Factorization Ranking Model for Fast Move Prediction in the Game of Go

by

Chenjun Xiao

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

In this thesis, we investigate the move prediction problem in the game of Go by proposing a new ranking model named Factorization Bradley Terry (FBT) model. This new model considers the move prediction problem as group competitions while also taking the interaction between features into account. A FBT model is able to provide a probability distribution that expresses a preference over moves. Therefore it can be easily compiled into an evaluation function and applied in a modern Go program. We propose a Stochastic Gradient Decent (SGD) algorithm to train a FBT model using expert game records, and provide two methods for fast computation of the gradient in order to speed up the training process. We also investigate the problem of integrating feature knowledge learned by the FBT model in Monte Carlo Tree Search (MCTS). Experimental results show that our FBT model outperforms the state-of-the-art fast move prediction system of Latent Factor Ranking, and it is useful in improving the performance of MCTS.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Prof. Martin Müller. He always provides me freedom to explore what I found interesting, and his valuable guidance and support helped me in all the time of research and writing of this thesis.

I gratefully acknowledge University of Alberta for financially supporting me. I would also like to thank Kenny Young, who helped me to present our work at the Computer Games Workshop at IJCAI 2016.

Finally, I want to thank my family. It is impossible to get through all these challenges and hard time without your supports. Thank you so much for your unconditional love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Games and Artificial Intelligence

Games, as a simplification of real-world decision making problems, provide an ideal environment and a big challenge for Artificial Intelligence (AI) research. Games research has pioneered many fields of AI research, such as Alpha-Beta search [22], Monte Carlo Tree Search [10] and Deep Reinforcement Learning [25, 33]. All of these algorithms were originally designed to play complex games, and have found applications in other areas [8, 21]. Computer programs using these techniques are capable of playing at the same level as, or even defeating the strongest human players in the world in many popular games:

- The **chess** program *Deep Blue*, developed by IBM used Alpha-Beta search implemented in a game-specific hardware running on a specific-purpose machine. It defeated the World Chess Champion Garry Kasparov by a score of two wins, one loss, and three draws in 1997 [32]. Current chess programs far surpass the best human players.

- The **checkers** program *Chinook* developed by Jonathan Schaeffer and his colleagues first won matches against top humans. Later, they solved the game [31, 30]. The program used parallel search and a large endgame database.

- The **poker** program *Cepheus* developed by Michael Bowling and his colleagues is capable of playing a nearly perfect game of Heads-up Limit Hold'em poker [7].

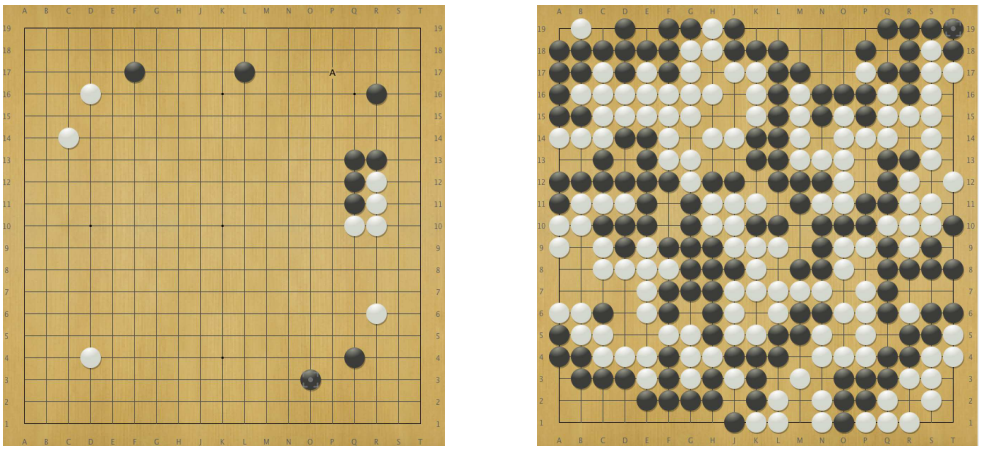


Figure 1.1: An example of a Go game: Opening (left) and end of the game (right).

- The **Atari 2600**[1] games playing program developed by Google Deepmind with a deep convolutional network trained by variants of Q-learning from self-played games achieved human-level performance in many games [25].

- The **Go** program *AlphaGo* developed by Google Deepmind recently won 4-1 against Lee Sedol, one of the best Go players in the world. The program applies several deep convolutional networks, trained by supervised learning and reinforcement learning, in a Monte Carlo Tree Search framework [33].

1.2 The Game of Go

Go is an ancient Chinese board game. The game is played by two players Black and White, who take turns to place a single stone of their own color on an empty intersection on a Go board. The standard board sizes are usually 9×9, 13×13 and 19×19. Stones can only be removed if they become surrounded by opponent stones. The goal of the game is to control a larger total area of the board with their stones than the opponent by the end of the game. Figure 1.1 shows an example of a typical Go game. The left picture shows the opening phase of the game, while the right picture shows the end of the game. Black wins the game at the end by 3.5 points using Chinese rules.

Although its rules are quite simple, Go is a very complex and deeply strategic

[1] Atari 2600 was a popular second generation game console originally released in 1977.

game. For many years, it has been considered as a grand challenge for Artificial Intelligence. In general, the difficulty of conquering computer Go comes from two aspects. First, the game of Go has a tremendously large search space. For example, 19×19 has up to 361 legal moves and more than 10^{170} game states [16]. This is many orders of magnitude too large for search algorithms that have been proven to be successful in games such as chess and checkers. Second, before *AlphaGo* [33] it has been considered very difficult to construct a suitable evaluation function [26] for Go positions.

1.3 Move Prediction in the Game of Go

Human experts rely heavily on their Go knowledge when playing the game. This knowledge is accumulated through a lifetime of playing and investigating game records. It can help human experts to recognize promising moves and important regions of a Go board without simulating many thousands of continuations like modern Go programs. In computer Go research, knowledge is usually represented by shape patterns and tactical features. A *move prediction system* is capable of acquiring feature knowledge from human game records or self-played games using machine learning techniques in order to mimic how experts play the game. Such a system can directly serve as a Go playing program, but more importantly, it can play the role of a selective move generator to guide the search tree by estimating how likely each candidate move is to be played by an expert. Selective search algorithms, such as Monte Carlo Tree Search (MCTS) [10], can then focus on the most promising moves. *Move prediction systems* play a very important part in state-of-the-art computer Go programs [8].

1.4 Contributions of this Thesis

This thesis focuses on investigating the problem of move prediction in the game of Go. In particular, we are interested in building a *fast move prediction system*, which is expected to predict expert moves as accurately as possible without losing too much computational efficiency. As in most popular *fast move predic-*

tion systems, we consider each move as a group of features, and learn each feature's weights to predict expert moves using a supervised learning method. In order to do this, we propose and evaluate a new ranking model called *Factorization Bradley-Terry* (FBT) model. The major innovation of this new model is to consider the interaction between individual features within the same group as part of a probability-based framework. This combines the strengths of two leading approaches: The probability-based Minorization-Maximization technique [11] and Latent Factor Ranking [38], a model which considers feature interactions.

This thesis is organized as follows: In Chapter 2, we review several techniques related to this thesis. In Chapter 3, we give the definition of the FBT model as well as a stochastic Gradient Descent (SGD) based algorithm to train it using professional game records. Two techniques are provided in order to accelerate the training process: an efficient incremental implementation of the gradient update, and an unbiased approximate gradient estimator. We also show how to integrate FBT knowledge in the MCTS algorithm. In Chapter 4, we vary the size of data sets as well as the expressiveness of the feature sets, in order to test the performance of the FBT model. Experimental results show that FBT achieves the state-of-the-art move prediction accuracy among all fast move prediction algorithms. We also show that the feature knowledge learned by the FBT model is useful in improving the strength of Monte Carlo Tree Search in the Go program Fuego [14]. The main results of this thesis have been published in the following publications [39, 40]:

- *Chenjun Xiao and Martin Müller. Factorization Ranking Model for Move Prediction in the Game of Go. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-2016), 1359-1365, 2016.*

- *Chenjun Xiao and Martin Müller. Integrating factorization ranked features into MCTS: an experimental study. Computer Games Workshop at IJCAI, 2016.*

Chapter 2

Literature Review

2.1 Supervised Learning

The objective of building a move prediction system is learning to predict expert moves from professional game records. This is a standard supervised learning task [37]. In supervised learning, the *training data* consists of a set of *training examples*, which are *pairs* (x, y) consisting of some input data x and its desired output label y. A supervised learning algorithm analyzes the training data and learns a prediction function that predicts the label of new examples.

Most supervised learning algorithms aim to minimize a *loss function* L over the training set. Suppose there is a labeled training set of N examples $D = \{(x_1, y_1), \ldots, (x_N, y_N)\}$. The input data x is usually represented by a feature vector. A supervised learning algorithm learns a function $f_\theta : X \to Y$ which predicts the label of given input. The prediction function f_θ is parameterized by a weight vector θ. For example, in the linear case, θ corresponds to the weights of the features. In the case where f_θ is approximated by a neural network, θ corresponds to the weight parameter of the neural network. A *loss function* $l : Y \times Y \to \mathbb{R}^{\geq 0}$ for one training example is defined in order to measure how well f fits the training data. The *loss function* over all training data is defined by

$$L(D, \theta) = \sum_{i=1}^{N} l(f_\theta(x_i), y_i) \tag{2.1}$$

The prediction function is learned by approximately solving an optimization prob-

Algorithm 1 Stochastic Gradient Descent

Input Initial parameters θ, learning rate α, training data D

while convergence condition not satisfied **do**
    Randomly shuffle the training data D
    for $i = 1, 2, \ldots, N$ **do**
        $\theta = \theta - \alpha \nabla l(f_\theta(x_i), y_i)$
    end for
end while

lem

$$\theta^* = \text{argmin}_\theta \, L(D, \theta) \tag{2.2}$$

Typical choices of the *loss function* l include l_1 *loss*: $|f_\theta(x) - y|$, l_2 *loss*: $\frac{1}{2}(f_\theta(x) - y))^2$ and *negative log loss*: $-\ln P(y|f_\theta(x))$. In particular, the *negative log loss* is designed for learning a probability-based model. Minimizing this loss corresponds to the maximum likelihood estimation in statistics [4].

One common problem in supervised learning is overfitting, which means that the learned model has poor performance for predicting new examples, since it overreacts to minor variations in the training data [4]. To avoid it, we usually solve the following optimization problem instead of (2.2)

$$\theta^* = \text{argmin}_\theta \, L(D, \theta) + \lambda R(\theta) \tag{2.3}$$

Here, R is a regularization function, which is typically a penalty on the complexity of the learned model, and λ is a constant controlling the scale of the regularization [4].

2.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a stochastic approximation of the gradient descent algorithm [5]. It was developed in order to solve large optimization problems. We use the optimization problem (2.2) to explain the idea of SGD. A standard gradient descent algorithm performs the following updates iteratively until a convergence condition is satisfied:

$$\theta \leftarrow \theta - \alpha \nabla L(D, \theta) = \theta - \alpha \sum_{i=1}^{N} \nabla l(f_\theta(x_i), y_i) \tag{2.4}$$

This method is very straightforward and easy to implement in practice. However, if the training set is very large and if no simple formula exists to compute the gradient $\nabla l(f_\theta(x_i), y_i)$, then evaluating the exact gradient $\nabla L(D, \theta)$ might require expensive computations. SGD deals with this issue by sampling a single example (x_i, y_i) from the data set, and performs the gradient update for that sample only:

$$\theta = \theta - \alpha \nabla l(f_\theta(x_i), y_i) \tag{2.5}$$

It is well known that the sampled gradient matches the exact gradient in expectation [5]. Although the theory demands selecting examples randomly, in practice it is usually faster to first randomly shuffle the data set, then go through it sequentially [6]. Pseudocode of SGD is provided in Algorithm 1.

2.3 Factorization Machine

There are two main ideas behind the model of Factorization Machine (FM) [27]: taking the interaction between features into account, and applying a factorization method to model the interaction. Including interactions between features in the model has the advantage that more information is taken into account. The model equation for a FM is defined as:

$$f_{w,v}(x) := w_0 + \sum_{i=1}^{m} w_i x_i + \sum_{i=1}^{m} \sum_{j=i+1}^{m} \langle v_i, v_j \rangle x_i x_j \tag{2.6}$$

where w_0 is the global bias, w_i is the strength of x_i and the dot product $\langle v_i, v_j \rangle$ models the interaction between x_i and x_j.

It has been shown that the interactions are very important in the move prediction problem [38]. The most straightforward way to model the interaction is to construct a matrix, where each element indicates the interaction between two features. However, this simple model is extremely hard to learn when dealing with the prediction problem under a sparse data set. Data is said to be sparse when almost all elements of the feature vector are zero, and a sparse data set contains all sparse data. It is not

hard to see that if two features rarely appear together in the data set, it is impossible to estimate their interaction. FM adopts a very smart way to deal with this issue: as shown in model (2.6), every feature in the model has an *interaction vector* whose dimension is a pre-defined parameter of the model, and the interaction between two features is the dot product of their *interaction vectors*. Thereby, the interactions between a feature with others can be estimated as long as this combination exists in the training data set.

2.4 Monte Carlo Tree Search

The basic idea of Monte Carlo Tree Search (MCTS) [10] is to evaluate a state by constructing a search tree $\mathcal{T}$ of game states evaluated by fast Monte Carlo simulations. Each node $n(s)$ of $\mathcal{T}$ corresponds to a game state s, and contains statistical information about s: a total visit count for the state $N(s)$, a value $Q(s,a)$ and the visit count $N(s,a)$ for each action available in s. Simulations start from the initial state s_0, and are divided into two stages: in-tree and rollout. When a state s_t is already represented in the current search tree, $s_t \in \mathcal{T}$, MCTS uses a *tree policy* to select an action to go to the next state. Otherwise, for a state out of the tree , $s_t \notin \mathcal{T}$, a *roll-out policy* is used to roll out simulations and eventually obtain a reward signal. At the end of a simulation, after a state action trajectory $s_0, a_0, s_1, a_1, \ldots, s_T$ with reward R is obtained, each node of $\{n(s_t)|s_t \in \mathcal{T}\}$ is updated according to

$$N(s_t) \leftarrow N(s_t) + 1 \tag{2.7}$$

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1 \tag{2.8}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{R - Q(s_t, a_t)}{N(s_t, a_t)} \tag{2.9}$$

In addition the search tree is grown. In the simplest scheme, the first visited node that is not yet in tree $\mathcal{T}$ is added to it.

The UCT algorithm [23] is the most well-known variant of MCTS. It uses the UCB1 [2] algorithm as the *tree policy* to select actions by treating each state of the search tree as a multi-armed bandit. The action value is augmented with an

exploration bonus, and the *tree policy* selects the action maximizing the augmented value

$$a^* = \text{argmax}_a Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \tag{2.10}$$

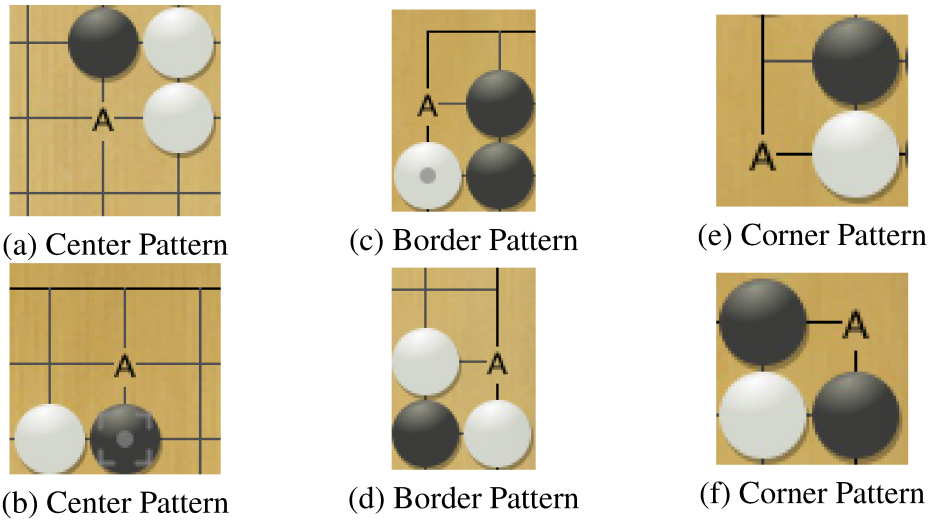
where c is a constant controlling the scale of the *exploration*.

2.5 Improving MCTS with Domain Knowledge

The quality of MCTS relies heavily on the performance of the Monte Carlo simulations. However in games with a large state space, simple simulation is unlikely to lead to accurate value estimation. Inaccurate estimation can mislead the search and can severely limit the strength of the program. Domain knowledge of the game can serve as heuristic information that benefits the search, and can significantly improve the performance of MCTS. Coulom [11] proposes Minorization-Maximization (MM) to learn feature knowledge offline and uses it to improve random simulation. Gelly and Silver [17] consider feature knowledge as a prior for evaluation when a new state is added to the search tree. AlphaGo [33] incorporates a supervised learned Deep Convolutional Neural Network (DCNN) as part of its in-tree policy for move selection, and further improves the network with reinforcement learning from games of self-play to get a powerful value estimation function. The integrated system became the first program to beat the world's top Go player [1].

2.6 Representing Moves with Features

In supervised learning for move prediction, the first step is to find a representation of each move as the input of the learning system. The common method is to represent each state-move pair as a set of active features. This representation has two advantages. First, it is well suited for generalization. The tabular case in reinforcement learning, directly using the state-move pair as input data of learning, would fail, since the training set cannot include all possible game states and might contain conflicting move selections in the same state. With such a representation, it is



(a) Center Pattern (c) Border Pattern (e) Corner Pattern

(b) Center Pattern (d) Border Pattern (f) Corner Pattern

Figure 2.1: Small 3×3 Pattern Examples: (a) and (b) 3×3 pattern in the center. (c) and (d) 3×3 pattern in the border. (e) and (f) 3×3 pattern in the corner.

almost impossible to learn useful generalized knowledge, for predicting moves for states that are not included in the training data. With a feature representation, the same feature can occur in different moves (included or not included in the training set), which both makes the learning process more efficient and able to generalize. A second advantage of a feature representation is that it is easy to hand-code domain knowledge in the feature set. For example, when building a move prediction system for Go, tactical features such as capture, extension and liberties are often used [11, 38, 37, 9, 24].

Popular features used in fast move prediction include simple features and shape patterns. Simple features include the location of the move, the distance to previous moves, and basic tactics such as capture and liberties. Two kinds of shape patterns are most often used: small patterns and large patterns. Small pattern features usually represent 3×3 or even smaller local shapes around a move. They are easy to compute but can only provide local information. In contrast, large patterns can cover a much wider region of the board. They are slower to compute but can provide more context. Therefore, small patterns are often used in a fast rollout policy, while large patterns are used as part of in-tree move selection knowledge, which is not so time-critical. Figure 2.1 shows some examples of small patterns. Figure 2.2 shows the large patterns template used in most move prediction systems [35, 11, 38, 37].

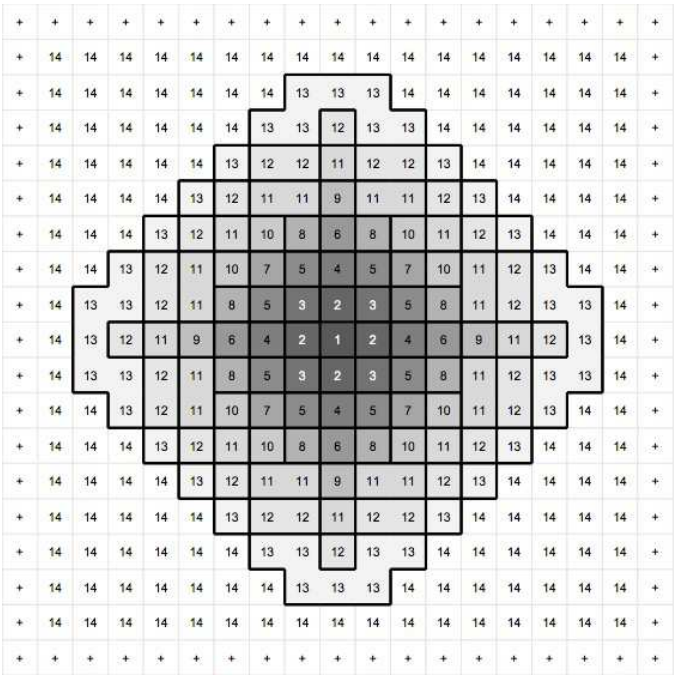


Figure 2.2: Large Pattern Template [35].

If a move has a particular feature, we say that feature is active in that move. With feature representation, each move is represented by a group of active features. Figure 2.3 shows an example. It shows the active features of move R12, including the location of the move, distance to previous move, game phase, CFG distance and matched shape pattern. Note that both 3×3 patterns and large patterns are rotation invariant.

A deep learning model is capable of learning high level representations automatically from raw input data [3]. This important property has made deep learning the state-of-the-art for many machine learning tasks, including move prediction in Go [9, 24]. However, a well-performing deep neural network is too slow to be used for fast move prediction. The features discussed above are still most popular here.

2.7 State-of-the-Art Fast Move Prediction

The most popular high-speed move prediction systems represent each move as a combination of a group of features. They learn weights for each feature from expert game records by supervised learning, and define an evaluation function based

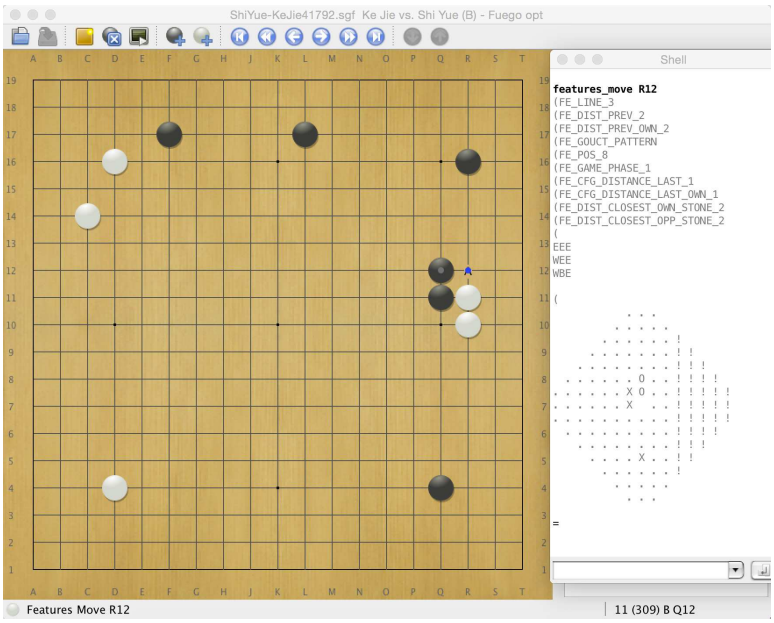


Figure 2.3: An example of representing a move by a group of features.

on these weights, which is used to rank moves. Examples include Bayesian Full Ranking [35], Bayesian Approximation for Online Ranking [37, 36], Minorization-Maximization (MM) [11] and Latent Factor Ranking (LFR) [38].

MM is an efficient learning algorithm which is very popular in Computer Go. In this model, move prediction is formulated as a competition among all possible moves in a Go position. Each move is represented as a group of features. A simple probabilistic model named the Generalized Bradley-Terry model [20] defines the probability of each feature group winning a competition. The strength of each feature is estimated by maximizing the likelihood of the expert moves winning all competitions over the whole (large) training set.

LFR improves the performance of move prediction by taking pairwise interactions between features into account. This is modelled using a Factorization Machine (FM) [27], an efficient model especially for problems with sparse interactions between features. Move prediction is considered as a binary classification problem, with the expert move in one class and all other legal moves in the other. A drawback of LFR is that the evaluation it produces does not provide a probability distribution over all possible moves. This makes it harder to combine it with other kinds of evaluation.

The successes of both LFR and MM highlight two different aspects: the interactions between features within a group, and an efficient ranking model to describe the interactions between all possible groups. The Factorization Bradley Terry model combines these aspects: like MM, move prediction is modeled as a competition among all moves using a Bradley-Terry model. As in LFR, a Factorization Machine models the interaction between the features of a move.

Recently, Deep Convolutional Neural Networks (DCNN) have achieved human-level performance and have outperformed traditional techniques for predicting expert moves [9, 24]. However, well-performing networks are very large, with millions of weights [24], and even on specialized hardware their evaluation speed is orders of magnitude slower than traditional techniques. Therefore, it makes sense to continue research on high performance fast move prediction algorithms as well. For example, in AlphaGo they are used as a temporary evaluation in a game tree, to give the search a good direction even before the slow neural network evaluation is available [33].

Chapter 3

The Factorization Bradley-Terry Model for Fast Move Prediction in the Game of Go

In this chapter, we first introduce the Factorization Bradley-Terry Model, then show how this model can be applied to the move prediction problem. We also provide an algorithm for parameter learning based on Stochastic Gradient Descent and two approaches to accelerate the training process.

3.1 The Factorization Bradley-Terry Model

The Factorization Bradley-Terry (FBT) model is introduced for estimating the probability of a feature group winning competitions with other groups, while taking into account the pairwise interactions between features within each group using a Factorization Machine (see Section 2.3). Let $k \in \mathbb{N}^+$ be the dimension of the factorization model. Let $\mathcal{F}$ be the set of all possible features. For feature $f \in \mathcal{F}$, let $w_f \in \mathbb{R}$ be the feature's (estimated) *strength*, and $v_f \in \mathbb{R}^k$ be its *factorized interaction vector*. Here, the *interaction strength* between two features f and g is modeled as $\langle v_f, v_g \rangle = \sum_{i=1}^{k} v_{f,i} \cdot v_{g,i}$. The parameter space of this model is $\mathbf{w} \in \mathbb{R}^{|\mathcal{F}|}$ and $\mathbf{v} \in \mathbb{R}^{|\mathcal{F}| \times k}$.

The idea of using factorized interaction vectors comes from Factorization Machines [27]. The matrix $\mathbf{v} \in \mathbb{R}^{|\mathcal{F}| \times k}$ is a sparse approximation of the full pairwise interaction matrix $\Phi \in \mathbb{R}^{|\mathcal{F}| \times |\mathcal{F}|}$, which is huge for large $|\mathcal{F}|$. A proper choice of

k makes such models especially efficient for generalization under sparse data sets (see Section 2.3) [27]. In Computer Go, settings $k = 5$ and $k = 10$ are popular [38]. Richer feature sets and larger training sets require increasing k for best performance, as will be shown in Table 4.1 in the Experiments Chapter.

The strength $E_{\mathcal{G}}$ of a group $\mathcal{G} \subseteq \mathcal{F}$ in FBT is defined as

$$E_{\mathcal{G}} = \sum_{f \in \mathcal{G}} w_f + \frac{1}{2} \sum_{f \in \mathcal{G}} \sum_{g \in \mathcal{G}, g \neq f} \langle v_f, v_g \rangle \tag{3.1}$$

In FBT, the popular exponential model [19] is used to model winning probabilities in competition among groups. Given N groups $\{\mathcal{G}^1, \ldots, \mathcal{G}^N\}$, the probability that group $\mathcal{G}^i$ wins is given by:

$$P(\mathcal{G}^i \text{ wins}) = \frac{exp(E_{\mathcal{G}^i})}{\sum_{j=1}^{N} exp(E_{\mathcal{G}^j})} \tag{3.2}$$

3.2 Move Prediction in Go using the Proposed Model

Let $\mathcal{S}$ be the set of possible Go positions and $\Gamma(s)$ be the set of legal moves in a specific position $s \in \mathcal{S}$. The objective of the *move prediction problem* is to learn from a training set to predict expert moves. *Features* $\mathcal{F}$ are used to describe moves in a given game state. Each move is represented by its set of active features $\mathcal{G} \subseteq \mathcal{F}$. The training set $\mathcal{D}$ consists of cases $\mathcal{D}_j$, with each case representing the possible move choices in one game position s_j.

$$\mathcal{D}_j = \{\ \mathcal{G}_j^i \mid \text{for } i = 1, \ldots, |\Gamma(s_j)|\}$$

As in MM, the process of choosing a move is modeled as a competition, using the ranking model defined before. In this setting, the set of features $\mathcal{F}$ is the set of "players" which compete in groups. Let the winner of the competition among groups in $\mathcal{D}_j$ be $\mathcal{G}_j^*$. From (3.2), the probability of the current test case is

$$P(\mathcal{D}_j) = \frac{exp(E_{\mathcal{G}_j^*})}{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})} \tag{3.3}$$

3.3 Parameter Estimation for FBT

In the proposed FBT model (3.3), each feature's *strength* and *factorized interaction vector* is learned from a training set $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$. This section first poses the learning of these parameters as an optimization problem, then describes a Stochastic Gradient Descent (SGD) algorithm for estimating them, and finally provides two methods for accelerating the learning process.

3.3.1 Definition of the Optimization Problem

Suitable parameters in FBT can be estimated by maximizing the likelihood of the training data. For the jth training case D_j, the negative log loss function is given by

$$l_j = -\ln \frac{exp(E_{\mathcal{G}_j^*})}{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})} = -E_{\mathcal{G}_j^*} + \ln \sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i}) \tag{3.4}$$

Assuming that competitions are independent, the loss function becomes simply the sum over all training examples:

$$L = \frac{1}{n} \sum_{j=1}^{n} l_j \tag{3.5}$$

The corresponding optimization problem is:

$$\min_{\mathbf{w} \in \mathbb{R}^{|\mathcal{F}|}, \mathbf{v} \in \mathbb{R}^{|\mathcal{F}| \times k}} L + \lambda_w R(\mathbf{w}) + \lambda_v R(\mathbf{v}) \tag{3.6}$$

We use L_2 regularization $\frac{1}{2}||\cdot||^2$ as our regularization function R to avoid overfitting. λ_w and λ_v are parameters. The choice of setting their values is discussed in the Experiments Chapter.

3.3.2 Parameter Learning with Stochastic Gradient Descent

We propose a SGD algorithm to learn the model parameters. Equation (3.1) can be expressed as a linear function with respect to every single model parameter $\theta \in \mathbf{w} \cup \mathbf{v}$ [28],

$$E_{\mathcal{G}} = \theta h_{\mathcal{G},\theta} + g_{\mathcal{G},\theta} \tag{3.7}$$

where if $\theta = w_s \in \mathbf{w}$, $h_{\mathcal{G},\theta} = 1$; if $\theta = v_{s,q} \in \mathbf{v}$, $h_{\mathcal{G},\theta} = \frac{1}{2}\sum_{t\in\mathcal{G},t\neq s} v_{t,q}$. In both cases, $g_{\mathcal{G},\theta} = E_{\mathcal{G}} - \theta h_{\mathcal{G},\theta}$. Note that $g_{\mathcal{G},\theta}$ are independent of the value of θ, and if $s \notin \mathcal{G}$, both $h_{\mathcal{G},\theta}$ and $g_{\mathcal{G},\theta}$ are simply zero. Using these facts, the gradient of the loss function l_j for parameter θ can be written as:

$$\begin{aligned}
\nabla_j\theta =& -\frac{\partial E_{\mathcal{G}_j^*}}{\partial\theta} + \frac{\partial \ln \sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})}{\partial\theta} \\
=& -h_{\mathcal{G}_j^*,\theta} + \frac{\sum_{i=1}^{|\Gamma(s_j)|} \partial exp(E_{\mathcal{G}_j^i})/\partial\theta}{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})} \\
=& -h_{\mathcal{G}_j^*,\theta} + \frac{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i}) h_{\mathcal{G}_j^i,\theta}}{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})}
\end{aligned} \tag{3.8}$$

The resulting gradients $\nabla_j\theta$ for updating a parameter θ are:

$$\begin{cases}
-1 + \frac{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})\mathbb{I}\{s\in\mathcal{G}_j^i\}}{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})} & \theta = w_s, s \in \mathcal{G}_j^* \\
-H_{\mathcal{G}_j^*,q} + \frac{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})\mathbb{I}\{s\in\mathcal{G}_j^i\}H_{\mathcal{G}_j^i,q}}{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})} & \theta = v_{s,q}, s \in \mathcal{G}_j^* \\
\frac{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})\mathbb{I}\{s\in\mathcal{G}_j^i\}}{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})} & \theta = w_s, s \notin \mathcal{G}_j^i \\
\frac{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})\mathbb{I}\{s\in\mathcal{G}_j^*\}H_{\mathcal{G}_j^i,q}}{\sum_{i=1}^{|\Gamma(s_j)|} exp(E_{\mathcal{G}_j^i})} & \theta = v_{s,q}, s \notin \mathcal{G}_j^*
\end{cases} \tag{3.9}$$

with indicator function $\mathbb{I}$ and $H_{\mathcal{G},q} = \frac{1}{2}\sum_{t\in\mathcal{G},t\neq s} v_{t,q}$.

3.3.3 Choice of Approach for Optimization

The new model (3.3) can be considered an extension of the generalized Bradley-Terry model [20], which applies a Factorization Machine to model the interaction between team members when computing a team's ability. It is very similar to the

conditional exponential model [12], which is widely used in computational linguistics. From this standpoint, it seems like optimization algorithms for solving these models such as Alternating Optimization (AO) [28], Maximization-Minimization (MM) [20] [11] and Improved Iterative Scaling (IIS) [12] might also be suitable for solving (3.6). Generally speaking, all of these methods try to solve the optimization problem iteratively. At each iteration, to update a parameter θ, they first construct a sub-optimization problem according to properties of the function to be optimized, such as Jensen's inequality or an lower bound for the logarithm function. Next, they find the optimizer as the starting point of the next iteration by fixing all the other parameters except θ and setting the derivative of θ to zero. In (3.6), the derivative of a group $\mathcal{G}_j^i$ for θ is

$$\frac{\partial exp(E_{\mathcal{G}_j^i})}{\partial \theta} = exp(\theta h_{\mathcal{G}_j^i,\theta}) exp(g_{\mathcal{G}_j^i}) h_{\mathcal{G}_j^i,\theta}$$

If $\theta \in \mathbf{v}$, the values $h_{\mathcal{G}_j^i,\theta}$ will change with different i and j, and thus the update of $\theta \in \mathbf{v}$ at each iteration does not have a closed form solution. A numerical method such as Newton's method is required to compute the update value for θ, which would introduce further computational complexity. In conclusion, it would be very inefficient to use such algorithms for problem (3.6).

3.3.4 Efficient Gradient Computation

The first bottleneck of computing the gradient of the loss function l_j via (3.8) is calculating $exp(E_{\mathcal{G}_j^i})$ for each $i \in \{1, \ldots, |\Gamma(s_j)|\}$. For one particular group $\mathcal{G}$, a direct computation of $exp(E_{\mathcal{G}})$ is $O(|\mathcal{G}|k)$. However, we can compute this term once and update it in constant time when parameters corresponding to some feature $s \in \mathcal{G}$ have changed. Suppose that we update the parameter θ to θ'. Then for a group $\mathcal{G}$, $E_{\mathcal{G}} = \theta h_{\mathcal{G},\theta} + g_{\mathcal{G},\theta}$ should be updated to

$$E'_{\mathcal{G}} = \theta' h_{\mathcal{G},\theta} + g_{\mathcal{G},\theta} = E_{\mathcal{G}} + (\theta' - \theta) h_{\mathcal{G},\theta} \tag{3.10}$$

Therefore, at the beginning of training we precompute all $E_{\mathcal{G}}$ in the training set once, then update them efficiently through equation (3.10) when necessary, which only takes constant time.

After this optimization, the computational complexity of updating $E_{\mathcal{G}}$ depends on the complexity of calculating $h_{\mathcal{G},\theta}$. For $\mathbf{w}$, the h term in (3.10) is just 1, so the update time is constant. For updating $\theta = v_{s,q} \in \mathbf{v}$, the direct computation of the h term in (3.10) takes $O(|\mathcal{G}|)$ time. However, it can be updated in constant time by first precomputing $R_{\mathcal{G},q} = \frac{1}{2}\sum_{t\in\mathcal{G}} v_{t,q}$ for each group in the training set, since

$$h_{\mathcal{G}}(v_{s,q}) = R_{\mathcal{G},q} - \frac{1}{2}v_{s,q} \tag{3.11}$$

After an update of $v_{s,q}$ to $v'_{s,q}$, $R_{\mathcal{G},q}$ can be updated in constant time by

$$R'_{\mathcal{G},q} = R_{\mathcal{G},q} + \frac{1}{2}(v'_{s,q} - v_{s,q}) \tag{3.12}$$

Using equations (3.10) to (3.12), $exp(E_{\mathcal{G}})$ can be updated in constant time.

Regarding the space complexity of the fast gradient computation method described above, let $\gamma = \max_{j\in\{1,\ldots,n\}}|\mathcal{D}_j|$. For each group $\mathcal{G}$ in $\mathcal{D}_j$, storing one $E_{\mathcal{G}}$ and one $R_{\mathcal{G},q}$ for each factorization dimension $1 \le q \le k$, gives total space complexity $O(n\gamma k)$. For example, for 19×19 Go, $\gamma \le 361$, so the total space complexity is $O(nk)$.

3.3.5 Approximate Gradient

Another bottleneck of computing the gradient via (3.8) is processing the set of all groups. For example, in the *move prediction problem* in 19×19 Go, the average value of $|\Gamma(s_j)|$ is about 200. A Monte Carlo approach can address this problem by sampling an approximate gradient that matches the real gradient in expectation.

Let P_j^i be the probability that $\mathcal{G}_j^i$ is the winner of $\mathcal{D}_j$ using model (3.2). The probability distribution $P_j = (P_j^1, \ldots, P_j^{|\Gamma(s_j)|})$ over $\mathcal{D}_j$ allows us to construct an unbiased approximate gradient. Consider a mini-batch of groups created by sampling M groups $\{\mathcal{G}^{(1)}, \ldots, \mathcal{G}^{(M)}\}$ from $\mathcal{D}_j$ according to the probability distribution P_j. The **sampled approximate gradient** of the loss function l_j for parameter θ is:

$$\widehat{\nabla_j\theta} = -h_{\mathcal{G}_j^*}(\theta) + \frac{1}{M}\sum_{i=1}^{M} h_{\mathcal{G}_j^{(i)},\theta} \tag{3.13}$$

We now show that the **sampled approximate gradient** matches the gradient (3.8) in expectation.

Lemma 1. $E_{P_j}[\widehat{\nabla_j \theta}] = \nabla_j \theta$

Proof.

$$\begin{aligned} E_{P_j}[\widehat{\nabla_j \theta}] &= -h_{\mathcal{G}_j^*,\theta} + E_{P_j}[\frac{1}{M}\sum_{i=1}^{M} h_{\mathcal{G}_j^{(i)},\theta}] \\ &= -h_{\mathcal{G}_j^*,\theta} + \sum_{i=1}^{|\Gamma(s_j)|} P_j^i h_{\mathcal{G}_j^i,\theta} = \nabla_j \theta \end{aligned} \tag{3.14}$$

□

3.4 Integrating FBT Knowledge in MCTS

A move prediction system provides useful initial recommendations to a search about which moves are likely to be the best. Selective search with a proper exploration scheme, such as MCTS, can further improve upon these recommendations. One favourable property of the FBT model is to produce a probability-based evaluation, which estimates how likely each move is going to be selected by a human expert in a given game state. Therefore, FBT knowledge can be used as part of an exploration strategy, to focus exploration on moves which are favoured by human experts.

We apply a variant of the PUCT formula [29] to integrate FBT knowledge in MCTS. This formula is also used in AlphaGo [33]. The idea of PUCT is to explore moves according to a value that is proportional to the predicted probability, but also decays with repeated visits, as in UCT [23]. When a new game state s is added to the search tree, we call a pre-trained FBT model to get a prediction $P_{FBT}(s, a)$, which assigns an exploration bonus $E_{FBT}(s, a)$ for each move $a \in \Gamma(s)$. In order to keep sufficient exploration, we set a lower cut threshold λ_{FBT}, where for all $a \in \Gamma(s)$ if $P_{FBT}(s, a) < \lambda_{FBT}$ then simply let $E_{FBT}(s, a) = \lambda_{FBT}$, otherwise $E_{FBT}(s, a) = P_{FBT}(s, a)$. During in-tree move selection at state s, the algorithm selects move

$$a^* = \text{argmax}_{a \in \Gamma(s)}(Q(s,a) + c_{puct} E_{FBT}(s,a) \sqrt{\frac{\lg(N(s))}{1 + N(s,a)}}) \tag{3.15}$$

where $Q(s,a)$ is the accumulated move value estimated by online simulation, c_{puct} is an exploration constant, $N(s,a)$ is the number of visits of move a in s, and $N(s) = \sum_{a \in \Gamma(s)} N(s,a)$.

Chapter 4

Experiments

The experiments in this section consist of two parts. In the first part, we evaluate the FBT model and the approximate gradient computation for the move prediction task in the game of Go. FBT is compared with the state-of-the-art move prediction algorithm of Latent Factor Ranking (LFR) [38]. In the second part, we integrate the factorization ranked features in the open source program Fuego [14], in order to test if FBT knowledge is helpful for improving the performance of MCTS. All experiments are performed on a 2.4 GHz Intel Xeon CPU with 64 GB memory and based on the latest Fuego (svn revision 2017).

4.1 Experiments of Move Prediction Performance

4.1.1 Setup

Both FBT and LFR use the same parameters as in [38], with learning rate $\alpha = 0.001$ and regularization parameters $\lambda_w = 0.001$, $\lambda_v = 0.002$. The same stopping criteria for the training process are applied: if an algorithm's prediction accuracy on a validation set is not increased for three iterations, the training is stopped and the best-performing weight set is returned. Three data sets of increasing size, $S_1 \subset S_2 \subset S_3$, are used, which contain 1000, 10000, and 20000 master games respectively. The games are in the public domain at `https://badukmovies.com/pro_games`. S_1 is from year 2013, S_2 is from years 2008-2013 and S_3 is from 1999-2013.

The learned model is evaluated on a test set. The test and validation set con-

tain 1000 games each, and are disjoint from the training sets and from each other. The prediction accuracy in all experiments is defined as the probability of choosing the expert moves over all test cases, not as the average accuracy over the 12 game phases used in [38]. That metric gives higher percentages since it weighs the opening and late endgame more, where the prediction rate is above average.

4.1.2 Choice of Features

When using large patterns, many algorithms can get a high move prediction accuracy at the beginning of the game [38]. This is because these moves are often standard opening moves which can be represented very accurately by large patterns. It is informative to also compare move prediction algorithms without such large patterns. Therefore, two different feature sets, the **small pattern feature set** F_{small} and the **large pattern feature set** F_{large} are used in all tests. Both sets contain the same non-pattern features. F_{small} adds only 3×3 patterns, while F_{large} includes large patterns. F_{small} helps provide a better comparison of FBT and LFR, while F_{large} yields a more powerful move prediction system overall.

The F_{small} features are the same as in the LFR implementation that is part of the open source Fuego program [14]. Large patterns in F_{large} were added for this study. Most features are similar to earlier work such as [11, 38]. For implementation details, see the Fuego code base [13].

The simple features used in this work are:

- **Pass**
- **Capture, Extension**[1]**, Atari**[2]**, Self-atari**[3] Tactical features similar to [11].
- **Line and Position (edge distance perpendicular to Line)** ranges from 1 to 10.
- **Distance to previous move** feature values are 2,..., 16, $\geq$ 17. The distance is measured by $d(\delta x, \delta y) = |\delta x| + |\delta y| + max\{|\delta x|, |\delta y|\}$.

[1] Extend an existing configuration of one or more stones along the side.

[2] A stone or chain of stones has only one liberty, and may be captured on the next move if no more additional liberties.

[3] Adding a stone that can make one's own stones in an Atari.

Table 4.1: Results for F_{small}: probability of predicting the expert move with FBT and LFR, for $k = 5$ and $k = 10$. Best results for each method in bold.

Training set	FBT5	FBT10	LFR5	LFR10
S_1	32.56%	**32.82%**	30.01%	**30.08%**
S_2	33.18%	**33.42%**	30.95%	**31.63%**
S_3	33.46%	**34.01%**	31.13%	**31.94%**

Table 4.2: Results for F_{large}, same format as Table 4.1.

Training set	FBT5	FBT10	LFR5	LFR10
S_1	35.83%	**35.96%**	34.38%	**34.47%**
S_2	38.26%	**38.31%**	37.12%	**37.34%**
S_3	38.48%	**38.75%**	37.56%	**37.69%**

- **Distance to second-last move** uses the same metric as previous move. The distance can be 0.
- **Fuego Playout Policy** These features correspond to the rules in the playout policy used in Fuego. Most are simple tactics related to stones with a low number of liberties.
- **Side Extension** The distance to the closest stones along the sides of the board.
- **Corner Opening Move** Standard opening moves.
- **CFG Distance** Distance when contracting all stones in a block to a single node in a graph [15].
- **Shape Patterns** The *small pattern* set contains all patterns of size 3×3. The *large pattern* set includes circular patterns with sizes from 2 to 14, harvested as in [35, 38]. All shape patterns are invariant to rotation, translation and mirroring.

The system contains 195 non-shape pattern features and 1089 3x3 patterns in total. The total number of large patterns depends on which training set is used, since all large shape patterns are harvested from the training set (as shown in Section 4.1.3). Thereby, we have $|F_{small}| = 1284$ and $|F_{large}| = 195 + N_L$ if N_L large patterns are harvested. For a move, we can define its feature vector with $x \in \{0, 1\}^{|F|}$. The feature group of a move is the set of all features that have the value of one in x.

4.1.3 Expert Move Prediction

To compare the prediction accuracy of FBT and LFR on each data set, two different models were trained for each algorithm, setting the dimension of the *factorized interaction vector* to $k = 5$ and $k = 10$. These methods with a specific k value are called FBTk and LFRk respectively. All results are averaged over five runs, since both FBT and LFR randomize initial parameter values.

Experiment with F_{small}

Results for the small pattern set F_{small} are presented in Table 4.1. Both methods improve with larger training sets. For each combination of training set S_i and k, FBT outperforms LFR. The best model learned by FBT, for $k = 10$ and S_3, outperforms the best LFR model by 2.07%. Note that the gap between the maximum and minimum prediction accuracy of the five runs over all tests of FBT is 0.64%, which shows that FBT is quite stable and the performance difference between LFR and FBT is significant. Prediction accuracy increases with growing k, confirming the observation in [38].

Figures 4.2(a) and (b) compare the details of the move prediction results of LFR and our method. Figure 4.2(a) compares the cumulative probability of predicting the expert's move within the top n ranked moves, for S_3 with $k = 10$. While both methods rank most expert moves within the top 20, the gap between FBT and LFR grows initially up to about rank 5, then holds steady throughout. Figure 4.2(b) presents the prediction accuracy per game stage, where each game phase consists of 30 moves as in [38]. FBT outperforms LFR at every stage of the game.

Experiment with F_{large}

For creating large pattern features, pattern harvesting collects all patterns that occur at least 10 times in the training set. For training sets S_1, S_2 and S_3, the number of such patterns is 9390, 84660 and 152872 respectively. Following [37] and [34], Figure 4.1 shows the distribution of largest matches for the different pattern sizes in each game phase. Large patterns dominate in the opening (phase 1), then disappear rapidly. Later in the game, only small size patterns are matched. Table 4.2 shows

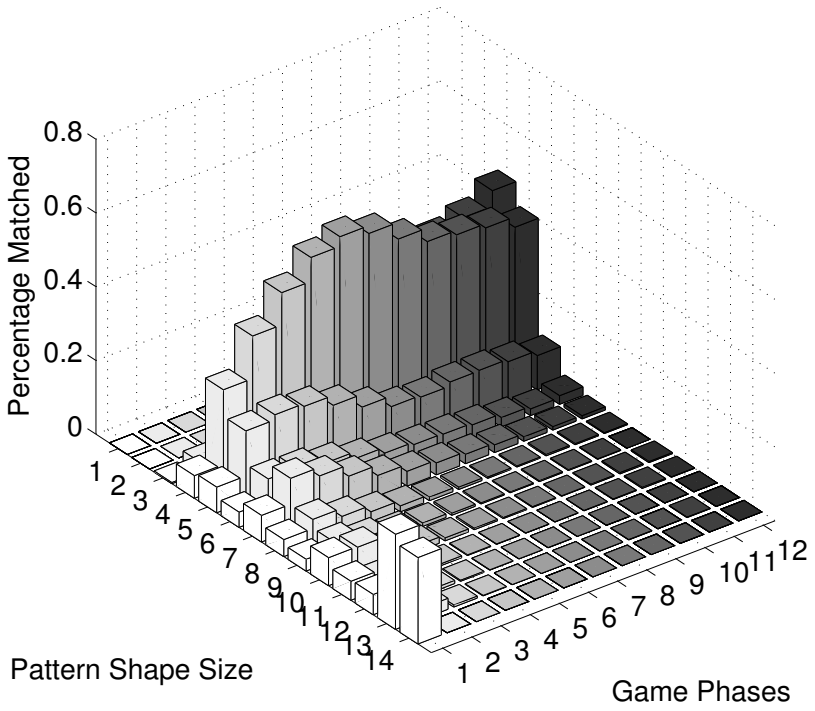
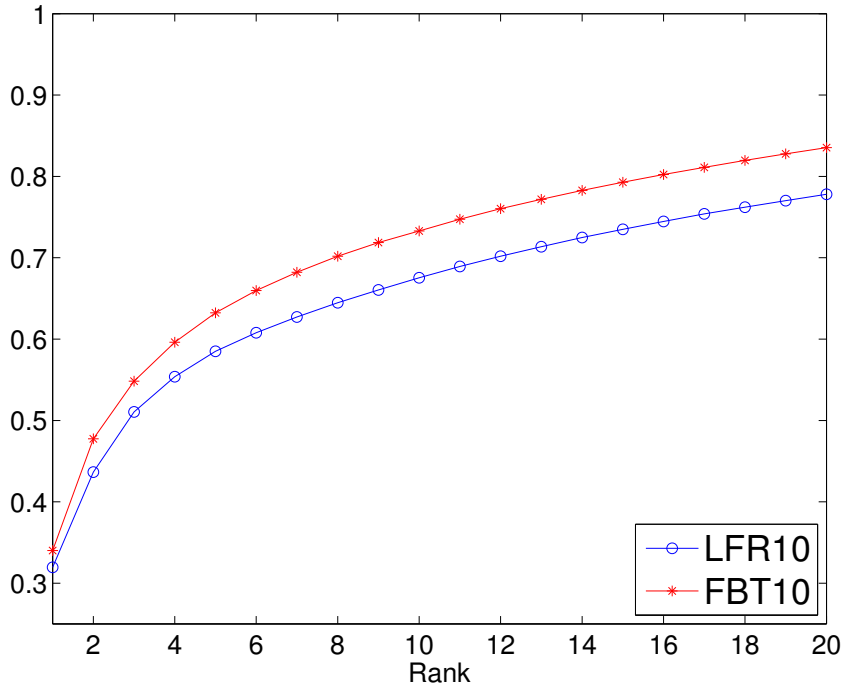


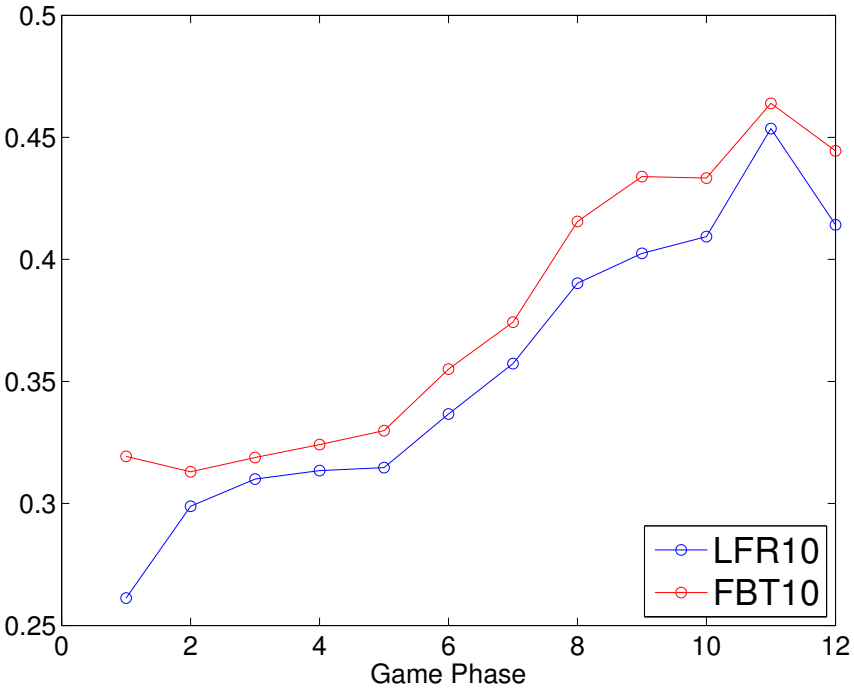
Figure 4.1: Distribution of patterns with different size harvested at least 10 times in different game phases.

the prediction accuracy of FBT and LFR trained with F_{large}. The gap between the maximum and minimum prediction accuracy of the five runs over all tests of FBT is 0.82%. Figures 4.3(a) and (b) show the cumulative prediction probabilities and the accuracy per game stage.

Both FBT and LFR learn better models with F_{large} than with F_{small}, with huge differences in the opening due to large patterns, and both methods achieve very high accuracy at the endgame. As with F_{small}, FBT outperforms LFR on every data set, but the differences between the two methods are smaller. This can be expected, especially for the opening stage, where both algorithms learn the same large-scale patterns for the standard opening moves. FBT retains its advantage for the middle game, which is important in practice since many games are decided there.

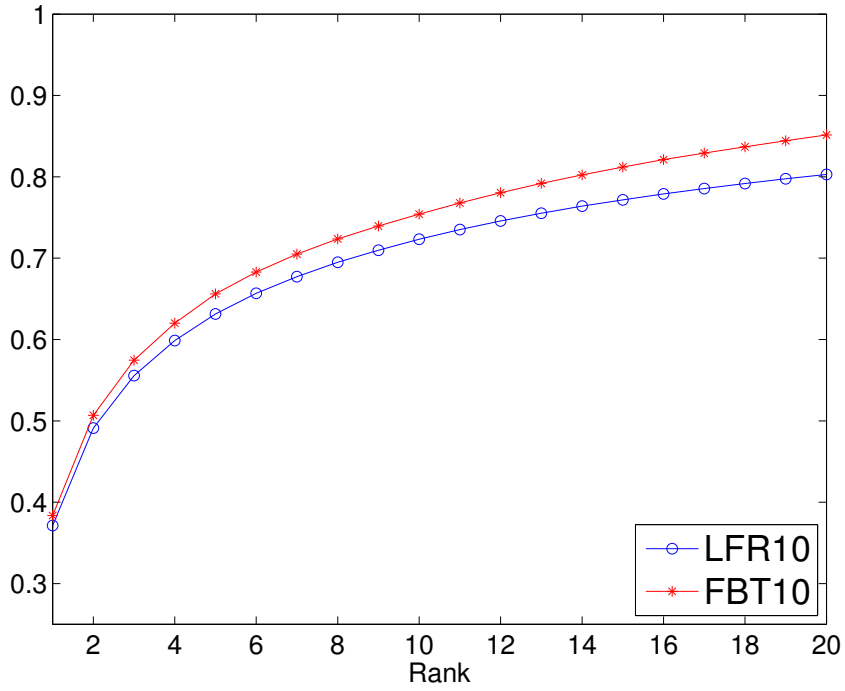


(a) F_{small} cumulative rank, $k = 10$

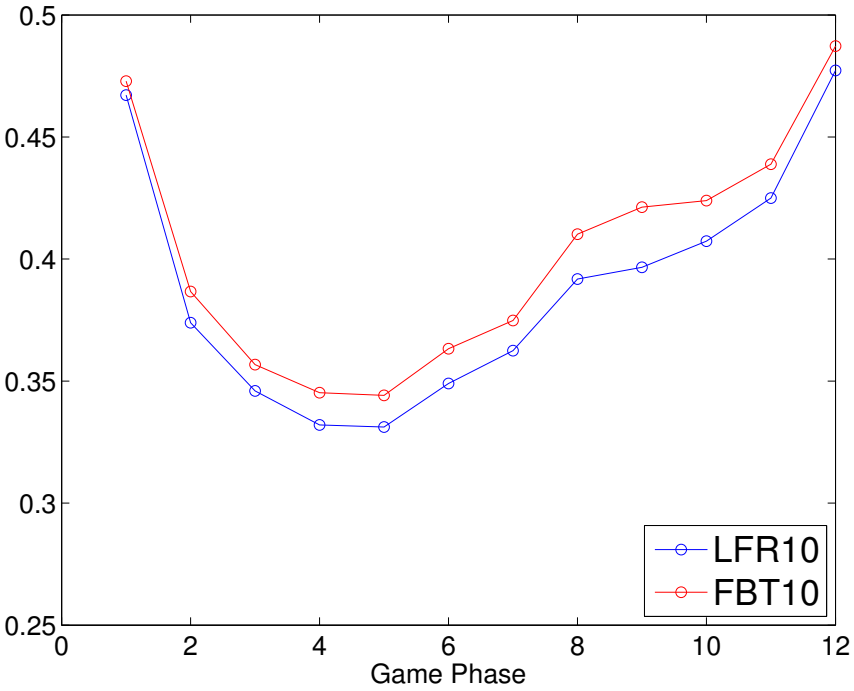


(b) F_{small} accuracy per game stage, $k = 10$

Figure 4.2: Move prediction experiments of FBT and LFR trained with F_{small}. y-axis is the prediction accuracy. In (a), the x-axis is the number of ranked moves, while in (b) the x-axis represents the game stage.



(a) F_{large} cumulative rank, $k = 10$



(b) F_{large} accuracy per game stage, $k = 10$

Figure 4.3: Move prediction experiments of FBT and LFR trained with F_{large}. y-axis is the prediction accuracy. In (a), the x-axis is the number of ranked moves, while in (b) the x-axis represents the game stage.

Table 4.3: Move prediction by FBT with different sample sizes on data sets $S_1 - S_3$. Results for LFR and FBT_Full shown for comparison. Bold values highlight best results among sampling methods.

Training set	FBT_S5	FBT_S10	FBT_S20	FBT_S30	LFR	FBT_Full
S_1	30.04%	30.81%	31.06%	**31.78** %	30.01%	32.56%
S_3	32.07%	32.90%	32.98%	**33.03**%	30.95%	33.18%
S_3	32.54%	33.01%	33.09%	**33.12**%	31.13%	33.46%

4.1.4 Sampling for Approximate Gradient Computation

To evaluate the new online sampling method for approximate gradient computation, experiments with $k = 5$ were run on the three training sets, varying the number of samples, $m \in \{5, 10, 20, 30\}$. In the following, FBT_Sm denotes approximate FBT with m samples, while FBT with full gradient computation is labeled FBT_Full. LFR is also included. The results are presented in Table 4.3. The performance of FBT_S increases with growing number of samples m, and approaches that of FBT_Full. FBT_S performs better than LFR even with a small number of samples. Regarding the cumulative rank, FBT_S30 achieves 82.26% for top 20 prediction, compared to 83.54% for FBT_Full and 77.81% for LFR (the right-most data points in Figure 4.2(a)).

The purpose of using Monte Carlo approximation in the gradient computation is to accelerate the training process while still keeping the theoretical soundness and practical accuracy. Figure 4.4 plots the prediction accuracy of FBT_Full and FBT_Sm as a function of time for all tested m. The plot shows the best prediction accuracy achieved so far on the validation set in 30 seconds intervals, starting from the same initial parameter value, on a training set of 1000 games with $k = 5$ using feature set F_{small}. All sampling algorithms finish the training process within 15 minutes, while FBT_Full needs about an hour. FBT_S20 and FBT_S30 outperform FBT_Full in the early phases. The performance of FBT_S30 comes very close to FBT_Full, while using less time to train (15 minutes vs 1 hour). The performance of the approximate gradient estimators increases with the number of samples, since the variance is reduced.

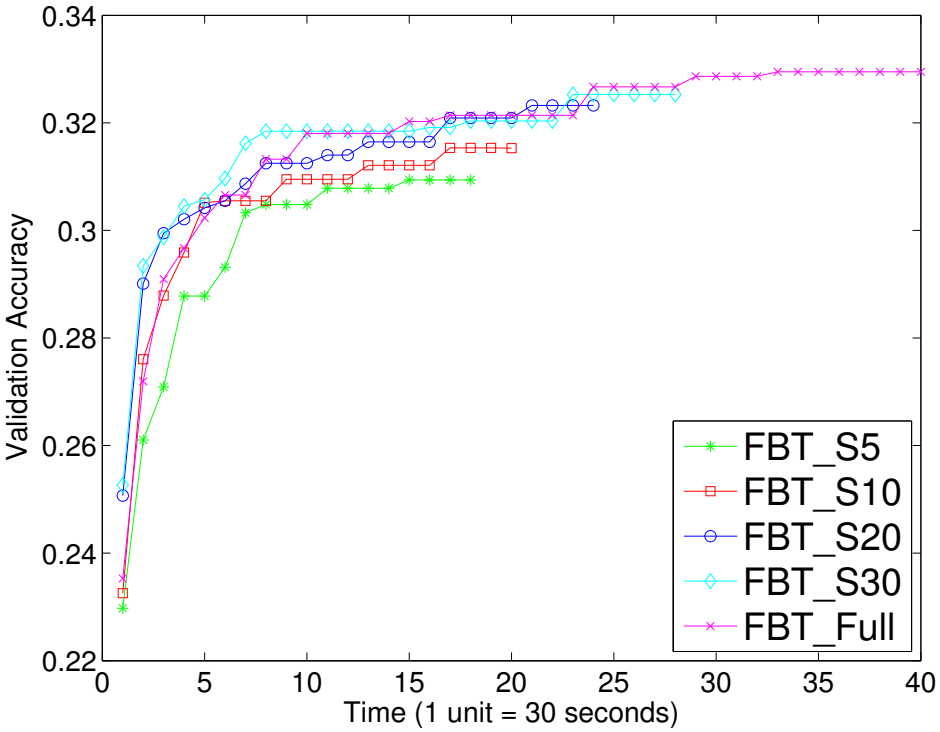


Figure 4.4: Accuracy of approximate sampling over time

4.2 Integrating FBT-Learned Knowledge in MCTS

We use the open source program Fuego [14] as our experimental platform to test if FBT knowledge is helpful for improving MCTS. We first introduce the feature knowledge in the current Fuego system, then introduce the setup of the experiment, and finally present the results.

4.2.1 Feature Knowledge for Move Selection in Fuego

Prior Feature Knowledge

Recent versions of Fuego such as svn version 2017 used in these experiments apply feature knowledge to initialize statistical information when a new node is added to the search tree. A set of features trained with LFR [38] and $k = 10$ is used. The LFR evaluation is a real value indicating the strength of the move without any probability based interpretation. Fuego designed a well-tuned formula to transfer the output value to the prior knowledge for initialization. It adopts a similar method

as suggested in [17], where the prior knowledge contains two parts: $N_{prior}(s,a)$ and $Q_{prior}(s,a)$. This indicates that MCTS would perform $N_{prior}(s,a)$ simulations to achieve an estimate of $Q_{prior}(s,a)$. Let $V_{LFR}(s,a)$ be the evaluation of move $a \in \Gamma(s)$, $V_{largest}$ and $V_{smallest}$ be the largest and smallest evaluated value respectively. Fuego uses the following formula to assign $N_{prior}(s,a)$ and $Q_{prior}(s,a)$,

$$N_{prior}(s,a) = \begin{cases} \frac{c_{LFR}*|\Gamma(s)|}{SA} * V_{LFR}(s,a) & \text{if } V_{LFR}(s,a) \geq 0 \\ -\frac{c_{LFR}*|\Gamma(s)|}{SA} * V_{LFR}(s,a) & \text{if } V_{LFR}(s,a) < 0 \end{cases} \tag{4.1}$$

$$Q_{prior}(s,a) = \begin{cases} 0.5 * (1 + V_{LFR}(s,a)/V_{largest}) & \text{if } V_{LFR}(s,a) \geq 0 \\ 0.5 * (1 - V_{LFR}(s,a)/V_{smallest}) & \text{if } V_{LFR}(s,a) < 0 \end{cases} \tag{4.2}$$

where $SA = \sum_i |V_{LFR}(s,i)|$ is the sum of absolute values of each move's evaluation. When a new game state is added to the search tree, Fuego initializes its statistics by setting $N(s,a) \leftarrow N_{prior}(s,a)$ and $Q(s,a) \leftarrow Q_{prior(s,a)}$.

Greenpeep Knowledge

A second kind of knowledge used in Fuego is in-tree move selection policy is called Greenpeep Knowledge. It uses a pre-defined table of diamond shape patterns (size 4 in Figure 2.2) to get probability based knowledge $P_g(s,a)$ about each move $a \in \Gamma(s)$. This knowledge is added as a bias for move selection according to a variant of the PUCT formula [29]. The reason why Fuego could not use LFR knowledge to completely replace the simpler Greenpeep knowledge might be that LFR cannot produce a probability-based evaluation. Details of the Greenpeep knowledge can be found in the Fuego source code base [13].

Move Selection in Fuego

In summary, Fuego adopts the following formula to select moves during in-tree search,

$$move = \text{argmax}_a(Q(s,a) - \frac{c_g}{\sqrt{P_g(s,a)}} \times \sqrt{\frac{N(s,a)}{N(s,a)+5}}) \tag{4.3}$$

where c_g is a parameter controlling the scale of the Greenpeep knowledge. $Q(s,a')$ is initialized according to formula (4.1) and (4.2), and further improved with Monte

Carlo simulation and Rapid Action Value Estimation (RAVE). Note that formula (4.3) does not have the UCB style exploration term, since the exploration constant is set to zero in Fuego. The only exploration comes from RAVE. Comparing formula (4.3) with (3.15), we could consider the FBT knowledge $P_{FBT}(s, a)$ as a replacement of the Greenpeep knowledge $P_g(s, a)$, but with a different way to be added as a bias and a different decay function.

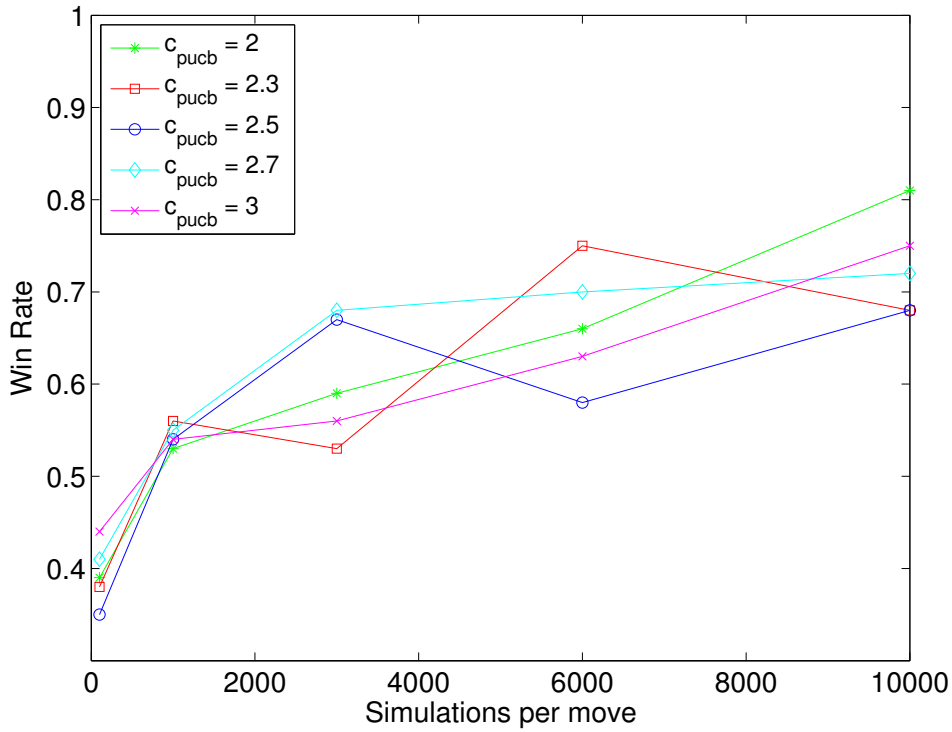
4.2.2 Setup

We call Fuego without LFR prior knowledge FuegoNoLFR, Fuego applying formula (1) to select moves in-tree FBT-Fuego, and Fuego without LFR but using formula (1) FBT-FuegoNoLFR. The lower cut threshold for FBT knowledge is set to $\lambda_{FBT} = 0.001$. All other parameters are left at the default settings of Fuego. The FBT model used in this experiment is trained on S_2 with F_{large} features and has interaction dimension $k = 5$.

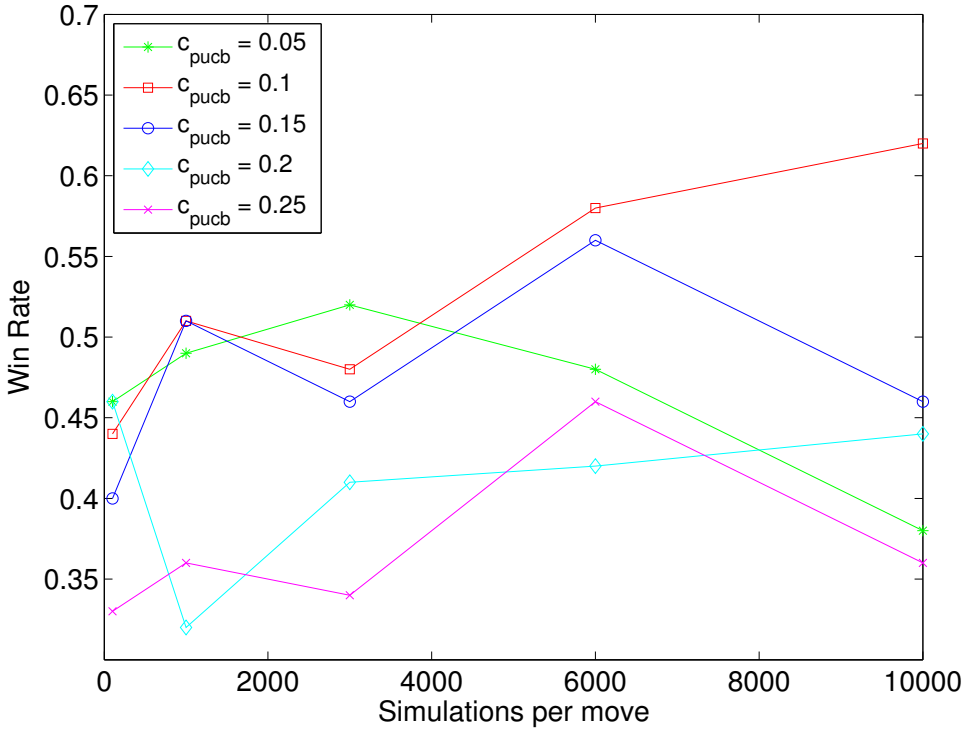
4.2.3 Experimental Results

We first compare FBT-FuegoNoLFR with FuegoNoLFR. This experiment is designed to show the strength of FBT knowledge without any influence from other kinds of knowledge. We test the performance of FBT-FuegoNoLFR against FuegoNoLFR with different exploration constants c_{puct}. After initial experiments, the range explored was $c_{puct} \in \{2, 2.3, 2.5, 2.7, 3\}$. Scaling with the number of simulations per move, N_{sim} was tested by setting $N_{sim} \in \{100, 1000, 3000, 6000, 10000\}$. Figure 4.5(a) shows the win rate of FBT-FuegoNoLFR against FuegoNoLFR. All data points are averaged over 1000 games. The results show that adding FBT knowledge can dramatically improve the performance of Fuego over the baseline without feature knowledge as prior. FBT-FuegoNoLFR scales well with more simulations per move. With $c_{pucb} = 2$ and 10000 simulations per move FBT-FuegoNoLFR can beat FuegoNoLFR, with a 81% winning rate.

We then compare FBT-Fuego with full Fuego, in order to investigate if the FBT knowledge is comparable with current feature knowledge in Fuego and able to improve the performance in general. In this case, c_{puct} is tuned over a different range,



(a) FBT-FuegoNoLFR vs FuegoNoLFR.



(b) FBT-Fuego vs Fuego.

Figure 4.5: Experimental results of integrating FBT knowledge in Fuego.

Program Name	100	1000	3000	6000	10000
FBT-FuegoNoLFR	11.8	192.4	704.1	1014.5	1394.9
FuegoNoLFR	5.1	55.7	148.7	225.2	354.4
FBT-Fuego	23.4	241.1	734.1	912.6	1417.5
Fuego	10.8	168.3	564.2	778.6	1161.2

Table 4.4: Running time comparison (specified in seconds) with different simulations for per move.

$c_{puct} \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$. $N_{sim} \in \{100, 1000, 3000, 6000, 10000\}$, and all data points are averaged over 1000 games as before. Results are presented in Figure 4.5(b). FBT-Fuego has worse performance in most settings of c_{puct}. But it can be made to work after careful tuning. As shown in Figure 4.5(b), with $c_{pucb} = 0.1$, FBT-Fuego scales well with the number of simulations per move, and achieves 62% winning rate against Fuego with 10000 simulations per move. One possible reason is that the FBT knowledge is not quite comparable with the LFR knowledge. The moves these two methods favour might be different in some situations, which makes it very hard to tune a well tuned system when adding another knowledge term.

Finally, we show the running time of our methods with different simulations per move in Table (1). FBT-FuegoNoLFR spends much more time than FuegoNoLFR, since FuegoNoLFR only uses Greenpeep knowledge for exploration and thus does not need to compute any feature knowledge. FBT-Fuego takes less time than FBT-FuegoNoLFR, since it does not compute the Greenpeep knowledge. The speed of FBT-Fuego is a little worse than Fuego. The slight time difference is spent on computing large patterns, while Fuego only uses small shape patterns.

Chapter 5

Conclusion and Future Work

The new Factorization Bradley-Terry (FBT) model, combined with an efficient Stochastic Gradient Descent algorithm, is applied to predicting expert moves in the game of Go. Experimental results show that FBT outperforms LFR, the previous state-of-the-art high-speed move predictor. FBT is also useful in improving the performance of Monte Carlo Tree Search.

Future work includes: 1. try to discover a method to transform FBT knowledge into prior knowledge for initialization. 2. try to apply the FBT knowledge for improving a fast roll-out policy. 3. combine FBT with Deep Convolutional Neural Networks (DCNN) to get a fast accurate move predictor. For example, in computer vision DCNN has been used to extract features of pictures and combined with a traditional classifier in a powerful object detection system [18]. A similar approach might also work for the move prediction problem: use DCNN to extract new features, then apply FBT to learn feature weights. Another possible approach is to automatically switch between FBT and DCNN, in order to get a high prediction accuracy while still keeping a low processing time.

Bibliography

[1] https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol. [Online; accessed 2016-07-26].

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47:235–256, May 2002.

[3] Y. Bengio. Learning deep architectures for AI. *Foundations and trends in Machine Learning*, 2(1):1–127, 2009.

[4] C.M. Bishop. Pattern recognition and machine learning. Springer, 2006.

[5] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[6] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

[7] M. Bowling, N. Burch, M. Johanson, and O. Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, January 2015.

[8] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[9] C. Clark and A. Storkey. Training deep convolutional neural networks to play Go. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Proceedings*, pages 1766–1774. JMLR.org, 2015.

[10] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In J. van den Herik, P. Ciancarini, and H. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630/2007 of *Lecture Notes in Computer Science*, pages 72–83, Turin, Italy, June 2006. Springer.

[11] R. Coulom. Computing Elo ratings of move patterns in the game of Go. In *Proc. Computer Games Workshop 2007 (CGW2007)*, pages 113–124. 2007.

[12] S. Della Pietra, V. Della Pietra, and J. Lafferty. Inducing features of random fields. *IEEE transactions on pattern analysis and machine intelligence*, 19(4):380–393, 1997.

[13] M. Enzenberger and M. Müller. Fuego, 2008-2015. http://fuego.sourceforge.net.

[14] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. Fuego - an open-source framework for board games and Go engine based on Monte Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[15] K. J. Friedenbach. *Abstraction Hierarchies: A Model of Perception and Cognition in the Game of Go*. PhD thesis, University of California, Santa Cruz, 1980.

[16] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

[17] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[18] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 580–587. IEEE, 2014.

[19] T.K. Huang, C.J. Lin, and R.C. Weng. Ranking individuals by group comparisons. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 425–432. ACM, 2006.

[20] D. Hunter. MM algorithms for generalized Bradley-Terry models. *Annals of Statistics*, pages 384–406, 2004.

[21] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[22] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1976.

[23] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer Berlin / Heidelberg, 2006.

[24] C. Maddison, A. Huang, I. Sutskever, and D. Silver. Move evaluation in Go using deep convolutional neural networks. *CoRR*, abs/1412.6564, 2014.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, and S. Petersen. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[26] M. Müller. Computer Go. *Artificial Intelligence*, 134(1–2):145–179, 2002.

[27] S. Rendle. Factorization machines. In G. Webb, B. Liu, C. Zhang, D. Gunopulos, and X. Wu, editors, *ICDM 2010, The 10th IEEE International Conference on Data Mining, Sydney, Australia, 14-17 December 2010*, pages 995–1000. IEEE Computer Society, 2010.

[28] S. Rendle, Z. Gantner, C. Freudenthaler, and L. Schmidt-Thieme. Fast context-aware recommendations with factorization machines. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 635–644. ACM, 2011.

[29] C. Rosin. Multi-armed bandits with episode context. *Ann. Math. Artif. Intell.*, 61(3):203–230, 2011.

[30] J. Schaeffer, N. Burch, Y. Bjrnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[31] J. Schaeffer, R. Lake, P. Lu, and M. Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21, 1996.

[32] J. Schaeffer and A. Plaat. Kasparov versus Deep Blue: The rematch. *ICCA Journal*, 20(2):95–101, 1997.

[33] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[34] D. Stern. *Modelling Uncertainty in the Game of Go*. PhD thesis, 2008.

[35] D. Stern, R. Herbrich, and T. Graepel. Bayesian pattern ranking for move prediction in the game of Go. In *Proceedings of the 23rd international conference on Machine learning*, pages 873–880. ACM, 2006.

[36] R.C. Weng and C.J. Lin. A bayesian approximation method for online ranking. *The Journal of Machine Learning Research*, 12:267–300, 2011.

[37] M. Wistuba, L. Schaefers, and M. Platzner. Comparison of bayesian move prediction systems for Computer Go. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 91–99. IEEE, 2012.

[38] M. Wistuba and L. Schmidt-Thieme. Move prediction in Go - modelling feature interactions using latent factors. In I. Timm and M. Thimm, editors, *KI2013*, volume 8077 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2013.

[39] C. Xiao and M. Müller. Factorization ranking model for move prediction in the game of Go. In *AAAI*, pages 1359–1365, 2016.

[40] C. Xiao and M. Müller. Integrating factorization ranked features into MCTS: an experimental study. In *Computer Games Workshop at IJCAI*, 2016.