Transferable Reinforcement Learning for Board Games

Natthaphon Hongcharoen

Thesis presented for Faculty of Engineering and Technology

Panyapiwat Insitute of Management

In study of bachelor of science, Computer Engineering

Studies year 2018

Abstract

The modern Reinforcement Learning has become widely interested recently, after the AlphaGo[1] became the first program to defeat a world champion in the game of Go. And by 2017, the AlphaGo Zero[2] and AlphaZero[3] programs achieved superhuman performance in the game of Go and Chess, by solely trained from games of self-play which require no human knowledge. But in contrast, both AlphaGo Zero and AlphaZero required extremely powerful processor as they need to random move in the early state of training which cost more expend. While in Computer Vision domain, we often use pretrained model from large dataset such as Imagenet[4] and retrain it for desired task which cost less time and achieved more accuracy than train it from scratch. In this thesis, we experiment a method to reuse the trained model of a game such as Othello, Connect4 or Gomoku and re-training with one different game by hoping it to be faster.

1 Introduction

1.1 Background

The modern Reinforcement Learning such as AlphaGo has showed outstanding performance by won against world champion in game of Go a decade earlier than predicted[5][6]. And the AlphaGo Zero has showed that it possible to train Reinforcement Learning without any human knowledge while still achieved good performance[2][3]. In contrast of good result, for AlphaGo, by using 'supervised learning from human expert games'[1] means it's require the expert games that might be difficult to obtained or inthe worst case, it might be impossible to get. And in case of AlphaGo Zero, according to original paper[2], the model need to be trained with 29 million games of self-play. Parameters were updated from 3.1 million mini-batches of 2,048 positions each. In Gian-Carlo Pascutto's experiments, every 2000 self-play moves generated by GTX-1080ti GPU would take 93 seconds, which means it need 1700 years to play all 29 million games[7]. It practically impossible to train AlphaGo Zero in personal level.

Meanwhile in Computer Vision domain, transferring weights which pretrained in large datasets and then re-train the model in preferred dataset is a widely used method as there are ImageNet[4] pretrained weights in many popular libraries[8][9][10]. The reason is it often improve regularizing and better performance and need less sample to train[11]. In the same way, Transfer Learning should be able to used in Deep Reinforcement Learning algorithm which use Deep Convolutional Neural Networks and help improve training speed and model's performance.

1.2 Objective

1. Create Reinforcement Learning models for board games such as Othello, Connect4 or Gomoku with AlphaZero algorithm.
2. Create Transferred Reinforcement Learning models from trained games.
3. Evaluate Transferred models performance against normal models.

1.3 Scope

1. Create at least 2 AlphaZero models and at least 2 Transferred models.
2. Evaluate performance of each algorithm in same iteration and same version.

2 Related Works

2.1 AlphaGo, Mastering the game of Go with deep neural networks and tree search

By March 2016 DeepMind AlphaGo has become the first computer program to won against the world champion in game of Go by defeated Lee Sedol, the winner of 18 world titles, which many years earlier than predicted[5][6]. The AlphaGo introduced a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state- of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. DeepMind also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away[1].

All abstract games have an optimal value function, $v^*(s)$, which determines the outcome, who would win the game from every board position or state s if both player play perfectly. These games may be solved by recursively computing outcome of every posible state, but in large games, such as chess[12] and especially Go[12], exhaustive search is infeasible[13][14], but the effective search space can be reduced by two general principles. First, the depth of the search may be reduced by position evaluation: truncating the search tree at state s and replacing the subtree below s by predicting the outcome from state s instead. This approach has led to superhuman performance in chess[15], checkers[16] and othello[17], but it was believed to be intractable in Go due to the complexity of the game[18]. Second, the breadth of the search may be reduced by sampling actions from a policy $p(a|s)$ that is a probability distribution over possible moves a in position s. For example, Monte Carlo rollouts[19] search to maximum depth without branching at all, by sampling long sequences of actions for both players from a policy p. Averaging over such rollouts can provide an effective position evaluation, achieving superhuman performance in backgammon[19] and Scrabble[20], and weak amateur level play in Go[21].

Monte Carlo tree search (MCTS)[22][23] uses Monte Carlo rollouts to estimate the value of each state in a search tree. As more simulations are executed, the search tree grows larger and the relevant values become more accurate. The policy used to select actions during search is also improved over time, by selecting children with higher values. Asymptotically, this policy converges to optimal play, and the evaluations converge to the optimal value function[23].

AlphaGo pass in the board position as a 19 × 19 image and use convolutional layers to construct a representation of the position. By sing these neural networks to reduce the effective depth and breadth of the search tree: evaluating positions using a value network, and sampling actions using a policy network. The neural networks trained using a pipeline consisting of several stages of machine learning. Begin by training a supervised learning (SL) policy network p_σ directly from expert human moves. This provides fast, efficient learning updates with immediate feedback and high-quality gradients. And also train a fast policy p_π that can rapidly sample actions during rollouts. Next, train a reinforcement learning (RL) policy network p_ρ that improves the SL policy network by optimizing the final outcome of games of self-play. This adjusts the policy towards the correct goal of winning games, rather than maximizing predictive accuracy. Finally, train a value network v_θ that predicts the winner of games played by the RL policy network against itself.

2.2 AlphaGo Zero, Mastering the Game of Go without Human Knowledge

AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. AlphaGo Zero introduced an algorithm based solely on reinforcement learning, without human data, guidance, or domain knowledge beyond game rules. AlphaGo becomes its own teacher: a neural network is trained to predict AlphaGo's own move selections and also the winner of AlphaGo's games. This neural network improves the strength of tree search, resulting in higher quality move selection and stronger self-play in the next iteration. Starting tabula rasa, AlphaGo Zero achieved superhuman performance, winning 100-0 against the previously published, champion-defeating AlphaGo.

AlphaGo Zero differs from first version AlphaGo in several important aspects. First and foremost, it is trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. Second, it only uses the black and white stones from the board as input features instead of many extracted features. Third, it uses a single neural network, rather than separate policy and value networks. Finally, it uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte-Carlo rollouts. To achieve these results, AlphaGo Zero introduced a new reinforcement learning algorithm that incorporates lookahead search inside the training loop, resulting in rapid improvement and precise and stable learning[2].

AlphaGo Zero uses a deep neural network f_θ with parameters θ. This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(p, v) = f_\theta(s)$. The vector of move probabilities p represents the probability of selecting each move (including pass), $p_a = P_r(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position s. This neural network combines the roles of both policy network and value network[1] into a single architecture. The neural network consists of many residual blocks[24] of convolutional layers[25] with batch normalisation[26] and rectifier non-linearities[27] (see Methods).

3 Methods

3.1 Reinforcement Learning

Our methods based on AlphaGo Zero's Reinforcement Learning algorithm. The neural network is trained from games of self-play by a novel reinforcement learning algorithm. In each state s of game, an MCTS search is executed, guided by the neural network f_θ . The MCTS search outputs probabilities π of playing each move (See Fig.1). These search probabilities usually select much stronger moves than the raw move probabilities p of of the neural network $f_\theta(s)$. The neural network's parameters are updated to make the predicted move probabilities and value $(p, v) = f_\theta(s)$ more closely match the MCTS improved search probabilities and self-play winner (π, z); these new parameters are used in the next iteration of self-play. The Monte-Carlo tree search uses the neural network f_θ to guide its simulations. Each edge (s, a) in the search tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, and an action-value $Q(s, a)$. Each simulation starts from the root state and iteratively selects moves that maximise an upper confidence bound $Q(s, a) + U(s, a)$, where $U(s, a) \propto P(s, a)/(1 + N(s, a))$[1], until a leaf node s' is encountered. This leaf position is expanded and evaluated just once by the network to generate both prior probabilities and evaluation, $(P(s', \cdot), V(s')) = f(s')$. Each edge (s, a) traversed in the simulation is updated to increment its visit count $N(s, a)$, and to P update its action-value to the mean evaluation over these simulations, $Q(s, a) = 1/N(s, a) \sum_{s'|s,a\to s'} V(s')$, where $s, a \to s'$ indicates that a simulation eventually reached s' after taking move a from position s (See Fig.2).

The neural network is trained by a self-play reinforcement learning algorithm that uses MCTS to play each move. First, the neural network is initialised to random weights θ_0. At each subsequent iteration $i \geq 1$, games of self-play are generated. At each time-step t, an MCTS search $\pi_t = \alpha_{\theta_{i-1}}(s_t)$ is executed using the previous iteration of neural network $f_{\theta_{i-1}}$, and a move is played by sampling the search probabilities π_t. A game terminates at step T when both players pass, when the search value drops below a resignation threshold, or when the game exceeds a maximum length; the game is then scored to give a final reward of $r_T \in \{-1, +1\}$. The data for each time-step t is stored as (s_t, π_t, z_t) where $z_t = \pm r_T$ is the game winner from the perspective of the current player at step t. In parallel, new network parameters θ_i are trained from data (s, π, z) sampled uniformly among all time-steps of the last iteration(s) of self-play. The neural network $(p, v) = f_{\theta_i}(s)$ is adjusted to minimise the error between the predicted value v and the self-play winner z, and to maximise the similarity of the neural network move probabilities p to the search probabilities π . Specifically, the parameters θ are adjusted by gradient descent on a loss function l that sums over mean-squared error and cross-entropy losses respectively,

$$(p, v) = f(s), \qquad l = (z - v)^2 - \pi^\top \log p \tag{1}$$

We applied our reinforcement learning pipeline to train our programs. The base neural networks model started training from completely random behaviour and continued without human intervention over 50 iterations for 512 games each, in total 25,600 games, using 25 simulations for each MCTS. Parameters were updated using Adaptive Moment Estimation[29] from recent 20 iterations play history with 4,096 mini-batch size(See Training Pipeline for more detail). The neural network contained 256 × 4 residual blocks(See Neural Network Architecture for more detail). This is very few number compare to 4.9 millions games and 1,600 MCTS simulations in original AlphaGo and AlphaGo Zero but it still took more than 10 days on a single K80 card.

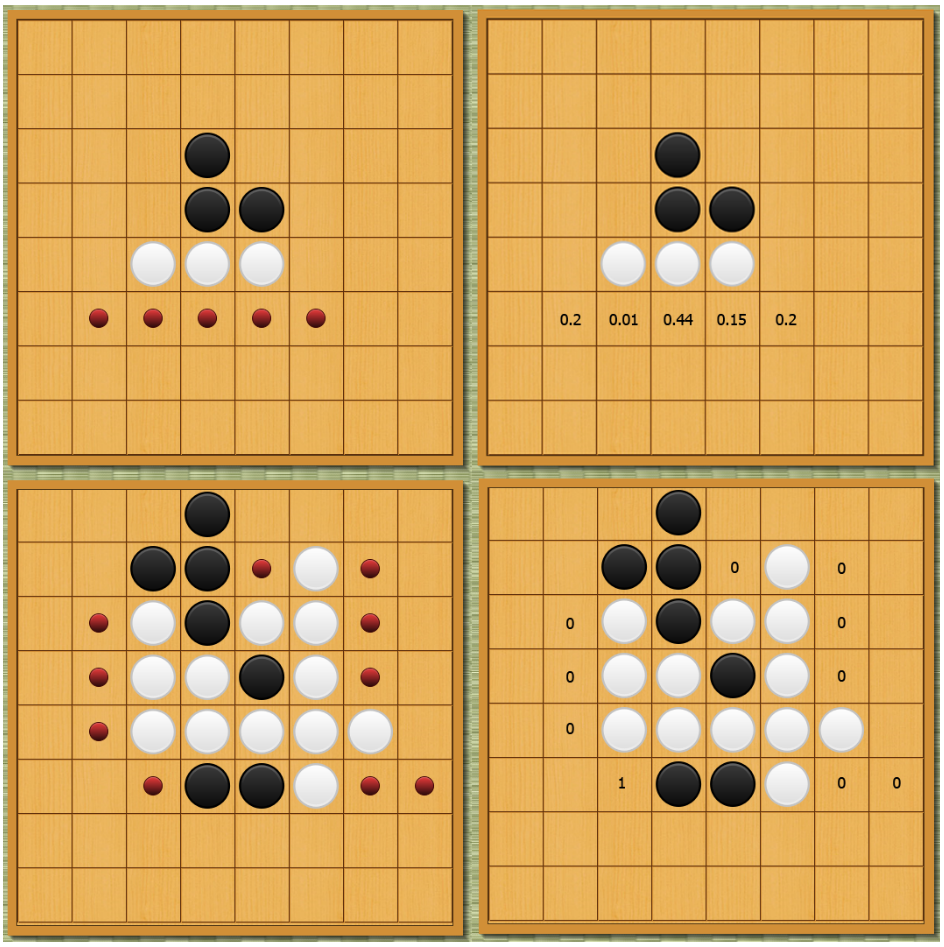


Figure 1: MCTS search outputs explanation. ***Top Left***: A state of Othello game after 2 moves, red dots are valid moves. ***Top Right***: MCTS outputs for first 15 moves of each game, the Probabilty of move are sparse, this means the training model has 20% chance to select the leftmost valid square or 44% for the middle square, this ensures a diverse set of positions are encountered . ***Buttom Left***: A state after 16 moves. ***Buttom Right***: MCTS output after 15 moves, this will ensure that the best move will always being selected.

3.2 Training Pipeline

Optimisation Each neural network f_{θ_i} is optimised on a single Tesla K80 GPU using Keras with TensorFlow backend. The batch-size is 4096. Each mini-batch of data is sampled uniformly at random from all positions from the most recent 10240 games(20 iterations) of self-play. Instead of stochastic gradient descent with momentum and learning rate annealing like AlphaGo and AlphaGo Zero, we optimised neural network parameters by Adaptive Moment Estimation[29]. The learning rate is fixed at 0.001, β_1 is 0.99 and β_2 is 0.9.

Evaluator To ensure we always generate the best quality data, we evaluate each new neural network checkpoint against the current best network f_{θ_*} before using it for data generation. Each evaluation consists of 100 games, using an MCTS with 25 simulations to select each move, using an infinitesimal temperature $\tau \to 0$ (i.e. we deterministically select the move with maximum visit count, to give the strongest possible play). If the new player wins by a margin of $> 55\%$ (to avoid selecting on noise alone) then it becomes the best player α_{θ_*}, and is subsequently used for self-play generation, and also becomes the baseline for subsequent comparisons.

Self-Play In each iteration, the best current player α_{θ_*} plays 512 games of self-play, using 25 simulations of MCTS to select each move. For the first 15 moves of each game, the temperature is set to $\tau = 1$; this selects moves proportionally to their visit count in MCTS, and ensures a diverse set of positions are encountered. For the remainder of the game, an infinitesimal temperature is used, $\tau = 0$.

3.3 Search Algorithm

The Search Algorithm is almost identical to AlphaGo Zero[2]; we recapitulate here for completeness.

Each node s in the search tree contains edges (s, a) for all legal actions $a \in A(s)$. Each edge stores a set of statistics,

$$\{N(s,a), W(s,a), Q(s,a), P(s,a)\},$$

where $N(s,a)$ is the visit count, $W(s,a)$ is the total action-value, $Q(s,a)$ is the mean action-value, and $P(s,a)$ is the prior probability of selecting that edge.

Select The first in-tree phase of each simulation begins at the root node of the search tree, s_0 , and finishes when the simulation reaches a leaf node s_L at time-step L. At each of these time-steps, $t < L$, an action is selected according to the statistics in the search tree, $a_t =$ argmax $Q(s_t, a) + U(s_t, a)$, using a variant of the PUCT algorithm,

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

where c_{puct} is a constant determining the level of exploration; this search control strategy initially prefers actions with high prior probability and low visit count, but asympotically prefers actions with high action-value.

Expand and evaluate Positions in the queue are evaluated by the neural network using a mini-batch size of 1; the search thread is locked until evaluation completes. The leaf node is expanded and each edge (s L , a) is initialised to $N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_a$; the value v is then backed up.

Backup The edge statistics are updated in a backward pass through each step $t \leq L$. The visit counts are incremented, $N(s_t, a_t) = N(s_t, a_t) + 1$, and the action-value is updated to the mean value, $W(s_t, a_t) = W(s_t, a_t) + v, Q(s_t, a_t) = \frac{W(s_t,a_t)}{N(s_t,a_t)}$.

Play At the end of the search AlphaGo Zero selects a move a to play in the root position s_0, proportional to its exponentiated visit count, $\pi(a|s_0) = N(s_0, a)^{1/\tau} / \sum_b N(s_0, b)^{1/\tau}$, where τ is a temperature parameter that controls the level of exploration. The search tree is reused at subsequent time-steps: the child node corresponding to the played action becomes the new root node; the subtree below this child is retained along with all its statistics, while the remainder of the tree is discarded.

3.4 Neural Network Architecture

The input to the neural network is a 8 × 8 × 2 image stack comprising 2 binary feature planes. The first plane X consist of binary values indicating the current player's stones position.The last plane Y represent the corresponding features for the opponent’s stones. These planes are concatenated together to give input features $s_t = [X, Y]$.

The input features s_t are processed by a residual tower that consists of a single convolutional block followed by 4 residual blocks[24].

The convolutional block applies the following modules (See Fig.3 for easier to understand):

1. A convolution of 256 filters of kernel size 3 × 3 with stride 1
2. Batch normalisation[26]
3. A rectifier non-linearity

Each residual block applies the following modules sequentially to its input:

1. A convolution of 256 filters of kernel size 3 × 3 with stride 1
2. Batch normalisation
3. A rectifier non-linearity
4. A convolution of 256 filters of kernel size 3 × 3 with stride 1
5. Batch normalisation
6. A skip connection that adds the input to the block

7. A rectifier non-linearity

The output of the residual tower is passed into two separate "heads" for computing the policy and value respectively. The policy head applies the following modules:

1. A convolution of 2 filters of kernel size 1 × 1 with stride 1
2. Batch normalisation
3. A rectifier non-linearity
4. A fully connected linear layer that outputs a vector of size $8^2 = 64$ corresponding to logit probabilities for all intersections.

The value head applies the following modules:

1. A convolution of 1 filter of kernel size 1 × 1 with stride 1
2. Batch normalisation
3. A rectifier non-linearity
4. A fully connected linear layer to a hidden layer of size 256
5. A rectifier non-linearity
6. A fully connected linear layer to a scalar
7. A tanh non-linearity outputting a scalar in the range [−1, 1]

The overall network depth is 8 parameterised layers for the residual tower, plus an additional 2 layers for the policy head and 3 layers for the value head.

3.5 Transfer Learning

We subsequently applied Transfer Learning to the Connect4 games trained neural network, by re-initialize parameters of the probabilities θ_ρ and value θ_υ output layers while using the same feature extractor layers, the new move probabilities and value would became $p = f_{\theta_\rho}(f_\theta(s))$ and $v = f_{\theta_\upsilon}(f_\theta(s))$ respectively.

The transferred neural network is trained for Othello games by the same procedure as the base network for another 25600 games. We performed 2 difference networks setup for transfered training, one trained the whole network just like the base model, another one is by fixed parameters of feature extractor layers and update only new initialized output layers.

We evaluated the transferred models using an internal tournament against fully randomize initializing model. The internal tournament consisting 2 part, for 3 and 25 MCTS simulations respectively. As the matter of time limit, we can only compare Othello game, trained from scratch and transferred from Connect4 game one each.

3.6 Shared Learning

We also experimented Shared Learning. By using same feature extractor layers f_{θ_γ} but different input θ_ι and output$(\theta_\rho, \theta_\upsilon)$(See Figure 4). Trained both games simultaneously to experiment that, is it possible to make a Neural Network that can do multiple task.

Training We aplied shering method for Othello and Connect4 games. Each games play 512 games per iteration. The parameters θ are updated using most recent 10240 games sample from one game(take it as Othello) and then updated using another game(take it as Connect4). The order of update is randomed to prevent bias. Another procedures are identical to prior experiment. We evaluate each new neural network checkpoint against the current best network f_{θ_*} before using it for data generation. The neural network f_{θ_i} is evaluated by the performance of an MCTS search α_{θ_i} that uses f_{θ_i} to evaluate leaf positions and prior probabilities. Each evaluation consists of 50 games for each games, intotal of 100, using an MCTS with 25 simulations to select each move. If the new player wins by a margin of $> 55\%$ for all result, and $> 48\%$ for each game(to avoid the network become good at on game but bad at another), then it becomes the best player α_{θ_*}, and is subsequently used for self-play generation, and also becomes the baseline for subsequent comparisons.

Evaluating We evaluated the model using an internal tournament against fully randomize initializing model from prior experiment. The internal tournament take match for every version of each game. As the matter of time, we can only evaluated the Othello models.

4 Result

4.1 Time Consumation

For the 4 blocks model, each iteration of 512 games takes about 5-6 hours for self-play and around 1 hours for evaluation, more than 300 hours for 50 iterations, around 2 weeks. And the 8 blocks model use twice time of this, at around 600 hours, almost a month. So we can only trained once per each experiment.

Here is the server's spec.

CPU	Intel Xeon E5-2630 v3 2.4 GHz 16 cores
GPU	nVidia Tesla K80
RAM	512GB DDR3

Table 1: The Train Server specification.

4.2 Transfered Othello

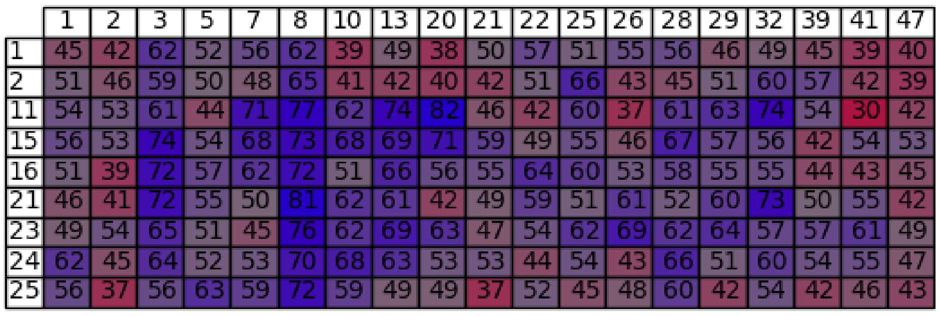
The Transfer Learning experimental result is very disapointed, as the Transferred model is totally outperformed by the Fully Random Initialize(Scratch) model. The evaluating result shown in Table 2.

MCTS sims	Scratch	Transferred
5	85	15
25	89	11

Table 2: **Match result**. Winning score of the internal tournament for scratch and transferred models.

4.3 Shared Learning

The Shared Learning experimental result in unexpected way. The early versions of Shared model are outperformed Normal model in every version but the latter versions are worse. Figure 5 show win percentage of Shared model against normal model.



	1	2	3	5	7	8	10	13	20	21	22	25	26	28	29	32	39	41	47
1	45	42	62	52	56	62	39	49	38	50	57	51	55	56	46	49	45	39	40
2	51	46	59	50	48	65	41	42	40	42	51	66	43	45	51	60	57	42	39
11	54	53	61	44	71	77	62	74	82	46	42	60	37	61	63	74	54	30	42
15	56	53	74	54	68	73	68	69	71	59	49	55	46	67	57	56	42	54	53
16	51	39	72	57	62	72	51	66	56	55	64	60	53	58	55	55	44	43	45
21	46	41	72	55	50	81	62	61	42	49	59	51	61	52	60	73	50	55	42
23	49	54	65	51	45	76	62	69	63	47	54	62	69	62	64	57	57	61	49
24	62	45	64	52	53	70	68	63	53	53	44	54	43	66	51	60	54	55	47
25	56	37	56	63	59	72	59	49	49	37	52	45	48	60	42	54	42	46	43

Figure 5: Shared model win percentage. Blue color means Shared model is better, red means worse.

5 Conclusions and Recommendations

As I have experienced it myself, Reinforcement Learning needs large amount of resource to make it done. As I have mentioned in Sec.4 that it took couple weeks just to make a small model(4 blocks/256 kernel) learn an easy game(Othello, which normally has only 3-6 valid moves per state or 8x8 Connect4 which has exactly 8). It's would difficult to train an RL model with personal computer since you would need to run at full GPU performance without any rest for many days or else, it would take even more time.

As for the experimental, there are some noticeable methods that have been used in the original AlphaGo Zero but I choose not to use them. First, adding Dirichlet noise to ensures that all moves may be tried. Second, I use Adam optimizer instead of SGD(See [2] Self-Play Training Pipeline section for more detail). I believe that they might make the model's convergence slower and want to try both use and not use them, but since only one experiment took more time than expected, I did not have change to prove my presumption. And the third, multi-thread MCTS search(Mentioned in [2]: Search Algorithm). I tried, but it bugged. And since it is multi-threaded tracking the error was really difficult so I gave up. Since the self-play process consume only 40-70% of GPU, multi-threading should speend up this process(I forget to capture GPU usage of the self-play so I did not mention this earlier)

References

[1] 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, pp. 484--489, 2016.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. H. A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, ``Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354--359, 2017.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, ``Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, ``Imagenet: A large-scale hierarchical image database," *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[5] A. Levinovitz, ``The mystery of go, the ancient game that computers still can't win." https://www.wired.com/2014/05/the-world-of-computer-go/, 2014.

[6] D. Silver and D. Hassabis, ``Alphago: Mastering the ancient game of go with machine learning." https://ai.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html, 2016.

[7] G.-C. Pascutto, ``[computer-go] zero performance." http://computer-go.org/pipermail/computer-go/2017-October/010307.html, 2017.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, ``Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016.

[9] Facebook, ``Tensors and dynamic neural networks in python with strong gpu acceleration." http://pytorch.org/.

[10] F. Chollet, ``Keras deep learning for humans." http://keras.io/.

[11] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, ``How transferable are features in deep neural networks?," *CoRR*, vol. abs/1411.1792, 2014.

[12] L. V. Allis, ``Searching for solutions in games and artificial intelligence," 1994.

[13] H. van den Herik, J. W.H.M.Uiterwijk, and J. van Rijswijck, ``Games solved: now and in the future," *Aritificial Intellegence*, vol. 134, pp. 277--311, 2002.

[14] J. Schaeffer, ``The games computers (and people) play," *Advances in Computers*, vol. 52, pp. 189--266, 2000.

[15] M. Campbell, A. J. Hoane, Jr., and F.-h. Hsu, ``Deep blue," *Artif. Intell.*, vol. 134, pp. 57--83, Jan. 2002.

[16] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron, ``A world championship caliber checkers program," *Artificial Intelligence*, vol. 53, no. 2, pp. 273 -- 289, 1992.

[17] M. Buro, ``From simple features to sophisticated evaluation functions,'' in *Proceedings of the First International Conference on Computers and Games*, CG '98, (London, UK, UK), pp. 126--145, Springer-Verlag, 1999.

[18] M. Muller, ``Computer go,'' *Artif. Intell.*, vol. 134, pp. 145--179, Jan. 2002.

[19] G. Tesauro and G. R. Galperin, ``On-line policy improvement using monte-carlo search,'' in *Advances in Neural Information Processing Systems 9* (M. C. Mozer, M. I. Jordan, and T. Petsche, eds.), pp. 1068--1074, MIT Press, 1997.

[20] B. Sheppard, ``World-championship-caliber scrabble,'' *Artif. Intell.*, vol. 134, pp. 241--275, Jan. 2002.

[21] B. Bouzy and B. Helmstetter, ``Monte-carlo go developments,'' in *Advances in Computer Games*, pp. 159--174, Springer US, 2004.

[22] R. Coulom, ``Efficient selectivity and backup operators in monte-carlo tree search,'' in *Proceedings of the 5th International Conference on Computers and Games*, CG'06, (Berlin, Heidelberg), pp. 72--83, Springer-Verlag, 2007.

[23] L. Kocsis and C. Szepesvari, ``Bandit based monte-carlo planning,'' in *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, (Berlin, Heidelberg), pp. 282--293, Springer-Verlag, 2006.

[24] K. He, X. Zhang, S. Ren, and J. Sun, ``Deep residual learning for image recognition,'' in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770--778, June 2016.

[25] Y. LeCun, ``Gradient-based learning applied to document recognition,'' 1986.

[26] S. Ioffe and C. Szegedy, ``Batch normalization: Accelerating deep network training by reducing internal covariate shift,'' in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 448--456, JMLR.org, 2015.

[27] R. H. R. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, ``Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit,'' *Nature*, vol. 405, pp. 947--951, 2000.

[28] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960.

[29] D. P. Kingma and J. Ba, ``Adam: A method for stochastic optimization,'' *CoRR*, vol. abs/1412.6980, 2014.