IMPROVING THE TACTICAL AWARENESS OF DEEP NEURAL NETWORK CHESS ENGINES

Mateen Ulhaq

April 19, 2021

***Abstract*—Deep neural network chess engines demonstrate a unique capacity for a positional understanding and long-term judgements not often seen in classical chess engines. Unfortunately, their tactical awareness is comparatively lacking: deep reinforcement learning-based chess engines miss relatively simple two-move and three-move tactical combinations and forced checkmates that classical chess engines often find quite quickly. This paper proposes methods to remedy this problem by introducing tactical data samples into the datasets that deep reinforcement learning-based chess engines are trained on. In particular, this paper demonstrates that a simplified and significantly smaller network architecture is capable of identifying the correct piece to move in tactical positions with a top-1 accuracy of 67% and a top-3 accuracy of 96%. Inclusion of tactically-aware sub-networks and training techniques can likely lead to stronger chess engines overall.**

I. INTRODUCTION

The recent advances in deep learning based chess engines have excited many enthusiasts within the chess engine community. These have brought to light interesting ideas and offered fresh perspectives in engine design. Indeed, even the dominant classical engine Stockfish [1] has adopted small neural network architectures to replace its evaluation head. Stockfish 13 was released with Efficiently Updatable Neural Networks (NNUE) [2], effectively demonstrating a surprisingly significant increase in engine strength over the purely classical predecessors. As human expert capacity reaches its limits for classical engine programmers, the need for learned architectures increases. Neural networks for chess are here to stay in one way or another.

A. Related works

One of the early works which sparked interest in deep learning for chess engines was the AlphaZero [3] engine. In December 2017, DeepMind demonstrated that with 40000 TPU-hours of training, a neural network-based chess engine could compete on-par with the state-of-the-art classical engine at the time, Stockfish 8. AlphaZero used a search algorithm based on PUCT (Polynomial Upper Confidence Trees) [4], which was a domain-specific improvement over Monte Carlo Tree Search (MCTS) that was used in AlphaGo [5]. In PUCT search, chess position nodes are expanded depending on the UCB (upper confidence bound) metric,

$$\text{UCB} = \frac{\text{Wins}}{\text{Trials}} + \lambda\sqrt{\frac{\ln\left(\text{Trials}_{\text{parent}}\right)}{\text{Trials}}},$$

which is a summation of the predicted win ratio for a given node's descendants and an upper confidence term that shrinks as more decendants are expanded in relation to the parent node, where λ is some exploration parameter. The network is used to provide policy and value assessments at each node, which determine which nodes to expand next and provide a score for the likelihood of the node representing a winnable chess position. The network was trained via a well-known technique from deep reinforcement learning: supervised updates were applied to the network using training data generated through engine self-play with that network. At each iteration, the updated network was used to generate further training data through self-play. The network architecture itself will be further discussed in Section II.

The success of AlphaZero sparked considerable interest in the chess engine community. In January 2018, Leela Chess Zero [6] (often abbreviated LCZero or Lc0), a new community-run project, was announced to replicate the results of AlphaZero. Since the computing resources required for this project are enormous, all training and self-play data generation is done by a distributed community effort. Over the last two years, Leela Chess Zero has made novel changes to its architecture, namely the introduction of Squeeze-and-Excitation (SE)

blocks [7], which allow feature maps to control the strength of their responses via trainable embeddings. Further changes include exploration of output formats for the policy and value heads.

B. Towards better tactical awareness

One well-known problem in Leela Chess Zero is that "shallow tactics" consisting of as few as simple two-move combinations are sometimes missed [8], [9]. Though the policy network performs exceedingly well at positional play and long-term judgements, this lack of short-term tactical awareness puts the network at a severe disadvantage against classical engines, which are nearly always very quick at finding forced tactical lines and mates, even several moves deep.

This work proposes one way to remedy this: by training specifically on classes of problems that the model finds difficulty in, it will be less likely to miss such tactics. For comparison, human grandmasters often also train on a healthy balance of tactical problems, in addition to learning from real games. Since training on the Leela Chess Zero architecture is a somewhat costly process, this paper is satisfied with presenting a smaller proof-of-concept. A simplified architecture based on the Leela Chess Zero architecture is used to assess a network's capacity to identify the most likely piece to move in a tactical position.

II. Methods

Tactics puzzles were taken from the lichess puzzles database [10], containing 1 million unique puzzles. Data was cleaned to put it into a usable format. In particular, the data comes with FEN positions of the move *before* the opponent's "blunder" that prompts the tactic. This was fixed by applying the first blunder to the position and updating the FEN position. The initial best moves for each of the tactical lines were also extracted.

A representation for boards consists of one-hot encoded 8x8 boards (similar to "bitboards" in chess programming literature). A separate 8x8 board is used for each piece type (pawn, knight, bishop, rook, queen, king) and each color (black, white), producing a total of 12 separate 8x8 boards. This is visualized in Figure 1. In Leela Chess Zero, a game history prior of the 8 latest moves is also taken as input. However, tactics are much more short-term and rapidly fluctuate across positions, so a history prior is unlikely to help, except for perhaps the previous move's "blunder". In addition, game details such as castling rights and the 50-move-rule have been omitted since all input tactical positions assume full castling rights (if valid) and absence of the 50-move-rule. Thus, to simplify and reduce the size of the architecture, this paper only uses a $12 \times 8 \times 8$ tensor as input to the network. It may later be adopted into a full $112 \times 8 \times 8$-input network by simply initializing the weights that are related to the other input channels to 0.

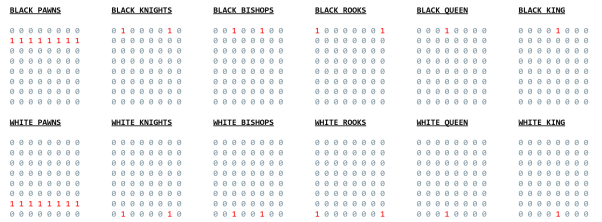

Figure 1. All 12 8x8 boards visualized for the standard starting position, `rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1` (FEN). Entries labelled "1" represent the presence of that particular piece type on that square.

Much of the network architecture this work uses is similar to the Leela Chess Zero architecture [11] — which, in turn, is similar to the AlphaGo Zero network architecture shown in Figure 2. Indeed, all architectures take a game state as an input. This is then processed through a "residual tower" of duplicated residual blocks. Each residual block outputs a $C \times 8 \times 8$ tensor, and its non-identity branch is comprised of conv + BatchNorm + ReLU + conv + BatchNorm layers. Following the addition of the identity and non-identity branches, a ReLU activation is applied. All convolutional layers are padded 3x3, in an effort to maintain a 8x8 board shape within the features.

The output of the residual tower is a common feature set. This $C \times 8 \times 8$ tensor is then passed to various output "heads", in a manner akin to multi-task learning. Both Leela Chess Zero and AlphaZero use a policy head and a value head, though this work differs in this respect. In this work, the policy and value heads have been removed. Substituted in their place is a "piece head", which outputs an 8x8 board probability map identifying the most likely piece to move.

This work proposes a "piece head" that takes a $C \times 8 \times 8$ feature tensor and transforms it to a 8×8 probability map by way of a convolutional layer and

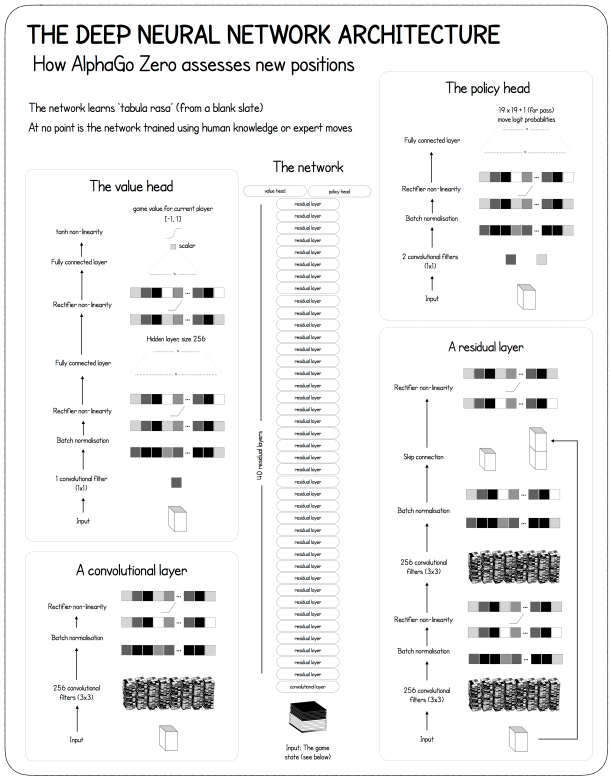

Figure 2. AlphaGo Zero network architecture. There are many similarities between the AlphaGo Zero and Leela Chess Zero architectures. Both architectures input a game state into the convolutional neural network. This game state is processed through a "residual tower", which acts as a feature extractor consisting of a number of residual blocks. The $C \times 8 \times 8$ (in $C \times H \times W$ format) output tensor of the residual tower is then passed into the output "heads", namely a "policy head" to determine the best move and a "value head" to assign a score to the input state. Picture adapted from [12].

dense layer. Training this head (along with the rest of the deep network) is equivalent to a traditional multi-class classification problem, so cross-entropy with softmax is used for the loss function.

III. Experiments and Results

The cleaned dataset of 1 million tactics positions was shuffled and split into a 70%-20%-10% train-test-validation split. The data was trained in batches of size 1024 with an AdamW optimizer with a weight decay parameter of 10^{-2} acting as a L2 regularizer integrated within the optimizer itself. A learning rate of 10^{-5} was used since larger learning rates saturated very quickly in very few epochs at a top-1 accuracy of roughly 50%. To improve generalization, Stochastic Weight Averaging

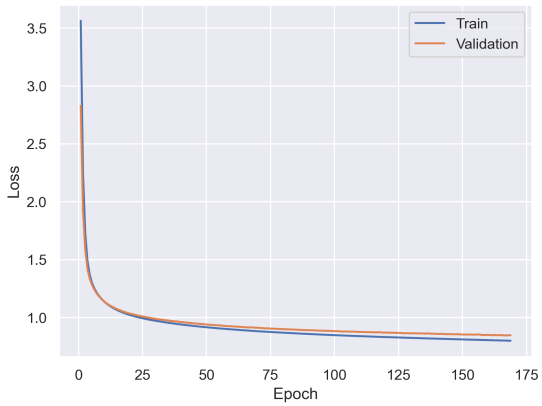

Figure 3. Loss curves for validation and training sets.

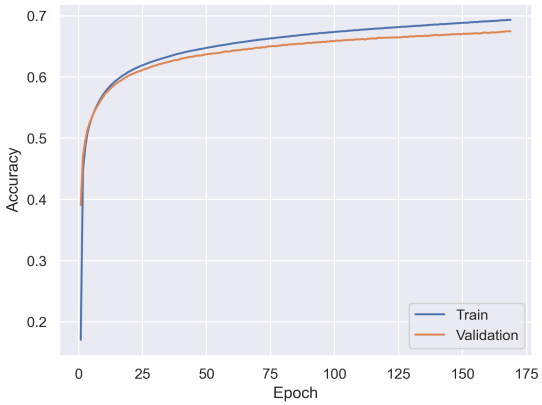

Figure 4. Top-1 accuracy curves for validation and training sets.

(SWA) [13] was also applied to enforce a weighted moving average of the weights.

After 190 epochs, the network reached a top-1 accuracy of 69% on the training set; for validation, it reached a top-1 accuracy of 67% and a top-3 accuracy of 96%. The loss curves are shown shown in Figure 3 and top-1 accuracy curves in Figure 4. These are discussed below.

IV. Discussion

The training and validation losses both continue to fall. Indeed, because the validation loss continues to decrease, it is unlikely that the network has been overfit. Though the training was stopped here, the validation top-

1 accuracy still appears to be growing linearly at a rate of 2% every 100 epochs. (This was measured over the last 50 epochs.) However, the growth rate is decaying, so it is unlikely that top-1 accuracy will experience much further growth.

Though the results look promising, one might ask: is the network simply naively learning to select pieces which are often involved in tactical situations? In descending order, the attacking pieces are most often queens and knights, with rooks and bishops to follow. Pawn attackers are largely absent from the dataset: likely, this is because pawns often function in a *positional* way, and only play supporting roles in tactics. Kings, of course, are much more often the *subject* of attacks! Likely, this is one of the reasons why classical chess engines have been parameter-tuned so that their weighting functions heavily depend on factors influencing king safety, particularly in the presence of an enemy queen.

Perhaps this is a large part of what the presented network is doing: merely learning to identify the most problematic pieces, given the presence of other pieces. This is of course done without human expert guidance, so there are perhaps novel relationships the network may also have discovered. Perhaps additional research and investigation are required. Indeed, one could investigate if a simple dense network architecture is capable of learning to solve the same tactical problems by way of an input representation that only provides information on whether a piece is on the board or not.

However, the fact that key insights such as this have not been addressed architecturally by the current dominant chess engine architectures suggests that there is significant room for improvement. Classical engines benefit greatly from decades of human expert programmers improving engine technology by discovering highly domain-specific methods, representations, and encodings. The lack of application of such domain-specific knowledge and data-driven design shows that there are great advances waiting to be made within this field.

V. Conclusion

This work investigated a "piece head" approach towards improving tactical awareness of deep reinforcement learning-based chess engines. It found that even a greatly simplified network architecture is capable of learning from a relatively small dataset (1 million samples). The simplified model managed to identify the correct piece with a top-1 accuracy of 67% and a top-3 accuracy of 96%. This proof-of-concept demonstrates that the larger, deeper network architectures used by real-world neural network chess engines — which this work was based on — should be capable of learning, at the very least, from similar techniques.

A. Future work

There are many ideas which have not yet been tried within deep reinforcement learning-based chess engines — or at any rate, they remain unpublished. Some promising ideas that the author proposes *in regards to improving tactical awareness* are outlined below.

1) Integrating proposed sub-network into primary architecture: It still needs to be shown that integrating the tactically-aware sub-network into the primary Leela Chess Zero architecture can lead to real-world improvements. The output of a "piece head" trained on tactics samples (as proposed by this work) may be added or multiplied to the policy head in an appropriate manner so as to improve the policy network's suggestions for node expansion. The piece head may be trained separately from the network, using the pre-trained network as a feature extractor. A technique along these lines could significantly improve "shallow tactics" performance at very little additional cost. Alternatively, a "zero-cost" solution would involve merging tactical problems into the dataset. However, this presents a class imbalance problem, since the tactics problems dataset is much smaller than the self-play generated dataset. The next point proposes a similar type of solution without this problem.

2) Verifying training data using classical engines for analysis: Classical engines are significantly better at tactical awareness, and often require very little computation to determine a good policy in most kinds of tactical situations. Thus, training data could be verified by a "quick" one-second analysis. This would help identify tactics that were missed by the network, so that the win/draw/loss (WDL) labelling of the training data could be appropriately corrected. This would likely lead to better tactical awareness since the training data itself would not contain obvious tactical errors! Self-play data with corrected errors could be further trained upon frequently to present the network with more challenging samples to learn from. Furthermore, this is far less computationally expensive than the process of generating the training

games themselves via self-play, so it is a comparatively small resource drain.

3) Input over-parameterization via attack maps: A history prior consisting of moves other than the one immediately preceding the present position is fairly unhelpful in the majority of tactical problems. However, one technique often used by classical chess engine developers are "attack maps" [14]. These are bitboard-based encodings that highlight squares (or enemy pieces) that a given piece attacks. Introducing predetermined encodings of this nature could have multiple benefits, including: reducing the required depth of the network (so that it no longer needs to dedicate layers towards determining these internally), improving tactical vision, and reducing runtime computational costs (since these encodings can be determined within nanoseconds via well-known bit-masking techniques). These would increase the input size of the architecture, though as Stockfish with NNUE (cited above) has shown, over-parameterized inputs allow for much smaller networks.

References

[1] M. Costalba, J. Kiiski, G. Linscott, T. Romstad, S. Nicolet, S. Geschwentner, and J. VandeVondele, *Stockfish*, version 13, Feb. 19, 2021. [Online]. Available: https://stockfishchess.org/.

[2] Y. Nasu, *NNUE efficiently updatable neural-network based evaluation functions for computer shogi*, Ziosoft Computer Shogi Club, 2018.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. arXiv: 1712.01815 [cs.AI].

[4] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Proceedings of the 17th European Conference on Machine Learning*, ser. ECML'06, Berlin, Germany: Springer-Verlag, 2006, pp. 282–293, ISBN: 354045375X. DOI: 10.1007/11871842_29. [Online]. Available: https://doi.org/10.1007/11871842_29.

[5] 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. 484–503, 2016. [Online]. Available: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[6] G.-C. Pascutto and G. Linscott, *Leela Chess Zero*, Mar. 8, 2019. [Online]. Available: http://lczero.org/.

[7] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, *Squeeze-and-excitation networks*, 2019. arXiv: 1709.01507 [cs.CV].

[8] (). "Missing shallow tactics," [Online]. Available: https://lczero.org/dev/wiki/missing-shallow-tactics/ (visited on 04/04/2021).

[9] (). "Missing mates," [Online]. Available: https://groups.google.com/g/lczero/c/NDv4xtE3W7k (visited on 04/04/2021).

[10] (). "Lichess puzzles open database," [Online]. Available: https://database.lichess.org/ (visited on 03/01/2021).

[11] (). "Neural network topology," [Online]. Available: https://lczero.org/dev/backend/nn/ (visited on 04/08/2021).

[12] D. Foster. (). "AlphaGo Zero explained in one diagram," [Online]. Available: https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0/ (visited on 04/10/2021).

[13] P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. G. Wilson, *Averaging weights leads to wider optima and better generalization*, 2019. arXiv: 1803.05407 [cs.LG].

[14] (). "Attack and defend maps," [Online]. Available: https://www.chessprogramming.org/Attack_and_Defend_Maps (visited on 04/16/2021).

APPENDIX

A. Installation

The code was tested on Python 3.9, but it should work with Python 3.6+. Please install the following packages:

```
pip install pytorch
pip install pytorch-lightning
```

Put both `project.zip` and `model.zip` into the same folder. Then open a Linux or bash terminal and run:

```
unzip project.zip
unzip model.zip
mv model/model.ckpt project/
mv model/data.csv project/
cd project/
```

B. Running the pre-trained model

If you want to manually run the model on a sample test batch:

```
# example.py

from common import setup
import torch.nn.functional as F

args, model, data_module = setup()
test_data_loader = data_module.test_dataloader()

for batch in iter(test_data_loader):
    inputs, targets = batch
    logits = model(inputs).view(-1, 8 * 8)
    y = F.softmax(logits, dim=1)

    preds = y.argmax(dim=1)
    top3 = logits.topk(k=3, dim=1).indices.t()
    top1_acc = (preds == targets).sum().item() / len(targets)
    top3_acc = (top3 == targets).sum().item() / len(targets)

    print("labels:      {}".format(targets))
    print("predictions: {}".format(preds))
    print("top-1 acc: {:.3f}".format(top1_acc))
    print("top-3 acc: {:.3f}".format(top3_acc))

    break
```

If you save this to a file called `example.py`, then you can run it with the **correct model checkpoint** by doing:

```
python example.py --resume_from_checkpoint=model.ckpt
```

C. Training

Run the following. The output model is in `lightning_logs/version_0/checkpoints`.

```
python train.py
```

D. Testing

Run the following:

```
python test.py --resume_from_checkpoint=model.ckpt --stochastic_weight_avg=False
```