A Scalable Neural Network Architecture for Board Games

Tom Schaul, Jürgen Schmidhuber

Abstract— This paper proposes to use Multi-dimensional Recurrent Neural Networks (MDRNNs) as a way to overcome one of the key problems in flexible-size board games: scalability. We show why this architecture is well suited to the domain and how it can be successfully trained to play those games, even without any domain-specific knowledge. We find that performance on small boards correlates well with performance on large ones, and that this property holds for networks trained by either evolution or coevolution.

I. INTRODUCTION

Games are a particularly interesting domain for studies of machine learning techniques. They form a class of clean and elegant environments, usually described by a small set of formal rules, have very clear success criteria, and yet they often involve highly complex strategies.

Board games generally exhibit these features to a high degree, and so it's not surprising that the field of machine learning has devoted major efforts to their study, with the result that in almost all popular board games, most notably chess, computer programs can beat all human players.

Probably the most interesting exception is the ancient game of *Go*, which can be solved for small boards [1] but is very challenging for larger ones [2], [3]. Go has a very high branching factor because at any moment there are about as many legal moves as there are free positions on the board (on average 200). This makes using traditional search-based methods prohibitively expensive.

It would be highly desirable if one could train a player on small boards, and then transfer that knowledge to bootstrap learning on larger boards (where training is much more expensive). Fortunately, the game's high degree of symmetry makes using patterns a viable strategy (which is heavily used by human expert players). This has led researchers to use tools that are good at pattern recognition, most of which are connectionist in nature, on Go and similar games – with varying degrees of success. So far, none of these methods has been found to exhibit a large degree of *scalability*, in the sense that it is directly usable on different board sizes and leads to similar behavior across board sizes.

In this paper we propose a neural network architecture that is scalable in this sense. Additionally, in order to keep our approach general, we keep it free from any domain-specific knowledge. We investigate the architecture's properties, determine the playing performance that can be reached by using standard evolutionary methods, and finally verify that its performance scales well to larger board sizes.

II. BACKGROUND

A. Flexible-size board games

There is a large variety of board games, many of which either have flexible board dimensions, or have rules that can be trivially adjusted to make them flexible.

The most prominent of them is the game of Go, research on which has been considering board sizes between the minimum of 5x5 and the regular 19x19. The rules are simple[4], but the strategies deriving from them are highly complex. Players alternately place stones onto any of the intersections of the board, with the goal of conquering maximal territory. A player can capture a single stone or a connected group of his opponent's stones by completely surrounding them with his own stones. A move is not legal if it leads to a previously seen board position (to avoid cycling). The game is over when both players pass.

Go exhibits a number of interesting symmetries. Apart from the four straightforward axes of symmetry, it also has an approximate translational invariance, which is clearer the further away from the border one is.

Among the practical difficulties with conducting experiments on Go are the need to distinguish dead groups from alive and seki ones, keep track of the history of all board configurations, and the difficulty of handling pass moves. Many other machine learning approaches to Go simplify the rules to prevent some or all of those problems.

A number of variations of Go, of varying degrees of similarity, are played on the same board and share the symmetry properties of Go. The most common ones are Irensei, Pente, Renju, Tanbo, Connect, Atari-Go and Gomoku. In this paper we will conduct experiments on the latter two.

Atari-Go, also known as Ponnuki-Go or 'Capture Game', is a simplified version of Go that is widely used for teaching the game of Go to new players, because it introduces many of the key concepts without the full complexity [5]. The rules are the same than for Go, except that passing is not allowed, and the first player to capture a predetermined number (usually one) of his opponent's stones wins (see figure 9 for an example).

Compared to Go, this variant makes playing independent of history and makes it straightforward to determine the winner, which in turn makes automated playing easier and faster. It retains the concept of territory: as in the end no player may pass, each one has to fill his own territory and therefore the player with most territory wins. Other strategies of Go, such as building eyes, or recognizing life-and-death situations may be less important in Atari-Go, but are still present.

We believe that all this is a good justification for using

Atari-Go as a test problem instead of Go, especially in early stages, before high-level performance is reached on Go.

Gomoku, also known as 'Five-in-a-row', is played on the same board as Go, but the rules are even simpler. Players alternate putting stones onto any of the intersections on the board. The first player to have 5 connected stones in a row, column or diagonal, wins.

While trying to block off the opponent's lines, a player tries to keep its own lines unblocked, and potentially do multiple attacks at the same time, not all of which can be countered. These strategies, again, are heavily pattern-based [6].

B. Scalable neural architectures

A large variety of neural network architectures have been proposed for solving board games. Here, we will briefly mention some of those that exhibit a certain degree of scalability with respect to board size.

One approach is to use a limited *focus* size on the board, use a form of preprocessing on it, and then reuse it in an identical fashion across the rest of the board. The outputs of that stage are then fed into some other architecture that combines them (e.g. [7]).

A variant of this idea are convolutional networks [8], which repeat this step on multiple levels, thus capturing more than just local patterns. Still, they are limited to (manually) fixed focus sizes at each level.

'Roving-eye'-based architectures [9] contain one component with a fixed focus size that can be aimed at any part of the board. This is then combined with an active component that can rove over the board until it feels ready to take a decision.

Other architectures have been proposed [10], [6] which make use of weight-sharing to capture domain-specific symmetries, but these are limited to a particular game, and also restricted in what kind of strategies they can learn.

Simultaneous Recurrent Networks [11] are structured like cellular automata. They successfully incorporate the whole context and make use of symmetries, but are not very efficient.

Graves [12] has developed a more general architecture, called *Multidimensional Recurrent Neural Networks* (MDRNNs), which is a special case of the DAG-RNNs proposed by Baldi [13]. While successfully used for vision tasks [14], they have been largely neglected in the domain of games, with the notable exception of Wu et al. [15], who applied them to supervised learning of expert moves for Go.

MDRNNs are efficient, and we believe that they have precisely the qualities of scalability that we are looking for, while remaining general enough to be used on many different games.

III. PROPOSED ARCHITECTURE

In this section we will provide a brief description of MDRNNs in general and give the details of the specific form used here.

Standard recurrent neural networks (RNNs) are inherently one-dimensional, they are effective for handling sequences with a single (time-) dimension. MDRNNs are a generalization of RNNs that overcome this limitation and handle multi-dimensional inputs. In our case the single time dimension is replaced by the two space dimensions of the board [12].

Intuitively, imagine a unit $u_\searrow$ that *swipes* diagonally over the the board from top-left to bottom-right. At each board position (i, j), it receives as an input the information from that board position $in_{i,j}$ plus its own output from when it was one step to the left $u_{\searrow(i-1,j)}$, and from when it was one step up $u_{\searrow(i,j-1)}$. It processes those inputs and produces an output $u_{\searrow(i,j)}$. See also figure 1 for an illustration.
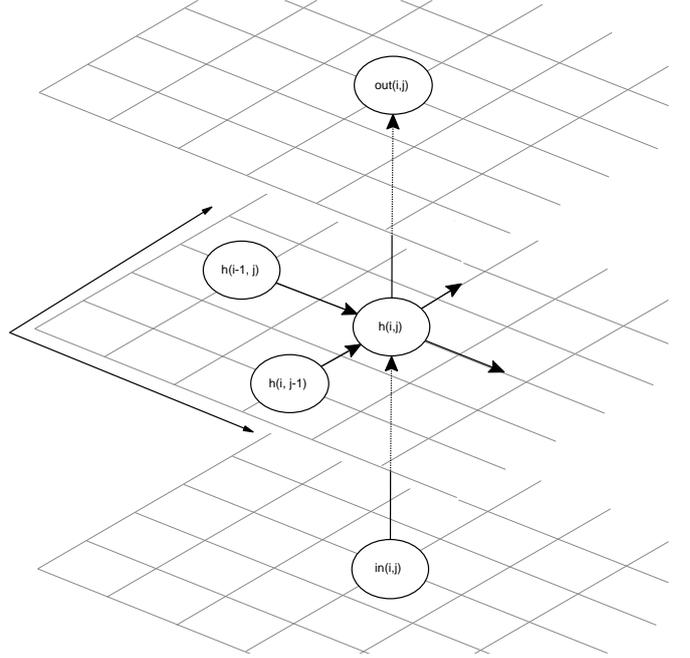

Fig. 1. MDRNN structure diagram.

Because of the recurrency, the unit has indirect access to board information from the whole rectangle between $(0, 0)$ and (i, j). It would be even more desirable to have such access to the whole board, which can be achieved by using 4 swiping units, one for each of the diagonal swiping directions in $D = \{\searrow, \nearrow, \swarrow, \nwarrow\}$ (this is a generalization of bidirectional RNNs). The output layer then, for every position, combines the outputs of the 4 units to a single value $out_{i,j}$ (which is potentially derived from the whole board information). The exact formulas are:

$$u_{\searrow(i,j)} = \tanh[w_i * in_{i,j} + w_h * u_{\searrow(i-1,j)} \\ + w_h * u_{\searrow(i,j-1)}]$$

(and analogous for the other directions)

$$out_{i,j} = \sum_{\diamond \in D} w_o * u_{\diamond(i,j)}$$

On the boundaries, where $u_\diamond$ is not defined, we replace it by a fixed value w_b (for all borders). In order to enforce symmetry, we use the same connection weights for all swiping units. Altogether, this then gives us 4 groups of weights in the network: w_i, w_h, w_o and w_b. The total number of weights is therefore $2k + k^2 + k + k = 4k + k^2$ where k is the number of hidden neurons in each swiping unit. In most of our experiments we use $k = 10$, and thus need to train only 140 weights, which is very little compared to other approaches.

At each position, the network takes two inputs which indicate the presence of a stone at this position. The first one is 1 if a stone of the networks own color is present and 0 otherwise, the second input encodes the presence of an opponent's stone in the same way. A black/white symmetric encoding, as used in other approaches (e.g. [10]) is not applicable here, because the output is not symmetrical: the best move for both players might be the same.

The output value at each position expresses the network's preference for playing there. A move is chosen by the network, by drawing a position from the Gibb's distribution (with adjustable temperature) of the output values. The choice is limited to legal moves, as provided by the game rules, so in practice the network outputs corresponding to illegal moves are ignored. If the temperature is set to zero, moves are selected greedily (randomly breaking ties).

For our implementation, we unfold the MDRNN along both dimensions and end up with a large but simple feed-forward network with a lot of weight-sharing. This makes evaluations efficient: on a normal desktop computer, the network needs about 2ms to choose a move on a 5x5 board, and 20ms on a 9x9 board.

Figure 2 illustrates how the network processes board inputs. In this example the network had 2 hidden neurons, and random weights. It is worth noting that in the space without any stones nearby the activations are symmetrical with respect to the border, but not around the stones.
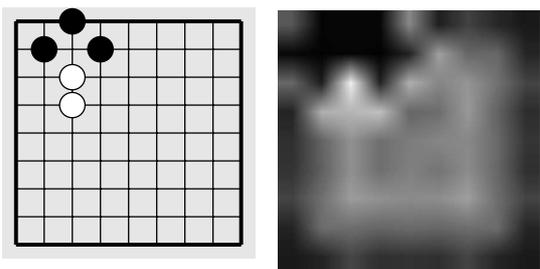

Fig. 2. Left: the board given as an input. Right: the output of the network (of the perspective of the white player), with brighter points corresponding to higher numbers.

IV. Methodology

A. Evolutionary methods

With a similar reasoning than for wanting to keep the architecture free from domain knowledge, we also want to use very common and general algorithms for training it. This makes sure that our results are more significant, as they are not an artifact of a particular algorithm, and can be easily reproduced.

We chose the well-established Evolution Strategies [16] for training the network weights directly. The fitness is determined by playing against a fixed opponent.

However, as training against a fixed opponent both biases the direction of evolution and limits performance to that of the opponent, we decided to also perform experiments with coevolution. For that, we use population-based competitive coevolution, based on the *host-parasite* paradigm (as described in [17]). In order to preserve diversity, we use the following standard enhancements (from [18]): *a*) shared fitness *b*) shared sampling *c*) hall of fame.

B. Experimental Setup

The two opponent players we are using throughout the experiments are:

- the *random player*, which randomly chooses any of the legal moves,
- the *naive player*, which does a one-ply search. If possible, it always picks a move that makes it win the game immediately, and never picks a move that would make it lose the game immediately. In all other cases (the large majority), it randomly picks a legal move.

As fitness we use the average score over 100 games against an opponent, alternating which player starts. In order to make it more informative than a pure win/lose score, we compute the score for a single game as follows:

$$
\text{score} = \begin{cases} 1 - p\frac{M - M_{min}}{M_{max} - M_{min}} & \text{if game won} \\[2ex] -1 + p\frac{M - M_{min}}{M_{max} - M_{min}} & \text{if game lost} \end{cases}
$$

with $p = 0.2$, M the number of moves done before the game is over, M_{min} the length of the shortest game and M_{max} the length of the longest game possible.

If not explicitly stated otherwise, we use the following settings:

- $k = 10$ hidden nodes (140 weights)
- greedy play (temperature = 0)
- random weights are drawn from the normal distribution $N(0, 1)$
- the default opponent is the naive player.

V. Experiments

This section provides the empirical results coming out of our study of applying our architecture to the two problem domains (Atari-Go and Gomoku).

We start by comparing the performance of our architecture with untrained weights to standard multi-layer perceptrons (MLP), also untrained. Next, we study its scalability by varying the board size. Then we train the architecture, using on the one hand simple evolution with as fitness the network's performance against the naive player, and on the

other hand competitive coevolution. Finally, we investigate whether the performance on those trained networks scales to larger board sizes.

A. Random weight performance

As a first experiment, we determine whether our architecture is suitable for the chosen problem domains. We therefore compare the untrained performance of our architecture with the performance of an untrained, standard MLP. We sample networks with a fixed architecture and random weights, and then evaluate their average performance over 100 games against one of the fixed opponents, and on different board sizes.

Here, the MLP are of the following standard form: input and output layers are the same than for the MDRNNs. There is a single fully connected hidden layer of sigmoid units. We experimented with different sizes for the hidden layer, and found the results not to be very sensitive to that choice. The presented results were produced with a hidden layer of 100 neurons.
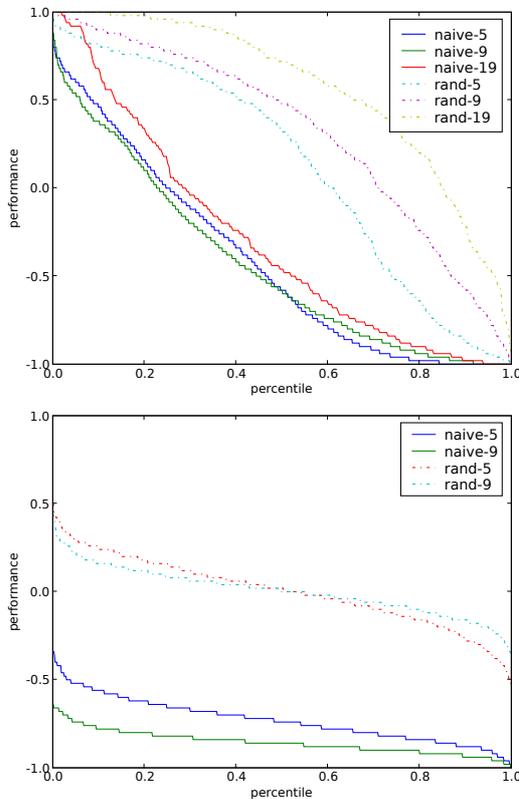

Fig. 3. Performance percentiles on Atari-Go, on board sizes 5x5 and 9x9 and against both the naive and the random opponent. Performance is the averaged score over 100 games. Above: MDRNN, below: MLP.

Figures 3 and 4 show the distribution of performance of networks with randomly guessed weights on both Atari-Go and Gomoku. The results are based on at least 200 samples per scenario. The results show clearly that it is much easier to get good or reasonable performance with random weights using an MDRNN compared to using a standard MLP with
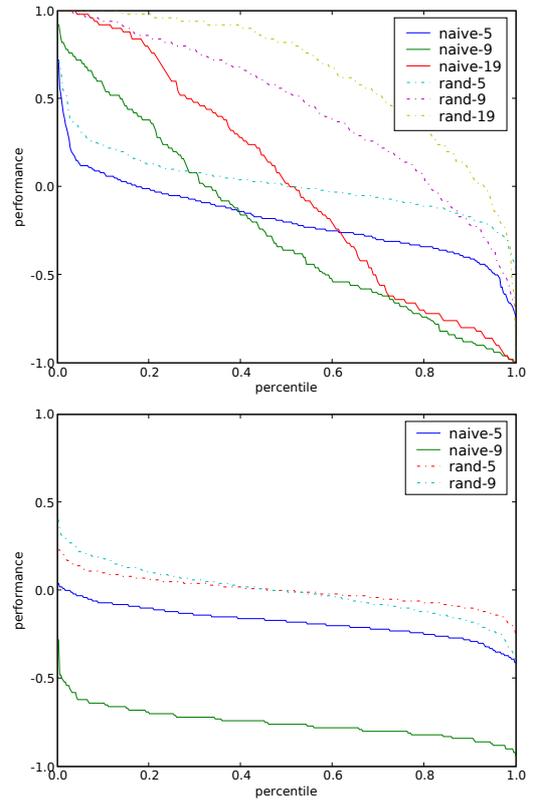

Fig. 4. Performance percentiles on Gomoku, on board sizes 5x5 and 9x9 and against both the naive and the random opponent. Performance is the averaged score over 100 games. Above: MDRNN, below: MLP.

the same number of hidden nodes. This is especially true on larger boards – in fact where the performance of MLPs goes down with increasing board size, it actually goes *up* for MDRNNs.

Interesting to note is also the performance distribution for Gomoku on board size 5x5: almost all networks of both kinds have a score close to zero, which means that the large majority of games ends in a draw. This is not surprising, as on such a small board even a random player easily disrupts the building of rows of five stones.

B. Scalability for untrained networks

The next experiment aims to determine the degree of scalability of our architecture. For that, we sample MDRNNs with random weights, and then measure their average performance (over 100 games) against one of the fixed opponents, on different board sizes. We then determine the linear correlation (Pearson coefficient), and also the proportion p of samples for which the score is higher on the larger board than on the smaller one.

Figure 5 plots the performance of random MDRNNs on 5x5 versus 9x9 against the random opponent. It provides a good visual intuition about the high correlation of performance between different board sizes.

Table I shows that this result holds for a broad range of scenarios. The results for all scenarios are based on at least
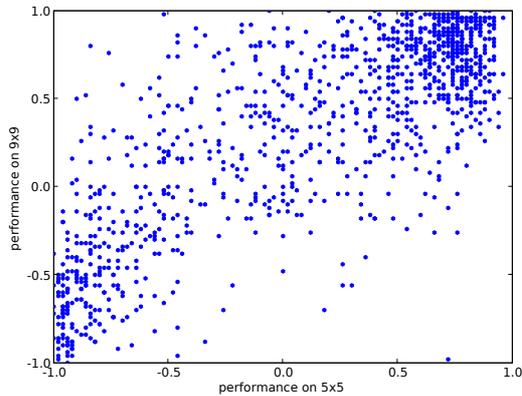
Fig. 5. Performance of the same random networks on different board sizes.

TABLE I
CORRELATION OF PLAY PERFORMANCE AGAINST A FIXED OPPONENT.

Game	Opponent	Sizes	Correlation	p
Atari-Go	Random	5x5 vs. 9x9	0.81	0.71
Atari-Go	Naive	5x5 vs. 9x9	0.70	0.49
Atari-Go	Random	7x7 vs. 11x11	0.78	0.73
Atari-Go	Naive	7x7 vs. 11x11	0.72	0.56
Atari-Go	Random	9x9 vs. 19x19	0.71	0.76
Atari-Go	Naive	9x9 vs. 19x19	0.66	0.61
Gomoku	Random	5x5 vs. 9x9	0.12	0.79
Gomoku	Naive	5x5 vs. 9x9	0.07	0.42
Gomoku	Random	7x7 vs. 11x11	0.60	0.78
Gomoku	Naive	7x7 vs. 11x11	0.64	0.58
Gomoku	Random	9x9 vs. 19x19	0.63	0.77
Gomoku	Naive	9x9 vs. 19x19	0.62	0.67

200 samples.

We find that the correlations are always positive. They are high in all scenarios, with the exception of Gomoku on size 5x5. We suspect that this is due to the large number of games ending in a draw (see section V-A) on that size. Another interesting observation is that in almost all cases p is significantly larger than the expected 0.5, which is another indication of the good scalability of the architecture.

In section V-E we will use the same methodology to determine the scalability for networks with trained weights.

C. Training against a fixed opponent

In order to determine whether our architecture can be trained to reach the level of play of a given fixed opponent, we use a simple evolution strategy.

Experimenting with many different parameters, we did not find their exact choice to be very sensitive, as long as the population size is large enough (simple hill-climbing performed much worse). We use the following settings:

- population size of 15, with an elitist selection of $\frac{1}{2}$, i.e. a (10+10) evolution strategy.
- no crossover or self-adaptation
- for mutation, each weight is changed by a number drawn from $N(0, 0.1)$

- the fitness evaluation is the one described in section IV-B, with respect to naive player.

All those settings are identical for both Atari-Go and Gomoku.

Figure 6 shows the performance during evolution. The results are the average values over 10 independent runs. As the fitness evaluation is noisy, we reevaluate it at every generation. This is also the reason that the performance of the best network per generation is not strictly increasing.
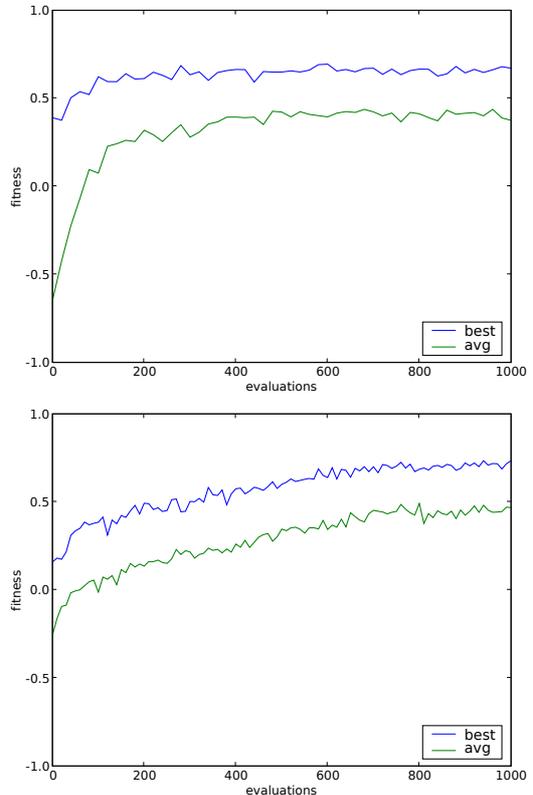

Fig. 6. Average and best population fitness during evolution. Above: Atari-Go, below: Gomoku.

The performance looks a bit better for Gomoku than for Atari-Go, but we can conclude that evolution easily finds weights that beat the naive player on most games, although is seems hard to reach really high performance.

D. Competitive Coevolution

To make our results independent of the biased fitness measure provided by a fixed opponent, we also train our architecture using coevolution.

Coevolution implies that there is no external fitness measure (as in section V-C), only a relative fitness. We compute the relative fitness of two players as the average score of an equal number of games with each player starting. In case both players are greedy, it is sufficient to play out two games per evaluation, otherwise we need to average over more, depending on the temperature of the Gibb's distribution.

We use a population size of two times 15, with an elitist selection of $\frac{1}{3}$ based on shared fitness. At every generation,

| Game	Parameters	Dominance number
Atari-Go	Non-elitist	14.8 ± 3
Atari-Go	Elitist	27.6 ± 14
Gomoku	Non-elitist	11.0 ± 5
Gomoku	Elitist	15.6 ± 11

Fig. 7. A typical coevolutionary run. Plotted is the performance of the generation champions of both populations against the naive player. The circles mark the dominant individuals (i.e. that beat all previous dominant ones).
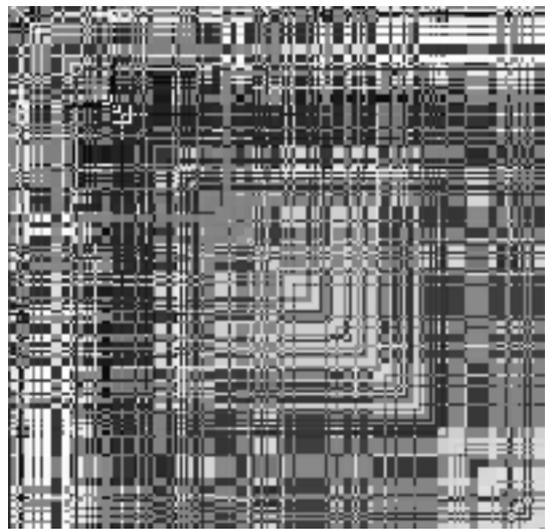

Fig. 8. Typical CIAO plot (same run than figure 7). Bright points correspond to a high score, dark ones to a low one.

every host is evaluated against 15 opponents, 5 of them parasites (according to shared sampling), and 10 players out of the hall of fame (which contains the player with the best relative fitness of each generation). Both populations exchange roles at each generation. We tried varying those settings, and did not find the exact parameters values to be very sensitive, as long as the population size is not too small and enough relative evaluations are done each generation. We also tried different temperatures for the move selection, but ended up falling back onto greedy play, as the results were not qualitatively different but much more expensive to compute due to the need for averaging.

There are a number of ways to track progress of coevolutionary runs. We will use three of them here: analyzing the absolute fitness of the evolved networks, CIAO plots ([19], see below) and dominance tournaments [20].

In a dominance tournament, we build a list of dominant individuals. The first one is the first generation champion. Then, while successively going through all other generation champions, we add it to the list if it can beat *all* previous dominant individuals. The length of that list then gives us the *dominance number*, and it is reasonable to consider that the higher that number is, the more progress has been made. Table II shows average dominance numbers for a number of different training scenarios.

Figure 7 shows a typical coevolution run. The performance plotted is the one measured against the naive player (which is inaccessible during evolution). It is interesting to note that this performance is not strictly increasing, and that even the champions of the dominance tournament do not always correspond to high absolute performance. As the high dominance numbers show progress nonetheless, this indicates that evolution based on relative fitness is coming up with different kinds of strategies than evolution based on absolute fitness.

To visualize the relative progress of the two populations during a coevolutionary run, we make CIAO plots, which show the performance of all generation champions of one population playing against all of those of the other. If coevolution is successful, we expect later generations to be better against earlier ones, thus to see brighter areas in the lower left and upper right. Figure 8 shows a typical CIAO plot which exhibits that property to a small degree. However, we can make two other, unpredicted, observations:

- the score values themselves are more extreme (i.e. the games are shorter) with players of earlier generations
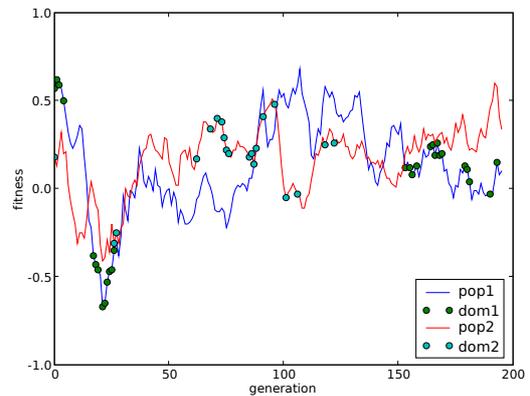
and more balanced in later generations (more grayish colors). This means that coevolution tends to lead to more careful players, which lose late if they do, but at the cost of not being able to win quickly against each other.

- carefully looking at individual lines in the plot, we find that often, if one player that is winning against one group of opponents and another player is losing to that group, then there is another group of opponents where those relations are exactly opposite. This seems to indicate a kind of rock-paper-scissors situation, which could be the reason why the absolute level of play does not progress as much as expected during coevolution.

Even with few weights, the MDRNN is capable of a wide *variety* of behaviors. When inspecting the generation champions of a coevolutionary run, we find a large diversity of those behaviors. Due to space constraints we can show only a few: figure 9 shows a few games, with the same

Game	Method	Train size	Test size	Correlation	p
Atari-Go	Coevolution	5	9	0.45	0.92
Atari-Go	Evolution	5	9	0.05	0.61
Atari-Go	Evolution	7	11	0.07	0.98
Gomoku	Coevolution	5	9	-0.11	0.70
Gomoku	Evolution	5	9	0.22	0.73
Gomoku	Evolution	7	11	0.43	0.85

Fig. 10. Performance of the same networks on different board sizes. All networks are generation champions of coevolutionary runs. Above: Atari-Go, below: Gomoku.

players (generation champions of a typical coevolutionary run) on board sizes 5x5 and 9x9. Naturally the behavioral patterns look different on a small board and on a large one, but they share common features.
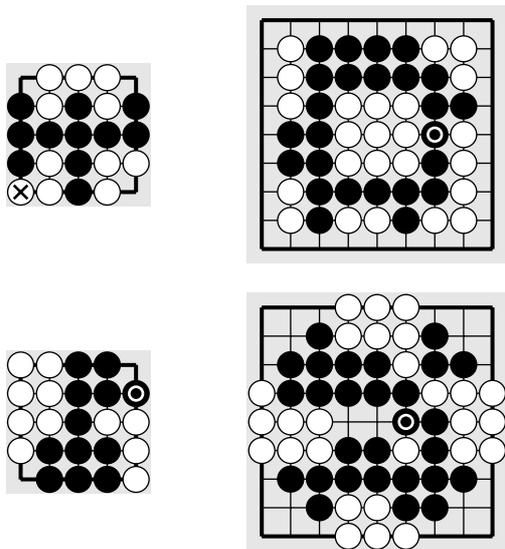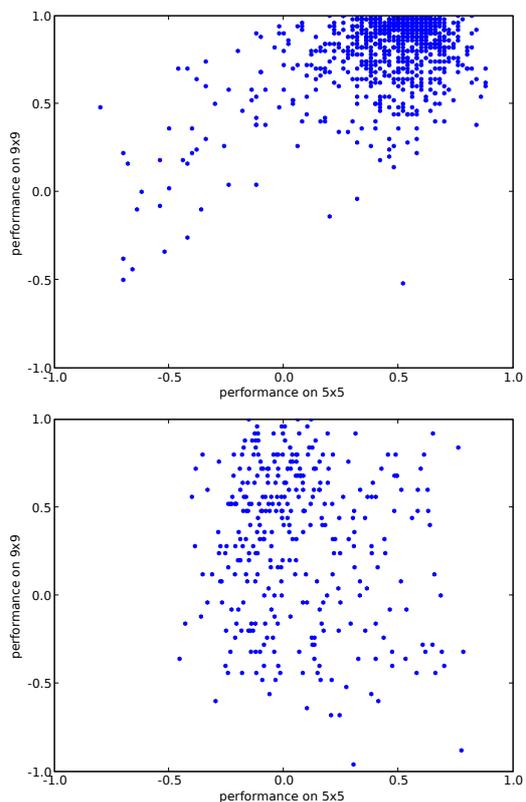

Fig. 9. Illustrative games of some champions: here black is always the same network, but the white players in the two rows are different.

E. Scalability for trained networks

The game-play on the 5x5 board is quite different from the one on 9x9, so there is a clear risk of evolution exploiting the training size and making the resulting networks not scale up anymore. Thus, as the final experiment, we attempt to determine the scalability of our architecture after training. For that, make use of the networks trained in sections V-C and V-D, and apply the same methodology than in section V-B.

Figure 10 shows how the performance (against the naive player) of networks trained on the 5x5 board scales to a board size of 9x9. Table III then gives the detailed results for all scenarios. The numbers are based on a minimum of 200 trained networks per scenario which are always generation champions (ignoring the champions of the first 50 generations).

Comparing the correlations and the proportions p to the values we found for untrained networks in section V-B, we find that generally the correlations are much lower, but p is much higher. This means that, for trained networks, the performance on a small board is less predictive of the performance on the large board, *but* it is almost always significantly higher on the large board.

VI. DISCUSSION AND FUTURE WORK

Revisiting the goals we set ourselves in the introduction, we can now draw some conclusions. By construction, our architecture is directly usable on different board sizes, even on different games. Apart from incorporating the board symmetries, it is free from domain-specific knowledge, just as the training methodology does not rely on any domain-specific heuristics: the naive player that is used for most purposes is only using the game rules.

The variety of empirical results provide a solid confirmation of our main hypothesis, that the architecture is scalable. We could show that training on a small board size transfers well onto larger ones. A very promising result is that the performance of the large majority of trained networks actually *increases* when they are tested on networks larger than the ones they were trained on.

A possible objection to the claim that our architecture is scalable is that part of the observed effect might be due to the weak opponent, which might scale very weakly. Further investigations, e.g. using a heuristic opponent, should be able to easily confirm or refute this interpretation.

The architecture appears to be appropriate for the domain, at least at low levels of play. To open up possibilities for higher levels of performance, we propose to address the following two weaknesses:

- to allow the network to make better use of *long-distance dependencies* (which are especially common in a game like Go), we suggest to replace the hidden units by LSTM cells [21].
- to allow for more complex strategies to evolve, we suggest to stack two (or more) MDRNNs on top of each other, so that the outputs of the swiping units of one form the set of inputs to the (independently) swiping units of the next one.

Preliminary experiments suggest that those enhancements may be worth it.

The fitness measure we used was sufficient for the purposes of this paper, but we believe that, in order to reach more ambitious performance goals, a *multi-objective* approach may be superior. Objectives could include: chance of winning with black, with white, number of moves until losing or winning, performance against different static opponents, and all those objectives used on multiple board sizes.

Our approach has been to strictly avoid domain knowledge, but it it clearly possible, even desirable, to incorporate some in order to reach competitive performance. Most of the standard ways for doing so can be directly applied to our architecture as well. For example, we could feed the network a number of domain-specific features [15] instead of the raw board. On the other hand we could easily adjust the output to generate a value function instead of moves, and then use that in a standard search-based framework, possibly with heuristic pruning of the search tree.

VII. Conclusion

We have developed and thoroughly investigated the properties of a scalable neural network architecture based on MDRNNs for board games. We could show that it is scalable, suitable for the domain, can be trained easily and the results of training scale well to larger boards.

As our aim was not to reach the highest performance, we avoided using any domain knowledge, which made it possible to use the same setup for two different games, and reach similar results. We therefore believe that it can be used on many similar problems as well, most notably on Go.

Naturally, our next goal will be to use the present results as a foundation for the more ambitions project of reaching state-of-the-art performance, by adding domain-specific knowledge on top of it.

Acknowledgments

References

[1] E. C. D. van der Werf, H. J. V. D. Herik, and J. W. H. M. Uiterwijk, "Solving go on small boards," *International Computer Games Association Journal*, vol. 26, pp. 10–7, 2003.

[2] T. P. Runarsson and S. M. Lucas, "Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go," *IEEE Transactions on Evolutionary Computation*, pp. 628–640, 2005.

[3] N. Richards, D. E. Moriarty, and R. Miikkulainen, "Evolving neural networks to play go," *Applied Intelligence*, vol. 8, pp. 85–96, 1997.

[4] K. Iwamoto, *Go for beginners*. Tokyo, Japan: Ishi Press, 1972.

[5] G. Konidaris, D. Shell, and N. Oren, "Evolving neural networks for the capture game," in *Proceedings of the SAICSIT Postgraduate Symposium*, 2002.

[6] B. Freisleben and H. Luttermann, "Learning to Play the Game of Go-Moku: A Neural Network Approach," *Australian Journal of Intelligent Information Processing Systems, Vol. 3, No. 2*, pp. 52 – 60, 1996.

[7] D. Silver, R. S. Sutton, and M. M. 0003, "Reinforcement learning of local shape in the game of go," in *IJCAI*, 2007, pp. 1053–1058.

[8] Y. Lecun and Y. Bengio, *Convolutional Networks for Images, Speech and Time Series*. The MIT Press, 1995, pp. 255–258.

[9] K. O. Stanley and R. Miikkulainen, "Evolving a roving eye for go," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2004.

[10] N. N. Schraudolph, P. Dayan, and T. J. Sejnowski, "Temporal difference learning of position evaluation in the game of go," in *Advances in Neural Information Processing Systems*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds., vol. 6. Morgan Kaufmann, San Francisco, 1994, pp. 817–824.

[11] X. Pang and P. J. Werbos, "Neural network design for j function approximation," in *in Dynamic Programming, Math. Modelling and Scientific Computing (a Principia Scientia journal)*, 1996.

[12] A. Graves, "Supervised sequence labelling with recurrent neural networks," Ph.D. in Informatics, Fakultat für Informatik – Technische Universität München, Boltzmannstrasse 3, D - 85748, Garching bei München, Germany, 2008.

[13] P. Baldi and G. Pollastri, "The principled design of large-scale recursive neural network architectures dag-rnns and the protein structure prediction problem," *Journal of Machine Learning Research*, vol. 4, pp. 575–602, 2003.

[14] A. Graves, S. Fernández, and J. Schmidhuber, "Multidimensional recurrent neural networks," in *Proceedings of the 2007 International Conference on Artificial Neural Networks*, Porto, Portugal, September 2007.

[15] L. Wu and P. Baldi, "A scalable machine learning approach to go," in *Advances in Neural Information Processing Systems 19*, B. Schölkopf, J. Platt, and T. Hoffman, Eds. Cambridge, MA: MIT Press, 2007, pp. 1521–1528.

[16] T. Bäck, F. Hoffmeister, and H.-P. Schwefel, "A survey of evolution strategies," in *Proc. Fourth Int'l Conf. Genetic Algorithms (ICGA'91), San Diego CA*, R. K. Belew and L. B. Booker, Eds. San Mateo CA: Morgan Kaufmann, 1991, pp. 2–9.

[17] A. Lubberts and R. Miikkulainen, "Co-evolving a go-playing neural network," in *Genetic and Evolutionary Computation Conference Workshop Program*. Morgan Kaufmann, 2001, pp. 14–19.

[18] C. D. Rosin and R. K. Belew, "Methods for competitive co-evolution: Finding opponents worth beating," in *Proceedings of the Sixth International Conference on Genetic Algorithms*, L. J. Eshelman, Ed. San Francisco, CA: Morgan Kaufmann, 1995. [Online]. Available: http://www.mpi-sb.mpg.de/services/library/proceedings/contents/icga95.html

[19] D. Cliff and G. F. Miller, "Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations," in *Advances In Artificial Life*. Springer Verlag, 1995, pp. 200–218.

[20] K. O. Stanley and R. Miikkulainen, "Competitive coevolution through evolutionary complexification," *Journal of Artificial Intelligence Research*, vol. 21, pp. 63–100, 2004.

[21] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 9, pp. 1735–1780, 1997.