Deep Reinforcement Learning for Morpion Solitaire

Boris Doux, Benjamin Negrevergne, and Tristan Cazenave

LAMSADE, Université Paris-Dauphine, PSL, CNRS, Paris, France
Boris.Doux@dauphine.psl.eu

Abstract. The efficiency of Monte-Carlo based algorithms heavily relies on a random search heuristic, which is often hand-crafted using domain knowledge. To improve the generality of these approaches, new algorithms such as Nested Rollout Policy Adaptation (NRPA), have replaced the hand crafted heuristic with one that is trained *online*, using data collected during the search. Despite the limited expressiveness of the policy model, NRPA is able to outperform traditional Monte-Carlo algorithms (i.e. without learning) on various games including Morpion Solitaire. In this paper, we combine Monte-Carlo search with a more expressive, non-linear policy model, based on a neural network trained beforehand. We then demonstrate how to use this network in order to obtain state-of-the-art results with this new technique on the game of Morpion Solitaire. We also use NeuralNRPA as an expert to train a model with Expert Iteration.

1 Introduction

Monte-Carlo search algorithms can discover good solutions for complex combinatorial optimization problems by running a large number of simulations. Internally, the simulations are used to evaluate each alternative branching decision, and the search algorithm successively commits to the best branching decision until a terminal state is reached. Thus, one can see simulations as a tool to turn uninformed (random) search policies into well informed ones, at the cost of computational power. Building on this observation, *Nested Monte Carlo Search* (NMCS) further improves the technique by running recursive (a.k.a. nested) simulations. At the lowest recursive level, the simulations are driven by a simple random search policy. At higher recursive levels, the simulations are driven by a search policy that is based on the simulations of the recursive level below. Nesting simulations greatly improve the quality of the solutions discovered, however it is generally impossible to run NMCS with more than 5 or 6 levels of recursion, due to the prohibitive cost of recursive simulations.

To further improve the quality of the results, it is often desirable to replace the purely random search policy with a hand crafted search heuristic, but building such heuristic is time consuming and requires expert knowledge which is difficult to encode in the search heuristic. To overcome this limitation and facilitate the adaptation of Monte-Carlo search to new problems, *Nested Rollout Policy Adaptation* [10] replaces the recursive policies, with a simple policy model that is *learned*, using data collected during the search. Thanks to this simple principle, NRPA is now the state of the art on different problems such as vehicle routing problems, network traffic engineering or RNA design as well as the game of *Morpion Solitaire* which became a testbed for several Monte-Carlo based algorithms such as NRPA and NMCS.

However, despite the success of learned policies, and a number of recent studies on the topic, the last major record break on Morpion Solitaire dates back from 2011. (Rosin obtained 82 on the 5D variant with a week long execution of NRPA).

Recently [2] has managed to rediscover the best score with optimized playouts, but despite many tries was unable to break the record. The recent success of AlphaGo/AlphaZero [12–14] suggests that combining Monte-Carlo search together with a neural network based heuristic can lead to important improvements. AlphaZero like Deep Reinforcement Learning has been tried for Morpion Solitaire with PUCT [15].

In this paper, we look into learning an expressive policy model for the Morpion Solitaire that is based on a deep neural network, and we use it to drive simulations at low computational cost. We then conduct thorough experiments to understand the behaviour of new and existing approaches, and to assess the quality of our policy models. Then we reintroduce this neural network based policy inside NMCS. We are able to obtain a policy which is almost as good as state-of-the-art NRPA algorithm with 3 nested levels, for a 2-3 times reduction of computational time. Finally, we experiment using self-play with a second approach based on Expert Iteration (Exit) with various experts. Our approach is able to learn a policy from scratch and outperforms previous work on selfplay in Morpion Solitaire by 6 points.

The rest of this paper is organized as follows: the second section describes related work on Monte Carlo Search. The third section explains search with a learned model. The fourth section shows how to combine neural networks and Monte Carlo Search. The fifth section shows how to apply Deep Reinforcement Learning using Neural NMCS and Neural NRPA. The sixth section outlines future work.

2 Preliminaries on Monte-Carlo search for game playing

Policies: A policy is a probability distribution p over a set of moves $\mathcal{M}$ that is conditioned on the current game state $s \in S$. For example, we often consider the uniform policy p_0, which assigns equal probability to all the moves that are legal in state s. I.e. $p_0(m|s) = \frac{1}{|M_s|}$.

In this paper, we also consider policies probability distributions p_W which are parameterized with a set of weights W. There is one real valued weight for each possible move, i.e. $W = w_{m_1}, \ldots, w_{m_{|\mathcal{M}|}}$, and the probability $p_W(m|s)$ is defined as follows:

$$p_W(m|s) = \frac{e^{w_m}}{\sum_{p \in M_s} e^{w_p}}$$

The softmax function enables to calculate the gradient for all the possible weights associated to the possible moves of a state and to learn a policy in NRPA using gradient descent.

Finally, we also consider more complex policies π_θ in which the probability of each move depends on a function of the state, represented using a neural network. Let $f_\theta : S \rightarrow \mathbb{R}^{|M|}$ be a neural network parameterized with θ, we can then define policy π_θ as follows:

$$\pi_\theta(m|s) = \frac{e^{(f_\theta(s))_m}}{\sum_{p \in M_s} e^{(f_\theta(s))_p}}$$

2.1 NMCS and NRPA

As most Monte-Carlo based algorithms, *Nested Monte Carlo Search* (NMCS) and *Nested Rollout Policy Adaptation* (NRPA) both generate a large number of random sequences of moves. The best sequence according to the scoring function is then returned as a solution to the problem. The quality of the final best sequence directly depends on the quality of the intermediate random sequences generated during the search, and thus on the random policy. Therefore NMCS and NRPA have introduced new techniques to improve the quality of the policy throughout the execution of the algorithm.

NMCS and NRPA are both recursive algorithms, and at the lowest recursive level, the generation of random sequences is done using playouts parameterized with a simple stochastic policy. If the user has access to background knowledge, it can be captured by using a non-uniform policy (typically by manually adjusting the weights W of a parameterized policy p_W). Otherwise, the uniform policy p_0 is used.

In NMCS, the policy remains the same throughout the execution of the algorithm. However, the policy is combined with a tree search to improve the quality over a simple random sequence generator. At every step, each possible move is evaluated by completing the partial solution into a complete one using moves sampled from the policy. Whichever intermediate move has led to the best completed sequence, is selected and added to the current sequence. The same procedure is repeated to choose the following move, until the sequence has reached a terminal state.

A major difference between NMCS and NRPA, is the fact that NRPA uses a stochastic policy that is *learned* during the search. At the beginning of the algorithm, the policy is initialized uniformly and later improved using gradient descent based the best sequence discovered so far. The policy weights are updated using gradient descent steps to increase the likelihood of the current best sequence under the current policy.

Finally, both algorithms are nested, meaning that at the lowest recursive level, weak random policies are used to sample a large number of low quality sequences, and produce a search policy of intermediate quality. At the recursive level above, this policy is used to produce sequences of high quality. This procedure is applied recursively. In both algorithms the recursive level (denoted $level$) is a crucial parameter. Increasing $level$ increases the quality of the final solution at the cost of more CPU time. In practice it is generally set to 4 or 5 recursive level depending on the time budget and the computational resources available.

2.2 Playing Morpion Solitaire with Monte Carlo search

The game of Morpion Solitaire Morpion Solitaire is a single player board game. The initial board state is shown in Figure 1 and a move consists of drawing a circle on an empty intersection, and drawing a line out of five neighboring circles including the new

one. A game is over when the player runs out of moves, and the goal of the game is to play as many moves as possible. The final score is simply the number of moves that have been played. There are two versions of the game called 5T (T for *touching*) and 5D (D for *disjoint*). In 5T two lines having the same direction can share a common circle, whereas in 5D they cannot.

The best human score for 5T is 170 moves and it has been discovered by Charles-Henri Bruneau who held this record for 34 years until he was beaten by an algorithm based on Monte-Carlo search. The current best score is 82 for 5D and 178 for 5T. Both records were established in August 2011 by Chris Rosin with an algorithm combining nested Monte-Carlo search and a playout policy learning (NRPA, [3, 10]).

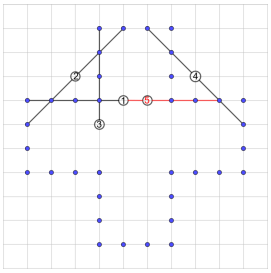

Fig. 1: Move 1, 2, 3 and 4 are legal for 5D and 5T. Move 5 is legal for 5T only

Modeling Morpion solitaire as a Monte-Carlo search problem Any game state is fully determined by the set of (oriented) segments connecting the circles. Thus, the initial game state s_0 is the empty set, and performing a move consists of adding a segment to the set of segments representing the current state. Each segment (or move) is determined by a 2D coordinates representing the starting point of the segment, and one direction among the 4 possible directions: left to right, top to bottom, top-left to bottom-right, and top-right to bottom-left. The game is over when the player reaches a terminal state i.e. a state s such that $M_s = \emptyset$.

Although the order in which the moves are added does not influence the final game state, (i.e. for any sequence of moves X and any permutation X' of X, we have $state(X) = state(X')$), it is generally difficult to compute the subset of moves that can be added without breaking the rules. Therefore the moves are drawn sequentially such that every intermediate state is also a legal state.

3 Imitating NRPA

In this section, we first focus on training a policy model that can be used to select good moves, without having to simulate a large number of games. We recall that a policy model is a conditional probability distribution $\pi_\theta(m|s)$ where s is a game state from the set of all possible game states S, and m is a move from the set of all possible move $\mathcal{M}$.

To obtain a good policy, we first train our policy model to learn to reproduce the sequences found by NRPA. The policy model is represented by a neural network, and is trained to predict the next NRPA move, given a description of the current game state. Each supervised example is a particular game state, labeled with the move that was chosen by NRPA during a previous run. (Note that since NRPA is a stochastic algorithm, identical game states may appear several times in the dataset, labeled with different moves.)

To successfully reproduce sequences found by NRPA, we need 1: a game state representation that contains the adequate features to accurately predict the next move by NRPA, and 2: a policy model that is expressive enough to capture the complex relation that exists between the game state and the best move selected by NRPA. In this section, we design and evaluate several training settings using different game state representations and different models. We then discuss the performance of these settings by using two criteria: the ability to mimic the behaviour of NRPA, and the quality of a play (i.e. the game score).

3.1 Game state representation

Although the game state is fully determined by the set of segments (as discussed in Section 2), this representation does not favor learning, and generalization over different but similar states. In this section, we discuss a better state representation, that explicitly captures important features and makes it possible to predict the behavior of NRPA, without having to run the costly simulations.

In all our models, the board is represented by five 30×30 binary valued matrices, which are large enough to capture any record holding boards. The first matrix is used to represent the occupied places (i.e. the circles in Figure 1) which are not directly available nor easy to compute from a board state represented as a set of segments. If the place i, j is occupied on a board, the corresponding value in the matrix is set to 1, and 0 otherwise.

Because this matrix alone does not fully determine the game state, the four extra binary valued matrices are used to represent the connecting segments, one matrix for each possible direction respectively: left to right, top to bottom, top-left to bottom-right, and top-right to bottom-left. A one in the first matrix (left to right) at position i, j signifies that there is a segment between the place i, j and the place $i + 5, j$ on the board. A one in the second matrix (top to bottom) at position i, j signifies that there is a line between position i, j and position $i, j + 5$ on the board and so on for each matrices. Every time a new place is occupied (i.e. the player makes a move) we set one boolean value in the first matrix, and one boolean value from one of the 4 remaining matrices.

In addition to the board representation, we extend the state representation with 4 extra matrices which are meant to represent all the possible moves for the next move (one matrix for each possible direction). We call this first representation **R1**.

To further improve temporal consistency of the policy model, we extend the first state representation with 8 extra matrices, to represent the 8 previous moves. We call this second representation **R2**.

3.2 Neural network architecture

We consider two neural network architectures. The first one is a fully convolutional neural network with 4 convolutional layers. The first 3 layers have 32 filters with 3x3 kernels, and the last convolutional layer has 4 filters with 1x1 kernels to match the output. The output is a vector of dimension n^2d where n is the dimension of the board and d the number of directions to represent all possible moves (in all our experiments use n=30 and d=4).

The second architecture is a residual neural network [8] with 4 convolutional layers with the same type and number of filters as the first architecture, and the same input/output definition.

We found the use of a fully convolutional model more effective than the policy heads used in AlphaGo and Alpha Zero which contain fully connected layers. A fully convolutional head is similar to the policy heads of Polygames [7] and Golois [6].

3.3 Training data & training procedure

We train the policy models using data generated with NRPA. Each example in the training set is a game state representation labelled with one move played by NRPA in this game state. To improve the quality of the training data, we can select only the moves from the NRPA games that scored well, however it is important to remark that there is a trade-off between the quality of the moves, and the diversity of the training data (a.k.a. the exploration vs. exploitation trade-off). To observe this phenomenon, we selected 10 000 games (800.000 game states) generated with NRPA that scored 80 or above (Figure 2 first plot), and 10 000 games (around 800.000 game states) generated with NRPA that scored between 70 and 82 (Figure 2 second plot). As we can see in the first plot, game states dramatically lack of diversity.

Based on this observation and other empirical analysis, we used NRPA to generate a large number of games, and selected 9141 games scoring between 70 and 82 for a total of 694 716 training examples (couples: game state, move). We use this data to train the neural networks models described above, using the two representations R1 and R2. We used a decaying learning rate starting at 0.01 and divided it by 10 every 40 epochs.

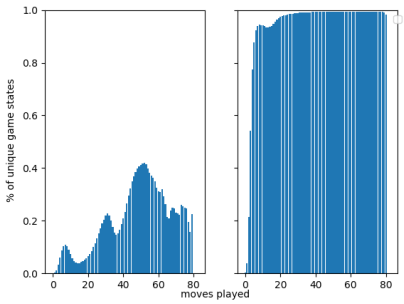

Fig. 2: NRPA data diversity

3.4 Model performance (without search)

We now compare the different policy models using two metrics. First we consider the test accuracy, that is how well the policy models are able to predict NRPA moves. Then we estimate the average and the maximum score obtained with a playing strategy which sample moves from the different policy models. The mean and the average are computed over 100 000 games.

Train/Test accuracy We first compare the two neural network architectures and the two game state representations that we have described in the previous section. The values of the loss functions during the training procedure for each architecture are shown in Figure 3, and a comparison of the accuracy achieved by each architecture and each state representation is shown in Figure 4. (We only show the comparison of the state representation using the Resnet architecture since it performs best.)

We first consider, the initial model with the game state representation R1 and the BasicCNN neural network architecture shown in Figure 3 (left). We observe that the training loss quickly reaches its lowest value, and that an important difference between the training and the testing loss remains. Unsurprisingly, this results in a poor model accuracy of 45.5% on the test set (as seen in Figure 1). Furthermore, this peculiar behaviour is not impacted by the use of a larger, more expressive neural network architecture such as the Resnet or by any more sophisticated training procedure.

To explain this behaviour, we recall that 1) NRPA is not deterministic, 2) the policy in NRPA is trained in a stochastic way and may vary significantly from one game to another. Non determinism leads to presence of a large number of identical examples labelled differently in the train and test set, which induces an incompressible Bayes risk, that cannot be removed, by increasing the expressivity of the model, or by improving the training procedure.

However, the behaviour is remarkably different on the second representation R2 which includes the previous moves in addition to current game state. This may be surprising, since with an unbiased algorithm, the best move only depends on the current state, and should not depend on the previous actions performed by the player. However, NRPA is biased by the learned policy, which differs from one game to another. The previous moves thus informs the neural network on the current policy, and the particular strategy that is being played, and ultimately reduces Bayes's risk. As a result, the neural network is able to better fit the training set (and benefits from additional epochs), the final loss is lower, the generalization gap is reduced, and the final accuracy reaches 70%.

This suggests that unlike the first two models based on R1, last model based on R2 is able to capture not just one strategy but several good strategies that were discovered by NRPA during the 9141 selected games.

Score To evaluate the quality of the policy models as players, we sample sequence of moves from each policy model and observe the score of the final state. The distribution of the scores across 100 000 sequences generated from each policy model is shown in Figure 5.

In both plots, we have a high probability of reaching a score between 57 and 62. However, the second model based on state representation R2 demonstrates better results

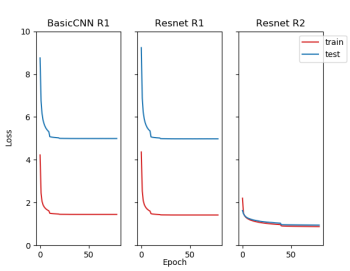

Fig. 3: Loss evolution during training

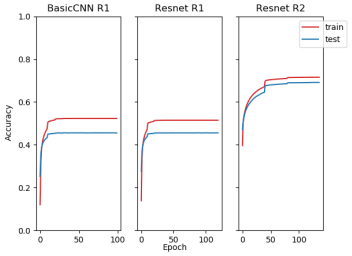

Fig. 4: Accuracy evolution during training

Epoch	BasicCNN R1	Resnet R1	Resnet R2
1	25%	27%	47%
40	45.5 %	45.3 %	67%
80	45.5 %	45.5 %	68.9%

Table 1: Accuracy for each tested configuration

in the early games, and there are fewer games that score less than 50 points. We believe that the second game state representation, which includes the previous moves, is able to achieve better temporal consistency and avoid simple mistakes which may be the consequence of mixing several NRPA strategies from the training set. The model based on R2 also exhibits the highest average score, and maximum score than the model based or R1.

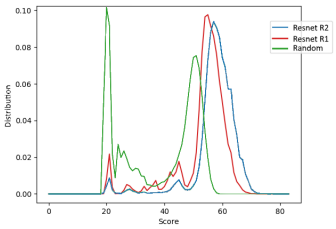

Fig. 5: Score distribution of Resnet R1 and Resnet R2

To accurately evaluate the quality of the models and to compare it with the original NRPA algorithm, we provide more precise score statistics which are available in Table 2.

In this table, *Uniform* is the performance of the uniform policy model p_0, the next 3 are the performance of NRPA with increasing level recursions, and the last 3 are our models, described in the previous sections. The statistics for the Uniform policy and

our models, are averaged over 100 000 games. However, generating NRPA games is computationally intensive so the statistics for NRPA(1), NRPA(2) and NRPA(3) are computed over 100 000, 10 000 and 400 games respectively where number between brackets refers to the number of recursive levels

We can see that the two neural network models based on R1 (without the previous moves) offer a little improvement over the baseline, but are outperformed by NRPA(1). However the neural network based on R2 performs significantly better than the baseline, (mean and max), and achieves better maximum scores than NRPA(1) and NRPA(2), without having to run a large number of rollouts.

	mean	max	$\sigma/\sqrt{n}$
Uniform	39.1	61	0.059
NRPA(1)	58.5	66	0.014
NRPA(2)	65.9	72	0.024
NRPA(3)	68.2	78	0.119
BasicCNN R1	41.7	60	0.024
Resnet R1	44.0	58	0.018
Resnet R2	50.5	74	0.032

Table 2: Results of our approaches compared to state of the art algorithm.

3.5 Combining MC search algorithms with a neural based search procedure

We now have a neural network that can act as an informed search heuristic comparable to a NRPA of level 2-3. To further improve the quality of the solutions, we incorporate the newly trained policy model inside existing search algorithms, in place of the random heuristic.

Table 3 summarizes results achieved by the different policy models. Nested(1) where number between brackets refers to the number of recursive levels, outperforms Resnet by 18,3 points in average and by 5 points for the maximum score. In this setup, our approach outperforms NRPA(2) in mean and maximum and perform very close to NRPA(3).

	mean	max	avg. game time
NRPA(3)	68.2	78	16:40
Nested(1) + Resnet R2	**68.8**	**79**	**6:26**
Resnet R2	50.5	74	0:01

Table 3: Comparison between different search algorithms

4 Self play with *Exit*

In the previous section, we were able to obtain a playing strategy by training a neural network with game data generated by NRPA. Although the resulting strategy is good

and computationally efficient, this technique remains entirely supervised by NRPA, and thus it is unlikely performing better than NRPA itself.

In this Section, we explore *self-play* and learn a new policy from scratch using an approach based on Exit [1]. In contrast with the previous approach in which the neural network is only used to store and generalize past experiences acquired through supervision, in Exit the expert is also based on a neural network and can be improved as we discover new good moves. This allows the expert to learn from scratch, and improve beyond the current best known strategy. (See [1] for details.)

Exit has been used in the notorious Alpha Zero [13] and Wang et al. [15] applied it for Morpion Solitaire. However, our approach is different since it does not use PUCT as the search algorithm. Instead we use an expert based on NRPA which is state-of-the art in Morpion Solitaire. Although using NRPA poses a number of challenges, we are able to outperform state-of-the-art in the self play setting by a significant margin.

Speeding up NeuralNRPA The main challenge that is to overcome if we want to use NeuralNRPA as an expert is the computational cost. Despite the improvement discussed in the previous Section training a policy from scratch using NeuralNRPA remains prohibitive.

In the previous approach, we make a forward pass at each step, which induces a significant computational cost. In the Morpion Solitaire, moves are often commutative, meaning that playing move a, then b leads to the same state than playing move b, then a. We can exploit this property and make a single forward pass for an entire game (including the many rollouts). This results in a small reduction of the average score, but a dramatic reduction of computational cost.

Training setting In our experiments, at each iteration we generated 10.000 boards with the learner. We train our model with a learning rate of 5.10^{-4} and 20 epochs.

We tested Exit using 3 different configurations: **NeuralNMCS(0)** is Exit with NMCS lvl 0 as expert which means the expert use the best sequence out of x rollouts played by the neural network. **NeuralNMCS(1)** is Exit with NMCS level 1 as expert, **NeuralNMCS(1)c** the expert is a early stopped version of NMCS where instead of calling NMCS after each move played, we stop the algorithm after the end of the first call from the highest nested level end and use the best sequence found as the label instead of choosing only one move after one call and repeat until the end of the sequence. **NeuralNRPA(1)** and **NeuralNRPA(2)** are GNRPA [5] of level 1 and 2 with the bias given by the policy output by the neural network. GNRPA had a bias to NRPA action's weight leading to a bootstraped NRPA into a specific direction.

Table 4 gives mean and max scores of neural networks trained by the different approaches. All of the approaches have been running for 180h. NeuralNRPA(2) A, NeuralNMCS(1)c A, NeuralNMCS(1) A and NeuralNMCS(0) A are the best approaches among all tested parameters.

The figure 6 displays the evolution of the maximum score evolution on 100 rollouts with the four best approaches of each type. NeuralNMCS(0) A is the fastest reaching 70 but it gets stuck quickly. NeuralNMCS(1) A shows poor exploration due to a low number of rollouts but it is also very slow regarding the number of its rollout parameter. NeuralNMCS(1)c A and NeuralNRPA(2) A are slower then NeuralNMCS(0) A but

ended up outperforming it. NeuralNMCS(1)c A is a bit faster than NeuralNRPA(2) A at the begining but NeuralNRPA(2) A gets the highest score at the end.

Approach	Temperature	Rollouts	NN mean score	NN best score	Best score in (Hours)
NeuralNMCS(0)	0.2	1	50.66	64	6
NeuralNMCS(0)	0.2	100	63.76	68	72
NeuralNMCS(0) A	0.4	100	54.33	70	51
NeuralNMCS(1) A	0.2	1	56.1	66	50
NeuralNMCS(1)c	0.2	1	61.42	68	63
NeuralNMCS(1)c A	0.2	10	64.38	72	134
NeuralNMCS(1)c	0.4	10	53.23	67	111
NeuralNRPA(1)	0.2	100	47.4	63	3
NeuralNRPA(2)	0.2	10	54.9	71	93
NeuralNRPA(2) A	**0.2**	**20**	**57.28**	**73**	**180**
NeuralNRPA(2)	0.2	40	53.78	70	151

Table 4: Comparison of approaches

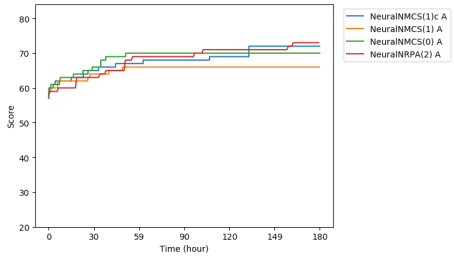

Fig. 6: Best approaches max score evolution

5 Conclusion

We have shown that it is possible to learn an exploratory policy for Morpion Solitaire from a set of states with score ranging from high average scores (70) to highest known scores (82) using a neural network. We also integrated this neural network in a Nested Monte Carlo search and showed it improves when sampling from its moves distribution reaching scores 3 moves away from the highest known score. We have also trained a network with an original version of Expert Iteration using Neural NRPA and Neural NMCS and found that Neural NRPA is the best expert performing 6 points higher than the reinforcement learning approach in [15].

In computer Go and more generally in board games the neural networks usually have more than one head. They have at least a policy head and a value head. The policy head is evaluated with the accuracy of predicting the moves of the games and the value head is evaluated with the Mean Squared Error (MSE) on the predictions of the outcomes of the games. The current state of the art for such networks is to use residual networks [4, 13, 14]. The architectures used for neural networks in supervised learning

and Deep Reinforcement Learning in games can greatly change the performances of the associated game playing programs. For example residual networks gave AlphaGo Zero a 600 ELO gain in playing strength compared to standard convolutional neural networks. Mobile Networks [9, 11] are commonly used in computer vision to classify images. They obtain high accuracy for standard computer vision datasets while keeping the number of parameters lower than other neural networks architectures. For board games and in particular for Computer Go it was shown recently that Mobile Networks have a better accuracy than residual networks [6].

We plan to try different architectures for Morpion Solitaire neural networks and compare their performances.

References

1. Anthony, T., Tian, Z., Barber, D.: Thinking fast and slow with deep learning and tree search. In: Advances in Neural Information Processing Systems. pp. 5360–5370 (2017)
2. Buzer, L., Cazenave, T.: Playout optimization for Monte Carlo search algorithms. application to Morpion Solitaire. In: IEEE Conference on Games (2021)
3. Cazenave, T.: Nested Monte-Carlo Search. In: Boutilier, C. (ed.) IJCAI. pp. 456–461 (2009)
4. Cazenave, T.: Residual networks for computer go. IEEE Transactions on Games **10**(1), 107–110 (2018)
5. Cazenave, T.: Generalized nested rollout policy adaptation. arXiv preprint arXiv:2003.10024 (2020)
6. Cazenave, T.: Mobile networks for computer go. IEEE Tansactions on Games (2020)
7. Cazenave, T., Chen, Y.C., Chen, G.W., Chen, S.Y., Chiu, X.D., Dehos, J., Elsa, M., Gong, Q., Hu, H., Khalidov, V., Cheng-Ling, L., Lin, H.I., Lin, Y.J., Martinet, X., Mella, V., Rapin, J., Roziere, B., Synnaeve, G., Teytaud, F., Teytaud, O., Ye, S.C., Ye, Y.J., Yen, S.J., Zagoruyko, S.: Polygames: Improved zero learning. ICGA Journal **42**(4) (December 2020)
8. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition pp. 770–778 (2016)
9. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
10. Rosin, C.D.: Nested rollout policy adaptation for Monte Carlo Tree Search. In: IJCAI. pp. 649–654 (2011)
11. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv2: Inverted residuals and linear bottlenecks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 4510–4520 (2018)
12. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484–489 (2016)
13. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
14. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of go without human knowledge. nature **550**(7676), 354–359 (2017)
15. Wang, H., Preuss, M., Emmerich, M., Plaat, A.: Tackling Morpion Solitaire with AlphaZero-like ranked reward reinforcement learning (2020)