Multi-Dimensional Deep Memory Atari-Go Players for Parameter Exploring Policy Gradients

Mandy Grüttner[1], Frank Sehnke[1], Tom Schaul[2], and Jürgen Schmidhuber[2]

[1]Faculty of Computer Science, Technische Universität München, Germany
[2]IDSIA, University of Lugano, Switzerland

Abstract. Developing superior artificial board-game players is a widely-studied area of Artificial Intelligence. Among the most challenging games is the Asian game of Go, which, despite its deceivingly simple rules, has eluded the development of artificial expert players. In this paper we attempt to tackle this challenge through a combination of two recent developments in Machine Learning. We employ Multi-Dimensional Recurrent Neural Networks with Long Short-Term Memory cells to handle the multi-dimensional data of the board game in a very natural way. In order to improve the convergence rate, as well as the ultimate performance, we train those networks using Policy Gradients with Parameter-based Exploration, a recently developed Reinforcement Learning algorithm which has been found to have numerous advantages over Evolution Strategies. Our empirical results confirm the promise of this approach, and we discuss how it can be scaled up to expert-level Go players.

1 Introduction

The two-player board game Go is one of the few such games that have resisted a panoply of attempts from Artificial Intelligence at building expert-level players. A broad range of techniques have been used, with some recent successes based on Monte Carlo Tree Search in combination with Reinforcement Learning (see e.g. [1, 2]). A large body of research has dealt with the problem using techniques based on Neural Networks (see e.g. [3] for an overview), and that is also the approach taken in this paper.

The recently developed Neural Network architecture called Multi-dimensional Recurrent Neural Networks (MDRNN [4]) has been shown to be highly suited to domains like board games with multi-dimensional inputs. Unlike typical flat networks (e.g. multi-layer perceptrons), they can incorporate spacial structure as well as symmetries in a very natural way. It has also been shown that MDRNNs trained on small game boards can be scaled up to play well on larger game boards, even without further training [5].

Training neural networks to play well with direct policy search (i.e. optimizing the controller network's parameters) can be done in a number of ways. Recent work [5] has used state-of-the-art black-box optimization methods like CMA-ES [6], which unfortunately does not scale well to larger numbers of weights, as required for more complex playing behavior. Other methods like Evolution Strategies (ES [7]) scale better but suffer from a relatively slow convergence. We therefore train our Go-playing networks using the novel Policy Gradients with Parameter-based Exploration (PGPE, [8]), which have recently been shown to be very successful at optimizing the parameters of large Neural Network controllers [9]. PGPE replaces the usual explicit policy of Reinforcement Learning with an implicit one, defined by a distribution over the parameters of the controller. The fitness for each sequence only depends on one sample and is therefore less noisy.

In section 2.1 we briefly introduce the game of Go and the simplified variant used here. The MDRNN architectures are described in Section 2.2, and sections 2.3. Section 2.4 introduces the three algorithms used (ES, CMA-ES and PGPE, respectively). Then, in section 3, we train MDRNNs using PGPE, CMA-ES and ES to play Go, empirically establishing the advantages of PGPE over ES and CMA-ES. Conclusions and an outlook on future work are presented in Section 6.

2 Method

In this section we give the needed background on the game of Go, the Neural Network architectures (MDRNN and MDLSTM) and the training algorithms (ES, CMA-ES and PGPE).

2.1 Go and Capture Game

For the comparison of the different methods we are using the Capture Game, a simplified version of the two-player board game Go, a game frequently used to demonstrate the power of algorithms [1, 11, 12]. In Go, the players alternately make a move by placing a stone on the board. They aim to capture groups of opposing stones by enclosing them, see Figure 1 for an illustration. The goal is to capture more stones and to surround more territory than the opponent (see [3] for more details).

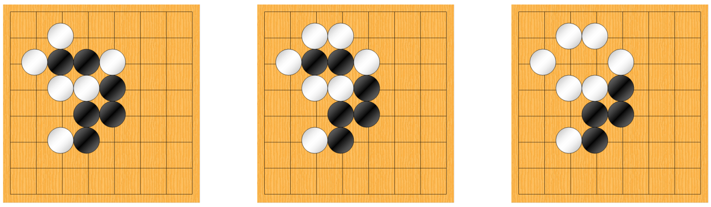

Fig. 1. A typical situation in Go: the turn is with white (left), who decides to capture a group of black stones (middle), which is then removed from the board (right).

The Capture Game, also called Atari-Go or Ponnuki-Go, uses the same rules as Go, except passing is not allowed and the goal of the game simplified: the first player who captures at least one opposing stone wins. As this goal is achieved earlier and with less complex strategies, this variant of Go is often used for teaching new players.

Go is very interesting in combination with MDRNNs and MDLSTMs, because scalability is an important issue for board games [10]. The original game board consists of 361 (19x19) fields, but it is possible to use smaller board sizes for teaching, or to shorten the game length. As the main strategies stay the same, it is possible to train on a small board size and then play on bigger ones. In our case we could use small Neural Networks for the training and afterwards use scaled versions to play on bigger boards.

2.2 Multi-dimensional Recurrent Neural Networks

Real world data often consists of multi-dimensional data such as videos, speech sequences or board games (as in our case). To use this data with regular Neural Networks (NN) the data must be transformed into a vector which leads to the loss of topological information about the inputs. Multi-dimensional Recurrent Neural Networks (MDRNN), instead, are capable of using high-dimensional data without this transformation. Furthermore MDRNNs can be trained on small problem instances (e.g. board sizes) and then used on bigger ones, a process we call *scaling*.

Compared to standard Recurrent Neural Networks (RNN), which can only deal with a single (time-)dimension, MDRNNs [4] are able to handle multi-dimensional sequences and were used successfully for vision [13], handwriting recognition [14] and different applications of Go [5, 15, 10, 3].

In the case of Go, the single time dimension is replaced by the two space dimensions of the game board. It would be worthwhile to get information about the whole board. Therefore we introduce *swiping* hidden layers which *swipe* diagonally over the board. The four directions that arise out of the described situation are the following: $D = \{\nearrow, \searrow, \nwarrow, \swarrow\}$.

As exemplary hidden layer we describe the layer $h_{\nearrow}$, which swipes diagonally over the board from bottom-left to top-right, in detail. At each position (i, j) of the board we define the activation $h_{\nearrow(i,j)}$ as a function of the weighted input $in_{(i,j)}$ and the weighted activations of the previous steps $h_{\nearrow(i-1,j)}$ and $h_{\nearrow(i,j-1)}$ which leads to:

$$h_{\nearrow(i,j)} = f(w_i * in_{(i,j)} + w_h * h_{\nearrow(i-1,j)} + w_h * h_{\nearrow(i,j-1)}) \tag{1}$$

where f is a function (e.g. $f = tanh$). On the boundaries fixed values are used: $h_{\nearrow(i,0)} = h_{\nearrow(0,i)} = w_b$. An illustration of $h_{\nearrow}$ for the game Go can be found in Figure 2. The output layer consists of the combination of all swiping directions and could be described as following:

$$out_{i,j} = g\left(\sum_{\diamond \in D} w_o * h_{\diamond(i,j)}\right) \tag{2}$$

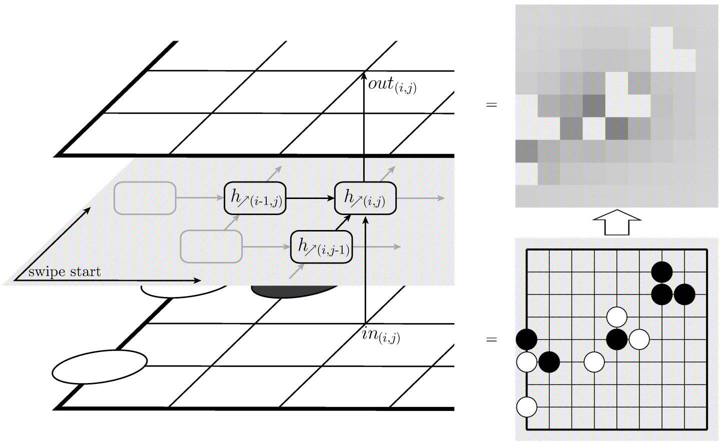

Fig. 2. On the left hand side the schematic illustration of a MDRNN shows how the output consists of a swiping hidden layer in one direction. The right hand side illustrates the output (top) to the corresponding input (bottom). The brighter the square, the lower the preference to perform the corresponding move (source [5]).

where g is typically the sigmoid function.

With the derived equation we have access to the whole game board. Nevertheless the reach of the access is limited by how fast the activations decay through the recurrent connections. This problem could be solved by using Long Short-Term Memory (LSTM) cells [4]. LSTMs are using gates to protect recurrent states over the time and where used successfully in [4, 3, 13]. The integration of LSTMs in MDRNNs by using swiping layers consisting of LSTM cells is called MDLSTM [5].

2.3 Evolution Strategies

Evolution Strategies (ES) are optimization techniques which are based on the principles of natural evolution, producing consecutive generations of individuals. During a generation a selection method is used to select specific individuals which form the new generation by recombination and mutation [16, 17]. Individuals can be solution candidates of any problem domain that is fully defined by a parameter set. Neural Networks (NN) and in this case MDRNNs fall into this class of problem domains, assuming the architecture of the NN is kept fixed, as the behavior of the NN is fully defined by its weight matrix.

Adapting continuous paramters by adding normally distributed noise is a typical mutation method. We use for our comparisons the local mutation operator and Covariance Matrix Adaption Evolution Strategy (CMA-ES) [6]. CMA-ES uses a covariance matrix $C \in \mathbb{R}^{n \times n}$, where n is the number of parameters for the mutation and achieves a derandomized correlated mutation. The covariance matrix approach is only feasible in relatively low-dimensional problem domains, because the size of the matrix grows with n^2. Here again it is advantageous that MDRNNs are scalable and that we can train the behavior on smaller instances of the game and scale it up to the full game size after learning.

2.4 Policy Gradients with Parameter-Based Exploration

In what follows, we briefly summarize [18], outlining the derivation that leads to PGPE. We give a short summary of the algorithm as far as it is needed for the rest of the paper.

In the standard Reinforcement Learning (RL) setting a reward signal at every time step in the Markovian decision process is given. We can associate a cumulative reward r with each history h by summing over the rewards at each time step: $r(h) = \sum_{t=1}^{T} r_t$. This makes the setting strictly episodic (natural for board games). In this setting, the goal of RL is to find the parameters θ that maximize the agent's expected reward

$$J(\theta) = \int_H p(h|\theta) r(h) dh \tag{3}$$

An obvious way to maximize $J(\theta)$ is to find $\nabla_\theta J$ and use it to carry out gradient ascent. Noting that the reward for a particular history is independent of θ, and using the standard identity $\nabla_x y(x) = y(x) \nabla_x \log y(x)$, we can write

$$\nabla_\theta J(\theta) = \int_H \nabla_\theta p(h|\theta) r(h) dh = \int_H p(h|\theta) \nabla_\theta \log p(h|\theta) r(h) dh \tag{4}$$

PGPE replaces the probabilistic policy commonly used in PG with a probability distribution over the parameters θ, where ρ are the parameters determining the distribution over θ. The expected reward with a given ρ is

$$J(\rho) = \int_\Theta \int_H p(h, \theta|\rho) r(h) dh d\theta. \tag{5}$$

Noting that h is conditionally independent of ρ given θ, we have $p(h, \theta|\rho) = p(h|\theta)p(\theta|\rho)$ and therefore $\nabla_\rho \log p(h, \theta|\rho) = \nabla_\rho \log p(\theta|\rho)$. Substituting this into Eq. (5) yields Eq. (6) under the notion of several conditionally independencies.

$$\nabla_\rho J(\rho) = \int_\Theta \int_H p(h|\theta) p(\theta|\rho) \nabla_\rho \log p(\theta|\rho) r(h) dh d\theta \tag{6}$$

where $p(h|\theta)$ is the probability distribution over the parameters θ and ρ are the parameters determining the distribution over θ. Clearly, integrating over the entire space of histories and parameters is unfeasible, and we therefore resort to sampling methods. This is done by first choosing θ from $p(\theta|\rho)$, then running the agent to generate h from $p(h|\theta)$:

$$\nabla_\rho J(\rho) \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_\rho \log p(\theta|\rho) r(h^n) \tag{7}$$

If we assume that ρ consists of a set of means $\{\mu_i\}$ and standard deviations $\{\sigma_i\}$ that determine an independent normal distribution for each parameter θ_i in θ. some rearrangement gives the following forms for the derivative of $\log p(\theta|\rho)$ with respect to μ_i and σ_i:

$$\nabla_{\mu_i} \log p(\theta|\rho) = \frac{(\theta_i - \mu_i)}{\sigma_i^2} \qquad \nabla_{\sigma_i} \log p(\theta|\rho) = \frac{(\theta_i - \mu_i)^2 - \sigma_i^2}{\sigma_i^3}, \quad (8)$$

which can then be substituted into (7) to approximate the μ and σ gradients that gives the PGPE update rules. We also used the for PGPE standard Symmetric Sampling (SyS) and the reward normalization commonly used for PGPE. See [18] for details.

3 Experiments

In this section we compare PGPE with ES and CMA-ES on different board sizes and with different MDRNNs. For ES we chose a (μ, λ)-strategy where the μ best individuals are chosen from the whole population which has size λ. In particular, we applied local mutation and used $\mu = 5$ and $\lambda = 30$ which are standard values. The implementations of the Capture Game, the algorithms and the Neural Network architectures are available in the open-source Machine Learning library PyBrain [19].

3.1 Fitness function

The evaluation of the individuals is realized with a Greedy Go Player, implemented in Java using depth-first search. It first checks whether it can capture and thereby defeat the opponent directly. Otherwise it tries to defend its position, by counting the number of liberties for its groups of stones. If one of its groups only has one liberty, and therefore he would be defeated during the next opponents move, the Greedy Player tries to enlarge this group. As a third choice the Greedy player uses a heuristic. Let p and q be the number of liberties of the weakest group of the Greedy Player and the opponent Player. The Greedy Player chooses a valid move which maximizes the sum $p - q$.

By reason of implementation the Greedy Player may pass. As the Capture Game does not allow this move, we replace it with a random move instead. Primarily this happens during games with strong opponents.

To calculate the fitness we averaged 40 games which were played against the Greedy Player. The fitness values are scaled from -1 (individual never wins against Greedy Player) up to +1 (individual always wins against Greedy Player).

3.2 Network Topology

With the given architectures of MDRNNs (MDLSTMs) it follows that we have 12 (52) parameters which have to be evaluated. We will give a short calculation for

MDRNNs. As mentioned in 2.2 our network consists of four (identical) hidden layers. The hidden layer is modeled by k neurons. Each neuron is connected with a weight w_o to the output layer and two weights w_i to the input layer. Furthermore the neurons of the hidden layer are fully connected to each other which leads to k^2 weights which we call w_h. Additionally we have k weights w_b which are fixed and model the boarders of the recurrent connections. All together we get $k + 2k + k^2 + k = 4k + k^2$ weights. Taking into consideration the additional weights of LSTM-cells, a similar reasoning gives us $16k + 5k^2$ weights for MDLSTMs. We decided to use $k = 2$ neurons, the smallest number that allows for qualitatively interesting strategies, which leads to 12 (52) weights. The decisions was taken concerning previous results (see [5]). A larger number of neurons mostly results in a faster conversion, but increases the complexity of the network and therefore the calculation time. However, the use of larger board sizes would make a larger number (up to $k = 5$) of neurons more feasible.

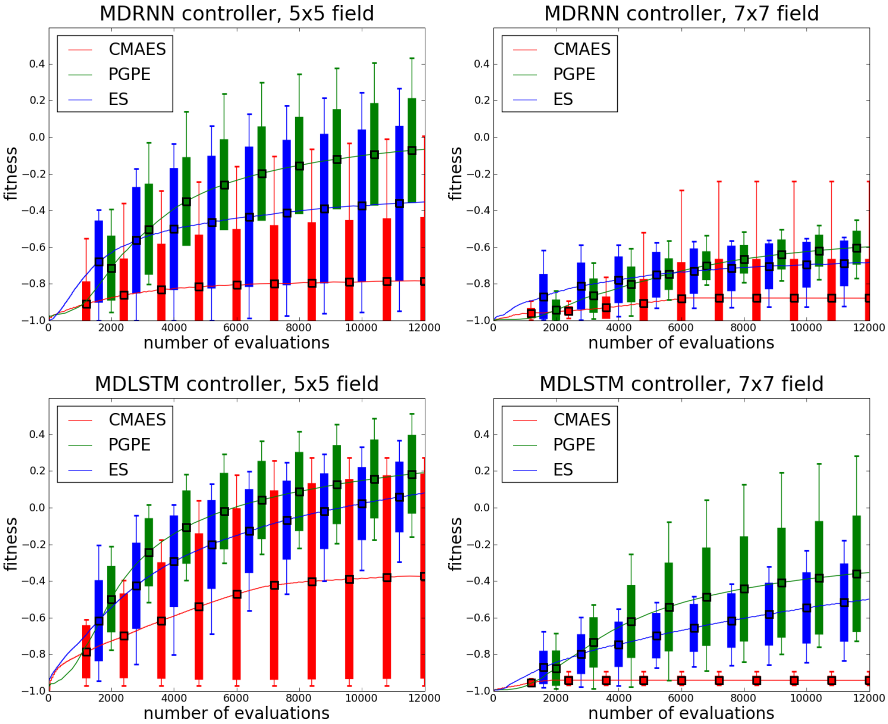

Fig. 3. Illustration of the four different types of networks. The plots give the fitness for each of the 12000 episodes as well as the standard deviation and min/max-values (average over 10 independent experiments).

3.3 Results

We trained MDRNNs and MDLSTMs for the board sizes 5 and 7. Furthermore we used 12000 episodes and averaged over 10 independent runs per data point. Figure 3 illustrates the results. The fitness value determines the average fitness of a generation. As we can see PGPE mostly converges faster than ES and CMA-ES. Primarily with the increasing of the number of parameters the advantages of PGPE towards ES increase.

Nevertheless neither ES nor PGPE has converged within the 12000 episodes to the maximum fitness value 1. This holds for the best individuals of each generation, too. In our experiments the best result of a single run of PGPE converges to 0.5 which is equivalent to a victory rate of 75% (see Figure 3 MDLSTM controller, 5x5 field). That is why ongoing learning could still improve the results.

Furthermore the use of MDLSTMs leads to better results than MDRNNs. This strength of MDLSTMs is accompanied by a long training time towards MDRNNs. Our observations are similar to [4, 5, 3].

Another fact we could read from our resulting plots is a big standard deviation. This observation leads to the suggestion (see section 6) that the standard meta parameters for PGPE and ES are not optimal for this problem domain and that meta-parameters that favor a more thorough exploration combined with longer learning cycles should provide better and more stable results.

4 Discussion

As is common for PGPE, the results of 3.3 start off with the rather slow phase of searching for the attractor of the global optima. This gives the PGPE curves the typical S-shape [18]. The ES curves form the usual saturation shape, with a faster convergence early on. However, PGPE takes over soon in the convergence process and then converges faster and onto a higher fitness level than ES. The resulting curve of CMA-ES does not reach the results of the other two methods. Especially while using a game board size of 7x7 CMA-ES prematurely converges to a low fitness value. CMA-ES seems to be to greedy for this task and thus converges premature.

One general observation from our experiments was that the longer the episodes and the higher the number of parameters, the more PGPE outperforms ES (in average fitness).

For general Go and other real-world problems more episodes are necessary. Future applications with stacked MDRNNs are possible, as suggested in [10], and for such applications PGPE seems more appropriate than ES or CMA-ES.

In summary, we find that PGPE performs better in finding good game behaviors, already on the smallest scaling level. It also scales better to scenarios with more episodes, and to higher dimensionalities of controllers.

5 Future Work

An interesting future application would be the research of the influence of PGPE on scaling MDRNNs as well as determining the best ratio between game board size and PGPE setup (especially using non standard meta-parameters like smaller step sizes for more thorough exploration and better final behavior). Besides, PGPE could be used for relearning the scaled controllers.

As suggested for ES in [3], we could use Co-Evolution to further improve the PGPE results. For PGPE this would mean the fitness is evaluated not only against the Java Player but also against the best learned controller(s) so far, and the controller defined by the mean of the current parameter set.

Furthermore, adaptively increasing the number of games per fitness evaluation could be used to speed up learning. In the early phase of learning, 3-4 games would be enough for an evaluation step, whereas up to 100 games might be necessary later on, to calculate a fitness value accurate enough to distinguish the slight changes in performance at that point.

As mentioned in section 3, the high standard deviation suggests that a higher rate of exploration would be favorable for the overall performance and stability. For PGPE this would correspond to decreasing the values of the two step sizes that are normally set to $\alpha_\mu = 0.2$ and $\alpha_\sigma = 0.1$. Not surprisingly however, this more thorough exploration comes at the price of longer convergence time.

6 Conclusion

In this paper we have introduced different methods of Machine Learning: PGPE, an algorithm based on a gradient based search through model parameter space, ES and CMA-ES, based on population based search. We compared these methods on the task of playing the Capture Game, a variant of Go, on small boards. Our experiments allow us to conclude that PGPE is advantageous on the given task, and also appears to scale better to larger and more difficult variants of the Go game. This is in line with similar results for PGPE on different benchmarks [18].

References

1. Bouzy, B., Chaslot, G.: Monte-Carlo Go Reinforcement Learning Experiments. In: In IEEE 2006 Symposium on Computational Intelligence in Games, IEEE (2006) 187–194
2. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: ICML; Vol. 227. (2007)
3. Grüttner, M.: Evolving Multidimensional Recurrent Neural Networks for the Capture Game in Go (2008)
4. Graves, A.: Supervised Sequence Labelling with Recurrent Neural Networks. PhD thesis, Technische Universität München (2007)
5. Schaul, T., Schmidhuber, J.: Scalable neural networks for board games. In: International Conference on Artificial Neural Networks (ICANN). (2009)

6. Hansen, N., Ostermeier, A.: Completely derandomized self-adaptation in evolution strategies. Evolutionary Computation **9** (2001) 159–195
7. Schwefel, H.: Evolution and optimum seeking. Wiley New York (1995)
8. Sehnke, F., Osendorfer, C., Rückstieß, T., Graves, A., Peters, J., Schmidhuber, J.: Policy gradients with parameter-based exploration for control. In: Springer LNCS proceedings of ICANN (in print). (2008)
9. Rückstieß, T., Sehnke, F., Schaul, T., Wierstra, D., Sun, Y., Schmidhuber, J.: Exploring parameter space in reinforcement learning. Paladyn **1**(1) (2010) 1–12
10. Schaul, T., Schmidhuber, J.: A scalable neural network architecture for board games. In: Proceedings of the IEEE Symposium on Computational Intelligence in Games (CIG 08). (2008)
11. Konidaris, G., Shell, D., Oren, N.: Evolving Neural Networks for the Capture Game. In: Proceedings of the SAICSIT Postgraduate Symposium. (2002)
12. Stanley, K.O., Miikkulainen, R.: Evolving a Roving Eye for Go (2004)
13. Graves, A., Fernández, S., Schmidhuber, J.: Multi-Dimensional Recurrent Neural Networks (2007)
14. Liwicki, M., Graves, A., Fernández, S., J., H.B., Schmidhuber: A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks. In: Proc. 9th Int. Conf. on Document Analysis and Recognition. (September 2007) 367–371
15. Wu, L., Baldi, P.: A scalable machine learning approach to go. In: in Advances in Neural Information Processing Systems 19, MIT Press (2007) 1521–1528
16. Streichert, F., Ulmer, H.: JavaEvA - A Java Framework for Evolutionary Algorithms. Technical Report WSI-2005-06, Centre for Bioinformatics Tübingen, University of Tübingen (2005)
17. Streichert, F.: Evolutionary Algorithms in Multi-Modal and Multi-Objective Environments. PhD thesis (2007)
18. Sehnke, F., Osendorfer, C., Rückstieß, T., Graves, A., Peters, J., Schmidhuber, J.: Parameter-exploring policy gradients. Neural Networks **23**(4) (2010) 551–559
19. Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstieß, T., Schmidhuber, J.: PyBrain. Journal of Machine Learning Research **11** (2010) 743–746