Net2Net Extension for the AlphaGo Zero Algorithm

Hsiao-Chung Hsieh, Ti-Rong Wu, Ting-Han Wei, and I-Chen Wu(✉)

Department of Computer Science, National Chiao Tung University, 1001 University Road, Hsinchu, Taiwan, ROC
{michael81420,kds285,ting,icwu}@aigames.nctu.edu.tw

Abstract. The number of residual network blocks in a computer Go program following the AlphaGo Zero algorithm is one of the key factors to the program's playing strength. In this paper, we propose a method to deepen the residual network without reducing performance. Next, as self-play tends to be the most time-consuming part of AlphaGo Zero training, we demonstrate how it is possible to continue training on this deepened residual network using the self-play records generated by the original network (for time saving). The deepening process is performed by inserting new layers into the original network. We present in this paper three insertion schemes based on the concept behind Net2Net. Lastly, of the many different ways to sample the previously generated self-play records, we propose two methods so that the deepened network can continue the training process. In our experiment on the extension from 20 residual blocks to 40 residual blocks for 9×9 Go, the results show that the best performing extension scheme is able to obtain 61.69% win rate against the unextended player (20 blocks) while greatly saving the time for self-play.

Keywords: AlphaGo Zero · Deep learning · Net2Net

1 Introduction

Since AlphaGo Zero's [7] recent achievement of reaching superhuman level in Go, there have been numerous projects to reproduce or analyze its core algorithm, such as Facebook AI Research's ELF OpenGo [9], the crowd-sourced Leela Zero [6], and CGI [10]. The AlphaGo Zero algorithm works by training deep convolution neural networks (CNNs) using self-play game records, which

H.-C. Hsieh and T.-R. Wu–Equal contribution.

This research is partially supported by the Ministry of Science and Technology (MOST) under Grant Number MOST 107-2634-F-009-011 and MOST 108-2634-F-009-011 through Pervasive Artificial Intelligence Research (PAIR) Labs, Taiwan and also partially supported by the Industrial Technology Research Institute (ITRI) of Taiwan under Grant Number B5-10804-HQ-01. The computing resource is partially supported by national center for high-performance computing (NCHC).

© Springer Nature Switzerland AG 2020
T. Cazenave et al. (Eds.): ACG 2019, LNCS 12516, pp. 131–142, 2020.
https://doi.org/10.1007/978-3-030-65883-0_11

requires a large amount of computing resources. During the prototyping process, a common approach is to train a relatively small network, say, 20 residual blocks [2], to ensure that the chosen hyper-parameters are viable, and that the overall algorithm has been implemented correctly. Once this prototype converges, it is a non-trivial problem to improve overall playing strength by extending the training to use a deeper or wider network. On the one hand, while it is simple to retrain completely using the same hyper-parameters, the process of generating the self-play records can be very costly. On the other hand, if we reuse the previously generated self-play records, it is not clear how to initialize the larger network's parameters, nor do we know how the self-play records should be sampled.

Techniques have been proposed to rapidly transfer the information stored in one neural network (NN) (referred to as the parent network) into another NN (referred to as the child network), so that the training process of the larger child network can be accelerated. Net2Net [1] is one such technique that can accelerate training by transferring an NN into another deeper or wider NN without reducing performance in image recognition. Net2Net expands the parent network by adding *identity layers* to it; the output of these identity layers are essentially the same as its inputs, so the child network behaves the same as the parent network initially. While Net2Net has been shown to be useful for CNN architectures, the same technique cannot be easily applied to computer Go, where the building blocks tend to consist of residual networks (ResNets), as in AlphaGo Zero's case [7]. Namely, ResNets contain *shortcut connections* [2] to deal with the degradation problem, which complicates the design of identity layers.

In this paper, we propose a new method to deepen a previously-trained parent ResNet following the AlphaGo Zero algorithm. The expanded child network is able to retain comparable performance, with further potential for training. We then propose two methods to train and further improve this child network, where the training data consists of the same self-play game records. This allows us to skip the most time consuming step in the AlphaGo Zero algorithm. Given the same collection of self-play records, by expanding the 20 block parent network at 3/4 of the overall training progress into 40 blocks, we were able to reach the same level of strength as a randomly initialized 40 block network in only 1/4 of the total training time.

2 Background

In this section, we briefly review residual networks, the AlphaGo Zero algorithm, and the key concepts of Net2Net network extension technique.

2.1 Residual Networks

The ResNet architecture was proposed by He et al. [2] to address the *degradation* problem in DNNs. In short, it is intuitive to assume that deeper networks tend to be better universal function approximators, and so earlier on, researchers have attempted to improve performance by simply increasing the depth of NNs.

Without going into details, two problems can arise from having networks that are simply too deep: the vanishing/exploding gradient problem (solved by techniques such as normalization layers [4]) and the degradation problem. When the degradation problem occurs, performance saturates and converges at a lower accuracy despite having more layers in the NN. Degradation has a different root cause than vanishing gradients, and is not caused by overfitting. By using *shortcut connections* to allow features to skip over one or more layers, as shown in Fig. 1, deeper ResNets are able to overcome the degradation problem and obtain better accuracy than shallower networks.

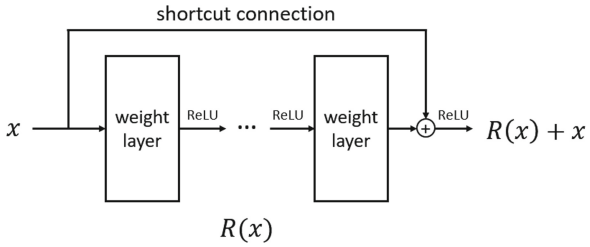

Fig. 1. Illustration of a ResNet block.

2.2 AlphaGo Zero

The goal of AlphaGo Zero is to train a Go agent using no human knowledge except the rules of the game. The algorithm is divided into 3 parts, self-play, optimization, and evaluation. Since this paper focuses on network training, the Monte Carlo tree search (MCTS) component of the AlphaGo Zero algorithm will not be discussed.

First, a randomly initialized network is used by the self-play player initially, denoted by p_0. In each epoch of training, this player continuously plays Go against itself to generate game records until a specified amount of games are collected. We refer to the set of game records generated in one epoch as a *collection*, denoted as c_k, and the self-play player as p_k, where k is the epoch number.

Second, during each epoch, a replay buffer is used to store the r most recent collections of game records, from which the optimization process involves sampling training data from this buffer to optimize the network. The replay buffer does not contain the full repository of game records because earlier collections (say, c_1) tend to be of too low quality for network optimization. In other words, the collections which are in the replay buffer need to match the current network's playing level. The hyperparameter r is referred to as the replay buffer size. During the predefined interval (from a collection newly used to the next), the network weights are saved as checkpoints, whose count is a hyperparameter.

Third, the network at each checkpoint is evaluated according to its win rate against the current self-play player. If the network at some checkpoint wins more than 55% of the games against the current self-play player, the former replaces

the latter as the new self-play player. By iterating these three steps, the strength of the self-play player generally improves. During the process the quality of game records will also increase.

2.3 Net2Net

Net2Net is a technique proposed to transfer a smaller network to a larger one [1]. With Net2Net, new layers are added to the original, smaller network (or *parent network*), to form a new, larger network (or *child network*). More specifically, a strategy called the *function-preserving initialization* is proposed, from which the parameter θ' of the child network g can be decided such that

$$\forall x, f(x, \theta) = g(x, \theta') \tag{1}$$

where x is the input data, f is the parent network, and θ is parent network's parameters. The strategy can also be use in partial consecutive networks. As long as strategy is satisfied, the output of the child network g will always be the same as the output of the parent network f. Following this strategy, the methods *Net2WiderNet* and *Net2DeeperNet* were both investigated, where the former tries to widen f and the latter tries to deepen f. Since the scope of this paper focuses on deepening ResNets for the AlphaGo Zero algorithm, we will not discuss Net2WiderNet further.

Net2DeeperNet uses identity layers I to deepen the parent network. Suppose the shape of the identity layer is (C_{in}, C_{out}, K, K) where C_{in} is the number of input channels, C_{out} is the number of output channels, and K is the kernel size, which is usually an odd number. Since the output must be equal to the input to satisfy the function-preserving strategy, $C_{in} = C_{out}$. The kernel of the identity layer at index (m, n) is then as follows,

$$I(m,n) = \begin{cases} \textit{identity kernel} & m = n \\ \textit{zero matrix} & \text{otherwise} \end{cases} \quad 1 \le m, n \le C_{in} = C_{out}. \tag{2}$$

The identity layer can be added anywhere in a network. However, we must take into consideration the activation functions. The network usually consists of an activation function ϕ after a convolution layer. To satisfy the function-preserving strategy, the activation function composition must satisfy

$$\forall v, \phi(I\phi(v)) = \phi(v) \tag{3}$$

where v is a vector. As an example, if ϕ is the sigmoid function, the condition would not be satisfied. On the other hand, a ReLU would be acceptable. As long as the strategy is satisfied, we can deepen a network by adding identity layers at any depth within the parent network.

3 Our Method

In this section, we describe our transfer method and how we train the child network using the parent network's collection of self-play records.

3.1 Transfer Method

We divide the method into two parts. First, we describe the *extension type*, which defines how we add new residual blocks. Second, we describe the *connection type*, which refers to where the new residual blocks are placed.

Extension Type. We introduction three extension types based on the strategy given in Eq. 1. For all three types, suppose that a block B is represented by a function $y = B(x, \theta)$, where y is the output. To add a new block B' after B, we must satisfy the following:

$$\forall x, B(x, \theta) = B'(B(x, \theta), \theta'), \tag{4}$$

where θ' is the parameter of the new block B'. We list the three extension types as follows.

1. Unit-extension: We add two identity layers in the new block. However, due to the shortcut connection architecture of ResNets, we must add a 1/2 scale operator at the end of the new block to ensure the sum remains at a similar scale.
2. Zero-extension: We add two convolutions in the new block. Instead of identity layers, we initialize the weights of the convolutions to be zeros, referred to as *zero layers*. There is no need to add the scale operator. However, this extension might be more difficult to converge during training.
3. Intra-extension: In Net2Net, several identity layers were added in every block. Therefore, we increase two identity layers within the original blocks. That is, with this extension, the number of blocks does not increase.

The activation function used in ResNets tend to be the ReLU function, so by definition Eq. 3 is satisfied. Furthermore, a small noise signal is added so that subsequent training of the child network will not remain at the local optimum of the parent network, and therefore lead to faster convergence. With this addition of noise, the performance of the child network is expected to be slightly worse than the parent network, but we believe that the child network should reach the parent network's performance rapidly, and eventually exceed it. This also ensures that the child network can learn to use the new blocks' capacity.

Connection Type. For clarity of communication, we introduce an encoding system that can represent where the new blocks are added into the parent networks in this paper. We use '1' to represent ten new blocks, and '0' to represent ten original blocks. Ext-ITL represents the new blocks and original blocks interleaved with each other. The encoding string from left to right refers to the child network architecture from the first layer to the last layer. Note that since the Intra-extension method does not create any new blocks, this encoding system does not apply to the method. To illustrate the various connection types, we extend the AlphaGo Zero training from 20 blocks to 40 blocks as an example, shown in Fig. 3.

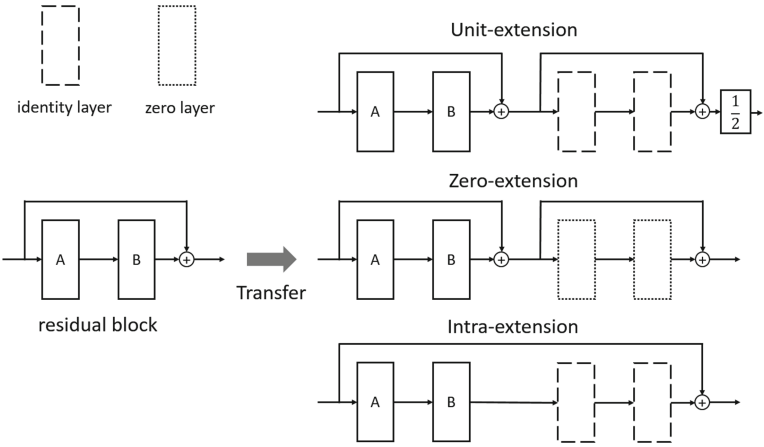

Fig. 2. Illustration of three extension types.

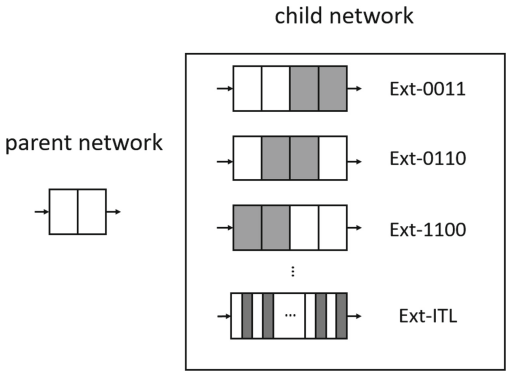

Fig. 3. Connection types. The white squares represent 10 original blocks, and the gray squares represent 10 new blocks. The Ext-ITL connection type represents 20 original blocks and 20 new blocks interleaved with each other.

3.2 Training Method

There are different approaches to sampling the game records when training the child network. More specifically, game records collections from earlier epochs tend to be less suitable for networks at later epochs, and vice-versa. Since the child network is expected to retain a similar level of strength as the parent network, once the parent network is transferred at some epoch, the child network should continue training with the game records from the replay buffer at that epoch. The problem is therefore at which epoch the transfer should take place. We now propose two different methods that describe when the parent network will be transferred, and how the child network should be subsequently trained.

End-Training. For end-training, the transfer occurs for the last self-play player p_n, that is, the last epoch is n. The r most recent collections (i.e. c_{n-r+1} to c_n) are loaded into the replay buffer, and the replay buffer does not need to be changed for the subsequent training of the child network. As shown in Algorithm 1, we transfer the player p_n to the child network. Then we load the last r collections into the replay buffer. Next, we simply train the child network using the replay buffer game records iteratively until a preset maximum number of iterations is reached. One of the caveats of end-training is that if the preset maximum number of iterations is small, the training may not be sufficient; on the other hand, if it is large, the same game records in the replay buffer will be trained many times, possibly leading to overfitting. Thus, another training method is proposed below.

Algorithm 1. End-Training

load game records in collections $c_{n-r+1} \sim c_n$;
while *maxIter is not reach* **do**
| select $c_{n-r+1} \sim c_n$ games to train the child network;
end

Shift-Training. When compared to the previous method, the parent network is transferred at epoch i. The game record collections matching the player p_i's level is loaded into the replay buffer. We then iteratively train the child network using the replay buffer, shifting the contents of the buffer to load more recent collections accordingly. As shown in Algorithm 2, we transfer the parent network for p_i to its child network, where i is referred to as the *transfer epoch*. Then we load at most r collections, specifically c_{i-r+1} to c_i, into the replay buffer. Next, we repeatedly load the next collection and train the child network using the replay buffer until the last collection c_n is loaded.

When training the child network, we increase the number of times t a sampled game is used to update the network. That is, since the child network is deeper than the parent network, our expectation is that more back-propagations need to be performed. For this reason, in this paper, each sampled game in shift-training is trained five times (i.e. $t = 5$), unless otherwise mentioned.

Algorithm 2. Shift-Training

```
if i − r + 1 > 0 then
    load game records in collections c_{i−r+1} ∼ c_{i−1};
    head = i − r + 1;
else
    load game records in collections c_1 ∼ c_{i−1};
    head = 1;
end
end = i;
while end ≤ n do
    load game records in collection c_end;
    if end - head = r then
        delete the c_head game records from replay buffer;
        head = head + 1;
    end
    select c_head ∼ c_end games to train the child network t times;
    end = end + 1;
end
```

4 Experiments

We demonstrate our method on 9×9 Go. We follow the AlphaGo Zero algorithm to train a Go agent with 20 ResNet blocks (and with 256 channels), at the end of which we obtain a player p_n. Our goal is to transfer the network to a deeper child network by adding 20 new blocks (consisting of 40 new convolution layers), following up with child network training, and evaluating the resulting player with p_n.

4.1 Experimental Setup

In the following experiments, we evaluate the strength of the resulting child network players by its win rate against the baseline p_n, where both players use 2 s of simulation time with a single NVIDIA Tesla V100 GPU. Since the child network is deeper than the parent network, the forward-pass of the child network takes more time. Nonetheless, the total simulation time is equally set to 2 s for both the child and the parent network. Each transfer method is expressed in terms of its extension type and its connection type. Furthermore, to speed up the overall experiments, for each match up, we continue playing against the baseline until a confidence interval of 95 % is reached to evaluate the probability of having a higher win rate against the baseline than 50%. We show the experiment setup in Table 1.

4.2 Experiment for Shift-Training

We use two experiments to analyze the effect of different transfer methods and transfer epoch i.

Table 1. Experiment hyperparameters and details.

Setting	
Replay buffer size r	20
Collection size	10000
Number of epochs n	243
Total batch size	2048
Learning rate	0.005
Weight decay	0.0001
SGD momentum	0.9
Local memory	754 GB
Thread count	36
Training hardware	8 GPUs (V100)

Comparison Between Transfer Methods. In this experiment, $i = 3n/4$. In addition to the transfer methods, we also trained a control group where the parent network is replaced by a randomly initialized 40 block network at $i = 3n/4$. This control group signifies what happens when no transfer occurs. Table 2 shows the highest win rate of various transfer methods. Unit-extension Ext-0011 has the best performance. It seems likely that the new blocks' capacity enhances the network to recognize more high level features. On the contrary, Zero-extension Ext-1100 performs the worse; it is possible that by introducing new layers, the low level features learned by the parent network were not carried over into the child network. With the exception of Zero-extension Ext-1100, all

Table 2. Result of various transfer methods.

Extension-type	Connection-type	Highest win rate of network (2 s)
Unit-extension	Ext-0011	61.69% (±4.00%)
	Ext-0110	55.45% (±3.88%)
	Ext-1100	57.07% (±3.29%)
	Ext-0101	47.85% (±3.78%)
	Ext-ITL	53.76% (±3.83%)
Zero-extension	Ext-0011	45.61% (±4.09%)
	Ext-0110	52.47% (±3.79%)
	Ext-1100	23.53% (±3.88%)
	Ext-0101	56.70% (±3.90%)
	Ext-ITL	48.15% (±3.77%)
Intra-extension	X	52.16% (±3.78%)
Random initialization (control)		42.60% (±4.34%)

other transfer methods exceed the random initialized control group, showing that the transfer methods can be used to preserve information learned by the parent network, with potential for further growth.

Comparison Between Transfer Epochs. Next, we set the transfer method to be Unit-extension Ext-0011. The transfer epoch i is set to $\{2n/3, 3n/4, 4n/5, 7n/8\}$, as shown in Table 3. The best performing transfer epoch is shown to be $3n/4$. Nonetheless, the win rates for all settings are higher than 50%, indicating that the resulting players can achieve at least p_n level. Earlier transfer epochs i imply more time spent training the more expensive child network, since the child network with shift-training uses 5 updates per sampled game. Therefore, there is a trade-off between transferring earlier (therefore spending more time training) and performance.

Table 3. Result of different transfer epochs.

i	Highest win rate of network (2 s)
$2n/3$	57.00% (±3.92%)
$3n/4$	61.69% (±4.00%)
$4n/5$	54.50% (±3.85%)
$7n/8$	51.92% (±3.77%)

Evaluation of Shift-Training. According to the previous experiments, Unit-extension Ext-0011 with $i = 3n/4$ is used as the setting for this experiment. We compare with the randomly initialized 40 block ResNet with $i = 1$ and $t = 5$, which is the naive method of retraining the 40 block network from scratch, with all other hyperparameters set equally. We refer to this group as the *40 block retrain* set. Compared with the retrained model, our method requires only about 1/4 of the training time, with its highest win rate to be as high as the retrained model, as shown in Fig. 4.

Additionally, we also retrained a separate 40 block ResNet with the same settings, except $t = 1.3$, so that the total number of training iterations are close to the transferred method. This second model is referred to as the *iteration normalized retrain* set. Our transfer method (Unit-extension Ext-0011) played against the 40 block retrain, iteration normalized retrain, and other high level players, where the result is shown in Table 4. The results show that Unit-extension Ext-0011 can match the 40 block retrain in performance, while exceeding the iteration normalized retrain model significantly. Furthermore, the win rates between Unit-extension Ext-0011 and other high level players are all higher than 50%. This shows that Unit-extension Ext-0011 does not exhibit signs of overfitting.

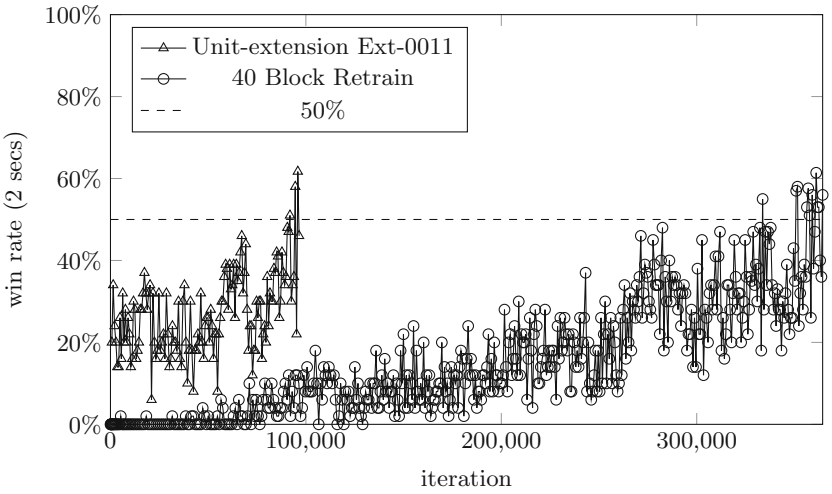

Fig. 4. The win rate curve of our method and the 40 block retrained model.

Table 4. Comparison with other high level players.

Player	Opponent	Win rate (2 s)
Unit-extension Ext-0011	40 block retrain	50.32% (±3.91%)
	Iteration normalized retrain	63.50% (±6.69%)
	p_{232}	89.00% (±4.35%)
	p_{235}	77.00% (±5.85%)
	p_{236}	72.50% (±6.20%)
	p_{238}	67.50% (±6.51%)
	p_{243}	61.69% (±4.00%)

Table 5. Results for end-training.

Extension-type	Connection-type	Highest win rate of network (2 s)
Unit-extension	Ext-0011	39.20% (±4.28%)
Intra-extension	×	64.76% (±4.03%)

4.3 Experiment for End-Training

In this experiment, we analyze two transfer methods of end-training. With end-training, the transfer epoch i can be thought of as fixed at n. As shown in Table 5, the results for Intra-extension seem to be very strong. This could be because the training data (from the replay buffer) is fixed, so it is relatively easier to fit. Playing between Intra-extension with end-training and the 40 block retrain with

2 s simulation time yields a win rate of 42.85%, so it is possible end-training can lead to overfitting.

5 Conclusion

We present a study on extending the ResNet of an agent trained using the AlphaGo Zero algorithm. We propose a scheme which transfers a parent network to a deeper child network without losing the learned knowledge; additionally, we propose two methods of sampling game records generated by the parent network for training the child network. Overall, the child network can retain the parent network's performance, or even surpass it with further training. From our experiments, we analyze a collection of hyperparameters to conclude that the Unit-extension Ext-0011 transfer method performs the best, with a win rate of 61.69% against the strongest player from the parent network. Finally, We demonstrate our method only requires about 1/4 of the training time as completely retraining a network of the same size, while simultaneously achieving the same level of performance.

Further research is left open to investigate more general extension-types in the future which can then be used for various network architectures, such as VGGNet [8], AlexNet [5], and DenseNet [3].

References

1. Chen, T., Goodfellow, I., Shlens, J.: Net2Net: accelerating learning via knowledge transfer. arXiv preprint arXiv:1511.05641 (2015)
2. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
3. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4700–4708 (2017)
4. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 (2015)
5. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
6. Pascutto, G.C.: Leela-Zero Github repository (2018). https://github.com/gcp/leela-zero
7. Silver, D., et al.: Mastering the game of Go without human knowledge. Nature **550**(7676), 354 (2017)
8. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
9. Tian, Y., et al.: ELF OpenGo: an analysis and open reimplementation of AlphaZero. arXiv preprint arXiv:1902.04522 (2019)
10. Wu, I.C., Wu, T.R., Liu, A.J., Guei, H., Wei, T.: On strength adjustment for MCTS-based programs. In: Thirty-Third AAAI Conference on Artificial Intelligence (2019)