the hyperparameter search. From broad to fine iterations, we shape them as circles, squares, diamonds, triangles, and crosses.

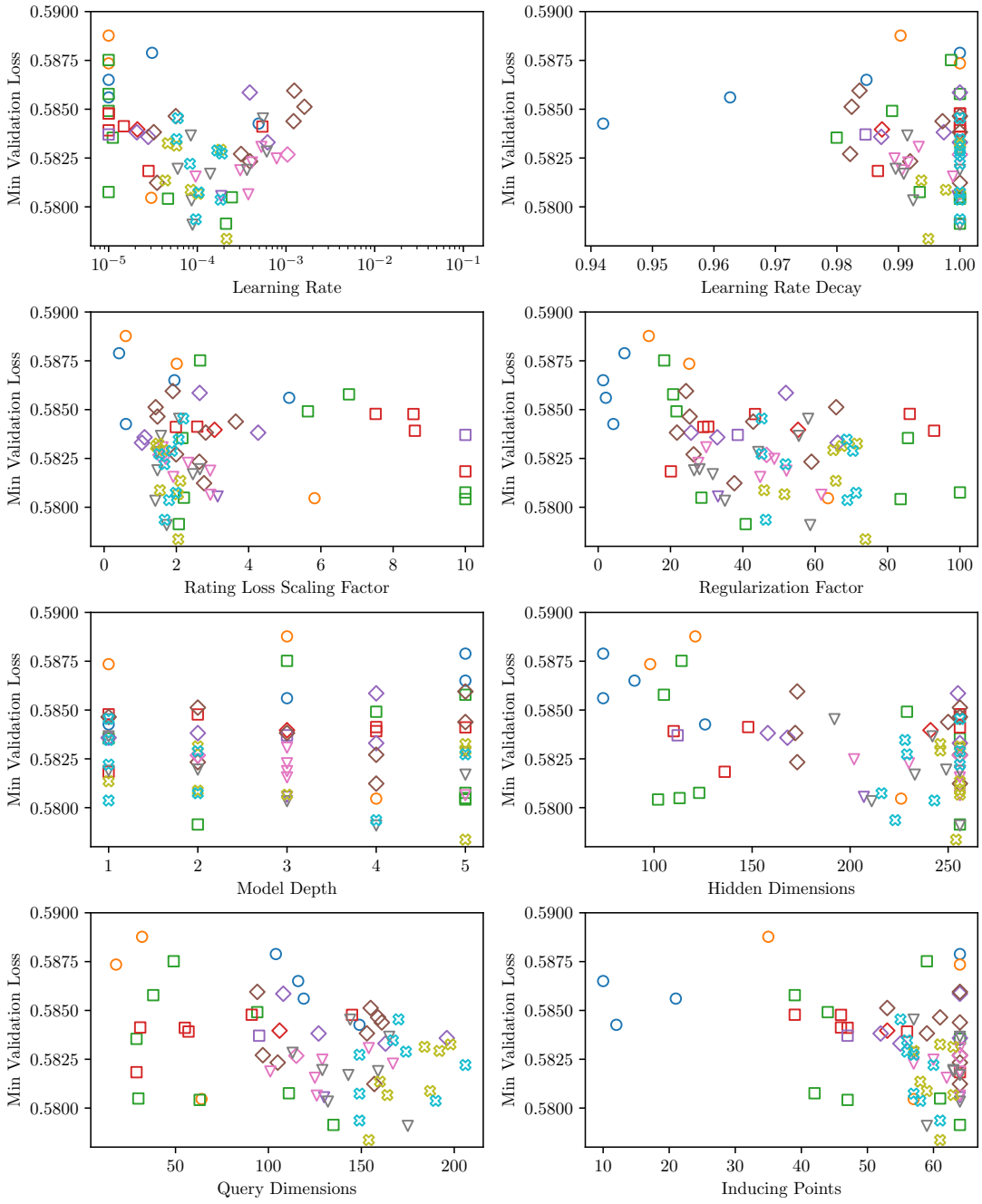


Figure 5.4: This plot shows only training runs that score $-\lambda \in [0.578, 0.59]$ on the validation set. As in the full view, each mark represents one training run.

model emerged at the larger end of the size spectrum afforded by our hyperparameter search space.

With our patience setting of 10, three of the early runs in the first search iteration indeed stopped after only 11 epochs, having seen no improvement after epoch 1. We did allow for up to 100 epochs, but the maximum training length under our early stopping policy was 78 epochs. On average, a training run stopped after 30 epochs.

5.3 Performance Comparison

Algorithm 5.1: This is the algorithm to evaluate a given fully-defined rating system as per the definitions in Section 2.3. For the dataset parameter $\mathcal{D}$, we pass—one after another—the training set $\mathcal{T}$, the validation set $\mathcal{V}$, the test set $\mathcal{E}$, and the exhibition set $\mathcal{X}$ in turn.

Function `evaluate` ($\langle \hat{r}, \hat{s} \rangle$: *rating system*, G: *pool*, $\mathcal{D} \subseteq G$: *dataset*)
  $\alpha \leftarrow 0$;
  $\lambda \leftarrow 0$;
  for $g_t = \langle b, w, s \rangle \in \mathcal{D}$ **do**
    $\hat{s}_g \leftarrow \hat{s}(t)$;
    $\alpha \leftarrow \alpha + \frac{1}{|\mathcal{D}|} |s - I_{\leq 0.5}(\hat{s}_g)|$;
    $\lambda \leftarrow \lambda + \ln(1 - |s - \hat{s}_g|)$;
  end
  return α *and* λ;

For the main result of this chapter, we present the full comparison between all our reference models and our own models across all drafted subsets of our pool from Section 4.7. The measures of comparison are the rating system quality criteria as defined in Section 2.3: the accuracy α and the log-likelihood λ. Algorithm 5.1 lists the pseudocode for the evaluation.

Our first baseline is the 50:50 prediction. Under this "model", we predict that white wins every game with a chance of exactly 50%. Any sensible model should be able to surpass this low bar.

Glicko-2, described in Section 3.1.3, is our reference and state-of-the-art rating system. As a binary rating system, it struggles on the exhibition set. This is expected due to lack of information about the players.

The pseudo-model *Future Glicko-2* "estimates" player ratings as their rating label $\hat{r}(p \mid t-1) = r(p \mid t)$ and match scores as $\hat{s}(t) = \hat{s}^{\text{B-T}}(t)$, using the estimated ratings without rating deviation. The information about the result is already contained in the label, which is taken from the future of the player's rating history. These predictions can therefore perform unrealistically well. We treat it as an upper bound of what can theoretically be achieved.

Model	Training Set		Validation Set		Test Set		Exhibition Set	
	α	λ	α	λ	α	λ	α	λ
50:50	70.1%	-0.69	61.1%	-0.69	61.9%	-0.69	55.6%	-0.69
Glicko-2	89.7%	-0.40	68.6%	-0.58	69.1%	-0.58	64.4%	-0.64
Future Glicko-2	100.0%	-0.34	74.6%	-0.52	75.4%	-0.51	83.4%	-0.38
Stochastic Model	65.7%	-0.94	59.2%	-1.21	58.5%	-1.18	64.4%	-2.03
Basic Model	76.1%	-0.50	65.2%	-0.63	65.2%	-0.61	70.9%	-0.57
Full Model	83.5%	-0.44	69.0%	-0.58	68.8%	**-0.58**	73.7%	**-0.55**

Table 5.2: These are the main results of our evaluation. We compare the different baselines, references, and our approaches. The bold emphasis marks the predictive log-likelihood of our full model on the representative test set, on par with the general performance of Glicko-2. Also listed in bold is its superior log-likelihood on our exhibition set between players with 4 or fewer recent games each.

Our stochastic model from Section 4.3 illustrates a lower bound for behavior-based rating estimation using ingame features. It performs below even the 50:50 model on representative game sets, but already meets the accuracy level of Glicko-2 on the exhibition set. It further tends to be overconfident in the result, yielding terrible log-likelihood statistics.

Our basic model from Section 4.5 shows how much less information the strength model can draw from the final outputs of the Go bot compared to what it can draw from the Go bot's internal representation. It achieves good accuracy on the favorable exhibition set, but it cannot match the binary reference system in the long term.

Our full model from Section 4.6 fulfills our expectations in that it surpasses the baseline models, matches the reference system, beats it under favorable conditions, and remains below the upper bound set by the precognoscious Future model.

The complete raw results are listed in Table 5.2.

Figure 5.6 compares the rating predictions of Glicko-2 and our labels. Figure 5.7 compares the score predictions of Glicko-2 with the actual outcome. The equivalent displays for our best full model are shown in Figure 5.8 and in Figure 5.9.

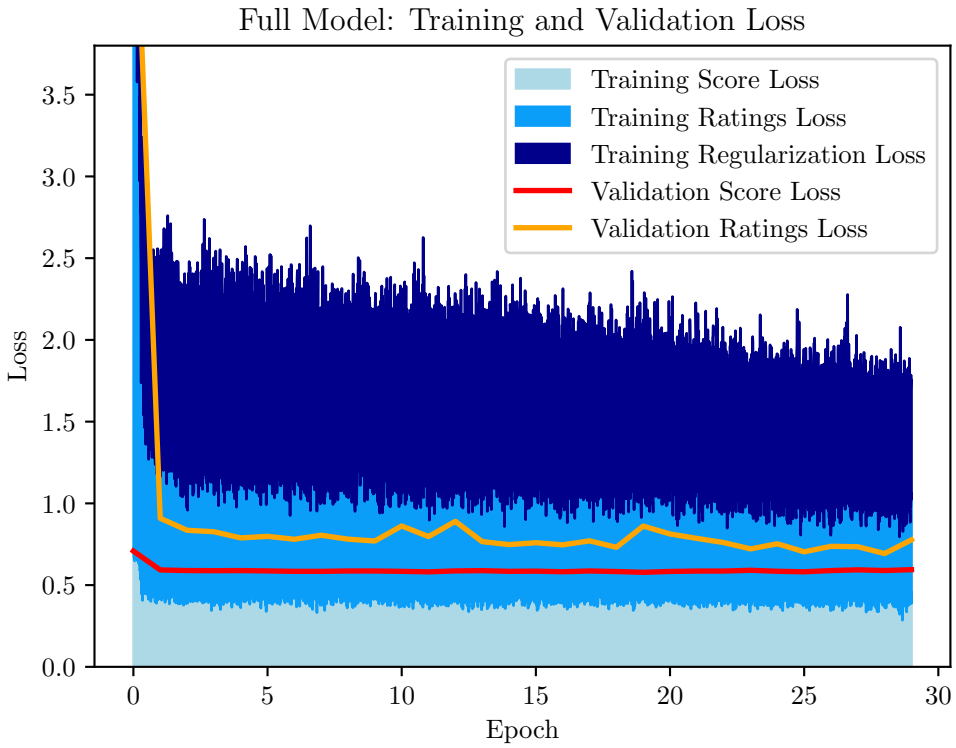


Figure 5.5: This is the training progress of the best-performing model out of the hyperparameter search. Despite the maximum of 100 epochs, the model reached peak performance on the validation set after 19 epochs and training subsequently stopped early. The three components of the training error are sampled at every training step and are stacked in three different shades of blue.

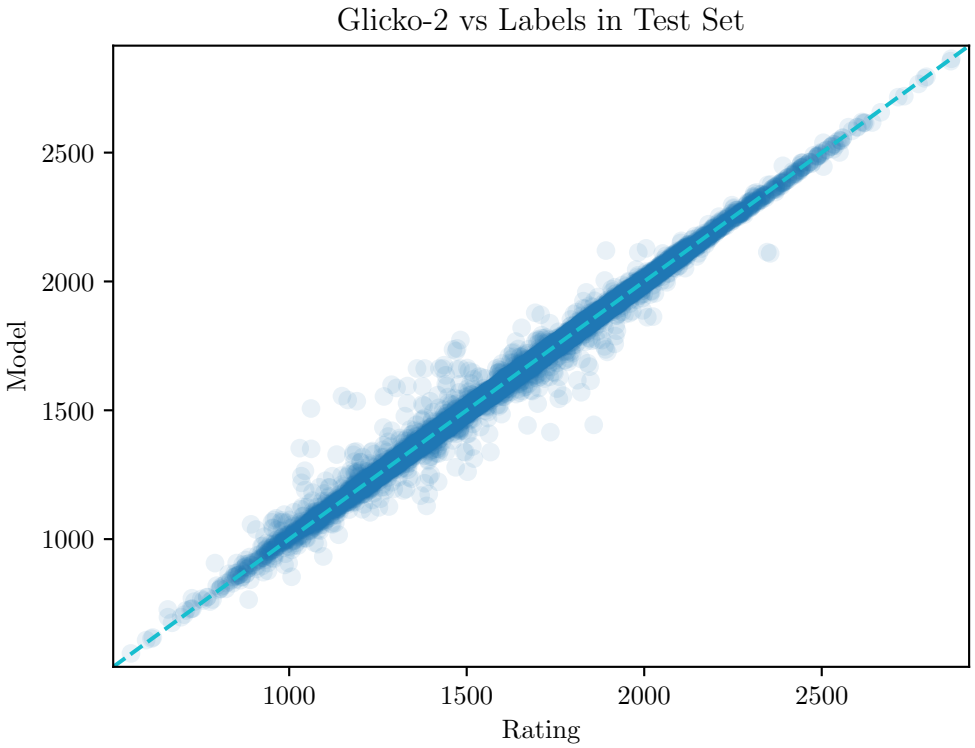


Figure 5.6: In this figure, we compare the rating determined by Glicko-2 before every game with the associated label. It shows that Glicko-2 tends to slightly err towards the mean. Because the labels are the Glicko-2 ratings of the same players from the future, we interpret the off marks as instances where the players' rating has not yet settled in the system.

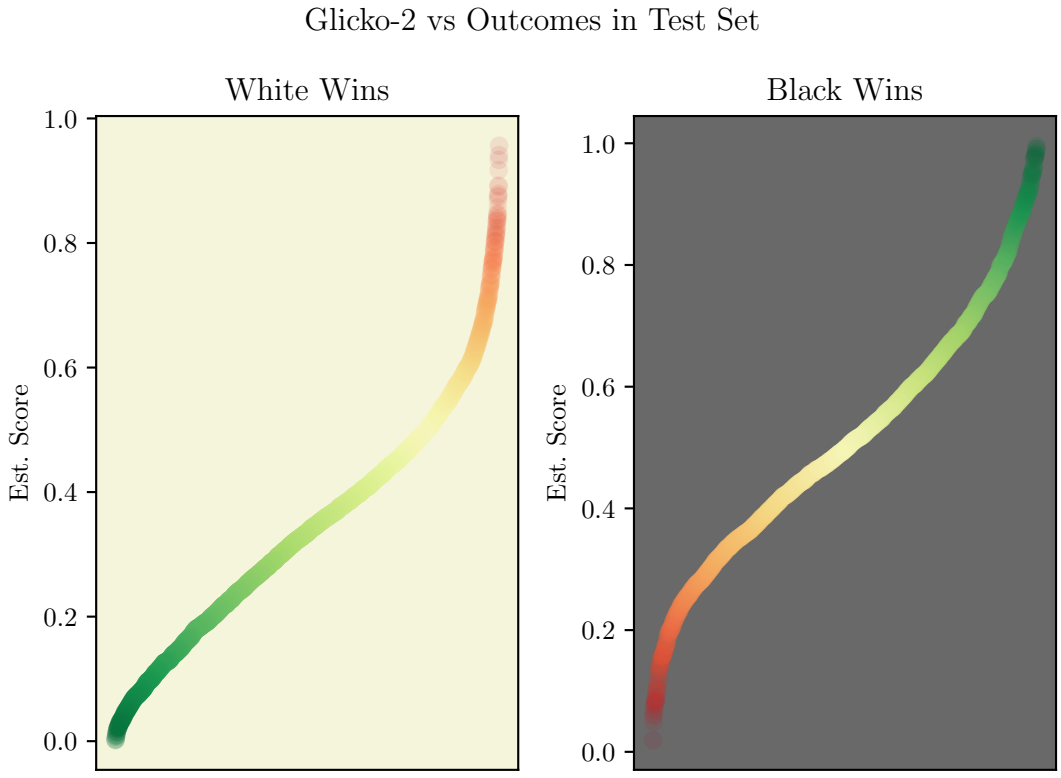


Figure 5.7: This is the distribution of estimated scores by Glicko-2. The left side shows games where the score was 0, i.e. the game outcome was a white win. The right side shows games with score 1. The minority cases in red are those where Glicko-2 was catastrophically wrong.

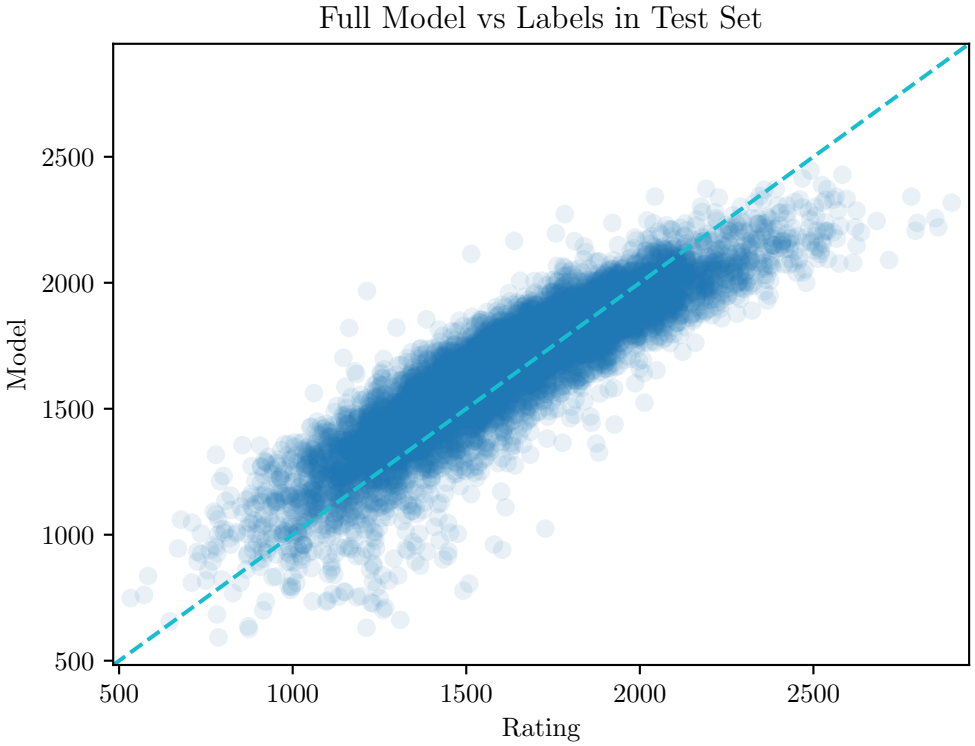


Figure 5.8: In this figure, we compare the rating determined by our full model before every game with the associated label. It shows that the model tends to err towards the mean, just like Glicko-2 does.

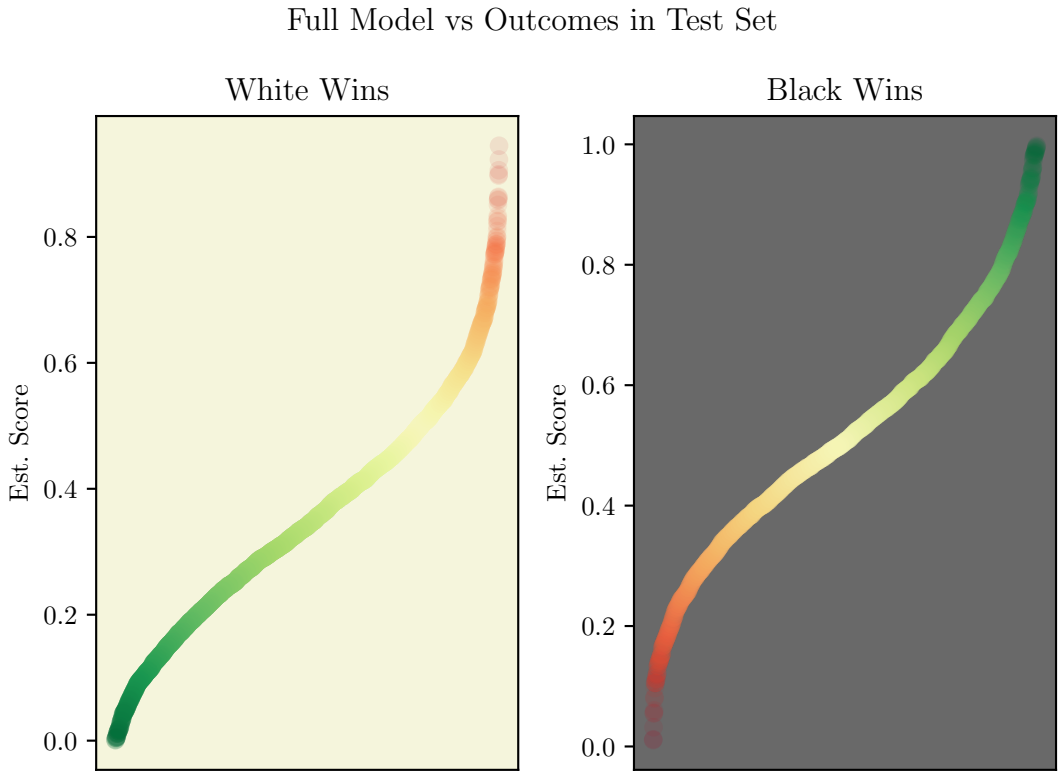


Figure 5.9: This is the distribution of estimated scores by the full model. The left side shows games where the score was 0, i.e. the game outcome was a white win. The right side shows games with score 1. The minority cases in red are those where the model was catastrophically wrong.

CHAPTER 6

Conclusions

We have examined a large pool of Go games and trained a neural network model to recognize the strength of a player from move data. We have drawn on an existing implementation of the Glicko-2 rating system as our reference and established several baseline models to compare their accuracy with our approach in the context of a rating system.

We have shown that our strength model produces estimates on par with Glicko-2 given a set of 500 recent moves, or about 4 average games' worth. When both contestants have short match histories of 4 games or fewer, our model can exploit the information advantage and achieve greater accuracy than the reference system.

The source code and weights for our strength model and for its training algorithms are available online[1]. Using the included scripts and the supplemental forks of both KataGo[2] and the OGS rating system with its Glicko-2 implementation[3], all the results and materials in this thesis can be replicated.

6.1 Performance Details

Our performance evaluation holds yet more noteworthy results. The selection criteria for the training set games dictate that the stronger-labeled player be the winner, and that both players have at least 10 games of future history to ensure an accurate label. This makes predictions easier to such a degree that Glicko-2 outperforms our full strength model, even after it has had the opportunity to overfit on the training data. Perhaps this indicates that the strength model is properly regularized, or that the labels are so inaccurate that the best way to match them is to imitate their methodology.

[1] https://github.com/Animiral/go-strength-model
[2] https://github.com/Animiral/KataGo
[3] https://github.com/Animiral/goratings

Although the points loss per move correlates with playing strength, our stochastic model illustrates that this feature alone is far from sufficient to make qualified predictions. Even our basic model with the same architecture as the full model does not quite live up to our reference. We have built our contribution on the hypothesis that KataGo's trunk output contains more extensive strength-relevant information than its distilled head output. Our results confirm this hypothesis.

6.2 Divergence from Labels

Having achieved satisfactory predictive accuracy with our strength model, we reflect on the extent of similarity between its rating outputs and our test set labels produced by Glicko-2, as shown in Figure 5.8. There are clearly visible divergences. Experimenting users of our public web application, to be introduced in Chapter 7, quickly encountered severe cases of outputs far from expectations. In low ratings, the model's predictions are more scattered. On the other end, the model appears to completely shy away from awarding the highest ratings.

Since we used the rating loss function only as a training loss component and not for validation, the final strength model is not fully optimized towards predicting the labels. Even among the epoch results of its training run, shown in Figure 5.5, there seems to have been an exchange of higher ratings loss for lower score loss in the peak epoch 19.

This general tradeoff does not justify the specific divergences in low and high rating ranges. One possible explanation is that the strength model is always limited by the information content handed down by the domain model. Even if KataGo can play at above-human strength with this information and its search algorithm, there may not be enough information to distinguish between the strongest ranks. Likewise, KataGo is not optimized towards discriminating bad moves.

Another probable factor is imbalance in the training data. We sample our training batches at random from a rating distribution that is representative of our pool. This distribution is dense in the center, as Figure 4.1 shows. Sampling the training data uniformly from the widest range of ratings is a potential future improvement outside the scope of this thesis.

Among the highest-rated players, the strength model shows a clear bias towards the specific playing style of KataGo. Our model judges even other powerful bots as lower-ranked when they prefer different moves than KataGo even though this should be considered a matter of taste.[4] The strength model inherits this bias from the domain model.

[4]Credit is due to the members of the Online Go Forum community, who experimented with various inputs, including games of other bots. See `https://forums.online-go.com/t/how-deep-is-your-go/53060`, visited: 2024-10-01.

6.3 Dataset Bias

One potential bias in our data is that our reference system Glicko-2 was also used for matchmaking in the construction of this same data. Players on the OGS platform may choose their opponents freely, with their rating number on public display. The platform also offers an "automatch" facility which aims to pair comparable opponents with close ratings.

If the matches in the dataset are consistently biased towards indecisiveness under a particular rating system, this system may be disadvantaged in comparison to other systems on the same data as its accuracy tends towards 50%. This caveat should be kept in mind regarding our results.

We must similarly consider mitigating effects on this matchmaking bias. First, we have not actually examined the particular preferences of any player. Some may deliberately seek out stronger or weaker matches. We merely conjecture that most want a fair fight and that opponent-seeking behaviors are clustered and not random, contributing to the bias.

Second, OGS has changed its rating system over time, applying the new system retroactively to the entire match history. The system used to be an Elo system variant until a 2017 switch to Glicko-2[5]. In 2021, the window size of games in a rating update was dropped to one and a bug was fixed that kept players' rating volatility too high[6]. These changes might disturb biases that occurred under the old ratings.

Third, as described in Section 4.1.1, we filter out some games that would originally have been part of the rating calculation, such as handicap games, further diluting the bias.

Fourth, all accurate systems must necessarily produce similar results, given the same pool. One system's matchmaking bias is every system's matchmaking bias, to a degree.

6.4 Domain Network Accuracy

Due to our design choice of preprocessing every move through a separate domain expert model, it naturally follows that our own model inherits all the noise and uncertainty from its dependency. Bots with neural networks may have achieved super-human Go-playing ability, but this characterization applies to a full engine with a search component. The primary training objective for the domain network is to guide the search algorithm effectively, not necessarily to be a strong player by itself. The raw KataGo network has a reputation for being as strong as a professional player, perhaps barely super-human. As a caveat, this is a general statement and does not necessarily apply to a specific game or a specific board situation.

[5] https://forums.online-go.com/t/ogs-has-a-new-glicko-2-based-rating-system-2017/13058, visited: 2024-10-01

[6] https://forums.online-go.com/t/2021-rating-and-rank-adjustments/33389, visited: 2024-10-01

All these things considered, it is less surprising that a competent strength estimate takes hundreds of moves, accounting for the non-informative chaff.

One promising way to lighten this inherited burden is to include the weights of the truncated domain expert network in the training of the strength model, a common transfer learning technique. This allows the whole network, including the domain component, to readjust to the new objective. This method was not available to us due to computational limitations. It remains an avenue for further research.

CHAPTER 7

Applications

We showcase our strength model in two applications. It has a straightforward use case as a supplement to—or even as a drop-in replacement for—a rating system. We have shown this viable in Chapter 5. Besides that, we make the model accessible to the public through a custom website and evaluate the strength of moves given in a book of Go problems. As with the strength model itself, the software associated with these applications is available online[1,2].

7.1 How Deep Is Your Go

We provide a small web application for the public to use the strength model[3]. Figure 7.1 shows the page design.

Users can upload one or multiple SGF files, the most widespread format for Go game records. The specification of the player name is only necessary for disambiguation if it cannot be automatically inferred from multiple game records as the unique omnipresent player. As described in Section 4.6, the application preprocesses the game records through KataGo for its trunk outputs and sends them through our strongest full strength model from Section 5.2. The page then shows the estimated rating number and corresponding traditional rank for the player on the OGS platform.

In model training, we chose to transform the network output by the mean and standard deviation of the rating labels in our dataset, estimated from the training set as described in Section 4.7.1. Even though we used the same Glicko-2 implementation as OGS, the resulting distribution of ratings may differ from the corresponding player ratings on

[1] https://github.com/Animiral/go-strength-model
[2] https://github.com/Animiral/KataGo
[3] http://howdeepisyourgo.org

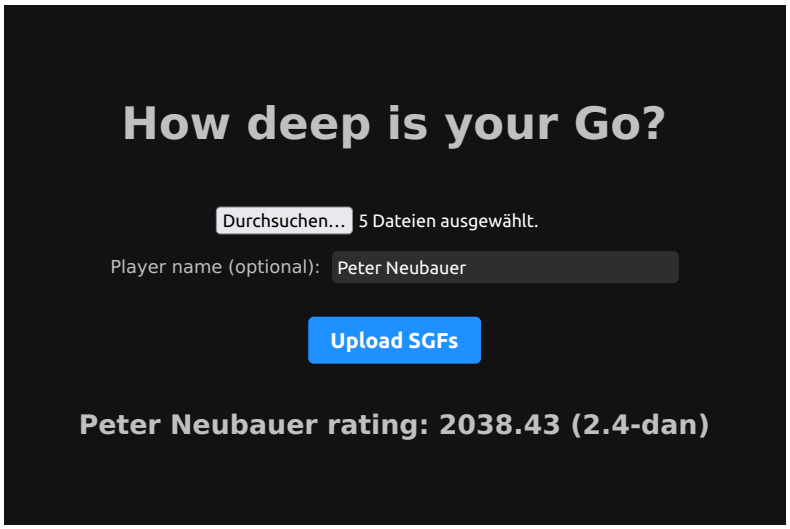


Figure 7.1: The design of our web application is a simple HTML form. Users can upload game records, optionally specify the player name, and submit them to the website. In response, the line at the bottom shows the player's estimated rating and the corresponding traditional rank.

OGS proper. To better match users' expectations, the web application transforms model outputs closer to the true OGS scale.

We estimate the coefficients for this transformation from 100 randomly selected training set games. The strength model produces 200 raw (unscaled) outputs $\mathbf{x}$ for these sample games. We also sample the historical ratings $\mathbf{y}$ of these players in these games from the public API of OGS. Then we fit the model

$$f(\mathbf{x} \mid a, b) = a\mathbf{x} + b$$

to minimize

$$\sum_i \left(f(x_i \mid a, b) - y_i\right)^2,$$

where x_i and y_i denote the i-th element of $\mathbf{x}$ and $\mathbf{y}$, respectively. The result is

$$a \approx 334.03, \; b \approx 1595.1.$$

The web application determines the rating $\hat{r}$ from raw model output x by $\hat{r} \leftarrow a\mathbf{x} + b$. These slightly lower, but more scattered rating numbers can be assumed to slightly degrade our strength model's predictions of match outcomes with a bias towards overconfidence.

The size of a query is limited to 100 kilobytes of data and 1000 moves. This is plenty, considering that our results in Section 5.3 show that our model using just 500 moves is generally on par with OGS's own rating system.

The appplication is written in Python using the `flask` library. A `Dockerfile` is included in the repository for creating an easy-to-deploy container image with all dependencies.

After the release announcement on the Online Go Forums, the website sparked curiosity, experimentation and discussion from the community. It received about 1000 visits in the first week.

7.2 Tricks In Joseki

Joseki are established patterns of moves with a reputation for giving both players an even result in a local board area. A *trick play* is a suboptimal move or combination which requires the opponent to find a *refutation*, i.e., a move or sequence of moves which preserves the good value of the position. Ideally, the refutation is hard to see for a weaker player, or one who shortsightedly plays by instinct and rules of thumb without a grasp on the key of the position created by the trick play.

Go enthusiasts use collections of problems to practice their skills. One such collection that this thesis's author happens to have at hand is the book "Tricks in Joseki" [Yan01]. Each of its 80 trick-play-themed problems is presented in the form of a local board position. It challenges the reader to find the correct next move for black. On the next page, it reveals two move sequences: the correct solution and an example of failure. We pose the question: At what level of skill can players be expected to fall for, and at what level to refute, the trick play?

In the eyes of our model, the answer is the estimated strength based on the failure variation. The refutation should in expectation be assessed as ideal and should receive the maximum rating, or a higher rating than the failure variation in any case. In this section, we present the strength model evaluation of every solution and every failure variation. Rating numbers are again scaled to OGS proportions using the coefficients $a \approx 334.03,\ b \approx 1595.1$ from Section 7.1.

7.2.1 Preparation

The book problems are given in a local context, like in Figure 7.2. Before we run our evaluations on the book sequences, we set each problem into a plausible whole-board context. We perform a linear search through all the ≈7 million games in our local dataset for positions that match the problem patterns, including board symmetries and color inversion. The full board of the game position then replaces the starting problem position. We add the continuing moves from the refutation and failure example respectively to the completed position, creating very short game records.

A handful of problems require our manual intervention. Our pattern search algorithm is too simple to handle capture moves. This affects only Problem 21, where we replace the captured stone location in the search pattern by a wildcard as shown in Figure 7.3.

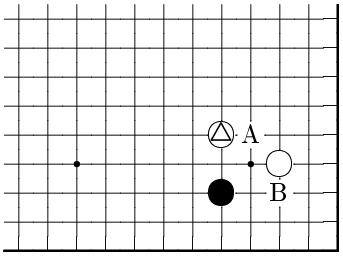


Figure 7.2: This is an example of a trick play problem. It is presented as a partial board position. After white has played the unusual marked stone △, black must respond correctly. To solve the problem, it helps to know that white often plays this stone at A instead, which aims at the right side and protects the corner too. Black deduces that the white shape is now thinner in the corner. This enables an opportunity for black to respond correctly at B.

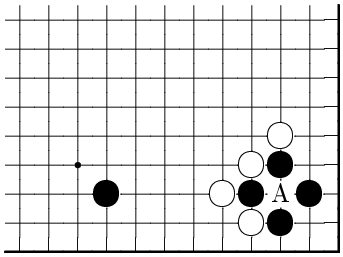


Figure 7.3: Before this position from Problem 21 emerges, black has captured a white stone at A. This is the only such circumstance among the 80 book problems. We replace this location with a wildcard character in our search pattern to make it compatible with our search algorithm and find a complete board position.

Problems 24, 40, 43, and 47, on display in Figure 7.4, do not occur in our dataset because these trick plays are too outrageous to be attempted even by weak amateurs. We complete these boards by removing the last white move that produced the patterns (as either indicated in the book or trivially inferred by the author's Go intuition), searching for the reduced pattern and re-adding the missing stone to the completed board. The statement for Problem 61 includes a global positional feature outside the locally visible part of the board, which we introduce to the completed position by judicious placement of a black stone in a plausible location, marked in Figure 7.7.

7.2.2 Results

The black moves in the book's refutation and failure example variations, set from our prepared completed problem boards, constitute the move inputs to our strength model.

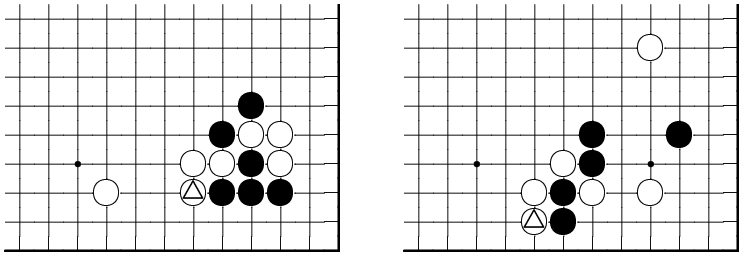
The evaluation of a sequence in our trick play collection works in the same way as the strength evaluation of a player's game. We preprocess all sequences through KataGo

Problem	Failure	Solution
1	2269.02 (4.9-dan)	2262.26 (4.8-dan)
2	1383.36 (7.6-kyu)	2055.44 (2.6-dan)
3	2112.78 (3.2-dan)	2168.48 (3.8-dan)
4	1839.44 (1.0-kyu)	2412.53 (6.3-dan)
5	1832.10 (1.1-kyu)	2252.43 (4.7-dan)
6	1772.66 (1.8-kyu)	2234.58 (4.5-dan)
7	**2498.12** (7.1-dan)	1280.68 (9.4-kyu)
8	**2511.21** (7.2-dan)	2209.89 (4.3-dan)
9	2325.49 (5.5-dan)	2543.01 (7.5-dan)
10	**2145.72** (3.6-dan)	1997.36 (1.9-dan)
11	1574.01 (4.6-kyu)	1860.14 (0.7-kyu)
12	1808.46 (1.4-kyu)	2245.66 (4.6-dan)
13	1822.72 (1.2-kyu)	2199.82 (4.2-dan)
14	**2357.52** (5.8-dan)	1690.07 (2.9-kyu)
15	2195.50 (4.1-dan)	2284.98 (5.0-dan)
16	2016.54 (2.2-dan)	2398.20 (6.2-dan)
17	1942.29 (1.3-dan)	2015.86 (2.1-dan)
18	2269.35 (4.9-dan)	2457.20 (6.7-dan)
19	1589.32 (4.4-kyu)	2206.71 (4.2-dan)
20	**2220.18** (4.4-dan)	2217.49 (4.4-dan)
21	**2058.38** (2.6-dan)	1803.06 (1.4-kyu)
22	1888.77 (0.4-kyu)	2003.46 (2.0-dan)
23	**1970.24** (1.6-dan)	1945.86 (1.3-dan)
24	1895.84 (0.3-kyu)	2405.52 (6.2-dan)
25	1572.03 (4.6-kyu)	2338.04 (5.6-dan)
26	**2409.34** (6.3-dan)	1782.57 (1.7-kyu)
27	1821.21 (1.2-kyu)	2014.88 (2.1-dan)
28	2231.75 (4.5-dan)	2247.87 (4.7-dan)
29	1761.05 (2.0-kyu)	2288.19 (5.1-dan)
30	1789.37 (1.6-kyu)	1697.26 (2.8-kyu)
31	**2168.54** (3.8-dan)	2027.85 (2.3-dan)
32	1772.84 (1.8-kyu)	1821.51 (1.2-kyu)
33	**2337.18** (5.6-dan)	2186.19 (4.0-dan)
34	**2555.07** (7.6-dan)	2299.80 (5.2-dan)
35	**2398.89** (6.2-dan)	2354.95 (5.7-dan)
36	1555.68 (4.9-kyu)	2335.51 (5.6-dan)
37	1308.62 (8.9-kyu)	2216.98 (4.3-dan)
38	1957.07 (1.5-dan)	2340.25 (5.6-dan)
39	2058.11 (2.6-dan)	2439.00 (6.6-dan)
40	1887.76 (0.4-kyu)	2173.01 (3.9-dan)

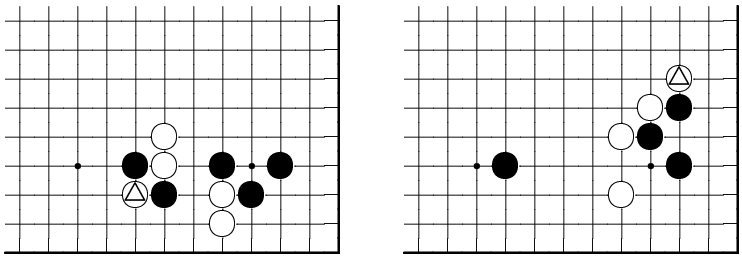
Table 7.1: This list contains evaluations of the strength model on the book problems 1-40. Failure ratings in bold indicate cases where the rating is lower than the rating for the corresponding refutation.

Problem	Failure	Solution
41	2055.24 (2.6-dan)	2291.06 (5.1-dan)
42	1769.17 (1.9-kyu)	2072.75 (2.8-dan)
43	1723.07 (2.5-kyu)	2438.43 (6.6-dan)
44	1928.47 (1.1-dan)	2058.03 (2.6-dan)
45	1898.33 (0.2-kyu)	2001.14 (2.0-dan)
46	**2449.39** (6.7-dan)	2254.57 (4.7-dan)
47	1742.48 (2.2-kyu)	2274.59 (4.9-dan)
48	2010.70 (2.1-dan)	2260.98 (4.8-dan)
49	1744.91 (2.2-kyu)	2403.78 (6.2-dan)
50	2001.70 (2.0-dan)	2203.52 (4.2-dan)
51	1306.38 (8.9-kyu)	2330.86 (5.5-dan)
52	1965.85 (1.6-dan)	2394.92 (6.1-dan)
53	1267.09 (9.6-kyu)	2271.40 (4.9-dan)
54	2242.02 (4.6-dan)	2357.79 (5.8-dan)
55	935.02 (16.6-kyu)	1672.87 (3.2-kyu)
56	2038.25 (2.4-dan)	2102.21 (3.1-dan)
57	1649.59 (3.5-kyu)	2537.86 (7.5-dan)
58	1868.05 (0.6-kyu)	2302.54 (5.2-dan)
59	1897.73 (0.3-kyu)	2153.94 (3.7-dan)
60	2108.14 (3.2-dan)	2184.08 (4.0-dan)
61	**2256.07** (4.8-dan)	2090.81 (3.0-dan)
62	2020.84 (2.2-dan)	2202.72 (4.2-dan)
63	1646.35 (3.5-kyu)	2054.23 (2.6-dan)
64	2058.06 (2.6-dan)	2551.25 (7.6-dan)
65	**2331.87** (5.5-dan)	2202.29 (4.2-dan)
66	2313.89 (5.3-dan)	2508.54 (7.2-dan)
67	1719.86 (2.5-kyu)	2128.77 (3.4-dan)
68	1225.34 (10.4-kyu)	2517.07 (7.3-dan)
69	1962.06 (1.5-dan)	2531.57 (7.4-dan)
70	1316.62 (8.7-kyu)	2378.77 (6.0-dan)
71	1874.47 (0.5-kyu)	2471.16 (6.9-dan)
72	**2206.36** (4.2-dan)	1970.31 (1.6-dan)
73	2081.70 (2.9-dan)	2218.04 (4.4-dan)
74	2117.88 (3.3-dan)	2196.50 (4.1-dan)
75	**2406.97** (6.3-dan)	1906.27 (0.1-kyu)
76	1584.61 (4.4-kyu)	2372.97 (5.9-dan)
77	2104.60 (3.1-dan)	2293.44 (5.1-dan)
78	1560.95 (4.8-kyu)	1997.82 (1.9-dan)
79	1377.99 (7.7-kyu)	1967.48 (1.6-dan)
80	1772.01 (1.8-kyu)	2086.37 (2.9-dan)

Table 7.2: This list contains evaluations of the strength model on the book problems 41-80. Failure ratings in bold indicate cases where the rating is lower than the rating for the corresponding refutation.



Problem 24 Problem 40



Problem 43 Problem 47

Figure 7.4: These problem positions do not occur in our dataset. We adjust our search patterns by removing the last white move leading up to the position, marked △ in each case.

to extract the trunk features at the next move coordinates of all board states in the sequence with black to play. Then we apply the strength model to compute the estimated rating of the sequences. Tables 7.1 and 7.2 show the raw outcome.

7.2.3 Discussion

Among the 80 examined problems, we uncovered 18 surprising cases where the failure variation is rated higher by our model than the book refutation. In this section, we examine these cases, the most interesting among them in detail. We propose semi-formal explanations for the discrepancies derived from manual examination of the positions in question with the KataGo program combined with the author's interpretation.

We visualize the results in Figure 7.5. Our general answer is that effective trick plays exist even into the low dan-player range (2000-2200). Players ranked at least 4-dan (≈2180) and above should be able to refute the trick plays in this set. We cannot interpret too much into the outliers here due to prudent skepticism about the accuracy of the individual data points. On top of the inherent uncertainty in assigning a rating to a very small sample of moves, we introduced extra context by completing the boards in preparation, imposing uneven disturbances across the cases.

In general, most surprising cases are not contradicted by further examination. This means that the proposed failure variation is not really worse, and in some instances indeed

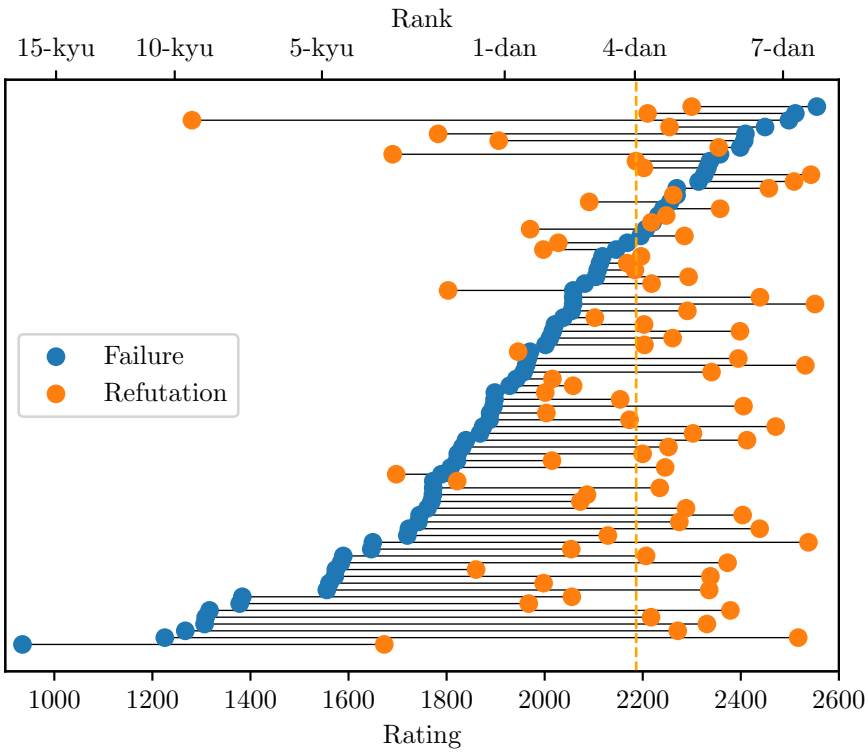
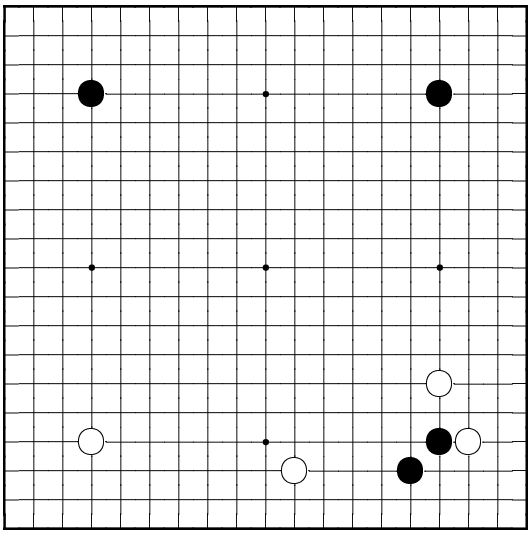


Figure 7.5: This distribution of strength assessments for each of the 80 problems is ordered by the failure rating. The dashed line indicates the mean rating of all refutation sequences.

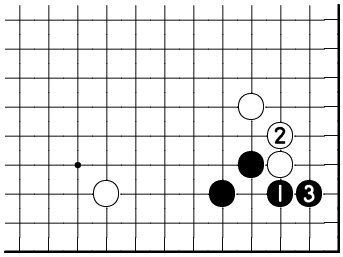
strictly preferable to the book refutation. The phenomenon illustrates the difference between the judgment of humans, even professionals, at a time when quality computer evaluation was not available or not trusted, and the present day, where free tools like KataGo are available to everyone, widely used and trustworthy. In traditional judgement, corner territory was generally undervalued, and small differences in move alternatives during the opening were given undue weight. The book's answers to 13 of the 18 surprising problems fall in this category. They represented the state of the art in 2001, the time of publication.

Problems 8 and 31 have the correct refutation, but later moves in the solution sequence are inaccuracies that taint the strength rating of the whole sequence. This can be blamed on the problem book format, where the black player resolves the local situation in proper form, even when the player would realistically play somewhere else in a real game.

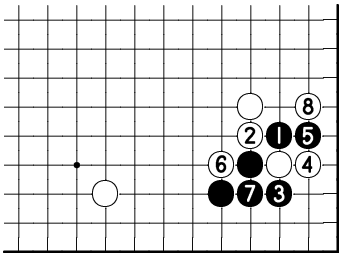
In Problem 23, shown in Figure 7.6, the difference between the two alternatives is negligible. Despite the refutation being the better path, our strength model inexplicably assigns a higher rank to the failure variation. The same happens in Problem 61, shown in Figure 7.7. This time, the refutation is clearly the better choice afforded by the favorable



Problem 23 (completed board)



Refutation

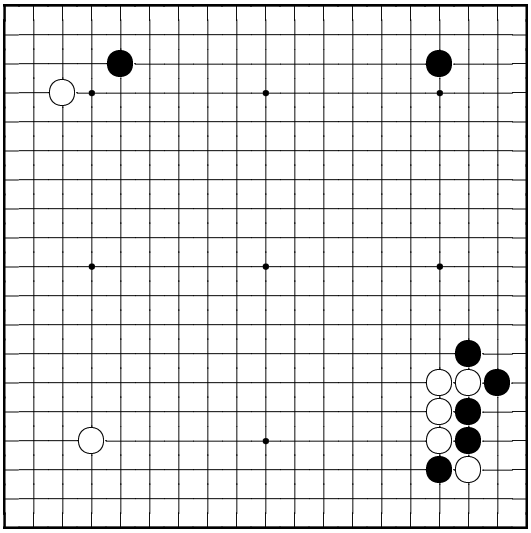


Failure

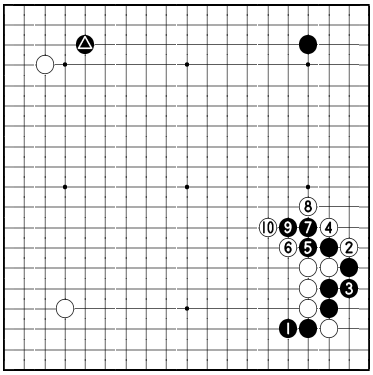
Figure 7.6: This problem causes our strength model to award 25 fewer rating points to the correct refutation on the left than to the failure variation on the right.

ladder situation for black. One possible explanation might be a case of misplaced expectations. We assume that the strength model judges the stronger move with a higher rating. In reality, there is no law of nature that guarantees that overall stronger players will pick the better move in these particular situations. Another possibility is simply that, despite all its training on up to 500-move histories, our strength model produces the wrong result when presented with these particular inputs and having so few moves on which to base its estimate.

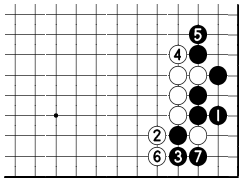
Problem 24 was initially in the suspicious set because our preparatory board completion step yielded a whole-board situation that thwarted the book's challenge. Figure 7.8 illustrates the point. Like Problem 61, the situation depends on a global positional feature outside the local problem statement, this time without explicit mention in the book. Like before, we changed the completed board to ensure that the local question remained intact, then re-evaluated the problem. After this intervention, the evaluation sharply changes in favor of the refutation.



Problem 61 (completed board)

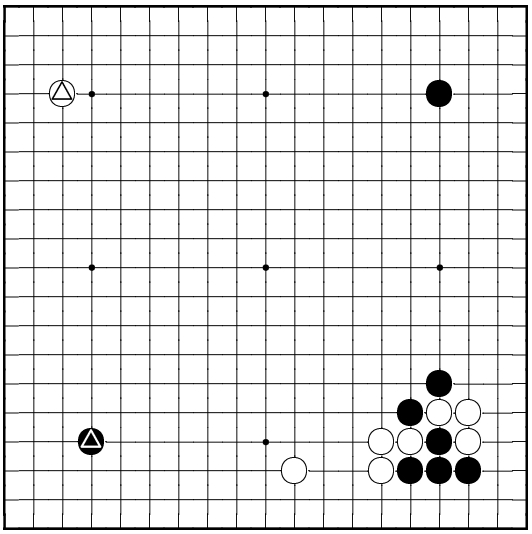


Refutation

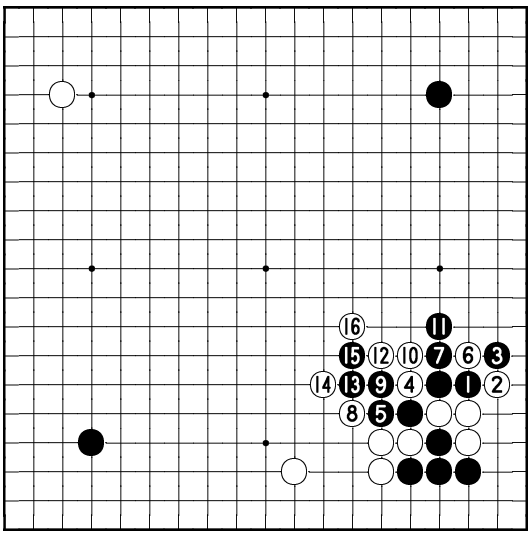


Failure

Figure 7.7: Again, our strength model associates the failure variation with the higer rating, this time by a wide margin of over 150 points. In this problem, it is a failure for black to cover with the usual defensive move shown in the Failure diagram. Black must take advantage of the stone marked with △, which prevents the white attempt to capture black by force, as shown in the Refutation diagram.



Problem 24 (completed board)



Failure

Figure 7.8: We fixed an issue with the preparatory board completion in Problem 24. The stone in the upper left must be white to ensure that the failure variation gives a bad result to black. The lower diagram illustrates how, if black attempts to enclose three white stones with 1 and 3, white can forcibly capture some black stones instead. Our fixed completed board is modified from the original game sample by exchanging the two stones marked with △.

Overview of Generative AI Tools Used

Generative AI tools were used extensively in the development of the software accompanying this thesis. The following contents were authored with influence from AI output:

- general layout and formatting of Python scripts,
- Python and C++ code samples for sub-tasks,
- Python code for specific script components, such as argument parsing, CSV file handling, creating plots, usage of libraries, manipulating data structures, and regular expressions,
- the project structure and the HTML template for the web application from Chapter 7,
- templates for shell scripts.

The tool for these tasks was ChatGPT in versions available from October 2023 to October 2024.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Ass] British Go Association. *History of Go-playing Programs*. URL: https://britgo.org/computergo/history (visited on 10/01/2024).

[BB12] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.2 (2012), pp. 281–305.

[BKH16] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: *arXiv e-prints* (2016). arXiv: 1607.06450.

[BT52] Ralph Allan Bradley and Milton E. Terry. "Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons". In: *Biometrika* 39.3/4 (1952), p. 324. DOI: 10.2307/2334029.

[BW95] Jay Burmeister and Janet Wiles. *Technical Report 339*. Department of Computer Science, The University of Queensland, 1995.

[CHH02] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1-2 (2002), pp. 57–83. DOI: 10.1016/S0004-3702(01)00129-1.

[Cou06] Rémi Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *Computers and Games*. Vol. 4630. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 72–83. DOI: 10.1007/978-3-540-75538-8_7.

[Cou07a] Rémi Coulom. "Computing Elo Ratings of Move Patterns in the Game of Go". In: *ICGA journal* 30.4 (2007), pp. 198–208.

[Cou07b] Rémi Coulom. "Monte-Carlo Tree Search in Crazy Stone". In: *12th Game Programming Workshop*. 2007, pp. 74–75.

[Cou08] Rémi Coulom. "Whole-History Rating: A Bayesian Rating System for Players of Time-Varying Strength". In: *International Conference on Computers and Games*. 2008, pp. 113–124.

[Dan+22] Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. "Policy improvement by planning with Gumbel". In: *International Conference on Learning Representations*. 2022.

[Dina] Alexander Dinerchtein. *Go-test*. URL: http://play.baduk.org/ (visited on 10/01/2024).

[Dinb] Alexander Dinerchtein. *What is your playing style?* URL: http://style.baduk.org/ (visited on 10/01/2024).

[Elo78] Arpad E. Elo. *The rating of chessplayers, past and present*. Arco Pub, 1978.

[Enz+10] Markus Enzenberger, Martin Müller, Broderick Arneson, and Richard Segal. "Fuego–An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 259–270.

[ET20] Attila Egri-Nagy and Antti Törmänen. "Derived metrics for the game of Go – intrinsic network strength assessment and cheat-detection". In: *2020 Eighth International Symposium on Computing and Networking (CANDAR)*. 2020, pp. 9–18. DOI: 10.1109/CANDAR51075.2020.00010.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[GEA11] Amr S. Ghoneim, Daryl L. Essam, and Hussein A. Abbass. "Competency Awareness in Strategic Decision Making". In: *2011 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA)*. IEEE, 2011, pp. 106–109. DOI: 10.1109/COGSIMA.2011.5753426.

[Gli01] Mark E. Glickman. "Dynamic paired comparison models with stochastic variances". In: *Applied Statistics* 28.6 (2001), pp. 673–689. DOI: 10.1080/02664760120059219.

[Gli16] Mark E. Glickman. *The Glicko system*. 2016. URL: http://glicko.net/glicko/glicko.pdf (visited on 10/01/2024).

[Gli22] Mark E. Glickman. *Example of the Glicko-2 system*. 2022. URL: http://www.glicko.net/glicko/glicko2.pdf (visited on 10/01/2024).

[Gli99] Mark E. Glickman. "Parameter estimation in large dynamic paired comparison experiments". In: *Applied Statistics* 48.3 (1999), pp. 377–394.

[GZL23] Yifan Gao, Danni Zhang, and Haoyue Li. "The Professional Go Annotation Dataset". In: *IEEE Transactions on Games* (2023). DOI: 10.1109/TG.2023.3275183.

[He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[HP11] H. Jaap van den Herik and Aske Plaat. "PACHI: State of the Art Open Source Go Program". In: *Advances in Computer Games*. Vol. 7168. Lecture Notes in Computer Science. 2011, pp. 24–38.

[IS15] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. Proceedings of Machine Learning Research. PMLR, 2015, pp. 448–456.

[Kai+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Identity Mappings in Deep Residual Networks”. In: *Computer Vision – ECCV 2016*. Vol. 9908. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 630–645. DOI: 10.1007/978-3-319-46493-0_38.

[KB17] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv e-prints* (2017). arXiv: 1412.6980.

[KI18] Yuuto Kosaka and Takeshi Ito. “Examination of Indicators for Estimating Players’ Strength by using Computer Go”. In: *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*. IEEE, 2018, pp. 96–101.

[KS06] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Machine Learning: ECML 2006*. Vol. 4212. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 282–293. DOI: 10.1007/11871842_29.

[Lee+19] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. “Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks”. In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 3744–3753.

[MBN15] Josef Moudřík, Petr Baudiš, and Roman Neruda. “Evaluating Go Game Records for Prediction of Player Attributes”. In: *2015 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2015, pp. 162–168. DOI: 10.1109/CIG.2015.7317909.

[McD+01] Ernest L. McDuffie, Marwan S. Al-Haik, Gunjan Gupta, and Xingming Liu. “On Neural Network Solutions for the Ancient Game of GO”. In: *Journal of Intelligent Systems* 11.3 (2001), pp. 203–215. DOI: 10.1515/JISYS.2001.11.3.203.

[MN16] Josef Moudřík and Roman Neruda. “Determining Player Skill in the Game of Go with Deep Neural Networks”. In: *Theory and Practice of Natural Computing*. Springer International Publishing, 2016, pp. 188–195. DOI: 10.1007/978-3-319-49001-4_15.

[NC] Akita Noek and Contributors. *goratings*. URL: https://github.com/online-go/goratings (visited on 10/01/2024).

[NK] Nihon Kiin and Kansai Kiin. *The Japanese Rules of Go*. Trans. by James Davies. URL: http://www.cs.cmu.edu/%7Ewjh/go/rules/Japanese.html (visited on 10/01/2024).

[Pas] Gian-Carlo Pascutto. *Leela Zero*. URL: https://zero.sjeng.org/ (visited on 10/01/2024).

[RN21] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Fourth Edition. Pearson series in artificial intelligence. Pearson, 2021.

[Sil+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[Sil+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. "Mastering the game of Go without human knowledge". In: *Nature* 550 (2017), pp. 354–359. DOI: 10.1038/nature24270.

[THW22] Dustin Tanksley, Daniel B. Hier, and Donald C. Wunsch II. "Reproducing Neural Network Research Findings via Reverse Engineering: Replication of AlphaGo Zero by Crowdsourced Leela Zero". In: *European Scientific Journal* 18.4 (2022), p. 61. DOI: 10.19044/esj.2022.v18n4p61.

[Van] Zachary Vance. *OGS 2021 collection of Go games*. URL: https://archive.org/details/ogs2021 (visited on 10/01/2024).

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017, pp. 5998–6008.

[Wua] David J. Wu. *New Human-like Play and Analysis*. URL: https://github.com/lightvector/KataGo/releases/tag/v1.15.0 (visited on 10/01/2024).

[Wub] David J. Wu. *Other Methods Implemented in KataGo*. URL: https://github.com/lightvector/KataGo/blob/master/docs/KataGoMethods.md (visited on 10/01/2024).

[Wu20] David J. Wu. "Accelerating Self-Play Learning in Go". In: *AAAI-20 Workshop on Reinforcement Learning in Games*. 2020.

[Yan01] Yilun Yang. *Tricks in Joseki*. Vol. 2. Pocket Skills. Yutopian, 2001.