Training Deep Convolutional Neural Networks to Play Go

Christopher Clark CHRISC@ALLENAI.ORG
Allen Institute for Artificial Intelligence*, 2157 N Northlake Way Suite 110, Seattle, WA 98103, USA

Amos Storkey A.STORKEY@ED.AC.UK
School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh, EH9 1DG, United Kingdom

Abstract

Mastering the game of Go has remained a long-standing challenge to the field of AI. Modern computer Go programs rely on processing millions of possible future positions to play well, but intuitively a stronger and more 'humanlike' way to play the game would be to rely on pattern recognition rather than brute force computation. Following this sentiment, we train deep convolutional neural networks to play Go by training them to predict the moves made by expert Go players. To solve this problem we introduce a number of novel techniques, including a method of tying weights in the network to 'hard code' symmetries that are expected to exist in the target function, and demonstrate in an ablation study they considerably improve performance. Our final networks are able to achieve move prediction accuracies of 41.1% and 44.4% on two different Go datasets, surpassing previous state of the art on this task by significant margins. Additionally, while previous move prediction systems have not yielded strong Go playing programs, we show that the networks trained in this work acquired high levels of skill. Our convolutional neural networks can consistently defeat the well known Go program GNU Go and win some games against state of the art Go playing program Fuego while using a fraction of the play time.

1. Introduction

Go is an ancient, deeply strategic board game that is notable for being one of the few board games where human experts are still comfortably ahead of computer programs in terms of skill. Predicting the moves made by expert players is an interesting and challenging machine learning task, and has immediate applications to computer Go. In this section we provide a brief overview of Go, previous work, and the motivation for our deep learning based approach.

*Work completed at the University of Edinburgh

Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37. Copyright 2015 by the author(s).

1.1. The Game of Go

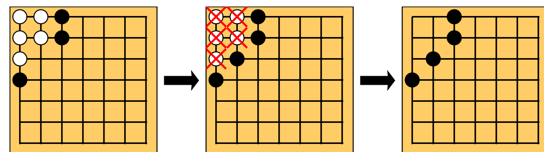


Figure 1. Capturing pieces in Go. Here white's stones in the upper left are connected to each other through adjacency so they form a single group (left panel). When black places a stone on the indicated grid point (middle panel) that group is surrounded, meaning there are no longer any empty grid points adjacent to it, so the entire group is removed from the board (right panel).

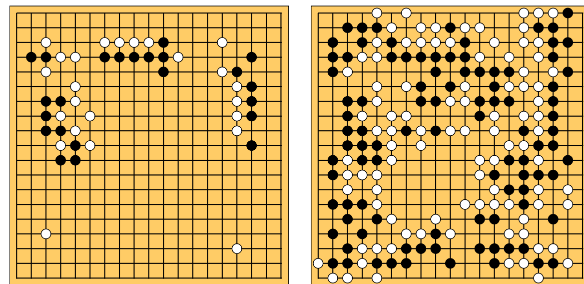


Figure 2. Example of positions from a game of Go after 50 moves have passed (left) and after 200 moves have passed (right). In the right panel it can be seen that white is gaining control of territory in the center and top of the board, while black is gaining influence over the left and right edges.

We give a very brief introduction to the rules of Go. We defer to (Bozulich, 1992) or (Müller, 2002) for a more comprehensive account of the rules. Go has a number of different rulesets that subtly differ as to when moves are illegal and how the game is scored, here we focus on generalities

that are common to all rulesets.

Go is a two-player game that is usually played on a 19x19 grid based board. The board typically starts empty. One player plays as white and one as black. White starts by placing a white stone on a grid point. Black then places a black stone on an unoccupied grid point, and play continues in this manner. Players can opt to pass instead of placing a stone, in which case their turn is skipped and their opponent may make a second move. Stones cannot be moved once they are placed, however a player can capture a group of their opponent's stones by surrounding that group with their own stones. In this case the surrounded group is removed from the board as shown in Figure 1. Broadly speaking, the objective of Go is to capture as many of the grid points on the board as possible by either occupying them or surrounding them with stones. The game is played until both players pass their turn, in which case the players come to an agreement about which player has control over which grid points and the game is scored.

Through the capturing mechanic it is possible to create infinite 'loops' in play as players repeatedly capture and replace the same pieces. Go rulesets include rules to prevent this from occurring. The simplest version of this rule is called the simple-ko rule, which states that players cannot play moves that would recreate the position that existed on their previous turn. Most Go rulesets contain stronger versions of this rule called super-ko rules, which prevent players from recreating any previously seen position. Figure 2 shows some example board positions from a game of Go.

State of the art computer Go programs such as Fuego (Enzenberger et al., 2010), Pachi (Baudiš & Gailly, 2012), or CrazyStone [1], can achieve the skill level of very strong amateur players, but are still behind professional level play. The difficulty computers have in this domain relative to other board games, such as chess, is often attributed to two things. First, in Go there are a very large number of possible moves. Players have $19 \times 19 = 361$ possible starting moves. As the board fills up the number of possible moves decreases, but can be expected to remain in the hundreds until late in the game. This is in contrast to chess where the number of possible moves might stay around fifty. Second, good heuristics for assessing which player is winning in Go have not been found. Counting the number of stones each player has is a poor indicator of who is winning, and it has proven to be difficult to build effective heuristics for estimating which player has the stronger position.

Current state of the art Go playing programs use Monte Carlo Tree Search (MCTS) algorithms. MCTS algorithms evaluate positions in Go using simulated 'playouts' where the game is played to completion from the current position assuming both players move randomly or follow some computationally cheap best move heuristic. Many playouts are carried out and it is then assumed good positions are ones where the program was the winner in the majority of them. See (Browne et al., 2012) for a recent survey of MCTS algorithms and (Rimmel et al., 2010) for a survey of some modern Go playing systems.

[1] http://remi.coulom.free.fr/CrazyStone/

1.2. Move Prediction in Go

Human Go experts rely heavily on pattern recognition when playing Go. Expert players can gain strong intuitions about what parts of the board will fall under whose control and what are the best moves to consider at a glance, and without needing to mentally simulate possible future positions. This is in contrast to typical computer Go algorithms, which simulate thousands of possible future positions and make minimal use of pattern recognition. This gives us reason to think that developing pattern recognition algorithms for Go might be the missing element needed to close the performance gap between computers and humans. In particular for Go, pattern recognition systems could provide ways to combat the high branching factor by making it possible to prune out many of the possible moves in the current position.

Outside of playing Go, move prediction is an interesting machine learning task in its own right. We expect the target function to be highly complex, since it is fair to assume human experts think in complex, non-linear ways when they choose moves. We also expect the target function to be non-smooth because a minor change to a position in Go (such as adding or removing a single stone) could be expected to dramatically alter which moves are likely to be played next. These properties make this learning task very challenging, however it has been argued that acquiring an ability to learn complex, non-smooth functions is of particular importance when it comes to solving AI (Bengio, 2009). These properties have also motivated us to attempt a deep learning approach, as it has been argued that deep learning is well suited to learning complex, non-smooth functions (Bengio, 2009; Bengio & LeCun, 2007). Move prediction for Go also provides an opportunity to test the abilities of deep learning on a domain that has a close connection to AI.

1.3. Previous Work

Previous work in move prediction for Go typically made use of feature construction or shallow neural networks. The former approach involves characterizing each legal move by a large number of features. These features include many 'shape' features, which take on a value of 1 if the stones around the move in question exactly match a predefined configuration of stones and 0 otherwise. Stone configurations can be as small as the nearest two or three squares and as large as the entire board. Very large numbers of stone configurations can be harvested by finding commonly occurring stone configurations in the training data. Shape features can be augmented with hand crafted features, such as distance of the move in question to the edge of the board,

whether making the move will capture or lose stones, its distance to previous moves, ect. Finally a model is trained to rank the legal moves based on their features. Work following this approach includes (Stern et al., 2006; Araki et al., 2007; Wistuba et al., 2012; Wistuba & Schmidt-Thieme, 2013; Coulom, 2007). Depending on the complexity of the model used, researchers have seen accuracies of 30 - 41% on move prediction for high-ranked amateur Go players.

Several researchers have made use of neural networks for move prediction. Werf et al. trained a neural network to predict expert moves using hand constructed features, preprocessing techniques to reduce the dimensionality of the data, and a two layer neural network (Van Der Werf et al., 2003). Our work builds upon work done by Sutskever et al., where two layer convolutional networks were trained for move prediction (Sutskever & Nair, 2008). They achieved an accuracy of 34% when predicting the moves made by professional Go players using a network that took both the current board position and the previous moves as input. An ensemble of networks reached 37% accuracy.

Our work will differ in several important ways. We use much deeper networks and propose several novel position encoding schemes and network designs that improve performance. We found that the most important one is a strategy of tying weights within the network to 'hard code' particular symmetries that are expected to exist in good move prediction functions. We also do not use the previously made moves as input. There are two motivations for choosing not to do so. First, classifiers trained using previous moves as input might come to rely on heuristics like 'move near the area where previous moves were made' instead of learning to evaluate positions based on the current stone configuration. While this might improve accuracy, our fundamental motivation is to see whether the classifier can capture Go knowledge, and the ability to borrow knowledge from experts by looking at their past moves cheapens this objective. Secondly, it is likely to reduce performance when it comes to playing as a stand-alone Go player. During play both an opponent and the network are liable to make much worse moves then would be made by Go experts, therefore coming to rely on the assumption that the previous moves were made by experts can be expected to yield poor results. This potential problem was also noted by (Araki et al., 2007). Our work is also the first to perform an evaluation across two datasets, providing an opportunity to compare how classifiers trained on these datasets differ in terms of Go playing abilities and move prediction accuracy.

Several of the works mentioned above analyze or comment on the strength of their move prediction program as a stand-alone Go player. In (Van Der Werf et al., 2003) researchers found that their neural network was consistently defeated by GNU Go and conclude their '...belief was confirmed that the local move predictor in itself is not sufficient to play a strong game.' Work by (Araki et al., 2007) also reports that their move predictor was beaten by GNU Go. Stern et al. report that other Go players estimated their move predictor as having a ranking of 10-15 kyu, but do not report its win rates against another computer Go opponent (Stern et al., 2006) . Both (Coulom, 2007; Wistuba & Schmidt-Thieme, 2013) do not give formal results, but suggest that their systems did not make strong stand-alone Go playing programs. In general past approaches to move prediction have not resulted in Go programs with much skill.

Subsequent to the public release of this work, Maddison et al. also released independent work exploring DCNNs for move prediction (Maddison et al., 2014). In that work they explore larger networks and different board representations. However they also use previous moves as input to their classifier, which we avoid. They show initial explorations of using convolutional networks as part of Monte-Carlo Tree Search.

The work presented here is based on the work done in (Clark, 2014).

2. Approach

2.1. Data Representation

As done by (Sutskever & Nair, 2008), the networks trained here take as input a representation of the current position and output a probability distribution over all grid points of the Go board, which are interpreted as a probability distribution over the possible places an expert player could place a stone. During testing probability given to grid points that would be an illegal move, either due to being occupied by a stone or due to the simple-ko rule, are set to zero and the remaining outputs renormalized. We follow (Sutskever & Nair, 2008) by encoding the current position in two 19x19 binary matrices. The first matrix has ones indicating the location of the stones of the player who is about to play, the second 19x19 matrix has ones marking where the opponent's stones are. We depart from (Sutskever & Nair, 2008) by additionally encoding the presence of a 'simple-ko constraint' if one is present in a third 19x19 matrix. Here simple-ko constraints refers to grid points that the current player is not allowed to place a stone on due to the simple-ko rule. In our dataset of professional games only 2.4% of moves were made with a simple-ko constraint present. However simple-ko constraints are often featured in Go tactics so we hypothesize they are still important to include as input. We elect not to encode move constraints beyond the ones created by the simple-ko rule, meaning constraints stemming from super-ko rules, because they are rare, harder to detected, ruleset-dependent, and less prominent in Go tactics. Thus the input has three channels and a height and width of 19. Again following (Sutskever & Nair, 2008), as well as other work that has found this to be a useful feature such as (Wistuba & Schmidt-Thieme, 2013), we tried encoding the board into 7 channels where

stones are encoded based on the number of 'liberties' they have, meaning the number of empty grid points that the opposing player would need to occupy to capture that stone. In this case channels 1-3 encode the current player's pieces that have 1, 2, or 3 or more liberties, channels 4-6 do the same for the opponent's pieces, and the 7th channel marks simple-ko constraints as before.

The classifier is not trained to predict when players choose to pass their move. Passing is extremely rare throughout most of the game because it is practically never beneficial to pass a move in Go. Thus passing is mainly used at the end of the game to signal willingness to end the game. This means players usually pass, not because there are no beneficial moves left to be played, but due to being in a situation where both players can agree the game is over. Modeling when this situation occurs is beyond the scope of this work.

2.2. Basic Architecture

Our most effective networks were composed of many convolutional layers. Since the input only has a height and width of 19 we found it important to zero pad the input to each convolutional layer to stop the size of the outputs of the higher layers becoming exceedingly small. We briefly experimented with some variations, but found that zero-padding each layer to ensure each layer's output has a 19x19 shape was a reasonable choice. In general using a fully connected top layer was more effective than a convolutional top layer as was used by (Sutskever & Nair, 2008). However, the performance gap between networks using a convolutional and fully connected top layer diminished as the networks increased in size. As the networks increased in size using more than one fully connected layer at the top of the network became unhelpful. Thus our DCNNs use many convolutional layers followed by a single fully connected top layer. We found the rectifier activation function to be slightly more effective than the tanh activation function, which is used in all DCNNs in this work.

We were limited primarily by running time, in almost all cases increasing the depth and number of filters of the network increased accuracy. This implies we have not hit the limit of what can be achieved by scaling up the network. We found that using many, smaller convolutional filters and deep networks was more efficient in terms of trading off runtime and accuracy than using fewer, larger filters.

2.3. Additional Design Features

Along with the network architecture described above we introduce a number of additional techniques that we found to be effective for this task.

2.3.1. Edge Encoding

In the neural networks described so far the first convolutional layer will zero pad its input. In the current board representation zeros represent empty grid points, so this results in the board 'appearing' to be surrounded by empty grid points. In other words, the first layer cannot capture differences between stones being next to an edge or an empty grid point. A solution is to reserve an additional channel to encode the out of bounds grid points. In this scheme an empty channel is added to the input. Then, instead of zero padding the input, the input is padded with ones around the new channel and padded with zeros around the other channels. We experimented with padding the first layer with ones in the channel used for the opponent's stones, but found this to be less effective.

2.3.2. Masked Training

In Go, some grid points of the board are not legal moves, either because they are already occupied by a stone or due to ko rules. Therefore these points can be eliminated as possible places an expert will move a priori. During testing we account for this, but this knowledge is not present in the network during training. Informal experiments show that the classifier is able to learn to avoid predicting illegal moves with very close to 100% accuracy, but we still speculate that accounting for this knowledge during training might be beneficial. To accomplish this we 'mask', or zero out, the outputs from the top layer that are illegal, and then apply the softmax operator, make predictions, and backprop the gradient across only these outputs during learning.

2.3.3. Reflection Preservation

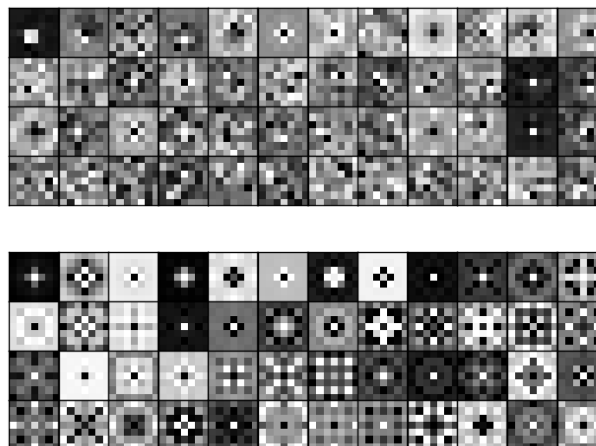
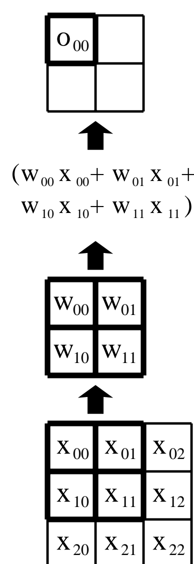


Figure 3. Visualization of the weights of randomly selected channels from randomly selected convolution filters of a five layer convolutional neural network trained on the GoGoD dataset. The network was trained once without (top) and once with (bottom) reflection preservation. It can be seen that even without weight tying some filters, such as row 1 column 6, have learned to acquire a symmetric property. This effect is even stronger in the weight visualization of (Sutskever & Nair, 2008).

.



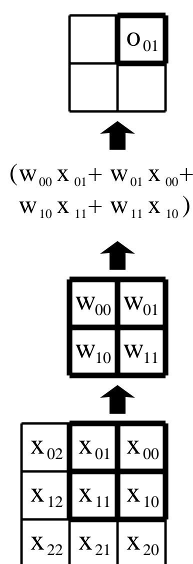


Figure 4. Applying the same convolutional filter to the upper left patch of an input (left) and to the upper right patch of the same input reflected across its y-axis (right). To enforce horizontal reflection preservation we will need to ensure $o_{00} = o_{01}$, for any x. This means we must have $w_{00} = w_{01}$ and $w_{10} = w_{11}$.

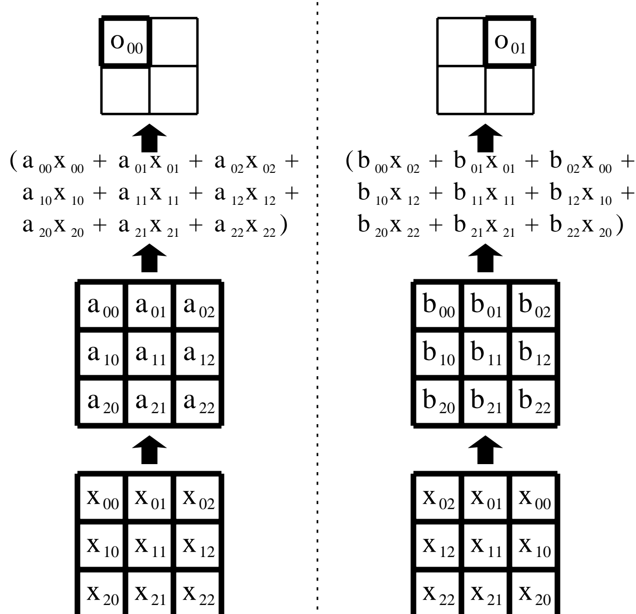


Figure 5. The computation for a single output unit, o_{00} with weights a_{ij}, for an input (left) and the computation for the output unit on the opposite side of the vertical axis, o_{01} with weights b_{ij} when the input is reflected across the y-axis (right). To enforce horizontal reflection preservation we need to ensure $o_{00} = o_{01}$ for any x. It is easy to see by examining the equations shown that this means we require $a_{00} = b_{02}$, $a_{01} = b_{01}$, $a_{02} = b_{00}$, ect.

In Go, if the board is reflected across either the x, y, or diagonal axis the game in some sense has not changed. The transformed position could be played out in an identical manner as the original position (relative to the transformation), and would result in the same scores for each player. Therefore we would expect that, if the Go board was reflected, the classifier's output should be reflected in the same manner. One way of exploiting this insight would be to train on various combinations of reflections of our training data, increasing the number of samples we could train on by eight folds. This tactic comes at a serious cost; our final network took four days to train for ten epochs. Increasing our data by eight folds means it would require almost a month to train for the same number of epochs.

A better route is to 'hard wire' this reflectional preserving property into the network. One way this can be done is by carefully tying the weights so that this property exists for each layer. For convolutional layers, this means tying the weights in each convolutional filter so that they are invariant to reflections. In other words enforcing that the weights are symmetrical around the horizontal, vertical, and diagonal dividing lines. An illustration of the kinds of weights this produces can be found in Figure 3. To see why this has the desired effect consider the application of a convolutional filter to an input with one channel. Let w_{ij} be the weights of the convolutional filter and x_{ij} be any square patch from the input of the same size where $0 <= i < n$ and $0 <= j < n$ and i and j index into the row and column of the input/weight in a top-to-bottom and left-to-right manner. The pre-activation output of this filter when applied to this patch of input is $s = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_{ij} w_{ij}$. Should the input be reflected horizontally, the patch of inputs x_{ij} will now be reflected and located on the opposite side of the vertical axis. Assuming the convolutional filter is applied in a dense manner (a stride of size 1), the same convolutional filter will be applied to the reflected patch. It is necessary and sufficient that the application of the convolutional filter to this reflected patch has a pre-activation value of s, the same as before, to meet our goal of having this layer preserve horizontal reflections applied to the input. Thus we want $\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} x_{ij} w_{ij} = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} x_{i(n-j)} w_{ij}$ for any x_{ij}. Thus we require $w_{ij} = w_{i(n-j)}$, in short that the convolutional filter is symmetric around the vertical axis. This is illustrated in Figure 4. A similar proof can be built for vertical or diagonal reflections.

For fully connected layers this property can also be encoded using weight tying in a similar manner. Assume the output and input of the fully connected layer have a square shape. Then enforcing horizontal invariance is a matter of requiring that, for any output unit, a, and the output unit on the opposite side of the vertical axis, b, that

$a_{ij} = b_{i(n-j)}$ for any i, j.

Where a_{ij}, $b_{i,j}$ refers to weight from input i, j to output units a and b respectively and n is the width of the input. This is shown in Figure 5. This reasoning can also be applied to vertical and diagonal reflections, the biases of each layer, and cases where there are multiple channels in the input and output. Encoding each reflection invariance ties each weight to another weight, so when accounting for all three possible reflections each each weight is tied to $2 \times 2 \times 2 = 8$ other weights. Likewise, the weight tying in the convolutional filters reduces the number of parameters in each filter by approximately an eighth. Thus we have reduced the number of parameters in the network by about eight folds. Tying weights requires a bit of extra computation, but it is a minor cost relative to forward and backward propagating batches of training data.

An alternative method of enforcing horizontal reflection preservation would be to apply the filters to half the input, reflect those filters and apply them to the other half of the input, and then concatenate the results. This would allow meeting the requirement that $w_{ij} = w_{i(n-j)}$ reducing the number of parameters, but we have not experimented with this approach.

While designed for Go, techniques in this vein have a ready application in image processing where we often expect the target function to be invariant to horizontal reflections. A minor adjustment of this technique could make a layer that is invariant to reflections rather than reflection preserving, meaning if its input is reflected its output will not change. Referring to Figure 5, this is just a matter of ensuring $b_{02} = b_{00}$, $b_{10} = b_{12}$, and $b_{20} = b_{22}$. Then one could build a network where all the lower layers preserve horizontal reflections and the final layer is invariant to horizontal reflections. The resulting network will be invariant to horizontal reflections, and have half the number of parameters of an untied network. This provides a way to account for the expected reflectional invariance property without having to double the amount of training data.

3. Training

3.1. Datasets

We use two datasets to evaluate our approach. The first is the Games of Go on Disk[2] (GoGoD) dataset consisting of 81,000 professional Go games. Games are played under a variety of rulesets (but usually Chinese rules) and have long time controls. The second dataset consists of 86,000 Go games played by high ranked players on the KGS Go server [3,4]. These games are all played under Japanese rules, have slightly lower player rankings, and generally use much faster time controls. We use two datasets because previous work in this field has typically used either one or the other. We use all available data from the GoGoD dataset and select 86,000 games form the KGS dataset (where games are on average slightly shorter) so that the number of position-moves pairs in each dataset that can be trained on is roughly equal. This yields about 16.5 million move-positions pairs for each dataset.

[2]http://gogodonline.co.uk/

[3]https://www.gokgs.com/

[4]http://u-go.net/gamerecords/

3.2. Methodology

Both datasets were partitioned into test, train, and validation sets each containing position-move pairs that are from disjoint games of Go. We use 8% (1.3 million) for testing, 4% (620 thousand) for validation, and the rest (14.7 million) for training. The validation set was used to monitor learning and to experiment with hyperparameters. We use vanilla gradient descent for training, although we found that it was important to anneal the learning rate towards the end of learning. Both convolutional and fully connected layers had their biases initialized to zero and weights drawn from a normal distribution with mean 0 and standard deviation 0.01.

4. Results

4.1. Ablation Study

Excluding	Accuracy	Rank	Probability
None	36.77%	7.50	0.0866
Ko Constraints	36.55%	7.59	0.0853
Edge Encoding	36.81%	7.64	0.0850
Masked Training	36.31%	7.66	0.0843
Liberties Encoding	35.65%	7.89	0.0811
Reflection Preserving	34.95%	8.32	0.0760
All but Liberties	34.48%	8.36	0.0755
All	33.45%	8.76	0.0707

Table 1. Ablation study. A medium scale network with four convolutional layers and one fully connected layer was trained on the GoGoD dataset while excluding different features. We report accuracy, average probability assigned to the correct move, and average rank given to the correct move relative to the other possible moves on the test set. The liberty encoding and reflection preserving techniques are the most useful, but all the suggested techniques improve average rank and average probability.

To evaluate some of the design choices made here an ablation study was performed. The study was done on a ‘medium scale’ network to allow multiple experiments to be conducted quickly. The network had one convolutional layer with 48 7x7 filters, three convolutional layers with 32 5x5 filters, and one fully connected layer. Networks were trained with mini-batch gradient descent with a batch size of 128, using a learning rate of 0.05 for 7 epochs, and 0.01 for 2 epochs, which took about a day on a Nvidia GTX 780 GPU. The results are in Table 1. The reflection preserving technique was extremely effective, leaving it out dropped accuracy by almost 2%. The liberty encoding scheme im-

proved performance but was not as essential, leaving it out dropped performance by 1%. The remaining optimizations had a less dramatic impact but still contributed nontrivially. Together these additions increased the overall accuracy by over 3% and increased accuracy relative to just using liberties encoding by over 2%.

4.2. Full Scale Network Evaluation

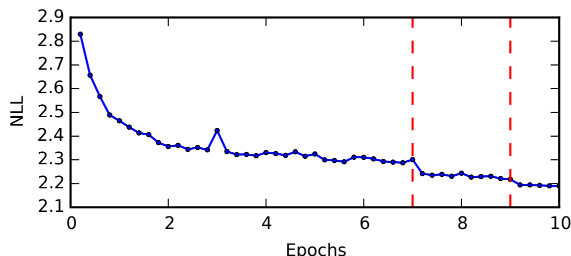


Figure 6. Negative log likelihood on the validation set while training the 8 layer DCNN trained on the GoGoD dataset. Vertical lines indicate when the learning rate was annealed. Improvement on the validation set has more or less slowed to a halt after 10 epochs of training.

	Accuracy	Rank	Probability
Test Data	41.06%	5.91	0.1117
Train Data	41.86%	5.78	0.1158

Table 2. Results for the 8 layer DCNN on the train and test set of the GoGoD dataset. Rank refers to the average rank the expert's move was given, Probability refers to the average probability assigned to the expert's move.

	Accuracy	Rank	Probability
Test Data	44.37%	5.21	0.1312
Train Data	45.24%	5.07	0.1367

Table 3. Results for the 8 layer DCNN on the train and test set of the KGS dataset. Rank refers to the average rank the expert's move was given, Probability refers to the average probability assigned to the expert's move

The best network had one convolutional layer with 64 7x7 filters, two convolutional layers with 64 5x5 filters, two layers with 48 5x5 filters, two layers with 32 5x5 filters, and one fully connected layer. The network used all the optimizations enumerated in the previous section. The network was trained for seven epochs at a learning rate of 0.05, two epochs at 0.01, and one epoch at 0.005 with a batch size of 128 which took roughly four days on a single Nvidia GTX 780 GPU. Figure 6 shows the learning speed as measured on the validation set. The network was trained and evaluated on the GoGoD and KGS dataset, as shown in Table 2 and Table 3 respectively. To our knowledge the

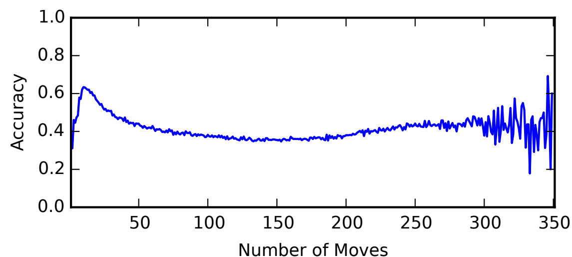


Figure 7. Accuracy when a fixed number of moves have passed on the GoGoD test set. The network is more accurate during the beginning and end game and less accurate during the middle game.

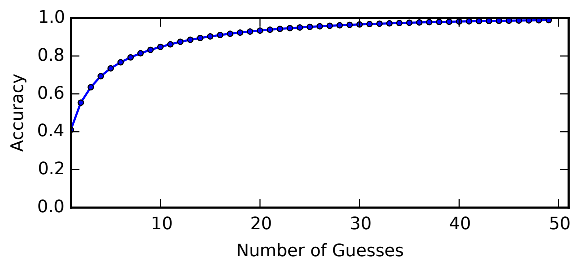


Figure 8. Accuracy when allowed a fixed number of guesses on the GoGoD test set. If the neural network's best guess is wrong, the right answer is often among its next few guesses. However there are occasions when the right move is ranked as the 30-50th best move by the network.

best reported result on the GoGoD dataset is 36.9% using an ensemble of shallow networks (Sutskever & Nair, 2008) and the best on the KGS dataset is 40.9% using feature engineering and latent factor ranking (Wistuba & Schmidt-Thieme, 2013). Our work was completed on more recent versions of these datasets, but in so much as they can be compared our work has surpassed these previous results by margins of 4.16% and 3.47% respectively. Additionally, this was done without using the previous moves as input. (Wistuba & Schmidt-Thieme, 2013) did not analyze the impact using this feature had, but (Sutskever & Nair, 2008) report the accuracy of one of their networks dropped from 34.6% to 21.8% when this feature was removed, implying their networks heavily relied on this feature.

We also examine the GoGoD test set accuracy of the network as a function of the number of moves made in the game, see Figure 7. Accuracy increases during the more predictable opening moves, decreases during the complex middle game, and increases again as the board fills up and the number of possible moves decreases towards the end game. Finally, we examine how accurate the network is when allowing the network to make multiple guesses, see Figure 8. It is encouraging to note that, if the network's highest-ranking move is incorrect, its second or

Opponent	KGS	GoGoD	GoGoD Small
GnuGo (C)	0.86	0.87	0.71
GnuGo (J)	0.85	0.91	0.67
Fuego (C)	0.12	0.14	0.00

Table 4. Win rates of three DCNNs against GNU Go level 10 using Chinese (C) and Japanese (J) rules and again Fuego 1.1. For each matchup 200 games were played. GoGoD and KGS refer to the full scale network trained on the named dataset, GoGoD Small refers to the exclude none network from Section 4.1. Even the smaller DCNN is able to consistently defeat GNU Go and the larger ones are able to win some games against Fuego.

third highest-ranked move is often correct. However there are times when the expert move was not among the top 40 ranked moves from the network. While it is not clear exactly how well a human expert would perform on this task, it seems likely that a human expert would practically always be able to guess where another expert would move given 40 guesses. Thus we do not think our DCNN has reached a human level of performance on this task.

4.3. Playing Go

The networks trained here were successful move predictors, but that does not necessary mean they will be strong Go players. There are two potential problems. First, during a game an opponent, or the classifier itself, are liable to make moves that create positions that are uncommon for games between experts. Since the networks have not been tested or trained on these kinds of positions there is no guarantee they will continue to perform well when this occurs. Second, even if the classifier is able to predict an expert player level move 90% of the time, if its other moves are extremely poor it could still be a terrible Go player. To test the strength of the networks as Go players they were played against two other computer Go programs, GNU Go 3.8[5] and Feugo 1.1 (Enzenberger et al., 2010). We test the final network trained on the KGS data, the GoGoD data, and the smaller 'exclude none' network from Section 4.1. The results can be found in Table 4. There is one complication; the DCNNs do not have the capability to pass during their turn. Therefore, should a game go on indefinitely, the networks will eventually run out of good moves to play and start making suicidal moves. To work around this issue we allow both Fuego and GNU Go to resign,. We additionally have the DCNN pass its turn whenever its opponent does thus ending the game. Games were scored using the opposing Go engine's scoring function.

Our results are very promising. Even though the networks are playing using a 'zero step look ahead' policy, and using a fraction of the computation time as their opponents, they are still able to consistently defeat GNU Go and take some games away from Fuego. Under the settings used here GNU Go might play at around a 6-8 kyu ranking and Fuego at 2-3 kyu, which implies the networks are achieving a ranking of approximately 4-5 kyu. For a human player reaching this ranking would normally require years of study. This indicates that sophisticated knowledge of the game was acquired. The smaller network we trained consistently defeated GNU Go despite being less accurate than some previous work at move prediction. Thus it seems likely that our choice not to use the previous move as input has helped our move predictors to generalize well from predicting expert Go player's moves to playing Go as standalone players. This might also be attributed to our deep learning based approach. Deep learning algorithms have been shown in particular to benefit from out of sample distributions (Bengio et al., 2011), which relates to our situation since our networks can be viewed as learning to play Go from a biased sample of positions. The network trained on the GoGoD dataset performed slightly better than the one trained on the KGS dataset. This is what we might expect since the KGS dataset contains many games of speed Go that are liable to be of lower quality.

[5]https://www.gnu.org/software/gnugo/

5. Conclusion and Future Work

In this work we have introduced the application of deep learning to the problem of predicting the moves made by expert Go players. Our contributions also include a number of techniques that were helpful for this task, including a powerful weight tying technique to take advantage of the symmetries we expect to exist in the target function. Our networks are state of the art at move prediction, despite not using the previous moves as input, and can play with an impressive amount skill even though future positions are not explicitly examined.

There is a great deal that could be done to extend this work. We limited the size of our networks to keep training time manageable, but using more data or larger networks will likely increase accuracy. We have only completed a preliminary exploration of the hyperparameter space and think better network architectures could be found. Curriculum learning (Bengio et al., 2009) and integration with reinforcement learning techniques might provide avenues for improvement. The excellent skill achieved with minimal computation could allow this approach to be used for a strong but fast Go playing mobile application. Finally, a trained DCNN could be integrated with a full-fledged Go playing system. For example, a DCNN could be run on a GPU in parallel with a MCTS Go program and be used to provide high quality priors for what the strongest moves to consider are. Such a combined system would be the first to bring sophisticated pattern recognitions abilities to playing Go, and we believe it would have a strong potential ability to surpass current computer Go programs.

Acknowledgments

We thank the anonymous reviewers for their helpful comments.

References

Araki, Nobuo, Yoshida, Kazuhiro, Tsuruoka, Yoshimasa, and Tsujii, Jun'ichi. Move Prediction in Go with the Maximum Entropy Method. In *Computational Intelligence and Games*, 2007.

Baudiš, Petr and Gailly, Jean-loup. Pachi: State of the Art Open Source Go Program. In *Advances in Computer Games*. 2012.

Bengio, Yoshua. Learning Deep Architectures for AI. *Foundations and trends® in Machine Learning*, 2009.

Bengio, Yoshua and LeCun, Yann. Scaling Learning Algorithms Towards AI. *Large-scale Kernel Machines*, 2007.

Bengio, Yoshua, Louradour, Jérôme, Collobert, Ronan, and Weston, Jason. Curriculum Learning. In *ICML*, 2009.

Bengio, Yoshua, Bastien, Frédéric, Bergeron, Arnaud, Boulanger-Lewandowski, Nicolas, Breuel, Thomas M, Chherawala, Youssouf, Cisse, Moustapha, Côté, Myriam, Erhan, Dumitru, Eustache, Jeremy, et al. Deep Learners Benefit More from Out-of-Distribution Examples. In *AISTATS*, 2011.

Bozulich, Richard. *The Go Player's Almanac*. Ishi Press, 1992.

Browne, Cameron B, Powley, Edward, Whitehouse, Daniel, Lucas, Simon M, Cowling, Peter I, Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon. A Survey of Monte Carlo Tree Search Methods. *Computational Intelligence and AI in Games*, 2012.

Clark, Christopher. Deep Go. Master's thesis, University of Edinburgh, 2014.

Coulom, Rémi. Computing Elo Ratings of Move Patterns in the Game of Go. In *Computer games workshop*, 2007.

Enzenberger, Markus, Muller, Martin, Arneson, Broderick, and Segal, Richard. Fuego: An Open-source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *Computational Intelligence and AI in Games*, 2010.

Maddison, Chris J., Huang, Aja, Sutskever, Ilya, and Silver, David. Move Evaluation in Go Using Deep Convolutional Neural Networks. *CoRR*, abs/1412.6564, 2014. URL http://arxiv.org/abs/1412.6564.

Müller, Martin. Computer Go. *Artificial Intelligence*, 2002.

Rimmel, Arpad, Teytaud, F, Lee, Chang-Shing, Yen, Shi-Jim, Wang, Mei-Hui, and Tsai, Shang-Rong. Current Frontiers in Computer Go. *Computational Intelligence and AI in Games*, 2010.

Stern, David, Herbrich, Ralf, and Graepel, Thore. Bayesian Pattern Ranking for Move Prediction in the Game of Go. In *ICML*, 2006.

Sutskever, Ilya and Nair, Vinod. Mimicking Go Experts with Convolutional Neural Networks. In *Artificial Neural Networks-ICANN*. 2008.

Van Der Werf, Erik, Uiterwijk, Jos WHM, Postma, Eric, and Van Den Herik, Jaap. Local Move Prediction in Go. In *Computers and Games*. 2003.

Wistuba, Martin and Schmidt-Thieme, Lars. Move Prediction in Go–Modelling Feature Interactions Using Latent Factors. In *KI 2013: Advances in Artificial Intelligence*. 2013.

Wistuba, Martin, Schaefers, Lars, and Platzner, Marco. Comparison of Bayesian Move Prediction Systems for Computer Go. In *Computational Intelligence and Games (CIG)*, 2012.