Population Based Training for Reinforcement Learning by the example of Leela Chess Zero

Lennart Wallis[1], Léon Lenzen[2]

Abstract

The aim of this paper is to show that PBT can reduce the additional effort involved in hyper parameter tuning. The already existing DNN *AlphaZero*, which is being expanded to include a PBT, serves as the basis for orientation. Determining appropriate hyperparameters requires a lot of experience or time trying different hyperparameters. For this reason we implemented PBT in *AlphaZero* in order to represent a direct comparison with a well-trained network. Our test results are able to show that the time or assumed experience can be reduced using PBT. Through PBT, the hyperparameters are adapted, whereby the hyperparameters always evolve towards the optimal value. In our tests, the PBT approach does not deliver perfect results such as the small sacrifice in accuracy. But it comes close to the result with the hyperparameters that have been tested over many years with a trend of achieving the same or even better results. For 200k steps the execution time of the PBT approach takes less than twice as long as with hand-tuned hyperparameters (Hand-Tuned approach: ∼37h Vs. PBT approach: ∼59h). Therefore more than one test run with hand-tuned hyperparameters consumes more time than the PBT approach.

[1] *Lennart.Wallis@smail.th-koeln.de*
[2] *Leon_Gerrit.Lenzen@smail.th-koeln.de*

Contents

1. Introduction

The development of artificial intelligence has made great strides in the recent years. Due to the constant improvement of the accessible hardware resources, it is possible to develop algorithms that are becoming more and more complex. The general availability of high-performance hardware enables even small developers to implement ambitious projects or, as in this case, to jump on major projects from the past. Due to the constant increase in system complexity, it is no longer possible for humans to precisely determine the optimal hyperparameters of the system. Even one of the most ambitious projects *AlphaZero* involves this problem. The preset hyperparameters of the *AlphaZero* were developed experimentally through several time-consuming test runs [1].
Secondly, the problem of comparability arises. The comparison of projects in the field of artificial intelligence in terms of speed shows too strong a dependency on the operating hardware [2]. Therefore, projects (especially crowd-sourced) can only be traced under the same hardware conditions in terms of improvement or should always be seen in their operating hardware context.
Within this paper, both problems within the adaptation of *AlphaZero* are to be addressed. Population Based Training (PBT) should make it possible to determine the optimal hyperparameters of the system. In addition, a short time comparison of both approaches should be shown, which could provide a conclusion about optimal hyperparameterization.

2. Background

For the sake of illustration, the Neural Network (NN) *AlphaZero* is explained below. In addition, the general approach of PBT is described. This should help to build an understanding of section 3.

2.1 AlphaZero

The main difference between *AlphaGo* and *AlphaGoZero* is the whole learning system of *AlphaGoZero* doesn't depend on human input data. The self-contained process evolves purely from the system's own experience solely learned by selfplays, starting with random weights. The goal of the algorithmic approach of *AlphaGoZero* in comparison to *AlphaGo* is gaining more generality by less complexity.

There are two Convoluted Neural Network (CNN)s:

- policy network: Gets an input of one single state of the game and determines the best next action. It acts as a recommendation system within the Monte Carlo Tree Search (MCTS) to narrow down possible choices.
- value network: Gets an input of one single end-state of the game recommended by the policy network and determines the winning probability for each player as a scalar value between -1 and +1.

The players compete in a defined number of games against themselves. Each step taken by the players results in a game state which is used to train the CNN. After the training with the new gameplays the resulting CNNs are evaluated by playing 400 games against the previous best CNNs. By achieving a winrate above 55% the CNNs are considered the new best player and therefore used for generating new selfplays. After every iteration higher quality data evolves within the networks. For more detailed information on this topic review [3],[4].

2.2 PBT

Hyperparameter tuning is one of the most complex tasks in building a Deep Neural Network (DNN). The PBT method introduced in [5] introduces a new method for Hyperparameter tuning. The optimal hyperparameters of the system can be determined by simulating different networks. The technique is a hybrid of the two most commonly used methods for hyperparameter optimisation:

- Random Search: Parallel independent training of DNN's and at the end of training the highest performing one is selected. Our Implementation of this is further explained in subsection 3.4
- Hand-Tuning: Based on experience of implementer. This also needs a lot of evaluation and most of the time tends to be a very repetitive and long process.

The PBT method also trains, like the random search approach, parallel networks but differs in the independence. It uses the information of the rest of the population to refine the hyperparameters and directs computational resources to the promising models. This approach is called *Exploiting*. In addition, hyperparameters that are not promising are tuned in different directions to explore other possibilities for the network. This approach is called *Exploration*. Both *Exploitation* and *Exploration* allow the network to put more resources onto promising models through appropriate hyperparameter tuning.

3. Implementation

This approach is based on an already existing program for training a DNN for *Leela Chess Zero* available under the name *lczero-training* on *Github* [6]. Here, before starting the training, a configuration file must be created in which both the model structure and the hyperparameters for the training are stored. Thus, although a staggered Learning Rate (LR) is supported, it can only be adjusted before the training starts.

The PBT process can be roughly divided into three steps that are executed over and over again. This is exactly how our implementation is structured. The three steps are:

1. Training of the agents
2. Tournament of the trained agents
3. Update of the agent configuration based on the tournament result

In the following, this process is called an evolution. Our implementation contains these evolutions. However, before performing these steps, a few preparations have to be made.

3.1 Data preparation

For each evolution a random selection from the dataset is made in advance. The used Computer Chess Rating Lists (CCRL) dataset consists of a total of 2.5 million games, where 20% are the test set and 80% are the training set. This pre-selection has to be done because otherwise the training algorithm would reimport all 2.5 million games (about 15GB) for each training [7]. This results in a longer preparation time, but the training time is significantly reduced.

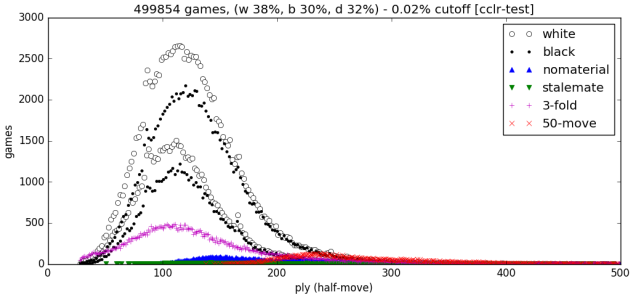

Figure 1. CCRL dataset
Source: `https://lczero.org/blog/2018/09/a-standard-dataset/`

Figure 1 shows the distribution of the games within the 500000 test dataset games. Thus, 38% were won with white,

30% with black. Additionally 32% of the games were draws. The double line for black and white indicate the games won by checkmate or by resignation. The missing 0.02% were removed from the graph because they contained more than 500 plies.

3.2 Initialisation

An initial population of any number of agents is generated for the first evolution. For this, the LR is set with a random value between 0 and 0.1. Each agent is assigned a Universally Unique Identifier (UUID), which allows the checkpoints and tensor boards to be assigned to an agent. To simplify the transfer of parameters for training, a configuration file is created for each agent. In addition to the UUID of the agent and the corresponding LR, the rest of the configuration for *lczero-training* is also stored in this file.

3.3 Training

As already mentioned at the beginning, we resort to *lczero-training*, since the actual training is already implemented here with the help of *tensorflow*. The model we used has 128 filters and 10 residual blocks. The previously created configuration file is now passed to the train script of *lczero-training*. Each agent is trained sequentially due to limited hardware resources. At the end of the training a *tensorflow* checkpoint is saved for each agent. The checkpoint contains information about the weights and all the data necessary to resume training. It will be used later for the tournament and the update of the population.

3.4 Evaluation

After all agents have been trained, the tournament is run to evaluate the strongest or weakest agent. For this the weights of the checkpoints are passed to the actual chess engine of the *Leela Chess Zero* project, *lc0* [6]. All possible pairings are played through. The two opponents, consisting of the trained agents, play a total of 50 games against each other. Afterwards, *lc0* outputs a win rate, which is used to determine which of the agents emerges as the winner from the 50 games. In addition a simple scoring system is used to build a ranking. The winner gets 3, the loser 0 and in case of a draw both get 1 point each. The winner is the agent who could score the most points.

3.5 Updating Population

This ranking list is then used to decide which agents should continue to exploit their LR and which should explore new LRs. For this purpose, the upper half of the ranking list is copied into the next evolution and thus continues to exploit their LR. The weights of the lower half are overwritten with the weights of the upper half and the LR is adjusted. The weights of the worst are overwritten with the weights of the best, then the second worst with the second best and so on. The same is done with the LR. In addition, an ε is added to the new LR. This ε consists of a random number, which lies between 20% of the current LR and -20% of the current LR as shown in Source Code 1. Now the agent configuration file is overwritten with the new values and the loop starts again at training.

Source Code 1. Calculation of ε

```
def get_epsilon(current_lr: float) -> float:
    return random.uniform(-current_lr*0.2,
                          current_lr*0.2)
```

4. Results

To evaluate the PBT method we developed, a base run was performed with the configuration recommended for the dataset. It should be noted that this configuration specifies several intervals for the LR. The values can be taken from Table 1 and are displayed in Figure 2 in blue.

Table 1. LR Intervals for Base Run

Steps	LR
0-80k	0.1
80k - 140k	0.01
140k - 180k	0.001
180k - 200k	0.0001

All following graphs use a smoothing factor of 0.7. Smoothing is applied because the PBT values do have quite a bit of noise. The faded lines do show the actual values. The PBT run was performed on the same hardware as the base run and used 8 different agents with a evolution size of 2000 steps. Within each graph, the original hand-tuning approach is shown in blue (test) and orange (train). The other Lines are the 8 agents of the PBT approach. It should be mentioned that the comparison of the PBT is compared with a mature hand-tuning approach representing a tough comparison.

4.1 Learning Rate

In the base run, the LR is adjusted at fixed intervals as already described in section 4. Here, either a lot trial and error has to happen or the engineer's wealth of experience has to be drawn upon to specify these intervals.

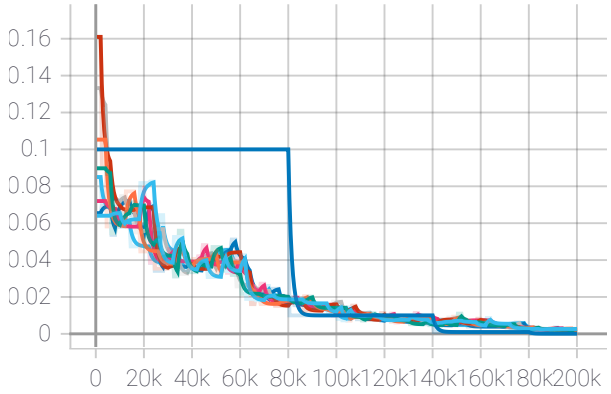

Figure 2. Comparison LR

A comparison of the learning rates of the two approaches is shown within Figure 2. The hand-tuned approximately represented approach includes a fixed drop as shown in Table 1. At the beginning it becomes clear that the PBT approach ranks much lower than hand tuning. The hand-tuning approach is ahead of the PBT model again from 80k steps, but is caught up again quickly. The same effect is also evident in the transition to 140k steps. Based on the trend of the PBT model, it can be assumed that a further improvement above the 200k steps limit will be shown. This is an advantage of PBT over the hand-tuned approach.

4.2 Policy Accuracy

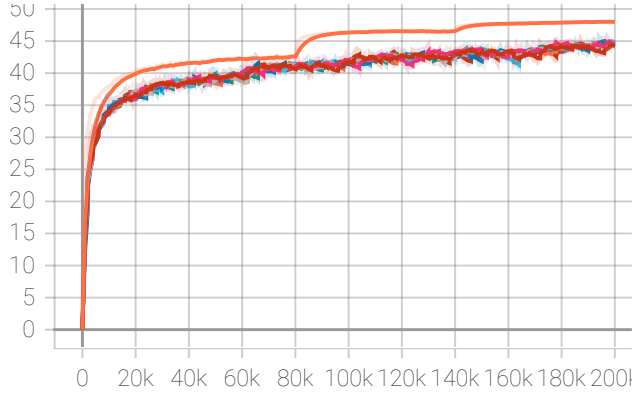

Figure 3. Comparison Policy Accuracy

The policy accuracy determines how accurate the DNN was able to predict the best next move. As shown in Figure 3 the first two decreases of the LR for the hand-tuned approach are clearly visible. The last decrease does not have a visible impact on the policy accuracy. Before each decrease of LR the curve plateaus suggesting the maximal policy accuracy for this LR is reached. The PBT approach on the other hand shows a steady positive gradient, correlating to the steadily decreasing LR. At the 80k steps mark (first LR decrease in hand-tuned) the PBT approach nearly closed the gab, which existed since the start due to slower learning in the beginning. In the end the hand-tuned DNN reached 48.05% and the PBT DNN reached 45.47%. Due to the still positive gradient in the PBT approach, the policy accuracy is likely to increase with more steps.

4.3 Value Accuracy

The value accuracy determines the probability of winning the game as described in subsection 2.1. The Comparison between the PBT approach and the hand-tuned approach is better in value accuracy than in policy accuracy. Figure 4 shows the same trend as the policy accuracy Figure 3. In the context of the hand-tuned network being fully developed the PBT approach shows acceptable results with a small sacrifice in accuracy.

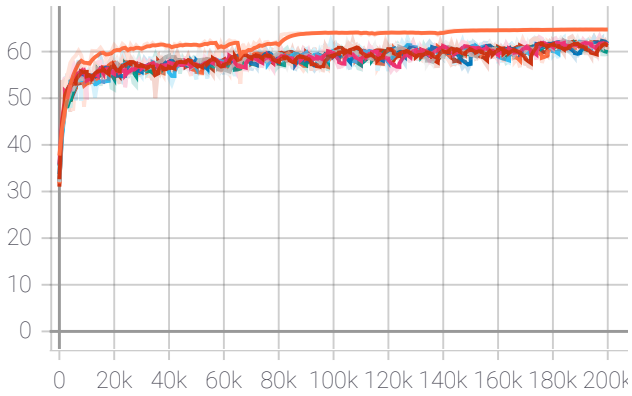

Figure 4. Comparison Value Accuracy

4.4 MSE Loss

The Mean Squared Error (MSE) loss curves of the hand-tuning approach, as shown in Figure 5, both graphs for training and testing are getting closer together and do not have much interference. On the other hand the PBT graphs are quite noisy and build a corridor around the hand-tunded curves with the training PBT MSE loss being below and the testing PBT MSE loss being above both hand-tuned losses.

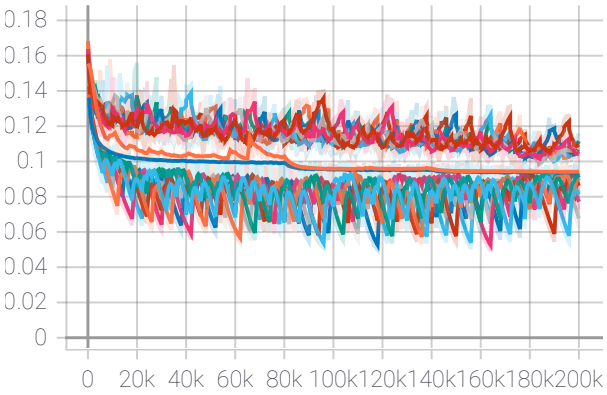

Figure 5. Comparison MSE Loss

On closer inspection the MSE loss shows signs of overfitting as the test value goes up and the train value yields in the opposite direction. Figure 6 illustrates this by showing a single agent in closer detail. This behavior persists even over several evolutions. Each evolution is getting new randomly selected games for training and testing. However, the graphs do not suggest this.

4.5 Policy Loss

The policy loss Figure 7 shows the same signs of overfitting as the MSE loss in subsection 4.4. After the first decrease of the LR for the hand-tuned approach both the training and testing policy loss of the PBT approach are above either values from the hand-tuned approach. But overall they are in the same region around 1.6 to 1.7 versus 1.55 for the hand-tuned approach.

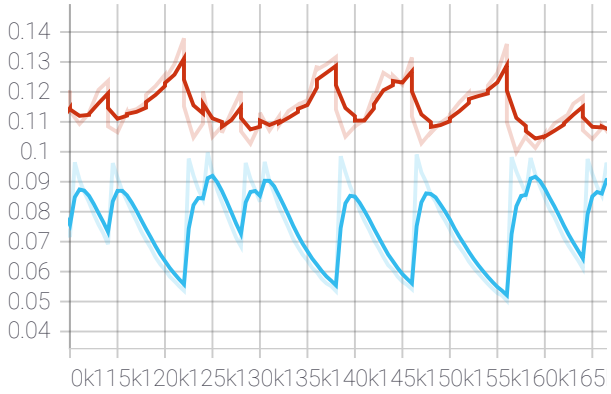

Figure 6. MSE Loss single Agent

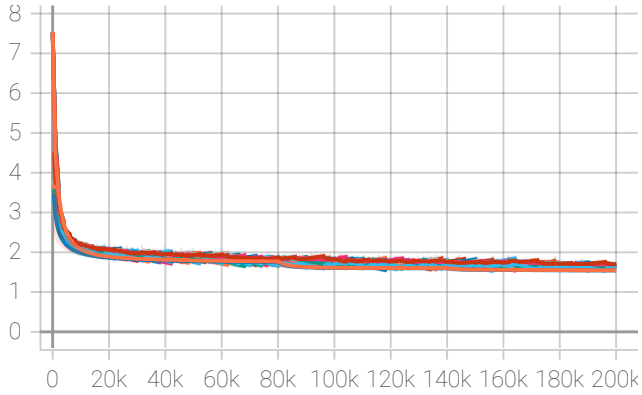

Figure 7. Comparison Policy Loss

4.6 Value Loss

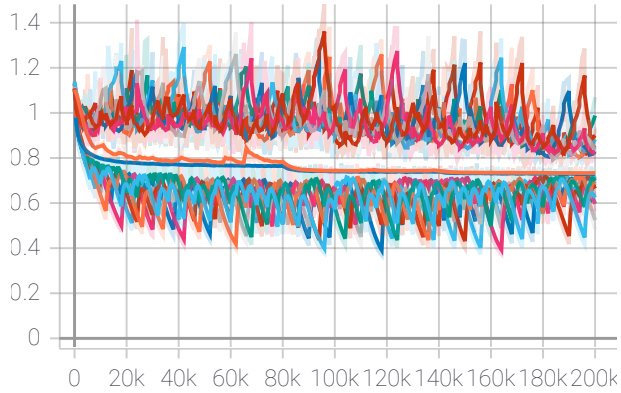

Figure 8. Comparison Value Loss

The overfitting issue from subsection 4.4 is also present at the value loss in Figure 8 resulting in value spikes. At the end, after 165k steps, the test value loss (upper corridor) from the PBT approach starts to decrease overall. Furthermore, it can be assumed that the spikes are getting smaller as the network evolves since the overfitting is decreasing because of the new generated data of each evolution.

5. Outlook

In subsection 4.2 it was shown that due to the still positive gradient in the PBT approach, the correctness of the guidelines can even be increased by further steps. This was not done due to lack of time.

Despite the overfitting issue discussed in subsection 4.4, the PBT approached managed to train a DNN which is just slightly weaker than the hand-tuned approach. Due to time constraints we were not able to run another PBT session to fix the overfitting. With increased evolution size and higher data volumes per evolution, noise is likely to be reduced because of the lower effect of overfitting. This should also increase the accuracy.

The script used for training has some quirks like running a test step on the first step on each evolution which pollutes the graphs resulting in high spikes.

The developed solution uses a *SQLite* database for the internal management of the entities, which creates a history independent of *tensorflow* logs. In addition, this lays the foundation for the individual agents to be executed in a distributed manner using a crowdsourced approach in a further development stage. It is already possible to track the status and progress of an evolution via the database, which in turn can be used to resume an aborted evolution.

List of Abbreviations

PBT Population Based Training

DNN Deep Neural Network

CNN Convoluted Neural Network

LR Learning Rate

CCRL Computer Chess Rating Lists

UUID Universally Unique Identifier

NN Neural Network

MCTS Monte Carlo Tree Search

MSE Mean Squared Error

References

[1] I-Chen Wu Ti-Rong Wu, Ting-Han Wei. Accelerating and improving alphazero using population based training. Paper, March 2020. https://arxiv.org/abs/2003.06212; viewed on 24. Nomvember 2020.

[2] Rishad Shafik, Adrian Wheeldon, and Alex Yakovlev. Explainability and dependability analysis of learning automata based ai hardware. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–4. IEEE, 2020.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, and Sifre. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[4] The Artificial Intelligence Channel. Deepmind alphazero - mastering games without human knowledge. Website, Januar 2018. https://www.youtube.com/watch?v=Wujy7OzvdJk&feature=emb_logo; viewed on 10. February 2021.

[5] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[6] LcZero Community. Lczero. Repository, 2020. https://github.com/LeelaChessZero; viewed on 24. Nomvember 2020.

[7] LcZero Community. A standard dataset. Website, September 2018. https://lczero.org/blog/2018/09/a-standard-dataset/; viewed on 20. Januar 2021.