APPLYING DEEP BELIEF NETWORKS TO THE GAME OF GO

A Capstone Experience Manuscript

Presented by

Peter Krafft

Completion Date:
May 2010

Approved by:

Professor Andrew Barto, Computer Science

George Konidaris, Computer Science

Abstract

Title: **Applying Deep Belief Networks in the Game of Go**
Author: **Peter Krafft, Mathematics and Statistics**
CE Type: **Independent Capstone Thesis**
Approved By: **Andrew Barto, Computer Science Dept**
Approved By: **George Konidaris, Computer Science Dept**

The game of Go poses an incredible challenge to the field of artificial intelligence. Whereas many games are now dominated by computers, most computer Go agents have met only limited success in competition with world-class human players. There is a large and growing body of research in the field of computer Go, but the path forward is still unclear. One avenue that researchers have not explored extensively is exploiting the hierarchical structure that is clearly present in Go. We hope to exploit this structure by using a deep architecture to find transformations that will capture hierarchical interactions between pieces and yield illuminating representations of board positions.

Recent breakthroughs in how to train deep architectures have made finding distributed, hierarchical representations tractable. We use these techniques to train one particular type of deep architecture, a deep belief network, to learn the distribution of Go boards that occur in expert gameplay. We also explore how different numbers of hidden nodes and different numbers of layers affect the performance of the model.

The deep belief network is constructed with layers of hidden variables that represent features of the data. The ultimate goal of this research is to use the hierarchical features we extract from expert games to train a reinforcement learning agent to play Go. Using these extracted features is more principled than choosing arbitrary features and less time intensive than hand-picking features. Though we do not complete the final step in this research here, we use the similar task of classifying the winner in a final board state as a proxy.

Abstract

Title: **Applying Deep Belief Networks in the Game of Go**
Author: **Peter Krafft, Mathematics**
CE Type: **Independent Capstone Thesis**
Approved By: **Andrew Barto, Computer Science Dept**
Approved By: **George Konidaris, Computer Science Dept**

The game of Go poses an incredible challenge to the field of artificial intelligence. Whereas many games are now dominated by computers, most computer Go agents have met only limited success in competition with world-class human players. There is a large and growing body of research in the field of computer Go, but the path forward is still unclear. One avenue that researchers have not explored extensively is exploiting the hierarchical structure that is clearly present in Go. We hope to exploit this structure by using a deep architecture to find transformations that will capture hierarchical interactions between pieces and yield illuminating representations of board positions.

Recent breakthroughs in how to train deep architectures have made finding distributed, hierarchical representations tractable. We use these techniques to train one particular type of deep architecture, a deep belief network, to learn the distribution of Go boards that occur in expert gameplay. We also explore how different numbers of hidden nodes and different numbers of layers affect the performance of the model.

The deep belief network is constructed with layers of hidden variables that represent features of the data. The ultimate goal of this research is to use the hierarchical features we extract from expert games to train a reinforcement learning agent to play Go. Using these extracted features is more principled than choosing arbitrary features and less time intensive than hand-picking features. Though we do not complete the final step in this research here, we use the similar task of classifying the winner in a final board state as a proxy.

Contents

Chapter 1

Introduction

The choice of basis functions in reinforcement learning is often time-consuming for a programmer. Our goal is to have a principled method for choosing a set of expressive basis functions to use in reinforcement learning.

One difficult problem for an AI researcher is to find problems that are hard enough that they can illustrate the advantage of their algorithms but easy enough so that they can actually be solved in a reasonable amount of time. Another difficulty in choosing problems is finding domains that exhibit characteristics that occur in practical applications. Using games as objects of research is often helpful because games satisfy these conditions.

1.1 Games as an Object of Study

Computer scientists have been interested in games as far back as Shannon [17] and Samuel [16]. Games are convenient as objects of study because they have simple, strict sets of rules, so their state spaces are well understood by human players and are easy to simulate. More important than their relative simplicity, though, is their difficulty. Games serve as benchmarks of human intelligence. Professional game players can make a living because the games they play are hard to master and because people

are interested enough in being good at games that they sponsor tournaments.

Of course, there are many hard problems that have greater economic or social importance than games, but writing a world-class chess algorithm is more feasible than, say, writing a program to predict the next economic meltdown. This is true simply because human players have shown that the former may be possible. That is, few humans can do much better than guess at the occurrence of major economic changes, so we have no reason to believe that a computer should be able to do it, but some humans are very good at chess—far better than beginners, who largely play at random. Games thus straddle the fine line between problems that are difficult enough to be interesting, yet easy enough to be approachable.

Since games reside on this line, they are in a unique position to test new algorithms and practices. Proving a technique successful in a game is a strong indication that the technique will be useful in other applications. TD-Gammon helped popularize reinforcement learning in the AI community. Deep Blue displayed the power of various search techniques. Watson, IBM's current project that will take on *Jeopardy*, promises to be a triumph for some information retrieval methods. In Go, we will discover how to learn policies in massive discrete state spaces.

1.2 The Problem of Go

Go is a particularly interesting problem for a few reasons. Most obviously, the state space is huge—there are $3^{19 \cdot 19} \approx 10^{172}$ distinct possible board states—and the game tree is correspondingly enormous [3]. This size would not be a problem if we had extremely effective search methods, but finding a good heuristic for evaluating positions in Go is uniquely difficult [14]. Finally, Go is distinctly visual. Players organize the stones on the board into distinct groups and learn to visually identify the influence of various groups. In particular, humans build a hierarchical view of the board that

programs have not been able to exploit. Because of these challenges, no computer has reached the level of a human professional on a full board.

Although these complications are certainly not independent, we will be primarily concerned with the last, with building a model that exploits the hierarchy apparent in Go. Therefore, we will treat Go as a microcosm, a particular instance of a wide variety of discrete problems that display hierarchical structure.

1.3 Deep Architectures

The concept of deep architecture is one manifestation of hierarchical design in unsupervised machine learning. Deep architectures are used to build hierarchies of features in a process called *deep learning*. We will try to mirror the hierarchical properties of Go with the hierarchical design of a deep architecture.

1.4 Overview of this Thesis

This thesis explores the application of one technique, deep belief networks, in the domain of Go. The goal of this thesis is two-fold. First, we hope to use Go as an example to learn about how to deal with complicated, high-dimensional data. Second, we propose a new technique to improve current computer Go algorithms.

Chapter 2 is a brief summary of the rules of Go, basic strategy in Go, some previous computer Go techniques, and what part of Go we will be interested in. Chapter 3 covers the basic theory of deep belief networks. Chapter 4 explains the specific research questions we looked at and the implementation details of our methods. Chapter 5 describes the results of our experiments. Finally, Chapter 6 discusses our results, other related techniques, and future directions for this research.

Chapter 2

Overview of Go

Originally popular in East Asia, Go has now spread around the world, especially over the last century. Strategy in Go is complicated and fascinating, and just as in chess, many professional players devote their entire lives to the study of the game.

Language plays an important role in a human Go player's learning. Experienced players develop a huge vocabulary associated with the game, for specific patterns, for rules of thumb, and for general strategies. English speaking players borrow many of the Japanese terms. One interesting pattern is that players use very different language as they advance in rank (rank is very important in Go, and online Go servers keep close track of players' ratings). Besides using more specific words and knowing the names for more patterns, players develop more abstract representations of the game as they get better. They ascend a hierarchy of abstraction.

Despite its rich strategy, the rules of play in Go are exceedingly simple.

2.1 The Rules of Go

We present a basic overview of the rules of Go, only reviewing what is necessary to understand this thesis. For a more thorough treatment see, for example, *Sensei's Library* (`senseis.xmp.net`).

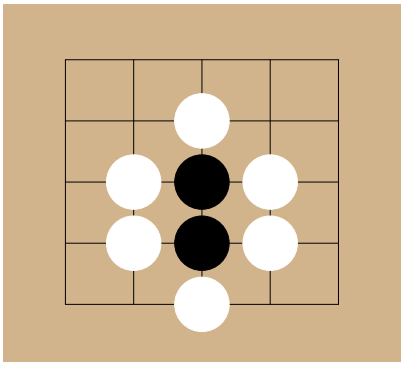

Figure 2.1: White has just played a stone atop black's two stones. White will now capture black's two stones.

Go is typically played on a square board, either 9x9 for short or practice games and 19x19 for full games, but any board size is possible (another common size is 13x13). The rules are as follows:

1. Two players alternate turns. One player uses white stones, the other black. The player using black stones plays first.
2. Each turn, a player must either place a stone on an empty space (an empty intersection on the board) or make no move (pass).
3. If a set of adjacent stones of the same color is completely surrounded by the opposite color, the opposite player captures those stones (see Figure 2.1).
4. The game ends after both players pass consecutively.

There are different methods for scoring the final position and determining a winner, but a player's score is typically the number of empty locations the player has surrounded (the amount of territory that player controls) plus the number of stones that player has captured.

2.2 Go Strategy

Go obviously has far too many nuances to explore here, but we will discuss some of the important aspects of Go strategy, as well as some relevant Go terminology.

The most basic element of gameplay in Go is simple tactics. A player must first learn how to take pieces, how to look ahead a small amount of moves, and how to identify the outcome of common tactical situations (such as traps). For example, two of the most common traps are called the ladder and net and are among the first examples shown in many Go books and tutorials. The only part of tactics that is important to us is its locality. Tactical fights usually take place in a small area on the Go board.

The most relevant aspects of Go strategy are the concepts related to groups. A group, just as it sounds, is a set of nearby, but not necessarily adjacent, allied stones. Single stones may have influence, but groups are ultimately what score you territory and capture your opponent's stones. Also unlike individual stones, groups have shape. Once again, the word is self-explanatory. The shape of a group of stones is simply the particular arrangement of the stones in that group.

A dedicated player quickly becomes familiar with common shapes because many shapes occur very frequently. The benefit of knowing about shapes is that a player can easily identify how easy it would be to protect or capture a particular group. A shape can be weak or strong, that is, the pieces in a shape can be easily taken or easily defended. Each player is interested in creating good shape for that player's stones and bad shape in the opponent's stones.

Shape is very important in this research because it indicates a certain amount of structure in Go games. Even though the state space of Go is enormous, the boards that actually occur in play only comprise a very limited subset of that space.

Figure 2.2 shows two common elementary shapes. The shape on the left, called a *hane at the head of two stones*, is a good shape for black because it severely limits white's territory, while increasing the potential for black's expansion. The shape on the right, the *ripped keima*, is a very bad shape for black. White has separated the black stones, making them unable to support each other. These shapes rarely occur

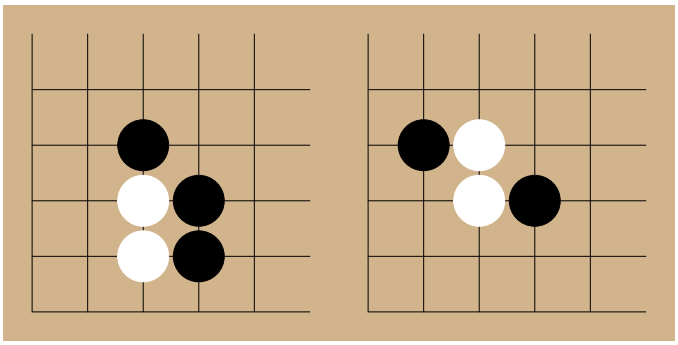

Figure 2.2: Here are two common shapes. The shape on the left is called a *hane at the head of two stones*. The shape on the right is called a *ripped keima*.

on their own, but they may appear in the context of a larger group.

The ideas of life and death naturally follow from the concept of shape. A group is alive if it cannot be taken (or at least if a reasonable player could keep it from being taken). The simplest example of life is a shape that entirely surrounds two separate spaces (each surrounded empty space is called an *eye*). Figure 2.3 shows this canonical example of life. This shape is alive because black would have to simultaneously play in both of the the surrounded spaces to capture any of the stones, which is impossible. A dead group is one that could be easily captured. The two black stones in Figure 2.1 were dead before the final white stone was place on their head.

Being able to recognize if a group is alive or dead is fundamental to good play and can be much more subtle than the illustrative examples we have shown. What is important to us is that life and death add another layer of abstraction. Whereas thinking in terms of groups embeds information about tactics, thinking in terms of life and death summarizes information about the interaction between groups. These concepts are further interesting because they are not explicit in the rules of the game: they are emergent.

Another aspect of Go strategy is the idea of joseki. A joseki is a string of moves that is known to have equal outcomes for both players. The important point about a joseki for us is that it will leave a group of stones with a certain shape. As with

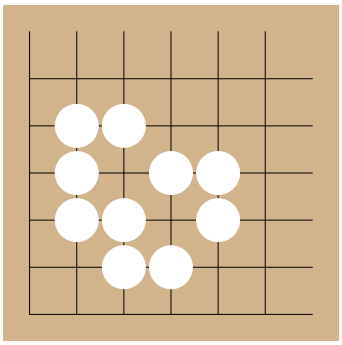

Figure 2.3: The white stones are alive as long as white doesn't fill one of its eyes.

knowledge of shape, joseki limit the space that human players have to search through when they are deciding how to move.

Players begin by thinking in terms of simple tactical situations. A player first learns how to look ahead a few moves, spot common traps, identify when to use some joseki, and understand other basic tactical situations. The player then combines these tools into knowledge of shape, which moves the player from thinking myopically to thinking in terms of the progression of the game. Next, the player learns how to interpret the interactions between shapes partly by thinking in terms of life and death.

A person learning the fundamentals of Go is thus climbing up a hierarchy of abstraction. At each level, the player learns how to predict more of the coming moves and how to integrate more of the current board. The progressing player changes from thinking locally to thinking globally.

Programmers have attempted to use these and other strategies in their computer Go agents, but they have met with only limited success.

2.3 Previous Computer Go

As is well known, there is no professional-strength computer Go agent. Regardless, computer Go has improved significantly in recent years, and some agents can now

play competitively against high rank humans on 9x9 boards. There are also many opportunities for computer agents to play against each other, perhaps the most famous being the Go sections of the computer Olympiad. We will review some of the most successful techniques in computer Go as well as some techniques that were less successful but relate to the topics at hand.

Currently, one of the most popular techniques, originally developed in the program MoGo by Gelly and Wang [8], is a search procedure called UCT. UCT is a tree-based extension of UCB1 that is used to improve the selection of moves in Monte-Carlo game simulations, which were applied to Go as early as 1993 [4]. UCT and UCB1 are approaches to the exploration-exploitation problem—how often to find new policies versus how often to exploit the policies you have found. In Go, UCT is used to efficiently choose paths to travel in a game tree. Gelly and Silver [7] recently used a modified version of UCT once again in MoGo to achieve master level play on the 9x9 board.

Although less successful, many computer Go programs have used reinforcement learning. Recently, researchers used reinforcement learning over a feature space of all possible combinations of stones in 1×1, 2×2, and 3×3 squares [18, 19]. With these million features, their agent learned a good position evaluation function without using any domain knowledge. Our hope is that with better features, this algorithm could perform very well.

Almost all Go programs use pattern databases [14], lookup tables that can be used to match subsets of the board with known successful responses to those positions. The pattern databases typically attempt to identify positions where a joseki or a well-known best move could be applied. With a few exception, these pattern databases are hand-coded. The idea of pattern databases is to effectively reduce the dimensionality of the search space by indicating what move is known to be effective in some contexts.

Our approach will attempt to reach the same goal as pattern databases in a very

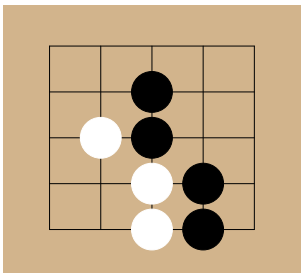

Figure 2.4: Two occurrences of the *hane at the head of two stones* that form a larger shape. Thinking of shapes in Go as being arranged hierarchically is useful.

different way. Instead of explicitly defining a searchable pattern database, we would like to learn a mapping function from boards to features that represent commonality between boards. Then by looking at the features activated by a certain position, we can know something about that position's relationship to other boards, for example its relative strength. After finding these features, we hope to use them in the context of reinforcement learning.

2.4 Hierarchical Views of Go

There are multiple ways to look at Go hierarchically. Foremost, the strategy is hierarchical. A good player must think in terms of tactical skirmishes, then in terms of shape, then in terms of life and death of groups, and finally in terms of influence over territory.

Go also displays a visual hierarchy. In the left panel of Figure 2.2 we showed a single shape, but this shape usually does not occur in isolation. On a real board it is combined hierarchically with other shapes. Figure 2.4 shows a more realistic situation for the *hane at the head of two stones*, where two of the same shape are built on top of each other. These two shapes combine to form a larger shape, which is a level up in the visual hierarchy.

Although a deep belief network has no guarantee of producing the hierarchical features we expect, since they are used as models of the human visual system, we

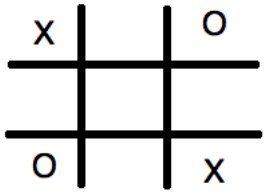

Figure 2.5: A position that is unlikely to occur in an expert game of tic-tac-toe.

would hope that they would at least be able to capture this second view.

2.5 The Distribution of Interest

As we observed previously, the board positions that occur in play show a surprising amount of regularity despite the large state space. This regularity allows the study of shape, joseki, and life and death to be useful. Also because of this regularity, human players are often easily able to see as many as 20 moves ahead [14]. That is, human players are often able to pick out what the best moves will be in many tactical situations. One explanation for this skill is that humans learn the distribution of board positions that occur in play. That is, they learn what is a likely move and what the board is likely to look like at a future stage.

Humans clearly integrate the previous moves in a game into their thinking about the current and future board positions in that game. Regardless, they still hone a notion of what is a usual or unusual appearance for the board. For a simpler example, consider tic-tac-toe. Any experienced tic-tac-toe player should know that the board in Figure 2.5 is unlikely to occur in play, even without knowledge about the previous moves in that game. Similarly, an experienced Go player develops a notion of the probability that a particular board would occur in play. Further, humans are never explicitly told what is or is not a likely position, so human players are learning this in an unsupervised fashion.

One of the difficulties in learning this distribution is that the position of a single

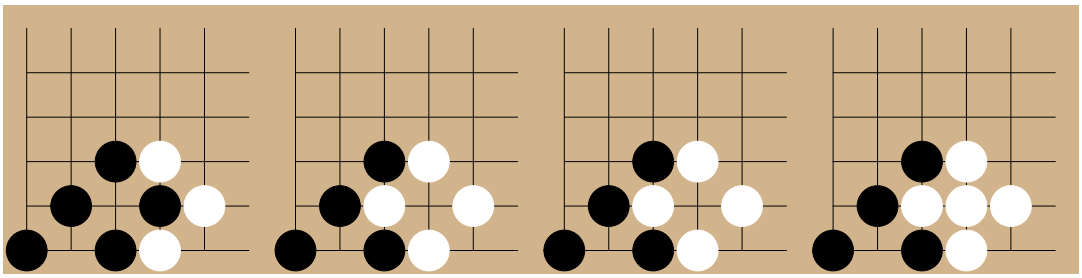

Figure 2.6: A *ko* fight. Black cannot respond in the third move by recapturing white, so she must play elsewhere. Because white ends the *ko* fight by filling his eye, that single move will have a large effect on how subsequent moves are played, making certain board that would have otherwise been likely, highly unlikely.

piece can drastically change the likelihood of a board. Formally, the distribution of Go boards that occur in play is highly varying. A highly varying function is one that would take a large number of linear pieces in a piecewise representation [2]. The difficulty of a highly varying function is that locations that are close in the input space may map to very different locations in the output space. We will not prove that the distribution of boards that occur in play is a highly varying function, but we will give an illustrative example to motivate it.

Figure 2.6 shows a corner of a board where a *ko* fight is happening. One rule we omitted before is the rule of *ko*, which says that a player cannot place a stone that will leave the board in the position it was in during that player's last turn. A result of this rule is that the players in Figure 2.6 cannot go back and forth taking each others' stone indefinitely. In particular, black cannot recapture white in the third move, so she must play elsewhere on the board.

Presumably, knowledge of this distribution helps humans prune the game tree that they have to search through when they are deciding how to move. We hope to learn and exploit this same distribution. Because of the highly varying nature of the distribution, and because of Go's clear hierarchical structure, we chose to use a deep architecture. In particular, we will also use an unsupervised method, a deep belief network.

Chapter 3

Overview of Deep Belief Networks

Graphical models can be used to represent joint probability distributions, where the nodes in the graph represent random variables and the edges represent dependence between those variables (for a brief but helpful description of graphical models see [15]). Typically, the overall structure of these dependencies (the graph topology) is known, but their magnitude must be learned. Unfortunately, this learning can often be time-consuming or intractable. In addition, some graphical models, single-layer neural networks for example, cannot express complicated functions.

The special structure of deep architectures alleviates the second problem, and Hinton's unsupervised greedy layer-wise training algorithm [11] alleviates the first.

3.1 Deep Architectures

Layered architectures stratify the variables in a model. Typically, the bottom layer is the visible layer, containing the observable data variables, and the higher layer are hidden layers, representing latent variables—features present but not explicit in the data. A typical layered neural network is shown in Figure 3.1. A deep architecture is simply a layered architecture that has many layers.

We are interested in two benefits that deep architectures provide. Foremost, they

build a hierarchical abstraction of the data. Each subsequent hidden layer in a deep architecture represents new information about the data by performing a transformation of the previous layer. Furthermore, these transformations are trained to represent some regularity in the data, so each higher layer should extract a higher order regularity.

Additionally, deep architectures are more expressive than many other models. Deep architectures can represent more functions than these models because they perform non-local generalization [2]. The most intuitive example of strictly local generalization is in a nearest neighbor algorithm. To find the classification of some new observation, the algorithm looks at the classification of training examples that were closest to that observation in the input space. The definition of local extends past the nearest neighbors to any method that only exploits training examples in some connected area around a new observation. A surprising number of methods use local generalization, including support vector machines, Gaussian processes, and kernel PCA [2].

Local generalization is sometimes unsuitable because it poorly characterizes the behavior of highly varying functions, so deeper architectures are desirable in those cases. Of course, deep architectures are harder to train than shallower models.

3.2 Deep Belief Networks

Deep belief networks (DBNs) are a popular type of deep architecture. The DBN is a multilayer neural network—a graphical model with the data variables in the first layer (the visible layer), connections from each of those variables to latent variables (also called hidden units or hidden nodes) in the layer above it, connections from those latent variables to another layer of hidden units, etc (See Figure 3.1)—with a specific structure.

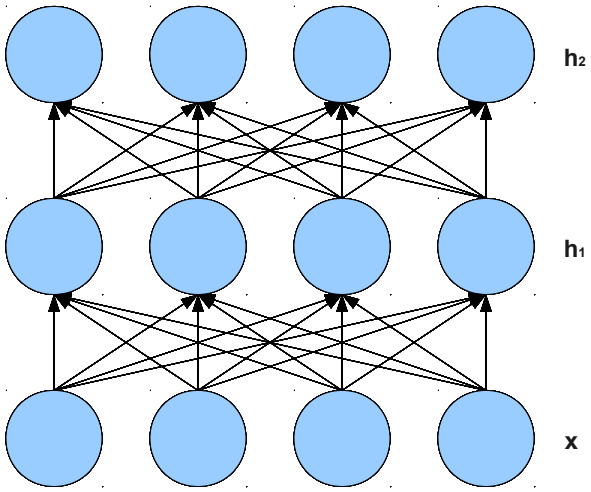

Figure 3.1: A three layer neural network with one visible layer and two hidden layers. The second hidden layer is the output layer. The output of this network is a vector.

DBNs build a layered abstraction, where each higher layer represents a higher order abstraction. In the DBN we use, each unit in the lowest hidden layer is a logistic transformation of a linear combination of the observable data. Each unit is trained to remember that a certain set of units are frequently simultaneously activated in the visible layer, so that each remembers a certain pattern of activation in the visible layer. A higher layer remembers the same type of interactions between units in the previous hidden layer it receives activation from, so each higher layer represents common combinations of common sets of activation.

Deep belief networks are convenient to use because their structure affords a fast unsupervised training algorithm. Additionally, that algorithm usually finds good values for the parameters of the model if no further training is performed, and it finds very good initial parameters for subsequent supervised learning [2, 11].

Because of the difficulties in training fully connected graphs, DBNs use restricted Boltzmann machines (RBMs) as their basis. RBMs are convenient because they are easy to sample from but they are still expressive. During the unsupervised training phase a DBN is equivalent to a stack of restricted Boltzmann machines, where the

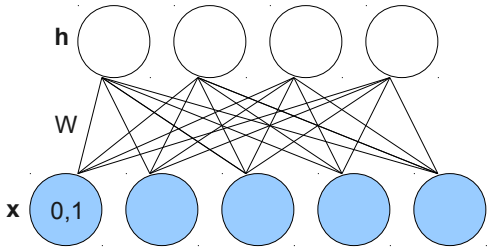

Figure 3.2: A simple restricted Boltzmann machine. The blue units (the vector $\mathbf{x}$) are the visible units, here assumed to be binary. The white units, $\mathbf{h}$, are the hidden units. The activation of some unit in $\mathbf{h}$ would be $\frac{1}{1+\exp(-\mathbf{x}W_{i\cdot} - c_i)}$, where c_i is an intercept term called the bias of the unit i.

hidden layer of each RBM forms the visible layer of the next.

3.2.1 Restricted Boltzmann Machines

Restricted Boltzmann machines (RBMs), originally formulated by Smolenski [20], consist of two layers of random variables, a visible layer and a hidden layer. Each of the units in the hidden layer is a transformation of the visible layer, and each unit in the visible layer is a transformation of the units in the hidden layer. We treat all units as binary random variables with the probability of activation being a logistic transformation of a linear combination of the other layer's units. Ours is the traditional choice, but the units can be taken from any exponential family of distributions [24].

The structure of the two layers is very important. In an RBM, the visible units are conditionally independent given the hidden units, and vice versa, so that both conditional distributions are tractable. A small RBM is shown in Figure 3.2.

The joint density of the distribution represented by an RBM is described in terms of energy. In general, the probability of the variables in an energy-based model having a certain configuration $\mathbf{y}$ is

$$P(\mathbf{y}) = \frac{\exp(-E(\mathbf{y}))}{\sum_{\mathbf{y}} \exp(-E(\mathbf{y}))},$$

where E is the energy function associated with the model. Configurations that have a lower energy are more likely to occur under the model. The denominator is a normalizing term called the partition function and is typically intractable to compute (for binary units, it requires computing the energy of every possible combination of the binary activations).

To write the joint distribution associated with the RBM, we partition the variables $\mathbf{y}$ into an $1 \times v$ vector of visible units from the first layer, $\mathbf{x}$, and an $1 \times h$ vector of hidden units from the second layer, $\mathbf{h}$. The parameters in the model are the weights between the visible and hidden units held in the $v \times h$ weight matrix W, and intercept terms for each visible unit and each hidden unit, a $1 \times n$ vector $\mathbf{b}$ and an $1 \times h$ vector $\mathbf{c}$, respectively. W is $v \times h$ rather than $(v + h) \times (v + h)$ because the connections between the units must be symmetric (so if a data variable is likely to activate a certain feature, that feature will in turn be correspondingly likely to activate the variable). Given this notation, the energy of the RBM is

$$E(\mathbf{x}, \mathbf{h}) = -\mathbf{b}\mathbf{x}' - \mathbf{c}\mathbf{h}' - \mathbf{h}W'\mathbf{x}'.$$

Due to the convenient linear form the energy function takes, we can avoid the partition function and factor the conditional distributions (see Bengio [2] for the algebra)

$$P(\mathbf{y}|\mathbf{h}) = \prod_i P(y_i|\mathbf{h}) \text{ and}$$
$$P(\mathbf{h}|\mathbf{y}) = \prod_i P(h_i|\mathbf{y}),$$

which is the entire point of using an RBM. In our case, using logistic units,

$$P(y_i|\mathbf{h}) = \frac{1}{1+\exp(-\mathbf{h}W_{\cdot i} - b_i)} \text{ and}$$
$$P(h_i|\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{x}W_{i\cdot} - c_i)},$$

where $W_{\cdot i}$ is the i^{th} column of W and $W_{i\cdot}$ is the i^{th} row.

Using the structure of a logistic RBM indicates that we are interested in some latent binary features associated with the data. These latent variables are meant to capture simultaneous activation patterns of the visible units, but this behavior is not apparent from the description of the joint density; it appears primarily in the learning rule for the RBM.

3.2.2 Training a Restricted Boltzmann Machine

The training rule for a logistic restricted Boltzmann machine is based on maximizing the probability of the observed data. This maximization is achieved by gradient descent along the derivative of the log probability of the data taken with respect to the weights (for a derivation see Hinton [11]). Before writing the gradient, it is useful to explain sampling from the RBM, because the gradient can be written concisely in terms of those samples.

We would like to sample from the joint distribution of the model, but it is difficult to compute exactly because of the partition function. Luckily we have simple equations for the conditional distributions $P(\mathbf{x}|\mathbf{h})$ and $P(\mathbf{h}|\mathbf{x})$, so we can easily perform block Gibbs sampling to obtain the samples we need.

To sample from the joint density we begin with the a data vector, sample $\mathbf{h}$ from its conditional distribution, then sample $\mathbf{x}$ from its conditional distribution given the $\mathbf{h}$ we just sampled, then continue this pattern indefinitely. At iteration infinity, we will be sampling directly from the joint density. Hinton calls a sample at that point

$(\mathbf{x}^\infty, \mathbf{h}^\infty)$ and the first sample (the data) $(\mathbf{x}^0, \mathbf{h}^0)$ [11]. The partial derivative of the log probability of a data vector with respect to the weight between a visible unit and a hidden unit, from which we get a learning rule, is the expected percent of the time that the hidden unit and the visible unit are on together given this data vector minus the expected percent of the time that the same two units are on together in the samples of the joint density,

$$\frac{\partial \log P(\mathbf{x}^0)}{\partial W_{ij}} = E[x_i^0 h_j^0] - E[x_i^\infty h_j^\infty],$$

and the learning rule that it implies is

$$\Delta W_{ij} = \delta(E[x_i^0 h_j^0] - E[x_i^\infty h_j^\infty]), \tag{3.1}$$

for some learning rate δ. This is a Hebbian learning rule. For example an extreme example, if $x_i^0 = 1$ and $P(h_i|\mathbf{x}) = 1$, then the first term will be exactly 1, so that the units always fire together with this data vector. Then if the units are ever not activated together in model distribution, the connection between the units will strengthen. On the other hand, if $x_i^0 = 0$, or if $P(h_i|\mathbf{x})$ is small, then the connection will tend to weaken unless the model distribution already knows the connection is weak. We will adopt Hinton's terminology in calling these products the correlation between the units, though it is a misnomer since $(x_i^0, h_j^0) = (1, 0)$ has the same effect as $(x_i^0, h_j^0) = (0, 0)$.

Equation 3.1 is the full learning rule and guarantees us that the log probability of the model will never decrease by adding another layer [11], but we do not have time to take an infinite number of samples from the model distribution. Contrastive divergence gives us a fast approximate alternative [10]. The idea of contrastive divergence is simple. Rather than taking an infinite number of samples, we truncate the sampling at a certain point. Although we can choose any value for this truncation, the typical

choice is one-step contrastive divergence, where only one sample is drawn from the model distribution. In one-step contrastive divergence the learning rule becomes

$$\Delta W_{ij} = \delta(E[x_i^0 h_j^0] - E[x_i^1 h_j^1]).$$

This learning rule has the interpretation of being the correlation of the two units in the draw from the data distribution minus the correlation of the two units in the reconstruction according to the model. The reconstruction interpretation comes from the fact that we sample $\mathbf{x}$ directly from the hidden units that are activated from the data. Contrastive divergence voids the theoretical guarantees of improvement but works well regardless [1, 5, 10].

The training of an RBM model attempts to find some set of features, i.e., some set of transformations of the data, that are able to exactly reconstruct the data. When the number of hidden units is less than the number of visible units, these features must represent statistical regularity within the data or suffer in the reconstruction error. By using random initialization of the weights and weight penalties, we can likely capture that same regularity even if the dimension of the data is not reduced. By stacking RBMs on top of one another, we can iteratively learn deeper regularities.

3.3 Forming a Deep Belief Network

Hinton developed an efficient algorithm for training deep belief networks. The idea is to stack restricted Boltzmann machines on top of one another, so the output of each machine is the input to another one. The basic structure is shown in Figure 3.3.

To train the network, we can use a greedy layer-wise training algorithm [11]. At each layer, the training is equivalent to training a restricted Boltzmann machine. Because we can use contrastive divergence, the training is very fast. At the same time, the deep structure allows the DBN to form a distributed representation of the

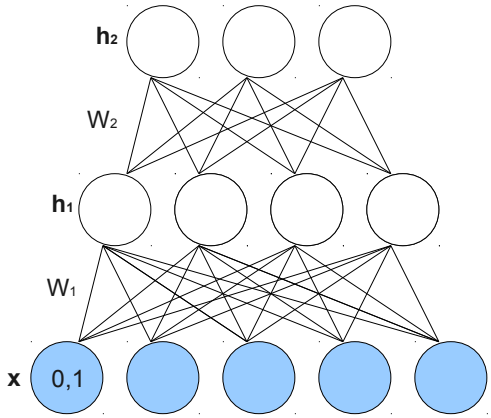

Figure 3.3: A deep belief network. Each pair of layers is trained as an RBM. The network can be unrolled into a directed model similar in structure to the network in Figure 3.1.

data distribution that builds higher order features at higher layers by combining the features of previous layers.

This algorithm is particularly useful as pretraining for a directed model such as the neural network in Figure 3.1. To get a model such as the one in Figure 3.1, we could relax the restriction that the weight matrices be symmetric so that the network can have separate recognition and generation weights. Taking the network in Figure 3.3 for example, the unrolled network would then have five layers. The fourth layer would be a new hidden layer with three units and initial weights W_2'. The fifth layer would be a hidden layer with five units and initial weights W_1'. All the connections in this new network would be directed upwards. With this unrolled network, we can then use a neural network gradient descent training rule to train the directed model, using the difference between the input and the output (the reconstruction) as an error signal. In this fine-tuning phase, we would use the weights learned from the unsupervised training as initial values that act as a prior to restrict the parameter space to an area close to an optimal solution [2].

We will not use this fine-tuning technique, but the results of the unsupervised learning phase are still good without it [11].

Chapter 4

Research Methodology

The goals of this research are to show that deep belief networks are able to represent the distribution of boards that occur in expert gameplay and to find the network topology that optimizes that representation. We can then use the results of this research to improve upon the approaches used in previous computer Go by providing a hierarchical representation of the Go board.

4.1 Implementation

We represented each space on the Go board with three units, one to indicate that the space should be black, one to indicate white, and one to indicate empty. We chose to include empty as a separate unit rather than as the value of the intercept for two reasons: first, we believe that empty squares are meaningful and thus should contribute directly the net's inferences, and second, zeroing out the board has a cleaner interpretation when all the units associated with a space being off indicates a lack of knowledge about the space, rather than that the space is empty.

Each dataset consists of some number of board positions in the representation described above. We downloaded hundreds of thousands of 9x9 games from the Computer Go Server archives (`http://cgos.boardspace.net/`), far more data than

we could use in our experiments, and converted each .sgf file into matrix format, where one data vector corresponds to a particular turn in a particular game.

The code for our DBN was based on code from Hinton and Salakhutdinov's *Science* supplement [12]. Although we changed much of the code, the learning parameters they used (learning rate for the weights of 0.1, learning rate for the visible biases of 0.1, learning rate for the hidden biases 0.1, weight penalty of 0.0002, initial momentum of 0.5, and final momentum of 0.9) suited our needs. We did not perform cross-validation to optimize those weights because our networks appeared to converge, and we had enough other parameters to optimize.

Also as in Hinton and Salakhutdinov [12], we trained each network in batches of 100 boards. Using batches saves on computation time because we only have to compute the weight updates every hundred data points instead of every single data point. We use the batch implementation at every layer, transforming each batch into the features of the next layer with the weights on the current layer.

Unlike Hinton and Salakhutdinov, we also use hectobatches to save on space. We only keep 10,000 data points (100 batches) loaded in memory at any one time. For large nets, this reduces memory consumption from multiple gigabytes to a few hundred megabytes.

4.2 Research Questions

At this stage of our research, we are primarily concerned with showing that deep belief networks can learn a representation of the distribution of boards that occur in expert gameplay and, after showing that, finding the network topology that provides the best representation. We are also interested in whether the network builds a sensible hierarchy that appeals to human intuition and if a network forms a useful representation, one that characterizes interactions that make regression or classification easier.

To test how well the network is representing the distribution of interest, we use two different quantitative metrics. Unfortunately, calculating the likelihood of a certain dataset is intractable for a network with more than one hidden layer, so we have to use *reconstruction* and *pattern completion*.

Reconstruction is the most natural metric since the network is trained on squared reconstruction error. To test the reconstruction error of a network, we propagate the data up to the top of the network, then propagate the top level representation back down, and compare the reconstructed data to the original. We can either use a single iteration of this process, or we can use the reconstructed data as probabilities, randomly select the binary states of the visible layer using those probabilities, then take multiple iterations to achieve a Gibbs sample.

Our second metric is based on pattern completion, similar to the completion of images in Lee et al. [13]. Instead of zeroing out half of a board, we zero out random positions on the board (each position has a 10% chance of being corrupted). We then perform reconstruction as before using the corrupted data and compare that reconstruction with the original data. The motivation for using completion is that, for example, if there is a common shape that has one position zeroed out, the network should learn the common value of that position.

We optimize over two variables: the number of hidden units and the number of hidden layers. To make the problem easier, we assume that each successive hidden layer has half as many units as the last layer. We test 7 different sizes, either 500, 1000, 1500, 2000, 2500, 3000, 3500 units in the bottom layer, and 4 different depths.

Once we decide on a network to use based on these two metrics, we make some subjective assessments of the network. We generate boards from the distribution represented by the network by taking a random vector at the top layer, performing Gibbs sampling with just the layer below it then propagating a sample from the invariant distribution down the network. We then observe whether the samples look

similar to real games. We also examine the effect of activating individual features by clamping a particular feature and performing Gibbs sampling on the layer containing that feature and the layer below it, then generating data with one of those samples.

We also test whether the representation provided by a particular network is useful for classification. We train the network to perform classification by adding two additional units (one corresponding to white winning and one corresponding to black winning) at a particular layer, then training the weights between that layer and the layer above it. We use the RBM training rule and the weights from the previous learning stage as initial parameters. (Alternatively we could clamp the values of the weight matrices we have already found, then performing the previous training method updating only the weights to the two additional units.) Also, we do not infer the units from below, we only generate them from above so the model on the lower layers is not affected. As training data, we only use final board positions, and we include the winner of the game as input into the two additional units (one for black winning and one for white winning). Once again, we use two units instead of one so we can have a sensible interpretation of zeroing both out during testing. We try this training at various layers. To evaluate the classification, we zero out the units and test whether the reconstruction of the units agrees with the original values. To decide which player the network intends to classify as the winner, we take the maximum of the activations of the two units. This modification is better than looking at the raw activations because one player might win by a very small amount.

We will not test the usefulness of the network in regression until we integrate the reinforcement learning approach in future work.

Chapter 5

Results

Figure 5.1 shows the reconstruction error for networks with between one and four layers and between 500 and 3500 units on the bottom layer. Each additional layer has half as many units as the last, and all the layers in this plot use 50000 training examples (about five hundred games). Interestingly, networks with one or two layers consistently perform better than networks with more layers and all perform about the same (the network with the least error is the single-layer 3500-unit network, but it only wins by a a small amount). This effect probably occurs for two reasons. First, halving the numbers of units in each layer affects the bigger networks more because it is hard for the next layer to recover all of the information from the layer below it. Second, adding a layer to the network without performing fine-tuning should make the reconstruction worse, because each additional layer loses some of the information from the last layer.

Considering the performance of all the layers, using a network with between 2000 and 3000 units at the lowest layer looks optimal. For these networks, all of the layers have low reconstruction error.

Figure 5.2 shows the results of the completion task. Instead of using squared error, we used the percent of incorrect guesses for the zeroed-out units. Here most of the

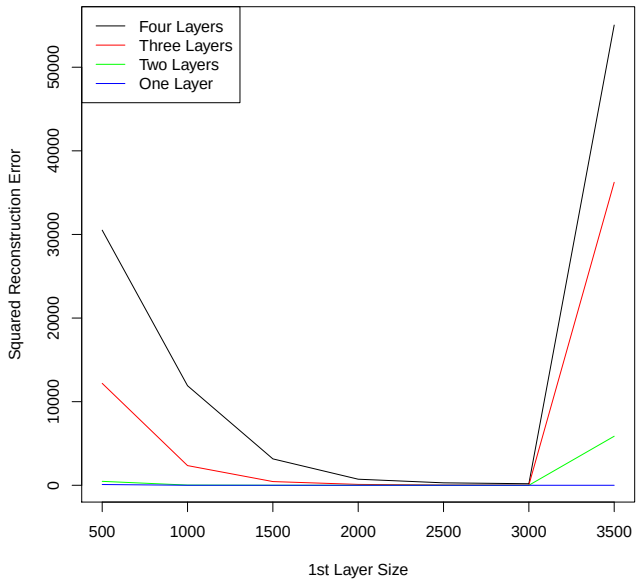

Figure 5.1: Squared error on the reconstruction task.

networks perform about the same: they all get about 22% incorrect (out of 1000). Purely guessing would get about 66% incorrect on average, so this number is good. Interestingly, the networks with more layers perform slightly better on this task. One optimistic interpretation of that noticeable increase in performance is that the higher layers are good enough at extracting important predictive information that they can compensate for their deficiency at reconstruction.

The results were once again different in the classification task (shown in Figure 5.3). Here we used only sizes between 500 and 2500, but the trend is clear. The networks with two layers perform much better than all the other networks. Of these three tasks, classification is the most similar to what we will expect from the network in reinforcement learning, so these results are especially significant. Here the network with 2500 units in the bottom layer and two layers performs the best with about 13.8% incorrect out of the 1000 trials (compare to random guessing at 50%), which suggests that the second layer is learning an important representation of the data, but that representation may be lost at the higher levels.

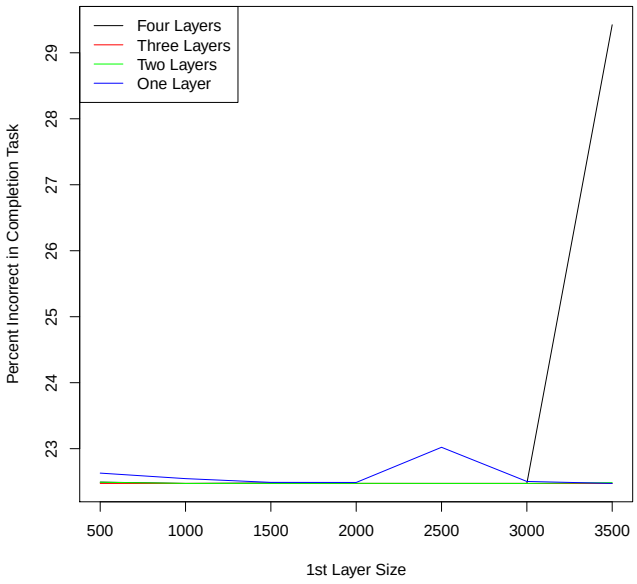

Figure 5.2: Percent of incorrect guesses on the completion task.

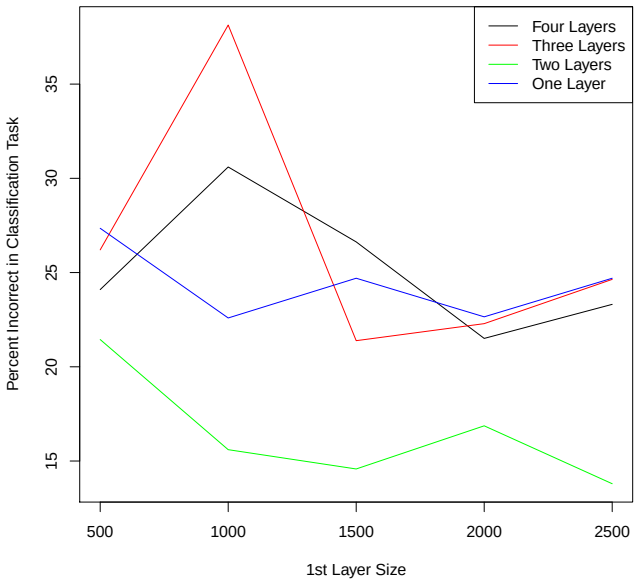

Figure 5.3: Percent incorrect classifications while classifying the winner of a game.

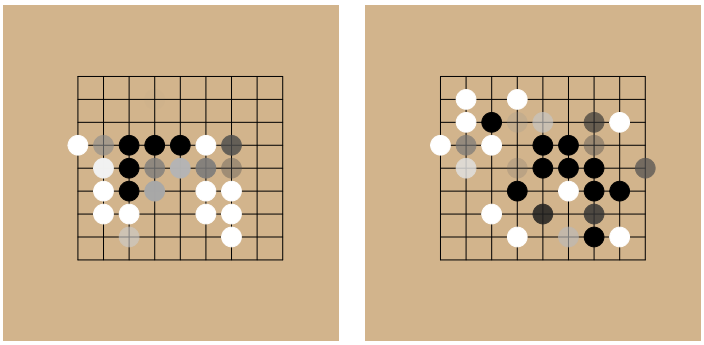

Figure 5.4: Boards drawn from the distribution of a network with a single 1000-unit hidden layer.

Besides these quantitative measurements, we were also interested in whether the network can produce subjectively reasonable results. Previous researchers have used Hinton diagrams to visualize the activations of individual hidden units. To generate a Hinton diagram, you simply activate a single unit in a hidden layer, then propagate the activation of that single unit down the network. A better way to do this is to perform Gibbs sampling with that unit held constant to get an actual sample from the distribution that would have that unit activated. The images produced by Hinton diagrams can be deceiving, though, because the hidden features are not additive [9], so we prefer not to use this method.

Instead, we draw samples from the generative model represented by the network. These samples should be interpreted as boards that the model thinks are likely to occur. Figure 5.4 shows two representative boards that we generated from a network with one 1000-unit hidden layer. A darker stone indicates the network thinks that stone is more likely to be black, a lighter stone indicates the white. A lower transparency indicates the stone is less likely to be present. These boards do not look very realistic. The board on the left looks more like a part of a game, and the board on the right has too much uncertainty (all the stones in a real game are purely black or white and completely opaque).

On the other hand, boards generated from a two layer network (shown in Figure

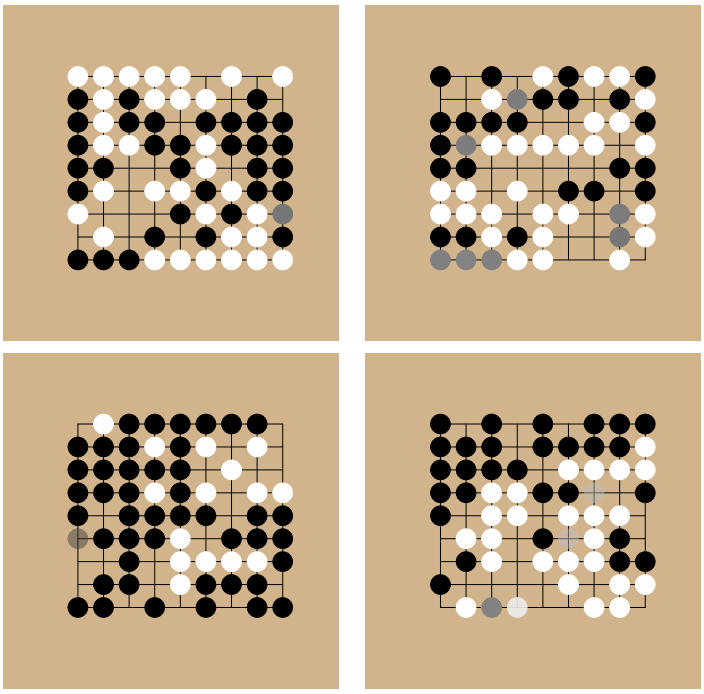

Figure 5.5: Boards drawn from the distribution of a two layer network with 1000 hidden units in the first layer and 500 hidden units in the second layer.

5.5 look much better. Besides a few strange occurrences of untaken surrounded stones, these boards look fairly realistic.

Strangely, boards generated from a three layer network look similar to those generated by the single layer network, and boards generated by a four layer network look similar in structure to those generated by the two layer network, though with more uncertainty (see Figure 5.6). Perhaps the third layer is just recapturing the patterns that the second layer was generated from, and the fourth is doing something similar from the third.

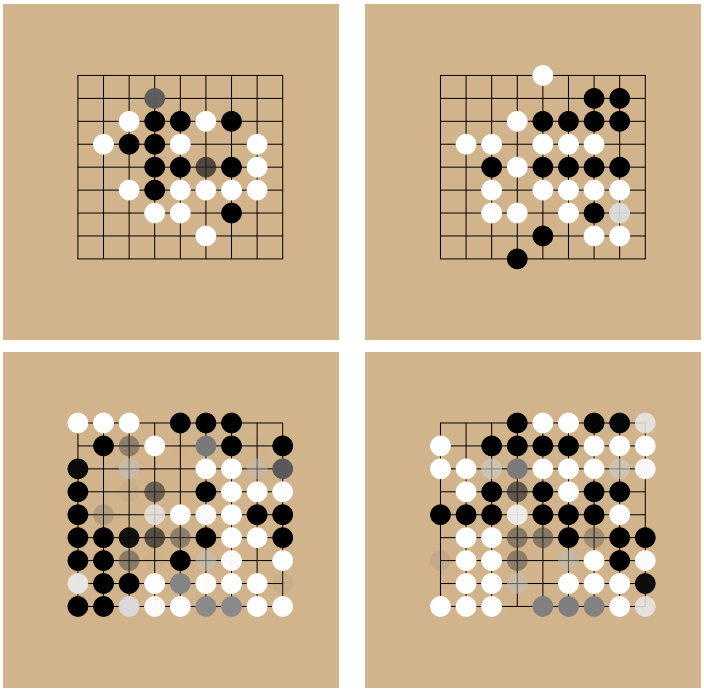

Figure 5.6: Top row: boards drawn from the distribution of a three layer, 1000-500-250 network. Bottom row: boards generated from a four layer, 1000-500-250-125 network.

Chapter 6

Conclusions

Our two most important metrics, classification and subjective assessment, point to networks with two hidden layers as being optimal for this problem. In the quantitative tasks, looking at the results for networks with two hidden layers, any network with more than 1000 units in the first hidden layer has about equal performance, so using a network with 1000 units may be best.

The network also appears to be hierarchically arranged for at least the first two layers. The boards sampled from the the network containing just the first hidden layer looked more like parts of games, while the boards generated from the second layer looked more like full games. Unfortunately, the network does not seem to represent any hierarchical representation similar to what humans use, but it still proved useful in the classification task, where the best network got a good score of about 86% correct.

Of course, these results could change drastically if we perform fine-tuning that integrates the layers in the network. There are also many other related methods we could use to improve the model.

6.1 Other Related Techniques

Fast persistent contrastive divergence [23] is a new technique to improve the performance of contrastive divergence. There are also simpler tricks involving learning weight decay schedules that are commonly used to improve the speed of learning. Convolutional deep belief networks [6, 13] are used to scale deep belief networks to larger visible layers, which may be useful for scaling up to 19x19 Go game.

One major weakness of our current model is that it does not account for the history of a game. The network has no concept of sequence. Researchers recently proposed two related models for modeling time series data, the conditional restricted Boltzmann machine and the factored conditional restricted Boltzmann machine [21, 22].

6.2 Future Research

We would like to be able to view deep learning as a transformation that succinctly characterizes the covariance in the data. Ideally, we should be able to treat position evaluation as a regression problem and use the transformed data to avoid correlation in the residuals. We have proposed reinforcement learning as a method for training this regression.

The fundamental problem with using a traditional DBN is that the features are not portable. The most important step in improving these models will be condensing and generalizing features so that features learned in one area of the state space, or even in one problem, can be used in another area or another problem.

In the meantime, we hope to scale these method to 19x19 boards and to use the features learned from our best network in a reinforcement learning algorithm for learning the concept of shape.

Bibliography

[1] Yoshua Bengio. Justifying and generalizing contrastive divergence. *Neural Computation*, 21(6):1601–1621, 2009.

[2] Yoshua Bengio. Learning deep architecture for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[3] Bruno Bouzy and Tristan Cazenave. Computer Go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[4] Bernd Brügmann. Monte Carlo Go, 1993.

[5] Miguel Carreira-Perpinàn and Geoffrey Hinton. On contrastive divergence learning. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*, pages 33–40. Society for Artificial Intelligence and Statistics, 2005.

[6] Guillaume Desjardins and Yoshua Bengio. Empirical evaluation of convolutional RBMs for vision. Technical report, Université de Montréal, 2008.

[7] Sylvain Gelly and David Silver. Achieving master level play in 9x9 computer Go. In *Proceedings of the* 23^{rd} *National Conference on Artificial Intelligence*, volume 3, 2008.

[8] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *Neural Information Processing Systems Conference On-line trading of Exploration Exploitation Workshop*, 2006.

[9] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2 edition, 2009.

[10] Geoffrey Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14:2002, 2002.

[11] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[12] Geoffrey Hinton and Ruslan Salakhutdinov. The dimensionality of data with neural networks. *Science*, 313:504–507, 2006.

[13] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, volume 382 of *ACM International Conference Proceeding Series*, pages 609–616, 2009.

[14] Martin Müller. Computer Go. *Artificial Intelligence*, pages 145–179, 2002.

[15] Kevin Murphy. A brief introduction to graphical models and Bayesian networks. Online, 1998. `http://www.cs.ubc.ca/ murphyk/Bayes/bnintro.html`.

[16] Arthur Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, pages 210–229, 1959.

[17] Claude Shannon. Programming a computer for playing chess. *Philosophical Magazine*, page 1950, 1950.

[18] David Silver. *Reinforcement Learning and Simulation-Based Search*. PhD thesis, University of Alberta, 2009.

[19] David Silver, Richard Sutton, and Martin Müller. Reinforcement learning of local shape in the game of Go. In 20^{th} *International Joint Conference on Aritificial Intelligence*, 2007.

[20] Paul Smolenski. *Parallel Distributed Processing*, volume 1, chapter Information Processing in Dynamical Systems: Foundations of Harmony Theory, pages 194–281. MIT Press, 1986.

[21] Graham W. Taylor and Geoffrey E. Hinton. Factored conditional restricted Boltzmann machines for modeling motion style. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1025–1032, New York, NY, USA, 2009. ACM.

[22] Graham W. Taylor, Geoffrey E. Hinton, and Sam Roweis. Modeling human motion using binary latent variables. In *Advances in Neural Information Processing Systems*, page 2007. MIT Press, 2006.

[23] Tijmen Tieleman and Geoffrey Hinton. Using fast weights to improve persistent contrastive divergence. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1033–1040, New York, NY, USA, 2009. ACM.

[24] Max Welling, Michal Rosen-Zvi, and Geoffrey Hinton. Exponential family harmoniums with an application to information retrieval. *Advances in Neural Information Processing Systems*, 17, 2005.