Mimicking Go Experts with Convolutional Neural Networks

Ilya Sutskever and Vinod Nair

Department of Computer Science
University of Toronto, Toronto, Ontario, Canada
{ilya,vnair}@cs.utoronto.ca

Abstract. Building a strong computer Go player is a longstanding open problem. In this paper we consider the related problem of predicting the moves made by Go experts in professional games. The ability to predict experts' moves is useful, because it can, in principle, be used to narrow the search done by a computer Go player. We applied an ensemble of convolutional neural networks to this problem. Our main result is that the ensemble learns to predict 36.9% of the moves made in test expert Go games, improving upon the state of the art, and that the best single convolutional neural network of the ensemble achieves 34% accuracy. This network has less than 10^4 parameters.

Key words: Go, Move prediction, Convolutional Neural Networks

1 Introduction

Go is an ancient and popular two-player board game in which players take turns to place pieces on the board, aiming to capture as many of the opponent's pieces and as much board territory as possible. The pieces are unmovable, but can be captured and removed from the board if surrounded by the opponent's pieces. Go is played on a 19×19 board; its rules are described by van der Werf [1, ch. 2]

Existing Go-playing computer programs are still not competitive with Go professionals on 19×19 boards (e.g., [2], [1], [3]). In contrast, some Chess-playing programs can play on par with world champions, even though Chess is not obviously easier than Go, except with respect to board size. Some checkers-playing programs are even more extreme and can play the *optimal* checkers strategy [4].

The techniques used for successful Chess play do not work for Go for several reasons. First, the nature of the rules of Chess makes it easy to estimate the player with the advantage, simply by counting the pieces. This and similar heuristics allow minimax search to compute a good move reasonably rapidly. In contrast, there is no easy way to determine the player with the advantage in Go. In particular, counting captured pieces (which works for Chess) does not work for Go because the value of a piece depends more heavily on its surroundings. The inability to rapidly evaluate Go positions prevents minimax searches from finding good moves. Second, typical Chess positions have 40 legal moves, while typical Go positions have 200 legal moves, so the search space is substantially larger. This partly explains why there are still no strong computer Go players.

The difficulty of Go suggests that it may be useful to start by solving a different problem whose solution would help create a strong Go player. In this paper we consider such a problem, which is to predict the moves made in Go experts' games. Predicting moves of Go experts is related to creating a good Go player, because an extremely accurate move predictor (e.g., a Go expert) ought to play Go well. However, constructing a fairly accurate move predictor is potentially easier than playing Go well, because an accurate move predictor can occasionally make devastating mistakes which would make it a poor Go player. For an illustration of the importance of the problem, consider an observer of a professional Go tournament. If the observer is able to predict, on average, every third move the players make, then it is plausible that the observer is a Go expert. Our convolutional neural networks can do precisely this. In addition, the move predictor can be used for educational purposes, since by outputting a probability distribution over all possible moves, it allows the student of expert Go games to see alternative good moves that could have been made (but obviously, were not). Finally, the move predictor can be used to narrow the search done by a full Go player.

We report the prediction accuracy of several convolutional neural networks [5], [6] on the problem of predicting Go experts' moves and also the accuracy when the networks are combined. Our main result shows that an ensemble of convolutional neural networks is able to predict 36.9% of the moves in the test Go games, while the most accurate previous method by Stern et al. [7] predicts 34% of the moves correctly. The best convolutional neural network of our ensemble has less than 10^4 parameters and achieves over 34% accuracy on the test set, while a very small convolutional neural network, with less than 2,000 parameters, achieves 30% prediction accuracy. Furthermore, the convolutional networks learn rapidly and reach 30% accuracy after learning on a small fraction of the training set.

We now outline the difference between Stern et al.'s approach [7] and ours. While our move predictor uses an ensemble of convolutional neural networks the best of which have a small number of parameters, Stern et al.'s predictor uses a 10^7-dimensional feature vector to represent the board, where each feature represents an exact pattern match (up to a translation and a symmetry) and is computed very efficiently. It copes with such high-dimensional feature vectors by using Bayesian methods, and is trained on 180,000 expert games. The nature of its exact pattern matching may make it difficult to generalize to non-expert games.

In addition to using fewer parameters, we used a smaller training set consisting of approximately 45,000 expert games [8]. Since the convolutional neural networks, especially the small ones, have relatively few parameters, and do not perform exact pattern matching, they have the potential to generalize to non-expert games.

The method of Stern et al. has an advantage over ours: it does not use the previously made moves as additional inputs. This is significant, because our results show that the moves made in previous timesteps greatly improve the performance of our convolutional neural networks (see table 1 for the extent to which it helps).

2 Convolutional Neural Networks

Convolutional neural networks [5] form a subclass of feedforward neural networks that have special weight constraints. They enable the network to learn much more efficiently from fewer examples, provided that the learning problem exhibits two-dimensional translation invariance.

Convolutional neural networks have been successfully applied to various problems, and have obtained the best classification accuracy on the MNIST digit dataset [11], [12], the NORB image dataset [13], and on the problem of handwritten zipcode understanding [14].

Convolutional neural networks are well suited for problems with a natural translation invariance, such as object recognition ([11], [13], [14]). Go has some translation invariance, because if all the pieces on a hypothetical Go board are shifted to the left, then the best move will also shift (with the exception of pieces that are on the boundary of the board). Consequently, many applications of neural networks to Go have used convolutional neural networks ([6], [15], [10], among others). A convolutional neural network is depicted in figure 1. Its construction is motivated by the observation that images (and Go boards) are expected to be processed in the same way in every small image patch; therefore, the weights of the convolutional neural network have a replicated structure which applies the same weights to every subrectangle of the image (the size of the subrectangles is always the same; fig. 1), producing the total inputs to the next layer. The weights of a convolutional neural network are also called *the convolutional kernel*, K, and its size, n, is the size of the subrectangles it considers. In particular, a fully connected feedforward neural network has many more parameters than a convolutional network of the same size. The relatively small number of weights makes parameter estimation require many fewer examples, so a good solution (if exists) can be found rapidly.

As in feedforward networks, applying the sigmoid function $(1 + e^{-t})^{-1}$ to the total inputs produces the values of the units in the hidden layer, which form several 2-dimensional laid out "images" that are processed in the same convolutional manner.

More concretely, a layer of a convolutional neural network is governed by the following equation

$$y_{x,y} = \left(1 + \exp\left(-\sum_{u=-(n-1)/2}^{(n-1)/2} \sum_{v=-(n-1)/2}^{(n-1)/2} x_{x+u,y+v} K_{u,v}\right)\right)^{-1} \tag{1}$$

where y is the activity of a hidden unit at position (x, y), and x are the input units.

A convolutional neural network can have several convolutional kernels; for instance, the network of figure 1 has three convolutional kernels in the first hidden layer. The convolutional kernel works in such a way that the sizes of the maps in the input, hidden, and output layers are the same. To accomplish this, some rectangles need to be partially outside of the board; in this case, the weights of the kernels corresponding to the outside-the-board region are unused.

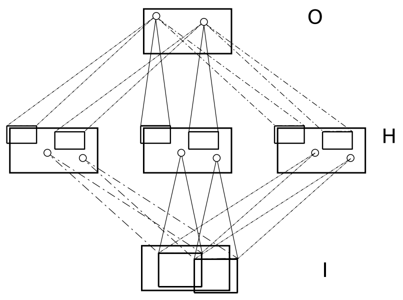

Fig. 1. A convolutional neural network with one hidden layer, and three convolutional kernels; the applications of the kernels to two rectangular regions of their inputs are shown. I, H, and O are the input, hidden, and output layers, respectively.

3 Experiments

In this section we describe the details of the convolutional neural networks that we used for our experiments and report their performance.

For our experiments, we used the Gogod collection of games [8] which contains about 45,000 expert games. D. Stern [7] generously provided us with a quarter of their test set for accurate comparisons. We processed the games in a simple manner so that it is the black's player move on each board, which involved reversing the colors of every second board (in a way that depends on first player's color).

The structure of the convolutional neural network used are as follows: The size of the convolutional kernels of the first layers are 9×9 or 7×7, and the size of the hidden-to-output convolutional kernels are 5×5. The hidden layer consists of 15 convolutional kernels. As usual, the output layer is the softmax of its input from the hidden layer, and the objective function is the log probability of the model given the labels.

Another variable in the experiments is the encoding of the input. We tried a raw input encoding, where each intersection on a board takes three possible

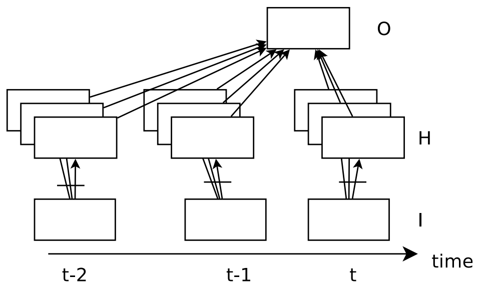

Fig. 2. A convolutional neural network that uses the moves made in previous timesteps. It receives, as inputs, several of the previous board positions. The marked convolutional maps are the same; thus every timestep is proceed in the same way, at first. The output layer consists of 361 softmax units, whose activities are the network's belief in the correctness of the given move.

values: empty, white and black, which is given to the convolutional layer in the form of two 19×19 bitmaps corresponding to the black and the white pieces.

We also tried an encoding that represents the number of liberties of each group (the liberty representation). A group is a connected set of pieces of the same color, and the number of liberties of a group is equal to the number of pieces the opponent needs to place in order to capture this group, without intervention. This feature was used earlier ([7], [15], [10]) and was shown to be useful. The number of liberties of a non-empty intersection is defined to be the number of liberties of the intersection's group, which can take the values $1, 2, \geq 3$ (it cannot be 0 if the intersection is nonempty). Combining the number of liberties with the color of the piece, each nonempty board intersection can take values in the set $Vels = \{1, 2, \geq 3\} \times \{black, white\}$ of size 6. Thus, the input representation is given in the form of 6 bitmaps of size 19×19 (cf. figure 3). An empty intersection causes every bitmap be 0 in this intersection; if the intersection is nonempty, then the bitmap corresponding to the feature in $Vels$ is set to 1 in the corresponding input intersection.

The final variable of the experiment is the number of previous boards used as inputs: instead of using only the current board as an input, it is also possible to use the board configurations of several previous timesteps. We used 0 and 4 previous timesteps. See figure 2.

We also tried using a much smaller a convolutional neural networks that have only 3 convolutional kernels of size 7×7, which achieved 30% accuracy, as well as one that has uses only one previous timestep.

The learning details are as follows: A learning rate of size 0.1 is used for the first 3000 weight updates, which is then reduced to 0.01. The momentum of size 0.9 is used. Learning proceeds for 10^5 weight updates, where each weight update corresponds to processing a single game in the Gogod dataset; thus, learning makes less than three passes over the training set. Finally, the gradient is always

Network Structure	Accuracy	Mean log loss
$7 \times 7, t = 0, l$	22.0%	4.9
$7 \times 7, t = 0, n$	17.5%	5.2
$7 \times 7, t = 4, l$	**34.1**%	4.0
$7 \times 7, t = 4, n$	31.7%	4.2
$9 \times 9, t = 0, l$	21.8%	4.9
$9 \times 9, t = 0, n$	18.2%	5.2
$9 \times 9, t = 4, l$	**34.6**%	4.0
$9 \times 9, t = 4, n$	32.3%	4.1
$7 \times 7, t = 4, l, h = 3$	30.0%	4.4

Table 1. Results of the small convolutional neural networks. Each simulation is described by there parameters. The first, $k \times k$, is the size of the input-to-hidden convolutional kernel; t is the number of previous timesteps used; l is written if the liberty representation is used, and n if the raw representation is used. Every network in this table has less than 10^4 parameters and 15 convolutional kernels, except for the last row, which has only 3 convolutional kernels and less than 2,000 parameters. Notice that the best network that does not make use of the previous timesteps achieves 22% accuracy, showing that the previous timesteps are extremely helpful for convolutional neural networks.

divided by 200 (before being multiplied by the learning rate) since the mean length of a Go game in our dataset is 206. No weight decay was used.

3.1 Averaging the predictions of different neural networks

In this subsection we describe the results that are obtained when all the networks are combined to make a large predictor. We train a neural network that takes the predictions of the individual nets listed in table 1 as inputs, and computes a single move prediction as output.

In addition to the networks in table 1, we also include a few simple move predictors in the ensemble that improve the accuracy of the ensemble, despite having low individual accuracy. The simple move predictor is a neural network that estimates the probability of an expert making a move at a particular location, given a local “patch” of the board centred at that location as input. For example, the input to the network would be a 9×9 patch of the board, and the output would be the probability of an expert making a move at the center of the patch. Such a model can be used for full-board move prediction by independently applying it to all possible candidate move locations on a board, and then selecting the one with the highest probability as the predicted expert move location. Such a purely local predictor will have limited accuracy because many expert moves in Go are decided based on knowledge about the entire board. Nevertheless, it will be good at predicting moves that are fully determined by local configurations of stones. We train networks on patches of size 9×9 (100 hidden units), 13×13 (100 hidden units), and 17×17 (200 hidden units), with test set move prediction accuracies of 18.4%, 18.9%, and 19.8%, respectively.

Although these networks perform poorly, they still improve the accuracy of the ensemble.

We selected a subset of the training set of size 1000 games to learn the weights of the ensemble network. We tried two averaging methods. The first method computes the weighted arithmetic means the predictions of all the networks, with the objective to maximize the log probability. This yields 35.5% accuracy. The second method computes the weighted geometric means of the predictions of all the networks and renormalizes the predictions, with the same objective. This approach, although yielding a lower average log probability, obtains 36.9% accuracy.

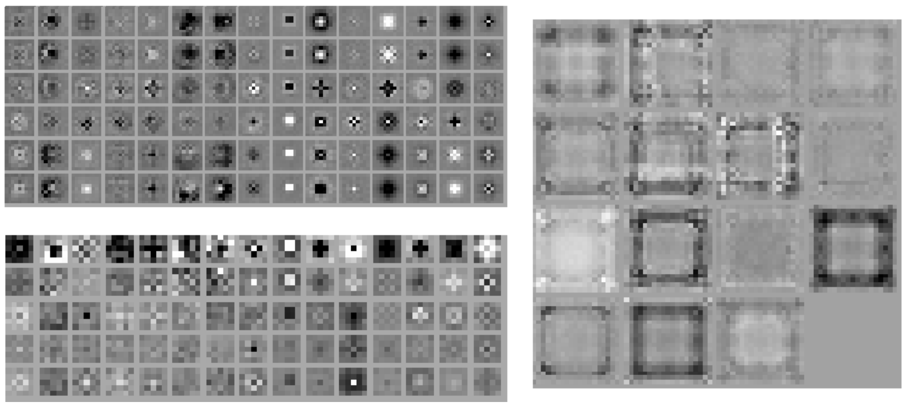
Fig. 3. The weights of a neural network achieving 34.6% accuracy that uses the liberties as an additional input. Recall that the input vector is represented by 6 bitmaps. Each column of the top-left image shows the convolutional kernels corresponding to each such bitmap, for each of the 15 convolutional maps. The image on the right shows the biases of the hidden units; it is displayed as 15 19×19 images. The image at the bottom shows the convolutional kernels mapping the 15 from the current and the 4 previous timesteps. The images are clipped at -1.5, 1.5, to enhance their contrast.

4 Related Work

A popular research direction for Go is to apply the idea that produced the Backgammon program [16]. The Backgammon playing program uses Reinforcement Learning [17] with self-play to learn a neural network that computes the Q-values of the different moves. This approach was attempted in [6] (but using $TD(0)$ [17]). It is not a straightforward application of $TD(0)$-learning to Go, because the neural network makes 361 predictions, one per intersection, on the identity of its owner in the end of the game. Doing this increases the amount of learning signal obtained from a single game. More elaborate variants of Q-learning with neural networks are also used [10], where the neural network is highly specialized and uses Q-learning to learn to predict many aspects of the final position (e.g., determine whether a given position will be an eye, or whether two positions will belong to the same group in the end of the game). This is motivated by the belief that a good Go player should be able to predict all these aspects of the final position, so the resulting hidden units will be more helpful for accurate board evaluation.

The most successful recent approaches at computer Go used Monte Carlo simulations to estimate the position's value [18]. While this approach yields the strongest existing computer Go player [19], we do not directly compare our move predictor to it, since a strong move predictor is not necessarily a strong Go player, and vice versa.

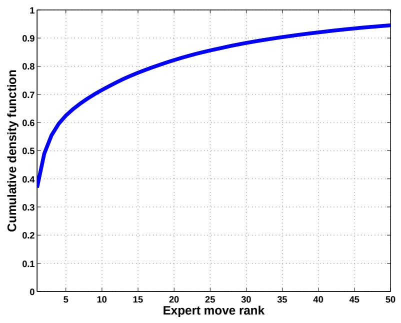

Fig. 4. The curve shows the fraction of the boards (vertical axis) whose correct move is among the top k predictions of the ensemble, where k ranges the horizontal axis.

There has also been some work on move prediction for computer Go [1], [15] using a small neural network resembling ours. It used some features, such as the number of liberties, as well as a feature that determines the distance to the previously made move, and were trained with a specially-chosen loss function. Its accuracy was 25%. A different specialized convolutional neural network trained with moderately strong play on 9×9 boards was able to generalize to 19×19 boards, obtaining 10% accuracy in expert's move prediction [20].

There has been more work [21] on move prediction that used high-dimensional feature vectors similar to those that Stern et al. used. In addition to using high-dimensional feature representation, Araki et al. [21] also used the previously made moves as additional features. They were only able to obtain prediction accuracy of 33%, having trained on half of the Gogod dataset.

5 Conclusions

We reported experimental results using neural networks for move prediction for Go. Our results show that small convolutional neural networks are a viable method for predicting Go expert moves. Our main discovery that knowledge of the previous timesteps combined with the convolutional neural networks produce a particularly accurate move predictor. Araki et al.'s [21] experience suggests that using the previous timesteps will be less helpful for feature-vector based approaches.

In particular, the small convolutional neural networks could be used to direct the search very efficiently, and their small number of parameters makes them easy to learn.

6 Acknowledgements

This research was partially supported by the Ontario Graduate Scholarship and by the Natural Sciences and Engineering Council of Canada.

References

1. van der Werf, E.: AI Techniques for the Game of Go. UPM, Universitaire Pers Maastricht (2004)
2. Müller, M.: Review: Computer Go 1984-2000. Lecture Notes In Computer Science (2000) 405–413
3. Bouzy, B., Cazenave, T.: Computer Go: An AI oriented survey. Artificial Intelligence **132**(1) (2001) 39–103
4. Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P., Sutphen, S.: Checkers Is Solved. Science **317**(5844) (2007) 1518
5. LeCun, Y., Boser, B., Denker, J., Howard, R., Habbard, W., Jackel, L., Henderson, D.: Handwritten digit recognition with a back-propagation network. Advances in neural information processing systems 2 table of contents (1990) 396–404
6. Schraudolph, N., Dayan, P., Sejnowski, T.: Temporal Difference Learning of Position Evaluation in the Game of Go. Advances in Neural Information Processing Systems **6** (1994) 817–824
7. Stern, D., Herbrich, R., Graepel, T.: Bayesian pattern ranking for move prediction in the game of Go. Proc. of the 23rd international conference on Machine learning (2006) 873–880
8. Hall, M.T., Fairbairn, J.: The Gogod Database and Encyclopaedia. www.gogod.co.uk (2006)
9. Sutton, R.: Learning to predict by the methods of temporal differences. Machine Learning **3**(1) (1988) 9–44
10. Enzenberger, M.: Evaluation in Go by a Neural Network using Soft Segmentation. Advances in Computer Games **10** (2003)
11. Simard, P., Steinkraus, D., Platt, J.: Best practices for convolutional neural networks applied to visual document analysis. Document Analysis and Recognition (2003) 958–963

12. Ranzato, M., LeCun, Y.: A sparse and locally shift invariant feature extractor applied to document images. In: Proc. International Conference on Document Analysis and Recognition (ICDAR). (2007)
13. LeCun, Y., Huang, F., Bottou, L.: Learning methods for generic object recognition with invariance to pose and lighting. Computer Vision and Pattern Recognition **2** (2004)
14. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE **86**(11) (1998)
15. van der Werf, E., Uiterwijk, J., Postma, E., van den Herik, J.: Local Move Prediction in Go. Computers and Games (2003)
16. Tesauro, G.: Temporal difference learning and TD-Gammon. Communications of the ACM **38**(3) (1995) 58–68
17. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press (1998)
18. Brugmann, B.: Monte Carlo Go. (1993)
19. Gelly, S., Wang, Y.: Exploration exploitation in Go: UCT for Monte-Carlo Go. NIPS-2006: On-line trading of Exploration and Exploitation Workshop, Whistler Canada (2006)
20. Wu, L., Baldi, P.: A Scalable Machine Learning Approach to Go. Neural Information Processing Systems (2007) 1521–1528
21. Araki, N., Yoshida, K., Tsuruoka, Y., Tsujii, J.: Move Prediction in Go with the Maximum Entropy Method. Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games (2007)