April 2015

UCT-Enhanced Deep Convolutional Neural Networks for Move Recommendation in Go

Pitchaya Wiratchotisatian
Worcester Polytechnic Institute

Sarun Paisarnsrisomsuk
Worcester Polytechnic Institute

MQP CDR#GXS 1501

UCT-Enhanced Deep Convolutional Neural Network for Move Recommendation in Go

a Major Qualifying Project Report
submitted to the faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science by

Sarun Paisarnsrisomsuk

Pitchaya Wiratchotisatian

April 30, 2015

Professor Gábor N. Sárközy, Major Advisor

Abstract

Deep convolutional neural networks (CNN) have been proved to be useful in predicting moves of human Go experts [10], [12]. Combining Upper Confidence bounds applied to Trees (UCT) with a large deep CNN creates an even more powerful artificial intelligence method in playing Go [12]. Our project introduced a new feature, board patterns at the end of a game, used as inputs to the network. We implemented a small, single-thread, 1-hidden-layer deep CNN without using GPU, and trained it by supervised learning from a database of human professional games. By adding the new feature, our best model was able to correctly predict the experts' move in 18% of the positions, compared to 6% without this feature. In practice, even though the board pattern at the end of the game is invisible to Go players until the very end of the game, collecting these statistics during each simulation in UCT can approximate the likelihood of which parts of the board belong to the players. We created another dataset by using UCT to simulate a number of playouts and calculating the average influence value of Black and White positions in those boards at the end of the simulations. We used this dataset as the input to the deep CNN and found that the network was able to correctly predict the experts' move in 9% of positions.

Acknowledgements

We would like to express our gratitude to the people whose assistance, hospitality, and advice helped the completion of this project:

- Levente Kocsis, Project Advisor and SZTAKI liaison
- Gabor Sarkozy, MQP Advisor
- Worcester Polytechnic Institute
- MTA-SZTAKI
- SZTAKI Colleagues
- Pachi and Gnu Go Development Team
- Ryan Harris on his YouTube videos about the back-propagation algorithm

Table of Contents

Table of Figures

1 Introduction

There have been countless developments of artificial intelligence that improved the ability to play games with computers. Computers are now expert players in many popular, two-player games, including Backgammon, Checkers, Chess, and Othello. Still the game of Go, an ancient board game, continues to elude the efforts of computer science researchers. Go is remarkable because of its simple rules but complex strategies. Because the size of the search space in the game is enormous and the game has the long-term influence of moves, creating a move prediction function in Go is a challenging task for computer science researchers.

In the past, researchers had failed to combine brute force tree-search with human knowledge or reinforcement learning [5]. The efforts to equalize the performance of computer Go to the performance of other two-player games were in vain, until 2006 when the Monte-Carlo tree search (MCTS) was introduced to the game of Go. The performance of Go computers was advanced by the development of machine learning schemes. Several techniques, such as upper confidence bounds applied to trees (UCT), rapid action-value estimation (RAVE), temporal-different search (TD-search), etc., are combined with MCTS, yet computers still cannot defeat human professional players.

The attempt to advance the performance of Go computers is now focused on increasing the use of pattern recognition in Go. Typical computer Go algorithms simulate thousands of possible future moves and make tiny use of pattern recognition. Clark and Storkey [10] believe that developing pattern recognition algorithms for Go might be the missing piece needed to close the performance gap between computers and humans. It could provide ways to combat a high branching factor since it might be possible to prune out many possible moves based on patterns in

the current position. Pattern recognition systems would allow computers to analyze Go in a much more humanlike manner. Eliminating most candidate moves based on learned patterns within the board would also provide more time to find and to think about more promising moves.

Since we expect moves made by professional Go players to be highly complex, the expected move prediction function is then intricate and non-linear. These properties make the move prediction task very difficult. Yet they motivate researchers to attempt a deep learning approach as it has been conveyed that deep learning is well suited to learn a complex, non-smooth function [10].

In the past few years, applying deep convolutional neural networks (CNN) to predict moves in the game of Go have become a popular topic. Previous work in move prediction for Go typically made use of feature construction or shallow neural networks. Depending on the complexity of the model used, researchers have achieved accuracies of 30% – 41% on move prediction for strong, amateur Go players [10]. Recently in 2014, Clark and Storkey [10] created an 8-layer deep CNN that won 12% of games against Fuego using time limits corresponding to approximately 5,000 rollouts per move. Maddison et al. [12] constructed a deep CNN that represents and learns a move evaluation function. They trained a large 12-layers convolutional neural network by supervised learning from a database of human professional Go games. The network predicts the expert moves correctly in 55% of positions which equals the accuracy of a 5-dan human player. Moreover, the work of Maddison et al. demonstrates that Go knowledge embodied by the CNN can be effectively combined with MCTS, by using a delayed prior knowledge procedure. The procedure evaluated the CNN asynchronously on a GPU, and results were incorporated into the main search procedure. Using 100,000 rollouts per move, their combined search defeated the raw CNN in 87% of games.

Our project shows that introducing a new feature, board pattern at the end of a game, as inputs to a deep CNN can improve the prediction accuracy of the network. We will call this new feature “final board pattern” throughout this report. We modified the Gnu Go program to import a Go database of over 170,000 human professional games, or over 34 million moves from the KGS Server in SGF format, and then extract features, including final board pattern, in each individual move. We implemented a small, single-thread, 1-hidden-layer deep CNN without using GPU, and trained it by supervised learning from the obtained dataset. Our best model was able to correctly predict the experts’ move in 18% of positions.

Even though the final board pattern is not visible to Go players until the very end of the game, collecting its statistics during each simulation in UCT can approximate the likelihood of which territory belongs to the players. We generated another dataset by modifying the Pachi program to create a UCT and average territory of Black and White in all final board patterns in the simulations. After training our deep convolutional neural network with this new dataset, the network was able to correctly predict the experts’ move in 9% of positions, which is better than training the network without final board pattern inputs that predicted correctly 6%.

2 Background

2.1 Game of Go

Go is an ancient board game originating in China more than 3000 years ago. It was spread to other countries in East Asia through trade and other contact between countries in the first millennium A.D. [1] and named differently as Weiqi in China, Igo in Japan, and Baduk in Korea. In ancient China, Go was one of the arts to be mastered by gentlemen of the society; by the 1600's, Chinese gentlemen must master the "Four Accomplishments" that consist of calligraphy, painting, playing the lute, and playing Go. Go is now a popular pastime in Japan. Go is not only played in China and Japan, but the International Go Federation published that millions of players in over seventy countries around the world enjoy playing and studying the game of Go [2].

2.1.1 Rules of Go

Go is a two-player game that is usually played on a 19x19 grid board. The board typically starts empty, but it can contain "handicap" stones placed prior to the start of the game to give one player an advantage in order to offset the difference in performance between differently ranked players. Each player places Black and White stones on the grids alternatively. A "liberty" is an open point (empty grid) next to a stone. A basic idea of the game is that a group of stones must have at least one liberty to remain on the board. Stones are "captured" and removed from the board when they are surrounded by the opponent's stones. An encircled liberty is called an "eye". A group of stones that contains at least two separated eyes is said to be unconditionally "alive", meaning it cannot be captured. "Dead" stones are stones surrounded in groups with poor shape that has one or no eyes, and would

be captured inevitably. A concept of “life and death,” indicating the status of stones, can be summarized by the following: a group of stones determined as being “alive” may remain on the board indefinitely, on the other hand, a group of stones determined as being “dead” may only be lost and captured eventually. Placing a stone where it has no liberties, called “suicide,” is illegal to perform; it is rarely beneficial even if the move is allowed. “Ko” meaning eternity in Japanese is a rule to prevent players to perform a loop of repeating the same position on the board.

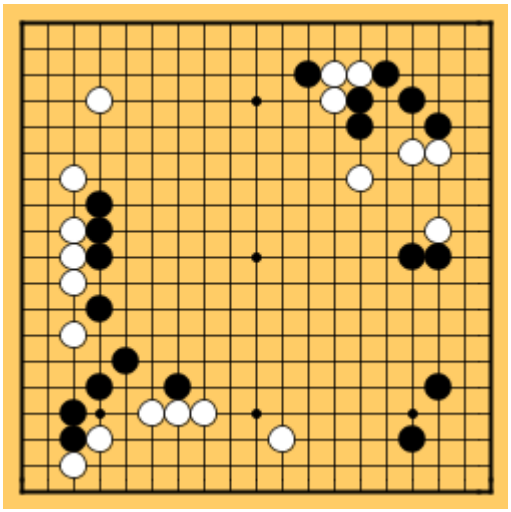

Figure 1. The standard 19x19 board after several moves

(http://www.newworldencyclopedia.org/entry/Go_(board_game))

The objective of the game of Go is to secure more “territory” (empty location on the board) than the opponent by either occupying or surrounding grid board portions with stones. Players can pass their turn. The game ends when both players pass, when in general they both agree that the game is over and the game is scored. Because Black has an advantage by playing the first move, the compensation point, called “komi”, can be given to white. According to the Chinese rule, each player gets one point for every location of territory they control, plus one point for their living stones. The Japanese rule in contrast, each player

receives one point for every location of territory they control, plus one point for each captured stone. Both rules usually generate the same score. At the end, a player with higher score wins.

2.1.2 Go Rank System

The difference in the rank of Go players corresponds to the number of handicap stones required to maintain parity. The rank of amateur players is ordered by decreasing kyu: beginning at 30 kyu and then going down to 1 kyu. The rank of professional players is ordered by increasing dan: beginning at 1 dan and then going up to 7 dan. Finally the rank of top professional Go players who currently have won at least one major tournament is ordered by increasing dan: beginning at 1 dan and then going up to 9 dan [3]. When players with different ranks play together, the difference in the rank can be compensated by the handicap system; roughly a difference of one rank corresponds to one handicap or one free move at the beginning of the game.

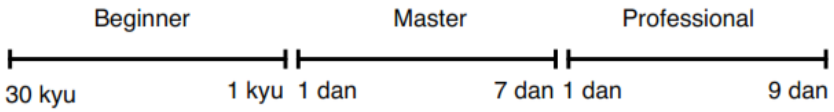

Figure 2. Performance ranks in Go, in increasing order of strength from left to right [3]

2.1.3 Go: An Ultimate Challenge for Artificial intelligence

Similar to other popular board games, namely Checkers, Chess, and Othello, Go has many of the same properties, as follows:

Two-player: This is the rudiment of multiplayer games.

Zero-sum: There is no cooperation in Go; if one player gains, another one loses.

Deterministic: There is no randomness involved in the game, unlike Monopoly that involve randomness, such as rolling a die.

Perfect-information: Both players can see the entire board all the time. When making decision, each player is perfectly informed of all events that have happened so far.

Although the rules of Go are simple and the game has similar characteristics compared to several other games, Go is difficult for a computer to play due to many reasons. Mainly, the combinatorial complication of the game is enormous; the state space of the game is not only huge (the number that represents black, white, and empty grid points on a 19x19 board is $3^{361} \approx 10^{172}$), but the branching factor of Go is also intractably huge. For example, the number of moves in each turn of Go is approximately 200 moves and the length of a typical game is around 300 turns; both numbers are more than five times compared to chess. Another reason that Go is mysteriously hard is the long term influence of a move in Go. A move made in the beginning of a game can affect the outcome of the game hundreds of moves later [3]. Therefore, unlike other popular board games, Go is one of the few board games where human experts have been comfortably defeated artificial intelligence. In addition, a legitimate Go strategy requires an extremely deep look-ahead, therefore, creating a heuristics to predict a move of Go experts is a challenging task.

2.2 Computer Go Techniques

As a result of artificial intelligence research, computers are now capable of playing a variety of popular games. Although computers are now experts in playing several games such as Backgammon, Checkers, Chess, and Othello, Go is an ancient game in which a computer cannot yet beat human experts [4]. Computer Go originally began with brute force tree-search with human knowledge or reinforcement learning [5]. However, these techniques have failed to achieve any solid success in the game of Go, until 2006 when Monte Carlo Tree Search had been introduced to the game of Go with the selectivity mechanism, UCT, [6] and then there have been substantial advancements in computer Go. The ranks of computer Go programs have jumped from weak kyu level to the professional dan level on a 9x9 Go board, and to strong amateur dan level on a 19x19 Go board [3].

2.2.1 Monte Carlo Tree Search (MCTS)

Before going into details of MCTS, let us start with the two ideas of Monte Carlo simulation and a minimax game tree search.

Investopedia [16] defines Monte Carlo simulation as a computational algorithm that approximates the probability of certain outcomes by running multiple trial runs, called simulations, using random variables. The method can give a good approximation to the expected value of a state in MCTS. A number of simulations are performed by sampling the actions according to the randomized policy. The expected rewards from the simulations are averaged to give the Monte-Carlo value estimate of the initial state.

A minimax game tree is used to represent a two-player zero-sum game tree that applies a minimax algorithm to decide the best move that maximizes the reward and minimizes the

loss. A node of the tree corresponds to a single state of the game and an edge between two nodes corresponds to a move in the game.

MCTS combines the two concepts of Monte Carlo simulation and minimax game tree search; it builds and expands the tree search while also evaluating the strength of individual moves by sampling the actions according to a randomized policy. Generally an MCTS consists of four stages processing repetitively until reaching the computational budget [7].

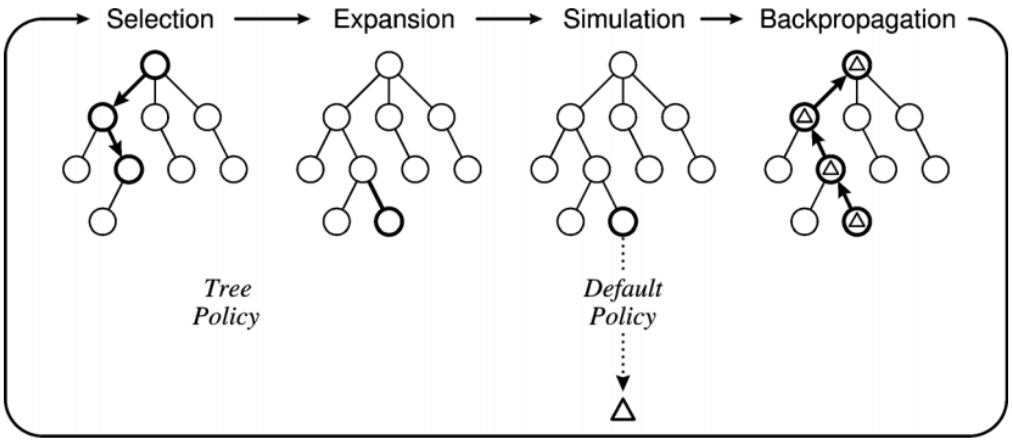

Figure 3. One iteration of a general MCTS approach [7]

Starting with the tree policy, beginning at a root node, a child selection policy is recursively applied through the tree until an expandable node is reached. Then child nodes are added to the tree. Following the default policy, a Monte-Carlo simulation is run from the new node(s) according to a certain policy to produce the estimated outcome. After these two policies, the simulation result is back-propagated through the selected nodes to update their statistics from leaf to root.

Many successful Go programs, for instance, MoGo, Pachi, Fuego, Gnu Go, and CrazyStone, are based on MCTS. Nonetheless, today MCTS Go programs are using either uniform sampling of actions or some heuristic biasing of the action selection probabilities that come with no guarantees [6]. The basic MCST algorithm can require prohibitively large

numbers of simulations when applied to very large search spaces, such as a 19x19 Go board, due to two main reasons [8]. First, each move position is evaluated independently without any generalization between similar positions. Second, Monte-Carlo simulation produces a high variance estimate of the value of each move position,

2.2.2 Upper Confidence Bounds Applied to Trees (UCB applied to trees / UCT)

Kocsis and Szepesvari introduced a new UCT algorithm that applied bandit-based method to guide Monte-Carlo planning [6]. The multi-armed bandit problem is a stochastic problem in which a gambler has to decide how to play K slot machines to maximize the total expected reward, where each machine gives a reward with a random distribution. He has to decide which machines to play, how many times to play each machine, and in which order to play them. The ideal solution to this problem is to always select the action whose reward value estimate is the largest, with an optimistic adjustment that takes account of the uncertainty of the value estimate. In this case, if the action does not result in an optimal action, it will still result in a reduction of the uncertainty associated with the value estimate of the chosen action. Therefore, suboptimal actions cannot be selected indeterminately [9]. There is an exploration-exploitation tradeoff in multi-armed bandit problems; the gambler need to quickly find the action with highest expected reward without losing too much reward along the way. The tradeoff can be displayed as the expected regret after playing *N* rounds:

$$R_N = \mu^* n - \sum_{j=1}^{K} \mu_j E[T_j(n)]. \quad (1)$$

Here μ_j is the value of the reward from the j[th] machine, $\mu^* = \max\{\mu_1, \mu_2, \ldots, \mu_K\}$, and $E[T_j(n)]$ is the expected number of plays on the j[th] machine. The regret is the loss caused by the policy not always playing the best machine and is the expected difference between the

reward sum associated with an optimal strategy and the sum of the collected rewards. Lei and Robbins [9] published a paper proving that there exists no policy whose regret would grow slower than $O(\log n)$.

UCT balances the exploitation and the exploration techniques in MCTS which improves the consistency in the tree search. In MCTS, the value of an action *a* of a Black player, *Z(a)*, is estimated as the following:

$$Z(a) = \frac{W(a)}{N(a)}. \tag{2}$$

Here *W(a)* is the total reward collected by a player in a game, and *N(a)* is the number of simulations in which a is the first action taken in the state s_0, the root of the game. The way that the UCT balances the exploration and the exploitation can be described as the following:

$$Z(a) = \frac{W(s,a)}{N(s,a)} + C\sqrt{\frac{logN(s)}{N(s,a)}} \tag{3}$$

$$the\ value\ of\ an\ action\ a\ in\ term\ of\ black\ player,$$

$$Z(a) = \frac{W(s,a)}{N(s,a)} - C\sqrt{\frac{logN(s)}{N(s,a)}} \tag{4}$$

$$the\ value\ of\ an\ action\ a\ in\ term\ of\ white\ player.$$

Here $C > 0$ is a tuning constant, *W(s,a)* is the total reward collected at terminal states during the simulations, *N(s,a)* is the number of simulations in which move *a* was selected from state *s*, and $N(s) = \sum_a N(s,a)$ is the number of simulations from state *s*. As the search tries to exploit actions with a high value reward, the second term that is added to the value estimate of an action *Z(a)* is the exploration bonus. The bonus is large if the algorithm inspects non-duplicate actions. This way, the most uncertain actions are eliminated and rarely explored moves are encouraged to be tried more frequently [9]. Given enough time, the search

of a UCT algorithm will find the optimal values for all nodes of the tree, and thus it can select the optimal action at the root state.

How do we apply a multi-arm bandit problem to UCT? The action selection problem in UCT is treated as a separate multi-armed bandit for every explored internal node. The arms correspond to actions associated with the tree's nodes, and the payoffs correspond to the cumulated rewards of the paths originating at the node or state s [6]. Unlike many other MCTS methods, if C in the equation (3) and (4) is large enough, then given enough simulations, the tree grows large enough so that the values of the nodes converge to the optimal values with probability one. The convergence of the values of nodes in the tree starts at the nodes close to the leaf nodes and eventually extends to all the nodes in the tree. The convergence of the values leads to the convergence of the algorithm so that we select an optimal action at the root node with probability one. The following is the theorem that Kocsis and Szepesvari [6] proved that the failure probability at the root of the tree converges to zero. We denote:

- $X_{i,t}$ = the payoff of machine *i* playing at time *t*,
- $\bar{X}_{i,n} = \frac{1}{n}\sum_{t=1}^{n} X_{i,t}$, the expected payoffs from playing the i^{th} arm,
- $\mathbb{I}(\cdot)$ the indicator function; $\mathbb{I}(I_s = i) = 1$ if $i = I_s$, otherwise 0,
- $T_i(t) = \sum_{s=1}^{t} \mathbb{I}(I_s = i)$, the number of times the i^{th} arm was played up to time *t*,
- $\mu_{i,n} = E\left[\bar{X}_{i,n}\right]$ and $\mu_i = lim_{n\to\infty}\mu_{i,n}$,
- $\mu^*, T^*(t), \bar{X}_t^*$, the quantities related to the optimal arm, where we assume that there exists a single optimal action, and
- $\Delta_i = \mu^* - \mu_i$.

Also for simplicity it is assumed that $0 \leq X_{i,t} \leq 1$ and $\bar{X}_{i,n}$converges.

Convergence of Failure Probability Theorem [6]: *Let* $\hat{I}_t = argmax_i \bar{X}_{i,T_i(t)}$.

Then $P(\hat{I}_t \neq i^*) \leq \alpha \left(\frac{1}{t}\right)^{\frac{\rho}{2} \frac{min_{i \neq i^*} \Delta_i{}^2}{36}}$ *with some constant* α. *In particular, it holds that*

$lim_{t \to \infty} P(\hat{I}_t \neq i^*) = 0$.

Explicitly, the theorem demonstrates that the probability that the selection of an action at the root node is not optimal is bounded by an upper bound value. In addition, given enough simulations, the probability that the selection of an action at the root node is not optimal converges to zero. However, in practice, the convergence may take a very long time. Nonetheless, the algorithm is easy to implement, extensible, and successful in many domains, such as MoGo, CrazyStone, Valkyria, Pachi, Fuego, Zen, etc.

2.2.3 Deep Convolutional Neural Networks (Deep CNN)

Move Prediction is an interesting task in machine learning. It asks how we can build a program that is trained on games of Go in order to predict the moves of a professional Go player. We expect moves in a game of Go to be highly complex. Human experts play Go with heavy analysis that relies heavily on pattern analysis; what parts of the board fall to whom and what should be the best move at a glance. In contrast, typical computer algorithms have to simulate thousands of possible future positions and make minimal use of pattern recognition [10]. Increasing pattern recognition performance in computers could contest the

high branching factor in Go since it is possible to prune out some of the possible moves based on learned patterns and efficiently gain more time to analyze promising moves.

Since human experts think in complex and non-linear ways when they choose moves, we expect the move prediction function to be highly complex and non-smooth as well. A minor change to a position in the game can dramatically alter which moves are likely to play next. In consequence, the learning task of move prediction is considerably challenging. A deep structure learning approach is therefore suitable to learn this complex and non-linear function.

During the several past years, deep structure learning, or commonly called deep learning or hierarchical learning has become a popular machine learning technique that impacts a wide range of signal and information processing work. Deng and Yu [11] defined deep learning as "a class of machine learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification". It is based on a set of algorithms that allow computers to approach human level pattern recognition abilities. It attempts to learn multiple levels of representation and abstraction that help to make sense of data by using model architectures with complex structures. Various deep learning architectures, such as deep neural networks (DNN), deep convolutional neural networks (Deep CNN), and deep belief networks (DBN), are used widely in image and speech recognition.

There have been some previous work in move prediction for Go that typically made use of feature construction or shallow neural networks. Depending on the complexity of the model used, researchers have achieved accuracies of 30% – 41% on move prediction for strong, amateur Go players [10]. However, in the past few years, applying deep convolutional

neural networks to the game of Go have become a popular topic. In particular, Maddison et al. constructed a deep CNN that represents and learns a move evaluation function. They trained a large 12-layers convolutional neural network by supervised learning from a database of human professional Go games. The network predicts the expert moves correctly in 55% of positions which equals the accuracy of a 5 dan human player. When the trained convolutional network was applied directly to play Go, it beat Gnu Go, the traditional-search program in 97% of the games without any search. It also matched the performance of a state-of-the art MCTS that simulates two million positions per move [12]. We will go into detail of DNN later in chapter 3.2.

3 Deep Convolutional Neural Networks

3.1 Artificial Neural Network (ANN)

ANN is a computational technique for recognizing patterns in data, which is used wildly to classify objects by features. It is popularly used in image recognition and video analysis. Also it has been used in Natural Language Processing and playing Go. Inspired by the structure and function of the human brain, the structure of ANN is comprised of a number of artificial nodes, known as "neurons" or units that are interconnected to construct a network. This network can compute values from a large amount of inputs and estimate non-linear, usually unknown, functions from the inputs.

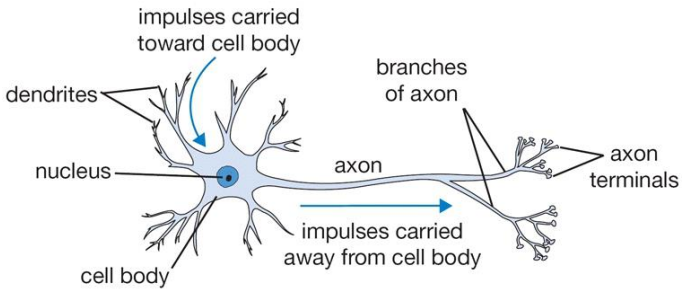

Figure 4. Structure of a neural network in the human brain

(http://cs231n.github.io/neural-networks-1)

Here are the simple steps of how ANN works. Starting from the input, each input is associated with weights and linked to neurons. The neural network can have several layers, which we call "hidden layers", and each layer can contain several neurons. Neurons in one layer are connected to neurons in the next layer with weights associated to them. The neural network architecture is often constructed in distinct layers of neurons; neurons in the same layer share no connections, but they are fully pairwise connected to neurons in the adjacent layers. Finally the output is obtained from the last neuron layer.

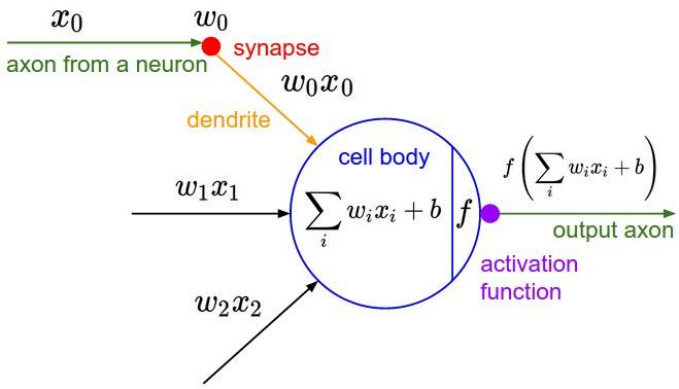

Figure 5. Architecture of an artificial neural network

(http://cs231n.github.io/neural-networks-1)

How does a neural network basically compute an output? Again each input is associated with a weight that is in general a floating point number. We will call the region of hidden layer(s) that contain a number of neurons, "nucleus" or "processing element". The process begins when each input x_i enters the nucleus, and then it is multiplied by its corresponding weight w_i. Then each product is summed together and compared to a threshold value. If the sum is greater than the threshold, the output is 1, otherwise the output is 0.

A function that transforms the input value to the output value of a neuron is called an "activation function". It is required to be continuous and differentiable. Here is a neural network where the activation function is a unit step function.

$$Output = \begin{cases} 1, & if \sum_{i=0}^{n} w_i x_i \geq 1 \\ 0, & if \sum_{i=0}^{n} w_i x_i < 1 \end{cases} \quad (5)$$

In general, a bias b, which is a random number, is added to the activation value by replacing the threshold. The bias measures how easily the processing element will produce 1.

$$Output = \begin{cases} 1, & if \sum_{i=0}^{n} w_i x_i + b > 0 \\ 0, & if \sum_{i=0}^{n} w_i x_i + b \leq 0 \end{cases} \quad (6)$$

In general, a log-sigmoid function σ, known as a logistic function or simply a sigmoid function, is used as an activation function because its derivative is easy to calculate. The following is a sigmoid function:

$$\sigma(X) = \frac{1}{1+e^{-\beta X}}. \quad (7)$$

Here β is a parameter which controls a slope of the curve; usually it is set to 1. A sigmoid function is similar to the step function, but with a region of uncertainty. The higher the value of β, the more the curve looks like a step function. Since the center of the sigmoid curve is at 0.5, negative activation values result in less than 0.5, and vice versa for positive activation values. A sigmoid function can also be constructed using the hyperbolic tangent function, in which case it would be called a tan-sigmoid function. There are several types of activation functions in practice. In our project, we employ a rectifier function (8) for hidden layers and a softmax function (9) for the output layer.

$$rectifier(x) = \max(0, x) \quad (8)$$

A unit rectifier function is also called a rectified linear unit (ReLU).

$$softmax(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, for\ j = 1, \ldots, K, \quad (9)$$

where *z* is a K-dimensional vector of arbitrary real values, $softmax(z)_j$ is a K-dimensional vector of real values in the range (0,1).

In conclusion of how ANNs process, initially, a set of input neurons is defined. Each input is associated with a weight. These input neurons are then transformed by a function designed by a creator, and the activations of these neurons are passed to other neurons layer-by-layer. While the error of biases Δb and the error of weights Δw (we call error or delta) are also calculated from one layer to the next layer and the weights and biases are adjusted with respect to the error. The process is repeated until an output neuron is activated at the end. Moreover, there are many types of ANNs varying from the basic structures of layers and directional logic of ANNs, to complicated multi-layer, multi-input, and many-directional-feedback-loop ANNs. Our project applies deep convolutional neural networks as move predictors in Go.

3.2 Deep Convolutional Neural Network (Deep CNN)

A Convolutional neural network is a type of feedforward neural network that has special weight constraints. It is comprised of one or more convolutional layers, which are often added with a subsampling step, and followed by one or more fully connected layers as in a standard multilayer ANN [13]. The architecture of a CNN is designed to take advantage of two-dimensional structure inputs that result in translation invariance features. Another benefit of a CNN is that it enables the network to learn more efficiently from fewer examples; with the same number of hidden nodes, a CNN is easier to train and has fewer parameters than a fully connected ANN. CNNs have been successfully applied to various problems, mostly in image and video recognitions. For example, CNNs have obtained the best classification accuracy on the MNIST (Mixed National Institute of Standards and Technology database) handwritten digits dataset in 2012. They have achieved a 0.23 percent error rate which is a near-human accuracy [14].

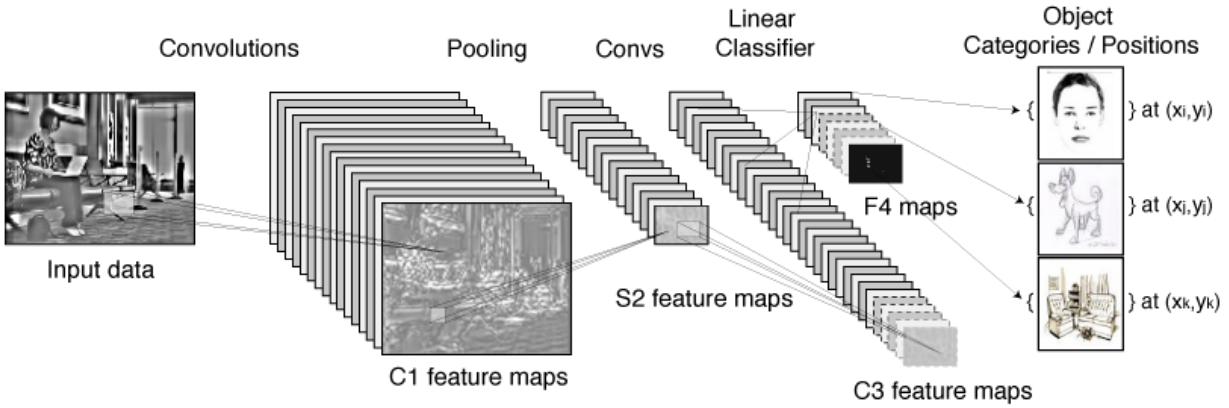

Figure 6. Convolutional Neural Network in Image Recognition

(http://torch.cogbits.com/doc/tutorials_supervised/convnet.png)

Why do we apply CNN to predict moves in Go? CNNs are well suited for problems with a natural translation invariance, such as object recognition. Go does have some translation invariance [15]. If all stones on a Go board are shifted to the left, then the best move will also shift, with the exception that stones are on the boundary of the board. Thus, CNNs have been long applied to the game of Go.

Previous move prediction for Go typically made use of feature construction or shallow neural networks [10]. The feature construction approach has been done by characterizing each legal move by a large number of shape features. Shape features can be augmented with hand crafted features, such as distance from the edge of the board, the number of captured or lost stones after making the move, its distance to the previous moves, etc. Finally, a model is trained to rank the legal moves based on their features. The features used as inputs to the CNN in our project were adapted from the work of Maddison et al. [12]. The details of our features are in the next chapter.

Feature	Planes	Description
Black / white / empty	3	Stone colour
Liberties	4	Number of liberties (empty adjacent points)
Liberties after move	6	Number of liberties after this move is played
Legality	1	Whether point is legal for current player
Turns since	5	How many turns since a move was played
Capture size	7	How many opponent stones would be captured
Ladder move	1	Whether a move at this point is a successful ladder capture
KGS rank	9	Rank of current player

Figure 7. Features used as inputs to the CNN in the work of Maddison et al. [12]

CNNs form a subclass of feed forward neural networks that have special weight constraints. Sutskever and Nair [15] described that Go boards in the CNN are expected to be processed in the same way in every small board patch. Therefore, the weights of the CNN have a replicated structure where the size of the sub-rectangular boards, as shown in figure 8, is always the same. The structure in one layer will produce the total inputs to the next layer. The weights of a CNN are called the convolutional kernel; its size is the size of the subrectangles it considers.

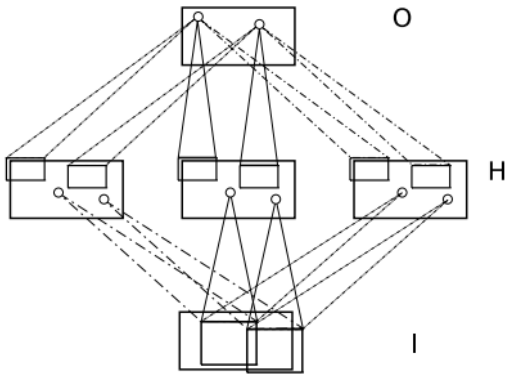

Figure 8. A CNN with one hidden layer, and three convolutional kernels;
the applications of the kernels to two rectangular regions of their inputs are shown.
I, H, and O are the input, hidden, and output layers, respectively. [15]

The CNN in figure 8 has one hidden layer and three convolutional kernels. In general, a CNN can have several convolutional kernels. The convolutional kernel works in such a way that the sizes of the maps in the input, hidden, and output layers are the same. To do so, some rectangles

need to be partially outside of the board; the weights of the kernels corresponding to the out-of-the-board region are unused.

After we design a CNN, we train the network and look for the minimum of the error function, the difference between desired outputs and outputs generated by the network. To approach this goal, we use the *backpropagation* algorithm, which is described in the next section.

3.3 Back-propagation Algorithm

The backward propagation of errors, or backpropagation, is a common method of training artificial neural networks. It is usually considered to be a supervised learning method; it requires a known, desired output for each input value. The method looks for the minimum of the error function in the weight space using an optimization method such as the gradient descent algorithm. The combination of weights which minimizes the error function is an essential solution of the learning problem.

Let x be an input and $t = y(x)$ be a target output. We use the gradient descent algorithm to find weights and biases so that the output from the network, $O(x)$, approximates the target output t for all training inputs x. To closely approximate the output, we have to minimize the error or the difference between actual outputs and outputs generated by the network. The error function can be displayed as the following:

$$E(w, b; x) = \frac{1}{2}\sum_{n \in N}\left\|t_n - O_{(w,b)_n}(x)\right\|^2. \tag{10}$$

Here N is the number of nodes in the output layer. The squared error is also called a "cost function", a "loss function", or an "objective function". The goal of the gradient descent algorithm is to minimize the error function $E(w,b;x)$, a function of the weights and biases. To approach this

goal, it calculates the gradient of the function with respect to all the weights and biases in the network, which is illustrated in the following equations:

$$\frac{\partial E}{\partial w} = \frac{\partial}{\partial w}\frac{1}{2}\sum_{n\in N}\left\|t_n - O_{(w,b)_n}(x)\right\|^2, \tag{11}$$

$$\frac{\partial E}{\partial b} = \frac{\partial}{\partial b}\frac{1}{2}\sum_{n\in N}\left\|t_n - O_{(w,b)_n}(x)\right\|^2. \tag{12}$$

When *f(·)* is an activation function and *x* is an input from a previous layer $l-1$ to a layer *l*, the results are shown as follows:

- The rate of change of the error with respect to the weight w_{jk} connecting from the j[th] node in the last hidden layer to the k[th] node in the output layer can be computed as follows:

$$\begin{aligned}\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial w_{jk}}\frac{1}{2}\sum_{n\in N}\left\|t_n - O_{(w,b)_n}(x)\right\|^2 = (O_k - t_k)\frac{\partial O_k}{\partial w_{jk}} = (O_k - t_k)\frac{\partial f(x_k)}{\partial w_{jk}} \\ = (O_k - t_k)f'(x_k)\frac{\partial x_k}{\partial w_{jk}} = (O_k - t_k)f'(x_k)O_j.\end{aligned} \tag{13}$$

If we denote $\delta_k = (O_k - t_k)f'(x_k)$, we can express the rate in term of the product of δ_k and the output from the j[th] node, which is the input of the k[th] node in the next layer.

$$\frac{\partial E}{\partial w_{jk}} = \delta_k O_j. \tag{14}$$

- The rate of change of the error with respect to the weight w_{ij} connecting from the i[th] node in the hidden layer *l-1* to the j[th] node in the hidden layer *l* can be computed as follows:

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial w_{ij}}\frac{1}{2}\sum_{n\in N}\left\|t_n - O_{(w,b)_n}(x)\right\|^2 = \sum_{n\in N}(O_n - t_n)\frac{\partial O_n}{\partial w_{ij}} \\ = \sum_{n\in N}(O_n - t_n)\frac{\partial f(x_n)}{\partial w_{ij}} = \sum_{n\in N}(O_n - t_n)\,f'(x_n)\frac{\partial x_n}{\partial w_{ij}} \\ = \sum_{n\in N}(O_n - t_n)\,f'(x_n)\frac{\partial x_n}{\partial O_j}\frac{\partial O_j}{\partial w_{ij}} = \sum_{n\in N}(O_n - t_n)\,f'(x_n)w_{jn}\frac{\partial O_j}{\partial w_{ij}}\end{aligned} \tag{15}$$

$$= \sum_{n\in N}(O_n - t_n)\, f'(x_n) w_{jn} \frac{\partial f(x_j)}{\partial w_{ij}} = \sum_{n\in N}(O_n - t_n)\, f'(x_n) w_{jn} f'(x_j) \frac{\partial x_j}{\partial w_{ij}}$$

$$= \sum_{n\in N}(O_n - t_n)\, f'(x_n) w_{jn} f'(x_j) O_i.$$

Again, if we denote $\delta_j = \sum_{n\in N}(O_n - t_n)\, f'(x_n) w_{jn} f'(x_j) = f'(x_j) \sum_{n\in N} \delta_n w_{jn}$, we can express the rate in term of the product of δ_j and the output from the i[th] node, which is the input of the j[th] node in the next layer.

$$\frac{\partial E}{\partial w_{ij}} = \delta_j O_i. \tag{16}$$

- In the same manner that we derived from equations (13), the rate of change of the error with respect to the bias b_k of the k[th] node in the output layer can be computed as follows:

$$\frac{\partial E}{\partial b_k} = \frac{\partial E}{\partial b_k} \frac{1}{2} \sum_{n\in N} \left\| t_n - O_{(w,b)_n}(x) \right\|^2 = (O_k - t_k) f'(x_k) = \delta_k. \tag{17}$$

- In the same manner that we derived from equations (15), the rate of change of the error with respect to the bias b_j of the j[th] node in the hidden layer l can be computed as follows:

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial b_j} \frac{1}{2} \sum_{n\in N} \left\| t_n - O_{(w,b)_n}(x) \right\|^2 = \sum_{n\in N}(O_n - t_n)\, f'(x_n) w_{jn} f'(x_j) = \delta_j. \tag{18}$$

After choosing the weights and biases of the network randomly, the backpropagation algorithm is used to adjust the weights and biases to minimize the error. The algorithm can be decomposed into the following four steps [17]:

i) Feed-forward computation: The algorithm runs the network forward with input data to get the network output. The vector of the output $\boldsymbol{O}$ is computed and stored to the network. The evaluated derivatives of the activation functions are also stored at each unit.

ii) Backpropagation to the output layer: For each output node k, it computes δ_k,

$$\delta_k = (O_k - t_k) f'(x_k).$$

iii) Backpropagation to the hidden layer: For each hidden node j, it calculates δ_j,

$$\delta_j = f'(x_j) \sum_{n \in N} \delta_n w_{jn}.$$

iv) Weight updates: It then updates the weights and biases as follows:

$$\Delta W_{ij} = -\alpha \delta_j O_i, \tag{19}$$

$$\Delta b_j = -\alpha \delta_j, \tag{20}$$

where α is the learning rate, which is a training parameter that controls the size of the weight and bias changes in the learning of the training algorithm.

The algorithm stops when the value of the error function has become sufficiently small.

Back-propagation networks are ideal for learning pattern recognition. By giving examples to the network, the algorithm learns and changes the weights and biases so that when the training is finished, it gives the desired output for a particular input.

4 Methodology

4.1 Existing Codebases

4.1.1 Gnu Go

Gnu Go [18] is a free Go software written in the language C. Its algorithms and source code are open and documented. It compiles on many platforms, including, GNU/Linux, Unix, Windows, and Mac OS. The latest stable release at the moment is Gnu Go 3.8, released in February 2009. Although the developers are now not maintaining it, the free program still boasts a great wealth of heuristics, patterns and tactical modules. Adapting it to MCTS appears to be a formidable task. It allows anyone to inspect or enhance, giving Gnu Go's descendants a certain competitive advantage.

The GNU Go 3.0.0 is the first version that implemented GTP (Go Text Protocol[1]). The implementation of GTP in GNU Go is distributed over three files, "interface/gtp.h", "interface/gtp.c", and "interface/play_gtp.c". The first two files implement a small library of helper functions which can be used by other programs. These files are licensed under the GNU GPL (GNU General Public License) in order to promote the GTP. The actual GTP commands are implemented in 'play_gtp.c', which has knowledge about the engine internals.

To start Gnu Go in GTP mode, we can simply invoke it with the option "--mode gtp". Here are simple examples of how to start learning about GTP:

list_commands: bring up a list of GTP commands type

showboard: view the Go board type

[1] More detail in section 4.1.3

play black q16: make a first move type, placing a black stone at row q and column 16

genmove white: make the engine move type.

We used Gnu Go to read Go records in SGF files (Smart Game Format[2]), and extract features from each move. One weakness we found on Gnu Go is that it supports UCT Monte-Carlo Tree Search only on a 9x9 board type.

4.1.2 Pachi

Another program that we implemented on is Pachi. Pachi [19] is an open source Go playing program. It is being actively developed by Petr Baudis and Jean-loup Gailly. The current stable Pachi version is 11.00 (codename Retsugen), released on April, 2015. The source code is written in C and capable to be compiled on Window, Linux, and Mac OS. The main source code brand is maintained within the Pachi.git repository.

By default, Pachi uses the UCT engine that combines Monte-Carlo approach with tree search; UCB1AMAF (UCB1 applied As Move As First policy) tree policy using the RAVE (Rapid Action Value Estimation) method is used for tree search, while the Moggy playout policy using 3x3 patterns and various tactical checks is used for the semi-random Monte-Carlo playouts [19].

We used Pachi to create UCT and collect statistics of a final board pattern in each simulation. One weakness that we found on Pachi is that it cannot undo previous moves.

4.1.3 GTP (Go Text Protocol)

The Go Text Protocol (GTP) [20] is a text based protocol for communication with computer Go programs. It is a modern alternative to the Go Modem Protocol (GMP). It is intended to be

[2] More detail in Appendix A.

used as auxiliary programs, to make it easier for Go programmers to connect to Go servers on the internet and do automatic regression testing. Once GTP is implemented, one will have a big choice of GUIs to use; one can have a program of one's own play on any of NNGS (No Name Go Server), IGS (Internet Go Server), or KGS (Kiseido Go Server); one can use existing test suites or existing tools with one's own test suite; finally, one can easily play matches against other GTP engines.

4.2 Implementation

4.2.1 Go Data

We collected over 170,000 complete Go games from the KGS Go Server [21], which are in SGF format[3]. The games consist of sequences of positions on a 19x19 board played by human experts in varying ranks. We developed features based on Maddison et al. [12] and extracted the following features:

Features	Planes	Description
Black/white/empty	3	Stone color
Liberties	4	Number of liberties (empty adjacent points)
~~Liberties after move~~	~~6~~	~~Number of liberties after this move is played~~
Legality	1	Whether point is legal for current player
~~Turns since~~	~~5~~	~~How many turns since a move was played~~
~~Capture size~~	~~7~~	~~How many opponent stones would be captured~~
~~Ladder move~~	~~1~~	~~Whether a move at this point is a successful ladder capture~~
KGS rank	9	Rank of current player (used as learning rate adjustment)
Final board pattern	9	Player who owns this point (our new feature)

Figure 9. The features we used in our deep network
We do not include 'Liberties after move', 'Turns since', 'Capture size', and 'Ladder move' as our features.
We use 'KGS rank' to adjust learning rate. Our new feature is 'Final board pattern'.

[3] See appendix for details of SGF format

All the features are split into multiple planes of binary values. Due to limit of time and resources, we did not include Liberties after move, Turns since, Capture size, and Ladder move features. Another way how our features and Maddison et al.'s features differ is that we did not use Rank as an input feature to our deep CNN; we use Rank as an adjustment factor to learning rate values in the training (explained more in section 4.2.2).

We added another feature into this table, which is final board pattern. This can be easily done by following moves in an SGF file, one by one, until the end of the game, and then backtracking all previous moves and adding the final board pattern. These steps were done by modifying a Gnu Go program in GTP (Go Text Protocol) mode.

The games usually end early by resigning, and some grid points left on the board are empty. We used the "initial_influence" command implemented in GTP mode of a Gnu Go program to approximate and fill out those empty spaces. The "initial_influence" function in a Gnu Go program returns an integer between -4 and 4 (inclusive) for each space, which shows how likely each space belongs to the Black player (positive number) or the White player (negative number); zero indicates equivalent influence of Black and White players or empty grid point. Therefore, we gave a plane of size 9 for the final board pattern feature.

At the end, we obtained over 34 million individual moves and 17 feature planes per each move.

4.2.2 Deep Convolutional Neural Network

Even though CNN was first introduced in 1980 [22] and deep CNN has been widely used in image processing, using deep convolutional neural network on predicting moves in Go is very recent [10], [12]. We tried to find existing deep CNN codebase, but we found that the input format

of image processing deep CNN is quite different from the input format we would like to use in our CNN. We also found that many of them are implemented in CUDA (Compute Unified Device Architecture), which our server machine does not support. Therefore, we decided to implement deep CNN ourselves from scratch. We followed the network architecture of Maddison et al. [12]; our deep CNN has the first hidden layer's filters of size 5x5 and the remainder layers' filter of size 3x3. Every layer operated on a 19x19 input, with no pooling. We also used position-dependent biases. Due to time and resource limits, our network has smaller number of hidden layers and kernels than Maddison et al.'s [12]. In the input layer, our network consists of 17 feature nodes for each of the 19x19 board positions. In the output layer, our network consists of 19x19 nodes, which produce values representing how likely each position should be chosen to play as the next move. Biases in our network was initially set to be 1.0 in every node and initial weight in each edge was uniformly distributed between [-0.5, 0.5]. During the training, we assigned different learnings rates to the training data varying between [0.00001, 0.00009], depending on the rank of the player who made that move. For example, we gave a higher learning rate value of 0.08 to a move made by an 8-dan player, and we gave a lower learning rate value of 0.03 to a move made by a 3-dan player.

We also added momentum value (η) of 0.999 in our training. Momentum is a technique for accelerating gradient descent. It adds a fraction of the previous weight update to the current one. The momentum can prevent the system from converging to a local minimum or saddle point. A high momentum can also increase the speed of convergence. However, if the momentum is set too high, it can cause the system to become unstable since the system can overshoot the minimum. In contrast, if the momentum is set too low, it can lead the system to local minima and slow down the training.

Our network also experimented with a variety of constructing deep CNN techniques to discover the best model for our inputs.

4.2.2.1 Weight Symmetries

We found that using weight symmetries or forcing weights to be rotationally and reflectionally symmetric [23] can improve the accuracy of our deep CNN.

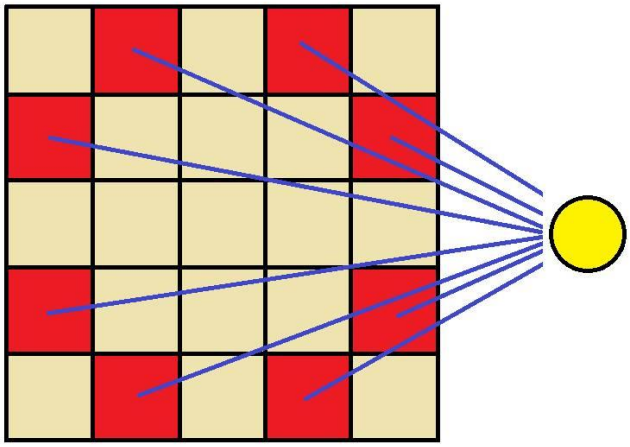

Figure 10. 8-fold weight symmetry group

We implemented this idea by making weights of all edges in the same "8-fold symmetry group" to be pointers that pointing to the same weight values. We updated these weight values by aggregating all weight adjustments over all edges in the group. As shown in figure 10, each red tile represents a grid point in the board. We applied weight symmetry such that the weight values connected to each red tile are the same.

4.2.2.2 Activation Functions

We experimented with several activation functions, such as the rectifier function, the softplus function, and the softmax function. We noticed that the rectifier function is not differentiable at $x = 0$, so we tried to use the softplus function, which is a smooth approximation function of the rectifier function.

$$softplus(x) = \ln(1 + e^{x}) \tag{21}$$

The derivative of the softplus function is exactly a sigmoid function. However, using the softplus function did not improve the accuracy of our deep CNN.

Using the rectifier function as the activation function for the output layer caused a problem to our deep CNN. In the first few trainings, our deep CNN sometimes created a big jump in weight adjustment, and our deep CNN produced a negative number as a result in the next training. Since the derivative of the rectifier function is zero when the input is a negative, the deep CNN adjusted the weight with zero rate.

$$rectifier'(x) = \begin{cases} 0 \; if \; x < 0 \\ 1 \; if \; x > 0 \end{cases} \tag{22}$$

To solve this problem, instead of using the rectifier function for the output layer, we used the softmax function.

Even though using the softmax function solved the problem mentioned above, our deep CNN sometimes still adjusted the weight with a very slow rate. If a result of a position is very low compared to other positions, the derivative of the softmax function at that position is very close to zero. To solve this problem, we decided to get rid of the derivative term of the adjusting weight equation (35). As a result, our adjusting weight equation for the output layer becomes

$$\Delta W_{ij} = -\alpha(O_k - t_k)O_j \tag{23}$$

4.2.2.3 Mini-Batches

Mini-batch is a stochastic gradient descent method, which computes the gradient against more than one training data at a step and applies the gradient once after a specific number of

training data [26]. For an example, using mini-batch size of 10 means we compute the gradients of weights and biases and apply them after 10 trainings.

In our project, we found that using 1000 mini-batches improved the performance of our deep CNN, compared to using 1 mini-batch.

4.2.3 UCT-simulated Final Board Pattern

Instead of generating final board patterns by following SGF files as mentioned in section 4.2.1, we created other datasets using UCT. We modified the Pachi program in option “uct” to collect statistics of final board patterns in each simulation, and to find the average of those final board patterns. We created multiple datasets with different number of simulations, which are 1, 5, 10, 50, 100, 200, 500, and 1000.

5 Experiments and Results

After we constructed the deep CNN, we did some experiments to find the best deep CNN model by varying its parameters. Here are the architectures and parameters that we used in our best deep CNN model:

- Architectures
    - The number of hidden layers = 1
    - The number of kernels = 10
    - The hidden layer's filter size is 5x5
    - No pooling layer
    - The output layer is not fully connected, but it has the same structure with convolutional layer with filter size 3x3
    - Hidden-layer activation function: linear rectifier
    - Output-layer activation function: softmax
    - Position-dependent biases
    - Weight symmetries
    - Initial biases are 1.0 for all nodes
    - Initial weights are uniformly distributed between [-0.5, 0.5]

- Parameters
    - Mini-batch size = 1000
    - Momentum parameter (η) = 0.999
    - Learning-rate parameter (α) = 0.0001
    - Learning rate are varying between [0.00001, 0.00009], as described in Section 4.2.2

Then, we constructed an experiment on different datasets by using the deep CNN with parameters mentioned above as a controlled variable. Here are datasets we used in the experiment:

- Extracted features as described in Section 4.2.1, without final board pattern feature
- Extracted features as described in Section 4.2.1, with actual final board pattern feature
- Extracted features as described in Section 4.2.1, with estimated final board pattern from UCT simulations as described in Section 4.2.3

We experimented on the accuracy in move predictions of the deep CNN over different number of trainings.

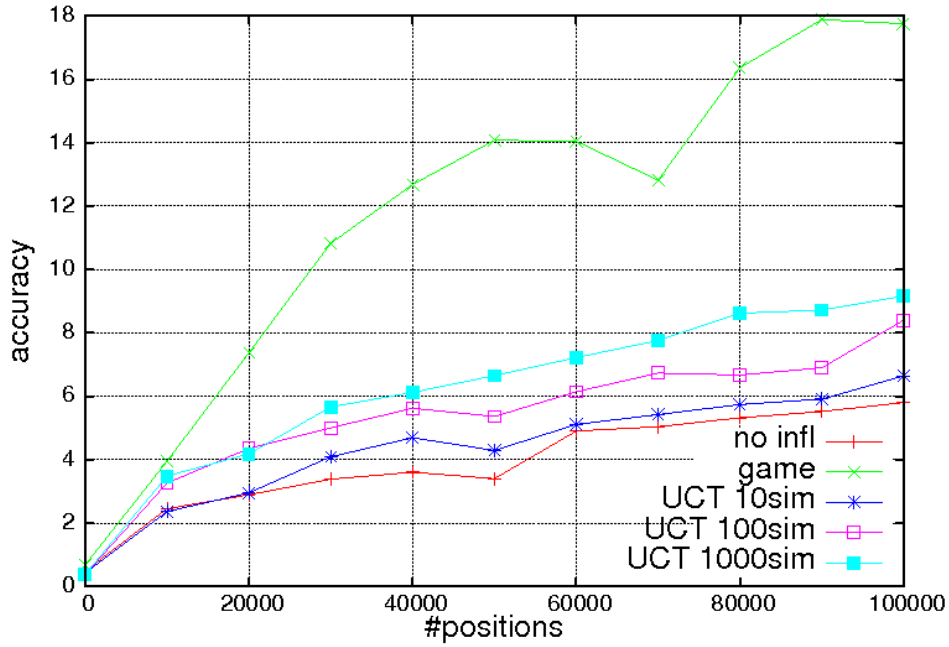

Figure 11. The accuracy comparison over different numbers of trainings (move positions) for different types of experiments

Figure 11 shows a comparative result of our deep CNN with different input datasets. Our base run is to use deep CNN to train the data without final board patterns (red line), the network was able to correctly predict 6% of expert moves. Adding the feature, end-game board pattern, which we obtained from SGF files (section 4.2.1), (green line) significantly improves the predictions of the network. We were able to achieve 18% accuracy. However, the actual final board pattern is not possible to be acquired until the very end of the game. Therefore, this accuracy is ideal and impossible to achieve in practice. A way we can predict the final board pattern of the current game is to collect the statistics in each simulation during UCT. The blue, pink, and cyan lines in the figure are the results of training the network with the inputs obtained by averaging the final board pattern in different number of simulations during UCT (section 4.2.3), 10, 100, and 1000 simulations respectively. Training with inputs from averaging 1000 simulations during UCT (cyan line) was able to correctly predict 9% of expert moves.

From the figure, even though the accuracy of prediction tends to increase as we increase the training data, adding the final board pattern feature to inputs of the deep network apparently improves the prediction accuracy, compared to the base run (red line).

Since deep CNN assigns values of being a potential position to each position in the board as its outputs, we not only can use the values to predict the position of the best next move, but we can also rank the positions as the second best next move, the third best next move, and so on.

We conducted another experiment on the accuracy of deep network if we allow the network to output, not only the best move but, top "k" moves. We measured what the percentage rate of times the correct move is chosen as one of the "k" moves picked by the deep CNN.

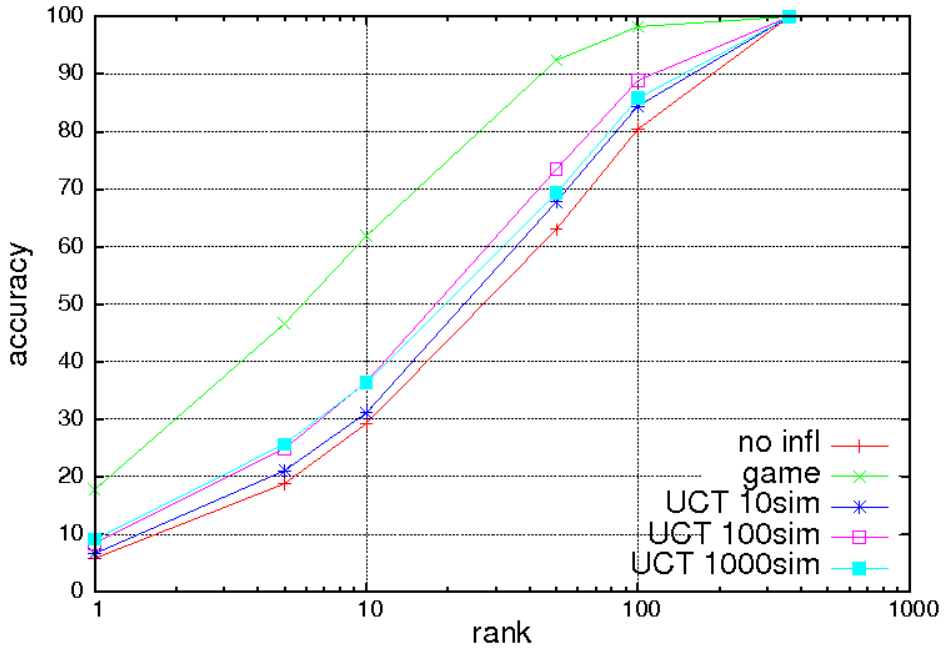

Figure 12. The accuracy of ranking top k move comparison over different types of experiments

In Figure 12, we analyzed how accurately the deep CNN can predict the expected move to be in the top "k" of its output, where the horizontal axis is the "k" values. At value k=10, adding final board pattern, which we obtained from SGF files (green line), can predict the correct move

as its top 10 best moves have prediction accuracy 63%, which is a significant improvement compared to the base run (red line, without final board pattern feature) with accuracy 29%. Again, as we mentioned before, the result from the green line is ideal and impossible to achieve in practice because the actual final board pattern is not possible to be acquired until the very end of the game. Instead, we generated board patterns at the end of games by averaging the final board games in each simulation of UCT. The blue, pink, and cyan lines are results from averaging the final board games of 10, 100, and 1000 simulations respectively. Training with inputs from averaging 1000 simulations during UCT (cyan line) was able to choose the correct expert moves as its top 10 moves with accuracy 37%.

Even though it might not be clear whether increasing the number of simulations during UCT will increase the accuracy of the network or not, it is clear that adding final board to inputs of the deep network improves the accuracy of predictions, compared to the base run (red line).

6 Conclusion and Future Work

6.1 Deep Convolutional Neural Network Improvement by Final Board Pattern

In recent papers, it has been shown that deep convolutional neural networks (CNN) can make a good prediction of human expert moves [10], [12]. The goal of our project was to improve deep CNN to achieve higher prediction accuracy. We introduced a new feature, final board pattern, as an input to our small, single-thread deep CNN. We extracted features, including final board pattern, from each individual moves of over 170,000 complete Go games from the KGS Go Server [21]. We found that adding this new feature to inputs of deep CNN significantly improves the accuracy of the network from 6% to 18%. However, this accuracy can only be achieved from the use of a complete Go database; in practice, it is not possible for Go players to foresee the final board pattern until the game is over.

6.2 UCT-simulated Final Board Pattern

Even though it is not possible to perfectly predict the final board pattern, it is possible to determine the likelihood of which territories belong to the players. We used UCT to approximate this likelihood by collecting the board patterns at the end of each simulation. Since previous work has shown that combining UCT with deep CNN significantly improves the potential of deep CNN [12], we can modify this UCT to collect the statistics at the end of each simulation with no extra cost. We generated another dataset which contains the approximation of final board pattern, obtained by UCT. With this dataset, our deep network was able to make a correct prediction 9% of expert moves, which is better than the base run (6% accurate).

6.3 Future Work

Due to limit of time and resources, we omitted some factors, which are left to be done as future work. Improving the network with these factors will ensure our conclusion that adding final board pattern improves the accuracy of deep network's prediction.

6.3.1 Deep Convolutional Neural Network Size and Speed

The deep network we used in this project is very small. It has only 1 hidden layer and 10 kernels. The number of hidden layers and the number of kernels are parameters in our program, which may be increased. However, increasing these numbers will make the network become larger and take much more time in trainings. Therefore, multi-threading and making use of GPU are needed in order to overcome this problem.

6.3.2 Data Usage

Even though we mentioned in Section 4.2.1 that we obtained over 34 million individual moves, in this project we only used 100,000 moves for the training set and 10,000 moves for the test set due to the time limit. As we can see in Figure 11, increasing the size of the training set can improve the accuracy of the network. Therefore, making use of all 34 million moves can provide potential improvement to our deep network.

6.3.3 Additional Features

As we mentioned in Section 4.2.1, we did not include Liberties after move, Turns since, Capture size, and Ladder move features to the input features. The main reason is that these features are obtained by adding a stone in each possible position, and then undoing the action. These actions

make the input feature take much more time to be generated. We encourage further research to add these features to inputs, which can improve the performance of the deep network.

6.3.4 Combining with UCT

Our deep network can be combined with UCT to improve the accuracy of prediction. Since a large deep CNN can take lots of time to evaluate the results, this step should be done by embedding the deep CNN into GPU to reduce the running time of deep CNN.

References

[1] American Go Association. http://www.usgo.org/brief-history-go, 1997.

[2] International Go Federation. http://intergofed.org/members.html, 2010.

[3] Gelly, Sylvain, et al. "The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions." *Communication of the ACM* 55.3 (2012): 106-13. Print.

[4] Computer-go.info. http://www.computer-go.info/h-c/index.html, 2015.

[5] J. Schaeffer. "The games computers (and people) play." *Advances in Computers* 52 (2000): 190–268.

[6] Kocsis, Levente, and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning." 4212 Vol. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. 282-293. Print.

[7] Browne, Cameron B., et al. "A Survey of Monte Carlo Tree Search Methods." *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012): 1-43. Print.

[8] Silver, David, Richard S. Sutton, and Martin Müller. "Temporal-Difference Search in Computer Go." *Machine Learning* 87.2 (2012): 183-219. Print.

[9] Lai, T. L., and Herbert Robbins. "Asymptotically Efficient Adaptive Allocation Rules." *Advances in Applied Mathematics*6.1 (1985): 4-22. Print.

[10] Clark, Christopher, and Amos Storkey. "Teaching Deep Convolutional Neural Networks to Play Go." (2014)Print.

[11] Yu, Dong, and Li Deng. “Deep Learning Methods and Applications.” *Foundations and Trends® in Signal Processing* 7.3-4 (2013): 197-387. Print.

[12] Maddison, Chris J., et al. "Move Evaluation in Go using Deep Convolutional Neural Networks." (2014)Print.

[13] Deep Learning. Computer Science Department, Stanford University. http://ufldl.stanford.edu/tutorial, 2013.

[14] Ciresan, D., U. Meier, and J. Schmidhuber. "Multi-Column Deep Neural Networks for Image Classification".IEEE , 2012. 3642-3649. Print.

[15] Nair, Vinod, and Sutskever, Llya. "Mimicking Go Experts with Convolutional Neural Networks." *Artificial Neural Networks - ICANN 2008: 18th International Conference, Prague, Czech Republic, September 3-6, 2008: Proceedings, Part II.* 5164 (2008): 101-110. Print.

[16] Investopedia.com. http://www.investopedia.com/terms/m/montecarlosimulation.asp, 2015.

[17] Rojas, Raúl, and SpringerLink ebooks - Computer Science (Archive). *Neural Networks: A Systematic Introduction*. New York: Springer-Verlag, 1996. Print.

[18] Gnu Go. https://www.gnu.org/software/gnugo, 2009.

[19] Pachi: Software for the Board Game of Go / Weiqi / Baduk. http://pachi.or.cz, 2015.

[20] GTP - Go Text Protocol. http://www.lysator.liu.se/~gunnar/gtp, 2012.

[21] Game records in SGF format. http://www.u-go.net/gamerecords, 2015.

[22] Ciresan, D., U. Meier, and J. Schmidhuber. "Multi-Column Deep Neural Networks for Image Classification".IEEE , 2012. 3642-3649. Print.

[23] Schraudolph, Nicol N, Dayan, Peter, and Sejnowski, Terrence J. "Temporal difference learning of position evaluation in the game of Go." *Advances in Neural Information Processing Systems*, (1994): 817–817.

[24] SGF File Format FF[4]. http://www.red-bean.com/sgf/, 2006.

[25] Perfilieva, Irina, and Vladik Kreinovich. "Towards a General Description of Translation-Invariant and Translation-Covariant Linear Transformations: A Natural Justification of Fourier Transforms and Fuzzy Transforms." (2011).

[26] Bottou, Léon. "Online Algorithms and Stochastic Approximations." *Online Learning and Neural Networks*, Cambridge University Press. (1998).

Appendix

A. Smart Game Format (SGF)

Smart Game Format [24], abbreviated as SGF, is a file format designed to store game records of board games for two players. It is a text only, tree based format. SGF was invented by Anders Kierulf in 1987 in the FF[1] format to record games of Go (called Smart Go Format). However it has become popular and has been extended to many other two-player board games.

SGF files cannot only store records of played games, but it can also provide features for storing annotated and analyzed games, such as board markup, variations, etc. Games stored in SGF format can easily be emailed, posted, or processed with text-based tools.

The file contains game trees of property lists with all their nodes and properties, and nothing more stored, written in pre-order. If a game needs to store more information on file with the document, a game-specific property must be defined for that purpose. The SGF's internal structure allows storing variations of main line of play.

Coordinate system for points and moves: The first letter designates the column of a board from left to right, the second letter designates the row of a board from top to bottom. The x-y coordinate system is used to read, starting from the upper left corner as the origin of a board. The author intentionally broke with the tradition of labeling moves (and points) with letters "A"-"T" (excluding "i") and numbers 1-19 for a 19x19 board. Two lowercase letters in the range "a"-"s" were used instead, for reasons of simplicity and compactness.

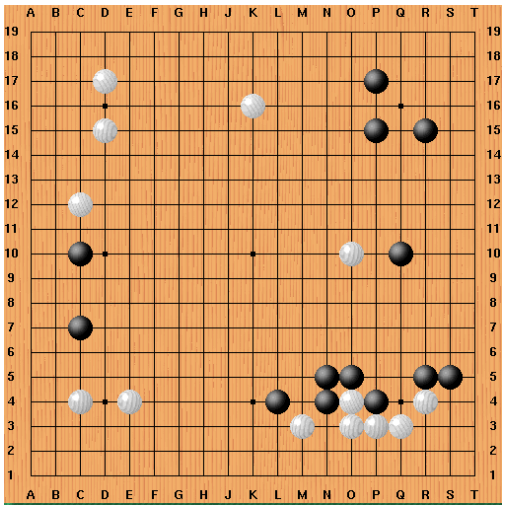

Figure 13. Coordinate system for points and moves in SGF files

(http://stromberg.dnsalias.org/~strombrg/software/newstosgf.gif)

SGF files are comprised of pairs of properties and property values, each of which describes a feature of the game. Here is the partial list:

AB: Add Black: locations of Black stones to be placed on the board prior to the first move.

AW: Add White: locations of White stones to be placed on the board prior to the first move.

AN: Annotations: name of the person commenting the game.

AP: Application: application that was used to create the SGF file (e.g. CGOban2, etc.)

B: a move by Black at the location specified by the property value.

BR: Black Rank: rank of the Black player.

BT: Black Team: name of the Black team.

C: Comment: a comment.

CP: Copyright: copyright information. See Kifu Copyright Discussion.

DT: Date: date of the game.

EV: Event: name of the event (e.g. 58th Honinbo Title Match).

FF: File format: version of SGF specification governing this SGF file.

GM: Game: type of game represented by this SGF file. A property value of 1 refers to Go.

GN: Game Name: name of the game record.

HA: Handicap: the number of handicap stones given to Black. Placement of the handicap stones are set using the AB property.

KM: Komi: komi.

ON: Opening: information about the opening (fuseki), rarely used in any file.

OT: Overtime: overtime system.

PB: Black Name: name of the black player.

PC: Place: place where the game was played (e.g.: Tokyo).

PL: Player: color of player to start.

PW: White Name: name of the white player.

RE: Result: result, usually in the format "B+R" (Black wins by resign) or "B+3.5" (black wins by 3.5 moku).

RO: Round: round (e.g.: 5th game).

RU: Rules: ruleset (e.g.: Japanese).

SO: Source: source of the SGF file.

SZ: Size: size of the board, non-square boards are supported.

TM: Time limit: time limit in seconds.

US: User: name of the person who created the SGF file.

W: a move by White at the location specified by the property value.

WR: White Rank: rank of the White player.

WT: White Team: name of the White team.

Here is an example of SGF Format from a tsumego problem on GoProblems.com:

```
(;FF[4]GM[1]SZ[19]
 GN[Copyright goproblems.com]
 PB[Black]
 HA[0]
 PW[White]
 KM[5.5]
 DT[1999-07-21]
 TM[1800]
 RU[Japanese]
 ;AW[bb][cb][cc][cd][de][df][cg][ch][dh][ai][bi][ci]
 AB[ba][ab][ac][bc][bd][be][cf][bg][bh]
 C[Black to play and live.]
 (;B[af];W[ah]
 (;B[ce];W[ag]C[only one eye this way])
 (;B[ag];W[ce]))
```

```
 (;B[ah];W[af]
 (;B[ae];W[bf];B[ag];W[bf]
 (;B[af];W[ce]C[oops! you can't take this stone])
 (;B[ce];W[af];B[bg]C[RIGHT black plays under the stones and
lives]))
 (;B[bf];W[ae]))
 (;B[ae];W[ag]))
```

B. Parameters used in our best DNN

- Mini-batch size = 1000
- Momentum parameter (η) = 0.999
- Learning-rate parameter (α) = 0.0001
- The number of kernels = 10
- The number of hidden layers = 1
- Output-layer activation function: softmax
- Hidden-layer activation function: linear rectifier (max(0,x))

C. Translational invariance

Go have some translation invariance. In formally, translation is to slide or to move a shape without rotating or flipping it. For example, suppose we have an input $x(s)$ at the starting point $s = 0$, if we shift all the moments of time by t, then in the new time coordinate s_1 that becomes $s - t$, the new input $x(s_1)$ has the form $x(s - t)$.

A general transformation linear 1-D transformation can be written as follows:

$Y(t) = c(t) + \int c(t, s) \cdot x(s)\, ds$, for appropriate functions $c(t)$ and $c(t, s)$. Define $y(t) = Y(t) - c(t)$. We can simplify the dependence on x(s) in a form:

$$y(t) = \int c(t,s) \cdot x(s) ds \tag{24}$$

We call *c(t,s)* a kernel of the transformation. Perfilieva and Kreinovich [25] described three definitions of translation invariant as follows:

1. We start with the input *x(s)* and produce the output *y(t)*. Then we select some time shift t_0 and take the input $x(s_1) = x(s - t_0)$ which is the new input in new translated coordinates $s_1 = s - t_0$. We expect the output $y_{t0}(t)$ also take the exact same form as before, i.e., $y_{t0}(t) = y(t_1) = y(t - t_0)$. We call the property that the relation between input and output does not change when we apply a time shift (translation), "invariant." In particular, such invariance is called translation-invariance.

2. We say that a linear transformation (MM) from functions to functions is translation-invariant if for every real number t_0, whenever the transformation transforms a function *x(s)* into a function *y(t)*, it also transforms a translated function $x(s - t_0)$ into the similarly translated function $y(t - t_0)$.

3. In signal processing, a linear transformation (MM) is translation-invariant if and only if the corresponding kernel *c(t, s)* has the form $A(t - s)$ for some function *A(t)*. The transformation (MM) then becomes $y(t) = \int A(t - s) \cdot x(s)\, ds$.