# Mobile Networks for Computer Go

Tristan Cazenave
LAMSADE, Université Paris-Dauphine, PSL, CNRS, Paris, France

**The architecture of the neural networks used in Deep Reinforcement Learning programs such as AlphaZero or Polygames has been shown to have a great impact on the performances of the resulting playing engines. For example the use of residual networks gave a 600 ELO increase in the strength of AlphaGo. This paper proposes to evaluate the interest of Mobile Networks for the game of Go using supervised learning as well as the use of a policy head and a value head different from the AlphaZero heads. The accuracy of the policy, the mean squared error of the value, the efficiency of the networks with the number of parameters, the playing speed and strength of the trained networks are evaluated.**

***Index Terms*—Deep Learning. Neural Networks. Board Games. Game of Go.**

## I. Introduction

This paper is about the efficiency of neural networks trained to play the game of Go. Mobile Networks [1], [2] 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.

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 [3], [4], [5].

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.

Residual networks will be compared to Mobile Networks for computer Go. Different options for the policy head and the value head will also be compared. The basic residual networks used for comparison are networks following exactly the AlphaGo Zero and AlphaZero architectures. The improvements due to Mobile Networks and changes in the policy head and the value head are not specific to computer Go and can be used without modifications for other games.

The remainder of the paper is organized as follows. The second section presents related works in Deep Reinforcement Learning for games. The third section describes the training and the test sets. The fourth section details the neural networks that are tested for the game of Go. The fifth section gives experimental results.

## II. Zero Learning

Monte Carlo Tree Search (MCTS) [6], [7] made a revolution in Artificial Intelligence applied to board games. A second revolution occurred when it was combined with Deep Reinforcement Learning which led to superhuman level of play in the game of Go with the AlphaGo program [8].

Residual networks [3], combined with policy and value heads sharing the same network and Expert Iteration [9] did improve much on AlphaGo leading to AlphaGo Zero [4] and zero learning. With these improvements AlphaGo Zero was able to learn the game of Go from scratch and surpassed AlphaGo.

Later AlphaZero successfully applied the same principles to the games of Chess and Shogi [5].

Other researchers developed programs using zero learning to play various games.

ELF/OpenGo [10] is an open-source implementation of AlphaGo Zero for the game of Go. After two weeks of training on 2 000 GPUs it reached superhuman level and beat professional Go players.

Leela Zero [11] is an open-source program that uses a community of contributors who donate GPU time to replicate the AlphaZero approach. It has been applied with success to Go and Chess.

Crazy Zero by Rémi Coulom is a zero learning framework that has been applied to the game of Go as well as Chess, Shogi, Gomoku, Renju, Othello and Ataxx. With limited hardware it was able to reach superhuman level at Go using large batches in self-play and improvements of the targets to learn such as learning territory in Go. Learning territory in Go increases considerably the speed of learning.

KataGo [12] is an open-source implementation of AlphaGo Zero that improves learning in many ways. It converges to superhuman level much faster than alternative approaches such as Elf/OpenGo or Leela Zero. It makes use of different optimizations such as using a low number of playouts for most of the moves in a game so as to have more data about the value in a shorter time. It also uses additional training target so as to regularize the networks.

Galvanise Zero [13] is an open-source program that is linked to General Game Playing (GGP) [14]. It uses rules of different games represented in the Game Description Language (GDL) [15], which makes it a truly general zero learning program able to be applied as is to many different games. The current games supported by Galvanise Zero are Chess, Connect6, Hex11,

Corresponding author: T. Cazenave (email: Tristan.Cazenave@dauphine.psl.eu)

Hex13, Hex19, Reversi8, Reversi10, Amazons, Breakthrough, International Draughts.

Polygames [16] is a generic implementation of AlphaZero that has been applied to many games, surpassing human players in difficult games such as Havannah and using architectural innovations such as a fully convolutional policy head.

## III. The Training and the Test Sets

We use two datasets for training the networks.

The first dataset used for training comes from the Leela Zero Go program self played games. The selected games are the last 2 000 000 games of self play, starting at game number 19 000 000. The input data is composed of 21 19x19 planes (color to play uses one plane, ladders use four planes: 2 for lost ladders and 2 for unsettled ladders, liberties use 6 planes for numbers of liberties from 1 to 6 or more, the current state uses 2 planes, the 4 previous states use 8 planes). The output targets are the policy (a vector of size 361 with 1.0 for the move played, 0.0 for the other moves), the value (1.0 if White won, 0.0 if Black won).

The second dataset is the ELF dataset. It is built from the last 1 347 184 games played by ELF, it contains 301 813 318 states.

At the beginning of training and for each dataset 100 000 games are taken at random as a validation set and one state is selected for each game to be included in the validation set. The validation set for the Leela dataset only contains games from Leela and the validation set for the ELF dataset only contains games from ELF. The same set of states in the validation sets are used for all networks. These games and states are never used for training, none of the states present in the same game as a state in the test set are used for training. We define one epoch as 1 000 000 samples. For each sample in the training set a random symmetry among the eight possible symmetries is chosen.

Both datasets contain games played at superhuman level. The Leela games are played at a better level than the ELF games since the latest versions of Leela are stronger than ELF.

## IV. Networks Architectures, Training and Use

### A. Residual Networks

Residual networks improve much on convolutional networks for the game of Go [3], [4]. In AlphaGo Zero they gave an increase of 600 ELO in the level of play. The principle of residual networks is to add the input of a residual block to its output. A residual block is composed of two convolutional layers with ReLU activations and batch normalization. For our experiments we use for AlphaZero like networks the same block as in AlphaGo Zero.

Another architecture optimization used in AlphaGo Zero is to combine the policy and the value in a single network with two heads. It also enables an increase of 600 ELO in the level of play [4]. All the networks we test have two heads, one for the policy and one for the value.

### B. Mobile Networks

MobileNet [1] followed by MobileNetV2 [2] provide a parameter efficient neural network architecture for computer vision. The principle of MobileNetV2 is to have blocks as in residual networks where the input of a block is added to its output. But instead of usual convolutional layers in the block they use depthwise convolutions. Moreover the number of channels at the input and the output of the blocks (in the trunk) is much smaller than the number of channels for the depthwise convolutions in the block. In order to efficiently pass from a small number of channels in the trunk to a greater number in the block, usual convolutions with cheap 1x1 filters are used at the entry of the block and at its output.

The Keras [17], [18] source code we used for the MobileNets models is given in the appendix.

### C. Optimizing the Heads

The AlphaGo Zero policy head uses 1x1 convolutions to project the 256 channels to two channels and then it flattens the channels and uses a dense layer with 362 outputs for all possible legal moves in Go. The AlphaGo Zero value head uses 1x1 convolutions to project the 256 channels to one channel and then it flattens the channel, connects it to a dense layer with 256 outputs and then connects these outputs to a single output for the value [4].

We experimented with different policy and value heads. For the policy head we tried a fully convolutional policy head. It does not use a dense layer. Instead it uses 1x1 convolutions to project the channels to a single channel, then it simply flattens the channel directly giving 361 outputs, one for each possible move except the pass move. The fully convolutional head has already been used in Polygames [16].

For the value head we experimented with average pooling. The use of Spatial Average Pooling in the value head has already been shown to be an improvement for Golois [19]. It was also used in Katago [12] and in Polygames [16]. In this paper we experiment with Global Average Pooling for the value head. Each channel is averaged among its whole 19x19 plane leading to a vector of size equal to the number of channels. It is then connected to a dense layer with 50 outputs. The last layer is a dense layer with one output for the value. We use a sigmoid as the activation function for the value since the labels are 0 or 1. The 0 and 1 labels for the value are better than -1 and 1 used in AlphaZero since we often use the Binary Crossentropy loss for the value.

### D. Training

Training of the networks uses the Keras/Tensorflow framework. We define an epoch as 1 000 000 states. The evaluation on the test set is computed every epoch. The loss used for the value in the AlphaZero papers is the mean squared error (MSE). We keep this loss for the validation and the tests of the networks in order to compare them on an equal basis. In some of the network we train the value with the binary cross entropy loss which seems more adapted to the learning of the value (i.e. we want to know if the game is won or lost). We also experiment with a weight on the value loss. The binary cross entropy loss is usually greater than the mean squared error loss, but we can make it even greater by multiplying the loss with a constant.

The batch size is fixed to 32. The annealing schedule is to train with a learning rate of 0.005 for the first 100 epochs. Then to train with a learning rate of 0.0005 from 100 to 150 epochs. Then to train with a learning rate of 0.00005 from 150 to 200 epochs. It enables to fine tune the networks when the learning stalls. This is similar to the AlphaZero annealing schedule which also divides the learning rate by ten every 200 epochs in the beginning and every 100 epochs in the end. Using this schedule the training of a large mobile network approximately takes 12 days with a V100 card.

For all networks we use a L2 regularization during training with a weight of 0.0001. We found that the validation loss and the level of the trained network is much better when using regularization.

### E. Inputs and Labels

The inputs of the networks use the colors of the stones, the liberties, the ladder status of the stone, the ladder status of adjacent strings (i.e. if an adjacent string is in ladder), the last 5 states and a plane for the color to play. The total number of planes used to encode a state is 21.

The labels are a 0 or a 1 for the value head. A 0 means Black has won the game and a 1 means it is White. For the policy head there is a 1 for the move played in the state and 0 for all other moves. The output for the policy head is different from the output used in AlphaZero since AlphaGo Zero and AlphaZero use Expert Iteration [9] which gives as output the number of time the moves has been tried in the PUCT search divided by the total number of evaluations in the PUCT search. We do not represent the pass move. When playing games the pass move is generated if all moves are on eyes. A possibility for representing the pass move with a fully convolutional policy is to add another plane for the policy which only gives the probability of playing the pass move. This is how the pass move is evaluated in Polygames for example. In the datasets we use the game is usually resigned before a pass move happens.

### F. Self Play Speed

A program that plays games against itself so as to generate more training data can be strongly parallelized. Parallelizing the different games being played can greatly speedup the overall reinforcement learning process. Both the forward pass of the network and the building of the batches can be parallelized. Parallelizing the forward passes is effectively done by building large batches of states with one state per self played game. The GPU is good at effectively parallelizing the forward pass on large batches. The building of the inputs of the large batches can also be strongly parallelized using threads.

Smaller networks are faster and enable larger batches for self play. This is why most programs start training with small networks and make them grow during learning.

## V. Experimental Results

We first give experiments for the Leela dataset and then for the ELF dataset. We start with small networks, then we detail unbounded networks, the parameter efficiency, the speed efficiency, training on the ELF dataset, the self play speed and we finish with a round robin tournament between different networks.

We had problems with the AlphaZero value head: it often did not learn even after many epochs so we replace it with another value head using average pooling. The use of average pooling layers for the value has been described previously in Golois [19], KataGo [12] and Polygames [16]. The value head we used has a global average pooling layer followed by a dense layer of 50 neurons and another dense layer with one output. We used the same value head for all our networks since it gave better results than the AlphaZero value head. Even with this value head it was necessary to launch multiple times the training of the large AlphaZero-like networks in order to start the convergence of the value. A possible solution to the difficulties we encountered with learning the value function could be to reduce the learning rate or to increase the batch size. In further experiments we made, dividing by 10 the initial learning rate made the training of large networks much more stable.

### A. Networks with less than one million parameters

The AlphaZero like network has 8 residual blocks of 66 filters and the AlphaZero policy and value heads. It has 988,405 parameters. During training it uses the MSE loss for the value and the Categorical Crossentropy loss for the policy. The network is called a0.small. The choice of 66 filters is made in order to stay lower than 1 million parameters.

The AlphaZero-like network with a Global Average Pooling value head has 10 residual blocks of 63 filters and the AlphaZero policy head. It has 986 748 parameters. During training it uses the MSE loss for the value and the Categorical Crossentropy loss for the policy. The network is called a0.small.avg.

The AlphaZero-like fully convolutional network has 13 residual blocks of 64 filters. For the policy head it does not use a dense layer, just a 1x1 convolution to a single plane and a flatten. The usual residual blocks used by AlphaZero can have problems with this policy head (the policy loss initially stays close to zero). It is better to use the Golois residual blocks [20]: the rectifier is after the convolution, the batch normalization is after the addition. It has 968 485 parameters. During training it uses the Binary Crossentropy loss for the value and the Categorical Crossentropy loss for the policy. The network is called a0.small.avg.conv.bin.

The fourth network uses 25 MobileNet blocks with a trunk of 64 and 200 filters inside the blocks. It uses the AlphaZero policy head. It has 997 506 parameters. During training it uses the MSE loss for the value and the Categorical Crossentropy loss for the policy. The network is called mobile.small.

The fifth network uses 33 MobileNet blocks with a trunk of 64 and 200 filters inside the blocks. It uses the fully convolutional policy head and the Global Average Pooling

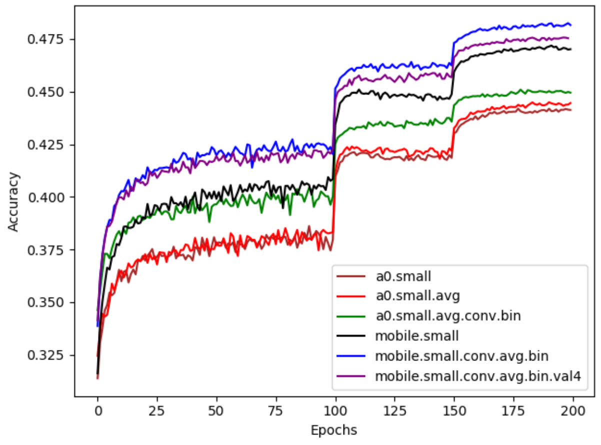

Fig. 1: The evolution of the policy validation accuracy for the different networks with less than one million parameters on the Leela dataset.

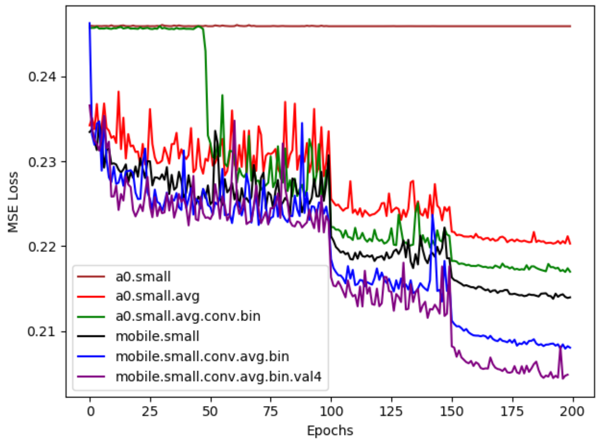

Fig. 2: The evolution of the value validation MSE loss for the different networks with less than one million parameters on the Leela dataset.

value head. It has 970 477 parameters. During training it uses the Binary Crossentropy loss for the value and the Categorical Crossentropy loss for the policy. The network is called mobile.small.conv.avg.bin.

The sixth network is the same as the fifth network except that it has a weight of 4 on the value loss. The network is called mobile.small.conv.avg.bin.val4.

Figure 1 gives the evolution of the accuracy for all small networks. The AlphaZero-like networks have a lower accuracy than the Mobile networks. The best network use MobileNet blocks together with a fully convolutional policy head and global average pooling for the value head. The AlphaZero-like network has the worst results. When removing the policy head and keeping only a 1x1 convolution the results get better. Using MobileNets with the AlphaZero policy head is close to the fully convolutional AlphaZero network. Training a fully convolutional MobileNet improves much the results. Finally putting a weight of four on the value loss of the fully convolutional MobileNet does not hurt much the training of the policy.

We can see in Figure 2 that the small AlphaZero-like

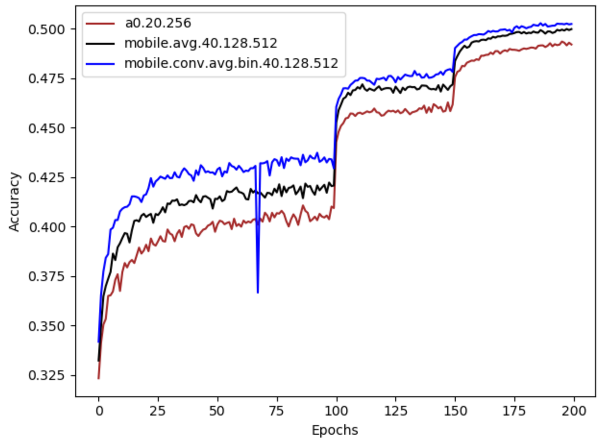

Fig. 3: The evolution of the policy validation accuracy for the different unbounded networks on the Leela dataset.

network does not learn the value within 200 epochs. We tried to launch the a0 training multiple times but did not succeed in learning both the policy and the value with a small network on the Leela dataset. The best value is obtained with a MobileNet with a weight of 4 on the value loss. With a weight of 1 the MobileNet is still the second best for the value, better than the AlphaZero-like networks.

## B. *Unbounded Networks*

We now experiments with large networks of sizes similar to the sizes of the AlphaZero networks.

The AlphaZero-like networks have $n$ residual blocks of 256 filters and the AlphaZero policy head. During training they uses the MSE loss for the value and the Categorical Crossentropy loss for the policy. The network are called a0.n.256 . We test the networks with 5, 10, 15, 20, 30 and 40 MobileNet blocks.

The MobileNets networks use $n$ MobileNet blocks with a trunk of 128 and 512 filters inside the blocks. They use the AlphaZero policy head. During training they use the MSE loss for the value and the Categorical Crossentropy loss for the policy. The networks are called mobile.avg.n.128.512. We test the networks with 10, 20, 40 and 60 MobileNet blocks.

The MobileNets fully convolutional networks use $n$ MobileNet blocks with a trunk of 128 and 512 filters inside the blocks. They use the fully convolutional policy head. During training they use the Binary Crossentropy loss for the value and the Categorical Crossentropy loss for the policy. The networks are called mobile.conv.avg.bin.n.128.512. We test the networks with 10, 20, 40 and 60 MobileNet blocks.

Figure 3 gives the validation policy accuracy for the AlphaZero-like network with 20 blocks and two Mobile networks with 40 blocks. These Mobile networks have much less parameters than the AlphaZero-like network and are faster on GPU. The Mobile networks have better accuracy and the fully convolutional policy head is slightly better.

Figure 4 show that the validation MSE loss of the value is also better for Mobile networks than for AlphaZero-like networks.

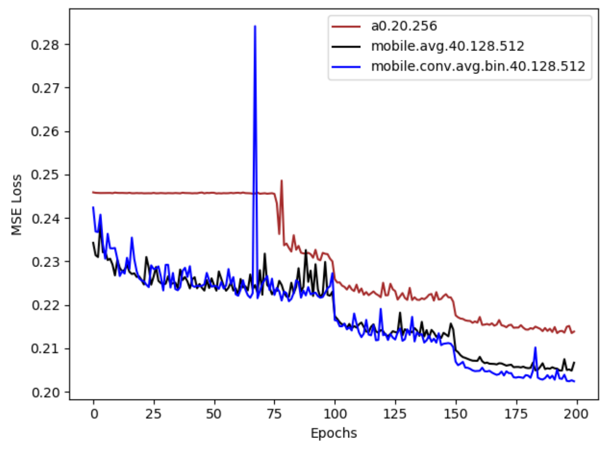

Fig. 4: The evolution of the validation value MSE loss for the different unbounded networks on the Leela dataset.

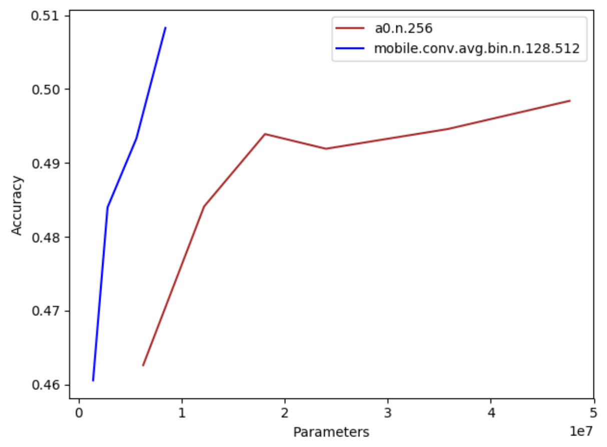

Fig. 5: The evolution of the policy validation accuracy with the number of parameters on the Leela dataset.

## C. Parameter Efficiency

We now give results for the validation accuracy and the validation MSE loss according to the number of parameters of the networks. We compare Mobile networks with fully convolutional policy head and global average pooling value head to AlphaZero residual networks.

Figure 5 gives the accuracy of the different networks according to the number of parameters. The Mobile networks that are trained have 10, 20, 40 and 60 Mobile blocks, a trunk of 128 and 512 filters inside the blocks. The AlphaZero networks have 5, 10, 15, 20, 30 and 40 residual blocks of 256 filters. Mobile networks have a better accuracy with much fewer parameters

Figure 6 gives the MSE loss of Mobile networks and residual network according to the number of parameters. Mobile networks have a much better evaluation than residual networks with much fewer parameters.

## D. Speed Efficiency

We now give results for the validation accuracy and the validation MSE loss according to the speed of the networks. We compare Mobile networks with fully convolutional policy

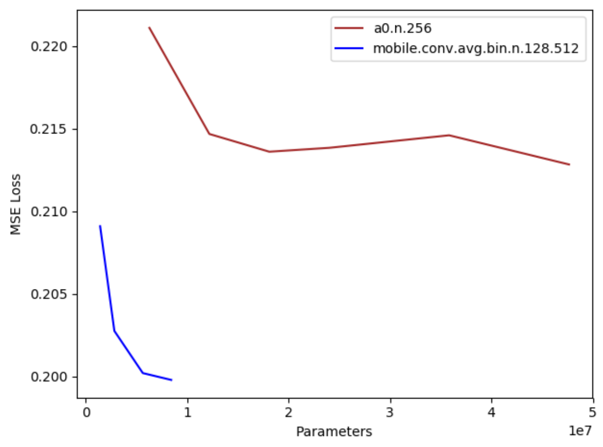

Fig. 6: The evolution of the value validation MSE Loss with the number of parameters on the Leela dataset.

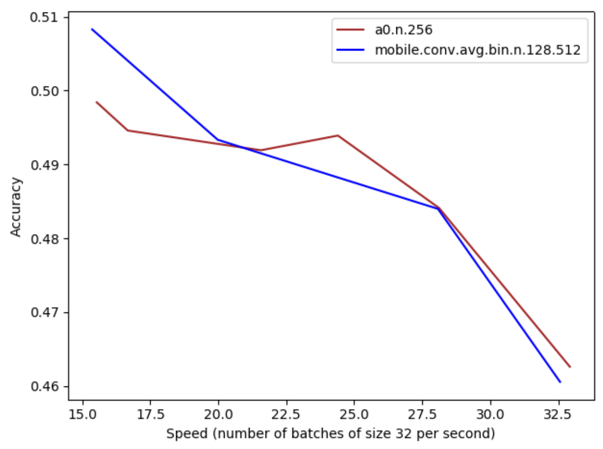

Fig. 7: The evolution of the policy validation accuracy with the speed of the networks on the Leela dataset.

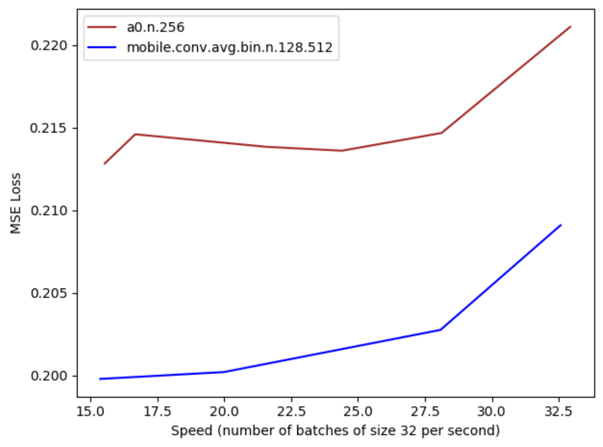

Fig. 8: The evolution of the value validation MSE Loss with the speed of the networks on the Leela dataset.

TABLE I: Batches per second for the tested networks.

| Network | Batches per second |
|---|---|
| a0.5.256 | 32.93 |
| a0.10.256 | 28.12 |
| a0.15.256 | 24.40 |
| a0.20.256 | 21.57 |
| a0.30.256 | 16.67 |
| a0.40.256 | 15.33 |
| mobile.conv.avg.bin.10.128.512 | 32.57 |
| mobile.conv.avg.bin.20.128.512 | 28.08 |
| mobile.conv.avg.bin.40.128.512 | 19.98 |
| mobile.conv.avg.bin.60.128.512 | 15.36 |

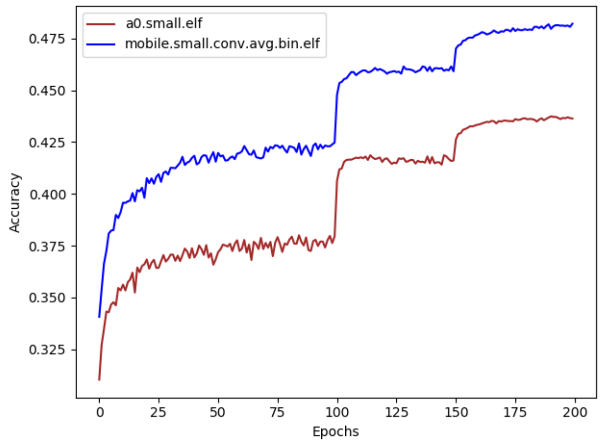

Fig. 9: The evolution of the policy validation accuracy for the different networks with less than one million parameters on the ELF dataset.

head and global average pooling value head to AlphaZero residual networks.

Table I gives the average number of batches per seconds reached by the networks when playing Go. The size of the batch is 32 since larger values did result in weaker play for the parallel PUCT algorithm.

Figure 7 gives the accuracy of the different networks according to the speed of the networks. We can see that for small high speed networks on the right of the figure the accuracy is similar but that for large low speed networks on the left the Mobile networks outperforms the AlphaZero-like networks.

Figure 8 gives the MSE loss of Mobile networks and residual network according to the speed of the networks. For all networks speeds, Mobile networks are better than AlphaZero-like networks.

### E. Training on ELF self-played games

Learning the value is difficult for AlphaZero-like networks on the Leela games. This may be due to Leela Zero resigning long before the endgame in states difficult to evaluate. The ELF self-played games are from a weaker engine and contains states easier to evaluate. The same networks as in the previous section are tested on the ELF dataset.

We can see in Figure 9 that the AlphaZero-like network is worse than a fully convolutional MobileNet on the ELF dataset with a network of less than 1 000 000 parameters.

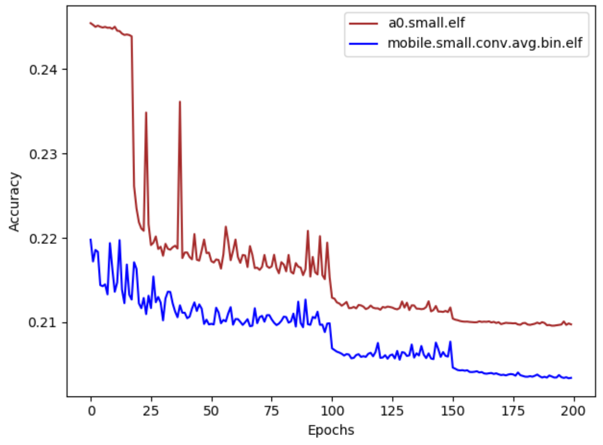

Fig. 10: The evolution of the value validation MSE loss for the different networks with less than one million parameters on the ELF dataset.

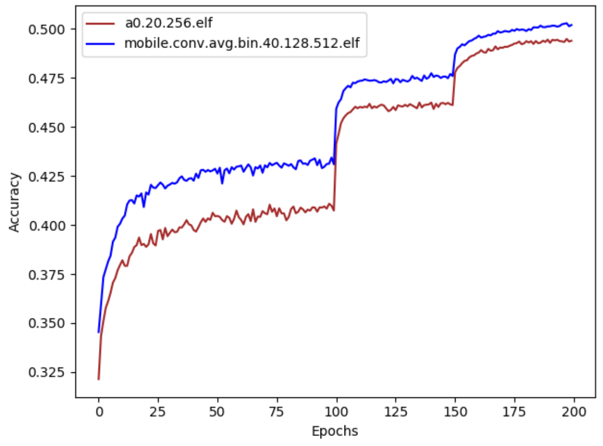

Fig. 11: The evolution of the validation policy accuracy for the different unbounded networks on the ELF dataset.

Figure 10 shows that small AlphaZero-like networks can learn the value of the ELF dataset when they could not on the Leela dataset. Nevertheless, the small Mobile networks still better learn the value than the AlphaZero-like networks.

We can see in Figure 11 that large Mobile networks have a better policy accuracy than large AlphaZero-like networks even if the Mobile network tested has much fewer parameters than the AlphaZero network.

Figure 12 show that the Mobile network we tested is slightly better for learning the value than the 20 blocks residual networks.

Figure 13 and Figure 14 show the parameter efficiency of Mobile and residual networks for the policy and the value on the ELF dataset. The policy accuracy and the value MSE loss are better for Mobile networks than for residual networks while using much fewer parameters. The networks used for this experiment are the 10, 20, 40 and 60 blocks Mobile networks and the 5, 10, 15 and 20 residual blocks networks.

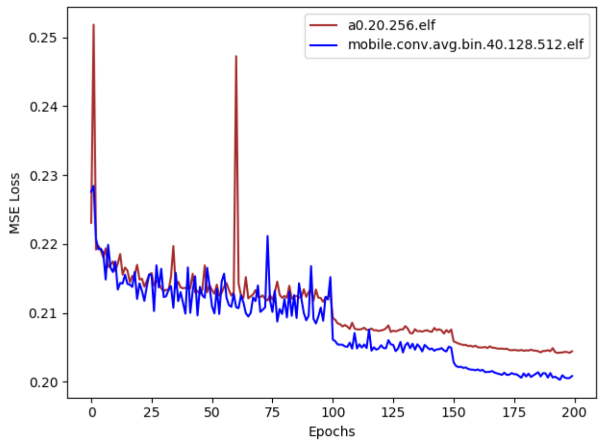

Fig. 12: The evolution of the validation value MSE loss for the different unbounded networks on the ELF dataset.

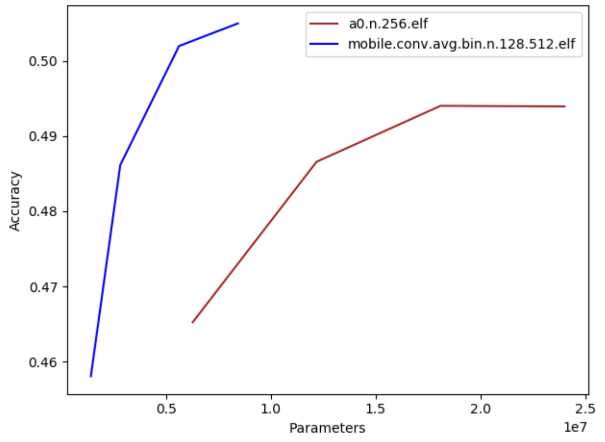

Fig. 13: The evolution of the policy validation accuracy with the number of parameters on the ELF dataset.

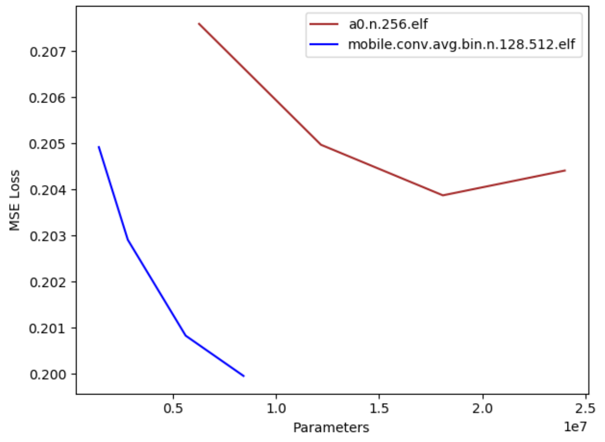

Fig. 14: The evolution of the value validation MSE Loss with the number of parameters on the ELF dataset.

TABLE II: Comparison of the number of forward pass per second.

| Network | Batch Size | Hardware | Forward/sec |
|---|---|---|---|
| a0.20.256 | 4 | CPU | 8.17 |
| a0.20.256 | 8 | CPU | 11.96 |
| a0.20.256 | 16 | CPU | 16.67 |
| a0.20.256 | 32 | CPU | 18.38 |
| a0.20.256 | 64 | CPU | 21.12 |
| a0.20.256 | 128 | CPU | 25.96 |
| a0.20.256 | 256 | CPU | 28.77 |
| a0.20.256 | 512 | CPU | 31.83 |
| mobile.conv.avg.bin.60.128.512 | 4 | CPU | 8.93 |
| mobile.conv.avg.bin.60.128.512 | 8 | CPU | 11.65 |
| mobile.conv.avg.bin.60.128.512 | 16 | CPU | 14.00 |
| mobile.conv.avg.bin.60.128.512 | 32 | CPU | 17.16 |
| mobile.conv.avg.bin.60.128.512 | 64 | CPU | 15.58 |
| mobile.conv.avg.bin.60.128.512 | 128 | CPU | 20.31 |
| mobile.conv.avg.bin.60.128.512 | 256 | CPU | 23.09 |
| mobile.conv.avg.bin.60.128.512 | 512 | CPU | 25.19 |
| a0.20.256 | 4 | GPU | 154.48 |
| a0.20.256 | 8 | GPU | 347.98 |
| a0.20.256 | 16 | GPU | 606.26 |
| a0.20.256 | 32 | GPU | 1003.37 |
| a0.20.256 | 64 | GPU | 1357.23 |
| a0.20.256 | 128 | GPU | 1672.95 |
| a0.20.256 | 256 | GPU | 1865.07 |
| a0.20.256 | 512 | GPU | 2025.22 |
| mobile.conv.avg.bin.60.128.512 | 4 | GPU | 196.51 |
| mobile.conv.avg.bin.60.128.512 | 8 | GPU | 361.92 |
| mobile.conv.avg.bin.60.128.512 | 16 | GPU | 575.12 |
| mobile.conv.avg.bin.60.128.512 | 32 | GPU | 1043.64 |
| mobile.conv.avg.bin.60.128.512 | 64 | GPU | 1434.10 |
| mobile.conv.avg.bin.60.128.512 | 128 | GPU | 1734.43 |
| mobile.conv.avg.bin.60.128.512 | 256 | GPU | 1911.81 |
| mobile.conv.avg.bin.60.128.512 | 512 | GPU | 2003.33 |

## F. *Self Play Speed*

We can see in Table II the number of forward passes per second of the networks according to the size of the batch in input of the networks. For small batches the 60 blocks Mobile network is faster than the 20 blocks residual network and we have seen that it is more accurate. For large batches residual networks are comparable to the speed of MobileNets. The GPU used for the experiments is a RTX 2080 Ti and the CPU is a 64 cores computer on Linux. The version of Tensorflow we used is 2.2.0 with cuda 10.2.89 and cudnn 7.6.5.

## G. *Making the networks play*

I made a round robin tournament between some of the networks in order to compare their level of play. The tournament gives each network one second per move using a RTX 2080 Ti. The MCTS algorithm used is PUCT [8]. The batch size for PUCT is set to 32. In order to have diversity in the games played by the same networks I randomized the choice of moves. Each move is ranked by the number of evaluations that are below it in the PUCT tree. If the second best move has more than 0.8 times the number of evaluations of the best move, it becomes a candidate for the move to be played. The engine chooses the second best move with a probability proportional to the number of playouts of the second best move divided by the number of playouts of the best move plus the number of playouts of the second best move, otherwise it plays the best move.

The results of the tournament are given in table III. The networks that play are networks trained on the Leela and

TABLE III: Round robin tournament between networks trained on the Leela and the ELF datasets.

| Network | Games | Winrate | $\sigma$ |
|---|---|---|---|
| mobile.conv.avg.bin.60.128.512 | 240 | 0.758 | 0.027 |
| mobile.conv.avg.bin.40.128.512 | 240 | 0.738 | 0.028 |
| mobile.conv.avg.bin.33.64.200 | 240 | 0.496 | 0.032 |
| mobile.conv.avg.bin.60.128.512.elf | 240 | 0.496 | 0.032 |
| a0.20.256 | 240 | 0.425 | 0.032 |
| a0.40.256 | 240 | 0.404 | 0.032 |
| a0.20.256.elf | 240 | 0.183 | 0.025 |

the ELF datasets. Mobile networks have better results than residual networks and networks trained on the Leela dataset have better results than networks trained on the ELF dataset. Even a small mobile network with less than one million parameters and 33 blocks has better results than large residual networks.

I also made the mobile.conv.avg.bin.40.128.512 network play on KGS. It plays instantly using the best move of the policy. It reached a stable 5 dan ranking. It is better than my previous residual policy network which reached a 4 dan ranking [3].

## VI. Conclusion

Residual networks were compared to Mobile networks with a fully convolutional policy head and a global average pooling value head. For the Leela dataset composed of games played at a superhuman level by a strong engine Mobile networks are better than residual networks both for small and for large networks. They have a better accuracy and value error. They are also better when compared according to the number of parameters of the networks. A tournament between the different networks using a fixed time per move confirmed that Mobile networks play better than residual networks that use many more parameters.

As future work it would be interesting to experiment with MobileNets in an AlphaZero-like deep reinforcement learning framework for games as well as in other combinatorial optimization problems.

## Acknowledgment

This work was granted access to the HPC resources of IDRIS under the allocation 2020-[AD011011826] made by GENCI. This work was supported in part by the French government under management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, reference ANR19-P3IA-0001 (PRAIRIE 3IA Institute).

## References

[1] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[2] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[3] T. Cazenave, "Residual networks for computer go," *IEEE Trans. Games*, vol. 10, no. 1, pp. 107–110, 2018. [Online]. Available: https://doi.org/10.1109/TCIAIG.2017.2681042

[4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, and T. Lillicrap, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[6] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, ser. Lecture Notes in Computer Science, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer, 2006, pp. 72–83.

[7] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *17th European Conference on Machine Learning (ECML'06)*, ser. LNCS, vol. 4212. Springer, 2006, pp. 282–293.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[9] T. Anthony, Z. Tian, and D. Barber, "Thinking fast and slow with deep learning and tree search," in *Advances in Neural Information Processing Systems*, 2017, pp. 5360–5370.

[10] Y. Tian, Jerry Ma*, Qucheng Gong*, Shubho Sengupta*, Z. Chen, J. Pinkerton, and C. L. Zitnick, "Elf opengo: An analysis and open reimplementation of alphazero," *CoRR*, vol. abs/1902.04522, 2019. [Online]. Available: http://arxiv.org/abs/1902.04522

[11] G.-C. Pascutto, "Leela zero," https://github.com/leela-zero/leela-zero, 2017.

[12] D. J. Wu, "Accelerating self-play learning in go," *CoRR*, vol. abs/1902.10565, 2019. [Online]. Available: http://arxiv.org/abs/1902.10565

[13] R. Emslie, "Galvanise zero," https://github.com/richemslie/galvanise_zero, 2019.

[14] J. Pitrat, "Realization of a general game-playing program," in *Information Processing, Proceedings of IFIP Congress 1968, Edinburgh, UK, 5-10 August 1968, Volume 2 - Hardware, Applications*, 1968, pp. 1570–1574.

[15] N. Love, T. Hinrichs, and M. Genesereth, "General game playing: Game description language specification," 2006.

[16] T. Cazenave, Y.-C. Chen, G.-W. Chen, S.-Y. Chen, X.-D. Chiu, J. Dehos, M. Elsa, Q. Gong, H. Hu, V. Khalidov, L. Cheng-Ling, H.-I. Lin, Y.-J. Lin, X. Martinet, V. Mella, J. Rapin, B. Roziere, G. Synnaeve, F. Teytaud, O. Teytaud, S.-C. Ye, Y.-J. Ye, S.-J. Yen, and S. Zagoruyko, "Polygames: Improved zero learning," *arXiv:2001.09832*, 2020.

[17] F. Chollet *et al.*, "Keras," 2015.

[18] F. Chollet, *Deep Learning with Python*. Manning, 2017.

[19] T. Cazenave, "Spatial average pooling for computer go," in *Computer Games - 7th Workshop, CGW 2018, Held in Conjunction with the 27th International Conference on Artificial Intelligence, IJCAI 2018, Stockholm, Sweden, July 13, 2018, Revised Selected Papers*, 2018, pp. 119–126.

[20] ——, "Improved policy networks for computer go," in *Advances in Computer Games - 15th International Conferences, ACG 2017, Leiden, The Netherlands, July 3-5, 2017, Revised Selected Papers*, 2017, pp. 90–100.

**Tristan Cazenave** Professor of Artificial Intelligence at LAMSADE, University Paris-Dauphine, PSL Research University and CNRS. Author of more than a hundred scientific papers about Artificial Intelligence in games. He started publishing commercial video games at the age of 16 and defended a PhD thesis on machine learning for computer Go in 1996 at Sorbonne University.

## APPENDIX
## SOURCE CODE

```
filters = 512
trunk = 128

def bottleneck_block(x, expand=filters, squeeze=trunk):
  m = layers.Conv2D(expand, (1,1),
                    kernel_regularizer=regularizers.l2(0.0001),
                    use_bias = False)(x)
  m = layers.BatchNormalization()(m)
  m = layers.Activation('relu')(m)
  m = layers.DepthwiseConv2D((3,3), padding='same',
                             kernel_regularizer=regularizers.l2(0.0001),
                             use_bias = False)(m)
  m = layers.BatchNormalization()(m)
  m = layers.Activation('relu')(m)
  m = layers.Conv2D(squeeze, (1,1),
                    kernel_regularizer=regularizers.l2(0.0001),
                    use_bias = False)(m)
  m = layers.BatchNormalization()(m)
  return layers.Add()([m, x])

def getModel ():
    input = keras.Input(shape=(19, 19, 21), name='board')
    x = layers.Conv2D(trunk, 1, padding='same',
                      kernel_regularizer=regularizers.l2(0.0001))(input)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    for i in range (blocks):
        x = bottleneck_block (x, filters, trunk)
    policy_head = layers.Conv2D(1, 1, activation='relu', padding='same',
                                use_bias = False,
                                kernel_regularizer=regularizers.l2(0.0001))(x)
    policy_head = layers.Flatten()(policy_head)
    policy_head = layers.Activation('softmax', name='policy')(policy_head)
    value_head = layers.GlobalAveragePooling2D()(x)
    value_head = layers.Dense(50, activation='relu',
                              kernel_regularizer=regularizers.l2(0.0001))(value_head)
    value_head = layers.Dense(1, activation='sigmoid', name='value',
                              kernel_regularizer=regularizers.l2(0.0001))(value_head)

    model = keras.Model(inputs=input, outputs=[policy_head, value_head])

    return model
```