Alpha Zero in Connect Four

Alexander Meyerowitz and John Zhou

Abstract

AlphaZero is a state-of-the-art DL/RL algorithm developed by the DeepMind team, capable of playing Go at superhuman levels. In this project, we implement a much smaller version of AlphaZero while preserving its basic structure, including MCTS and policy-value network. Our implementation achieved near optimal play against a random policy baseline in Connect Two, and 75% winrate in Connect 4. However, the model is limited by available compute and particularly the speed of MCTS, which slows training data generation and limits model complexity.

1 Introduction

Game-playing algorithms have long served as a benchmark for progress in computer science. The first program capable of defeating a human chess player was created in 1956. It utilized hard-coded heuristics to evaluate non-terminal board states to determine which move to make. That basic idea persisted for decades; except for simple games where every state could be exhaustively searched, the best performing algorithms usually made use of hand-crafted heuristics. However, with the recent progress in deep learning and neural networks, these algorithms have become increasingly self-sufficient, and are in large part able to train themselves. MDeepmind's Alpha-series game algorithms are at the forefront of this shift. Their AlphaZero in particular proved capable of matching or exceeding the performance of other state-of-the-art Go and chess algorithms with practically no human guidance, requiring only the rules of the game to train and a neural net that matched the game state space.

The core logic of AlphaZero revolves around Monte Carlo Tree Search (MCTS) and a neural network (NN). Running MCTS on a state will return both probabilities corresponding to each move and the expected value of the state. Running the NN on a state will return the same things as MCTS[1].

Because MCTS can evaluate games to their terminal state and use this to inform its move visitations (equivalently the move probability) and its expected value, it generates better probabilities and values than the raw NN. Furthermore, since MCTS uses the NN as its policy during rollout, as the NN improves

the MCTS will also get better, since it has a 'smarter' agent playing out the test games. This means that the output of MCTS, stored with the game state, can be used to train the NN, which will learn to approximate the MCTS and so improve. Ideally, this results in a highly accurate MCTS and NN, either of which can be used during testing - the MCTS will be slower but more accurate than the NN alone.

In our implementation, we focus on the Connect Four game (and a derivative Connect Two for testing). Connect Four is more suitable for us given its dramatically smaller action space, with a max action space size of 7 to Go's 361. This allowed us to scale down model complexity and the training set to a size feasible for us to generate with our compute while maintaining model performance. Still, we discovered that without some further optimization, even this proved too complex to train a model to human level.

2 Methods

Game Representation. The Connect Four board was represented as an 6x7 Numpy array, with empty tiles set to 0, player 1 tiles set to 1, and player 2 tiles set to -1. The game terminated when 4 of the same pieces were set in column, row, or diagonal, and the reward on game completion was 1 for the winning player, -1 for the losing player, and 0 for both players in case of a tie.

The Connect Two board was represented as a 1x4 Numpy array, where two of the same tiles would result in a victory. In all other ways it was the same as the Connect Four game.

Monte Carlo Tree Search. Each NN is trained to generate a probability distribution over the action space and an expected reward for the current state. Moves with high probability correlate to decisions within the game that lead to positive rewards for the player. In order to get training data to learn this task, we employ MCTS, an iterative tree-building method that uses simulated games to approximate the expected value for choosing certain moves.

Each node in the tree contains a state of the board, a counter for the number of times MCTS has visited it, the total value that has been passed up through the node, and a prior which is the probability the NN model assigned the action that resulted in this state. The root of the tree is initialized to the starting state, with zeros for the total value and visitations, a one for the prior, and no children. Fully generating the tree consists of four sub-processes: tree traversal, rollout, backup, and node expansion.

On each iteration of MCTS, we must traverse down from the root of the tree and find a leaf, a node with no children. To choose between the children of a node we compare their UCB1 scores, and go the one that maximizes this score. To calculate the UCB1 score of a node, S, we use the following equation:

$$\text{UCB1}(S) = \overline{v} + p \cdot C\sqrt{\frac{\log N}{n}}$$

where $\overline{v}$ is the average reward at this state, p is the prior, C is a constant to balance exploration against exploitation, N is the number of visitations to the parent node, and n is the number of visitations to the current node. Note that when $n = 0$, $\text{UCB1}(S) = \infty$. This means we will always preferentially go to unvisited nodes.

Once we find a leaf via tree traversal, we initiate rollout if it has never been visited before. This entails simulating a game from the leaf node's state (using the NN to make move decisions) and returning the reward once this simulation terminates. We adjust the reward to be from the perspective of the node's player, then add it to the total value stored at the leaf node. We also increment the visitation count of the node by one.

Once we have added the resulting value of rollout to the leaf node, we backup the value. From the current node we recursively pass the value up the tree, adding it to the total values stored at each node, and increment the visitation count. Critically, on each recursive call we *negate* the value. This is because a good reward for the current player, is a bad reward for the opposing player.

If at the end of tree traversal the leaf node has been visited before, we expand it. For each of the possible actions in the leaf node's state, we initialize a child node where that action has been taken. We use the NN to generate a probability distribution over these actions and use this to initialize the priors of the child nodes. After expansion, we traverse to the first child node and run rollout and backup from it.

After we have run MCTS on the desired number of iterations, we are able to generate the probability distribution over the action space and the expected reward. Our probability distribution is the normalized visitation counts for each of the root's child nodes. This is counter-intuitive since it does not take into account the average value at these states, but because visitation is determined by UCB1 scores (which takes into account these values) it indirectly accounts for them. To determine the expected reward we take the dot product of the actions probability distribution and the average rewards at the child nodes.

Neural Network IO. Each NN took in an observation of the game state, which transforms the board so that it is as if the current player is player 1. This is necessary because player 1 and 2 have opposite action policies. Because the game is perfectly symmetrical, it is more convenient to modify the input data than learn both of them.

Each NN produced a (probabilities, value) tuple for each state, where the shape of probabilities matches the shape of the action space. The value is a single float.

Neural Network Architecture.

1. Custom Connect Four Model

   The probability and value generators of the NN both had the following layers:

Two convolution layers with 42 and 84 channels respectively, kernel size of (3, 3) and stride of (1, 1), each followed by a ReLu.
A (504, 64) linear layer followed by a ReLu
The probability layer had a final (64, 7) linear layer with softmax.
The value layer had a final (64, 1) linear layer with tanh.

2. AlphaGo Zero architecture as described on the 2nd page of Methods[2]
   We only used 6 residual blocks, and 64 channels instead of 256, and had IO matching Connect Four
3. AlphaGo Zero architecture but with no residual layers.
4. Custom Connect Two Model
   (4, 16) linear layers for both probabilities and values, followed by ReLu, a linear layer to convert probabilities to size 4 and values to size 1, and softmax/tanh respectively.

Training Loop. Generation of training data is based on episodes, where each episode results in one game played to completion, and each turn has an associated (board, probabilities, value). The probabilities are generated via MCTS on each turn, and sampling from this allows a move to the next turn. The values are generated at game end based on the winning player, and assigned to each board based on the player about to make a move in each state. After playing some number of episodes, the data is shuffled and used to train a copy of the NN. If the NN performs better than the old one in a direct competition, the NN is updated to the new one. Any future MCTS calls then make use of this new model, and the loop repeats. We were able to create 10 episodes per loop run; this happened slowly and was the main factor limiting our model accuracy.

The NN loss was calculated via mean-squared error between MCTS and NN value and cross-entropy loss between MCTS and NN probabilities:

$$(p_{\text{NN}}, v_{\text{NN}}) = \text{NN}(s)$$
$$(p_{\text{MCTS}}, v_{\text{MCTS}}) = \text{MCTS}(\text{NN}, s)$$
$$l = (v_{\text{NN}} - v_{\text{MCTS}})^2 - p_{\text{MCTS}} \cdot log(p_{\text{NN}}) + c\|\text{NN}\|$$

where p is action probabilities, v is value, and $c\|\text{NN}\|$ is L2 weight regularization.

After each episode generation and subsequent model training, we test the model against a random opponent to determine improvement. The models are in general not very well suited for dealing with random opponents, since the episodes are generated via MCTS and so fail to cover the board states the random model tends to create.

3 Results

Fig. 1-4 are across 200 game samples, where each player moves first 100 times. Net score is calculated as wins-losses.

Fig. 1 shows how the implementation behaves with the very small action space of Connect 2. Note that with a random opponent in Connect 2, if the

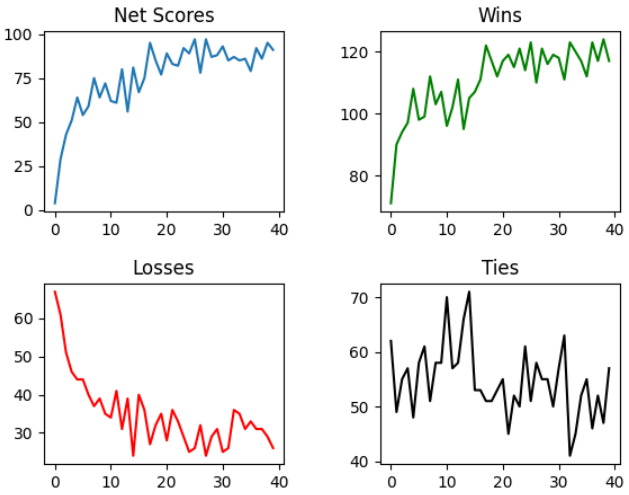

Fig. 1 These graphs show the record our custom Connect Two model had against a random agent following each training cycle.

opponent moves first an unavoidable loss occurs 25% of the time, and since the random player moves first 100 times there is a baseline loss count of 25, which the model approaches. It appears the model is converging to the ideal player, which is trivial in Connect 2 but demonstrates the viability of the algorithm.

Fig. 2 shows our algorithm on the full Connect Four board with our custom NN. Net score plateaus at around 100, with the algorithm winning 3/4 of the games. Ties are negligible due to the nature of Connect Four with a random model. Ideal winrate would be close to 100%. The lower actual winrate is likely due to several factors. First, with only 2 convolution layers and 2 linear layers, the model may not be complex enough to capture ideal strategies in Connect Four, i.e. approximation error. Furthermore, since the MCTS is limited to 50 rollouts and each training loop run only generates 10 episodes, the samples have probabilities and values which vary considerably from the ideal, and the NN may get stuck overfitting to these flawed samples in each run of the training loop. These issues may be exacerbated by the nature of the model training and selection process, which assumes an opponent identical to the model. MCTS is likely to focus on explorations on actions which result in victory, so states where one player acts randomly are underexplored relative to other states. Therefore, the model must apply patterns from totally different game states to the states created by the random model.

Fig. 3 shows our algorithm using the AlphaGo Zero inspired NN, which is much deeper than our custom model (it also uses batch normalization and skip layers, which the custom model does not). This depth lets it learn much more complex patterns. Unfortunately, Fig. 3 shows this effect is dominated by the overfitting problem. A larger model means longer MCTS runtimes and episode

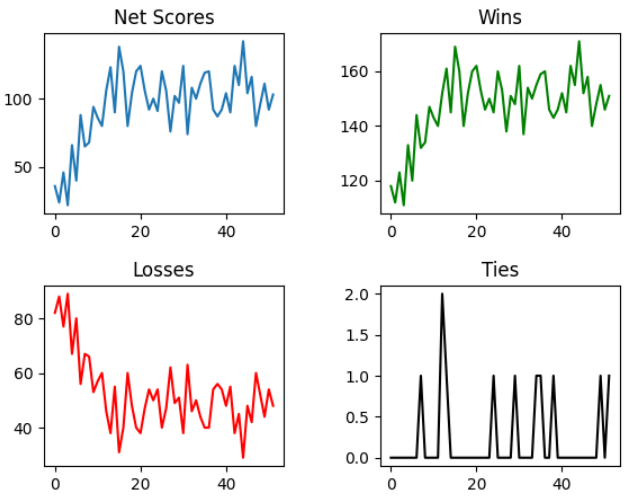

Fig. 2 These graphs show the record our custom Connect Four model had against a random agent following each training cycle

generations. This model was trained on the same 10 episode training samples, further augmented by saving training samples from previous iterations, but only managed a net score of around 50 with huge variations. The additional model complexity made it easier to find a model that beat the old one via spurious actions which did not translate well to beating the random model.

Fig. 4 shows our algorithm using the AlphaGo Zero inspired NN but without the residual layers. Depth is similar to our custom model, but it utilizes value and probability heads instead of having separate models for value and probability. This converged to around the same performance as our custom model, supporting the idea that the full AlphaGo Zero NN is too complex to be trainined in reasonable time from the data we can generate.

4 Discussion

AlphaZero and its relatives represent an extremely powerful new class of algorithms, and is part of a paradigm shift in AI towards full self-learning. They are designed to scale to fit the compute capabilities of large server clusters, and with these resources are able to solve problems traditionally considered unapproachable by AI.

Still, these algorithms have their limitations; namely, while they can be trained to superhuman levels with sufficient compute, they need a large amount of training to compete even with simple heuristics. For instance, even our best Connect Four model regularly fails to place winning pieces; running MCTS to decide which move to take alleviates this particular problem, but in general the model has not learnt these heuristics very strongly.

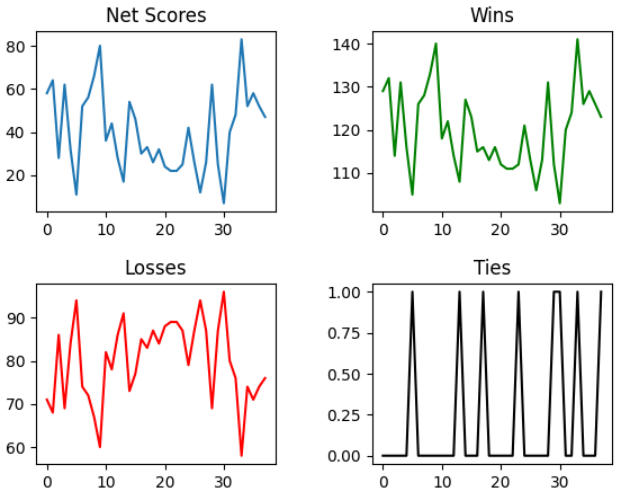

Fig. 3 These graphs show the record our AlphaGo Zero based Connect Four model had against a random agent following each training cycle

Our attempts at fixing this involved increasing model complexity, increasing the rollouts in MCTS, and increasing the episodes per training. Increasing model complexity largely failed; as shown in the results, the volume of data we could generate with MCTS was insufficient to train models with complexities on par with AlphaGo Zero. For the same reason increasing rollouts in MCTS, which it would improve probability and value estimation, was ultimately somewhat self-defeating since longer episode generation times would lower the data available for training. Increasing the number of episodes per training session had mixed success. We could not simply increase episode generation, since we were bottlenecked by that in the first place. As a result, we tried storing data until we found a better model, and storing episodes within a buffer of fixed size. Storing data until a better model was found has the problem that the resulting model will be as overfit to the data as it can be while still beating the old model, so this was eventually replaced with storing data within a buffer of fixed size. Once max size was reached, the buffer would remove the oldest episodes. This strategy only works if the model isn't changing very much after each training session relative to the size of the buffer. If the model changes quickly, the data in the buffer will still be somewhat useful insofar as it is generated by MCTS, which gets decent probabilities and values even with no model. However, there is no NN improving MCTS improving NN feedback loop as discussed in the introduction, because the NN is learning on probability and value estimates generated by MCTS with a worse model. Increasing buffer size made this problem worse but helped counter overfitting; we settled on a buffer size of 512, which constitutes roughly 5 episode generation cycles.

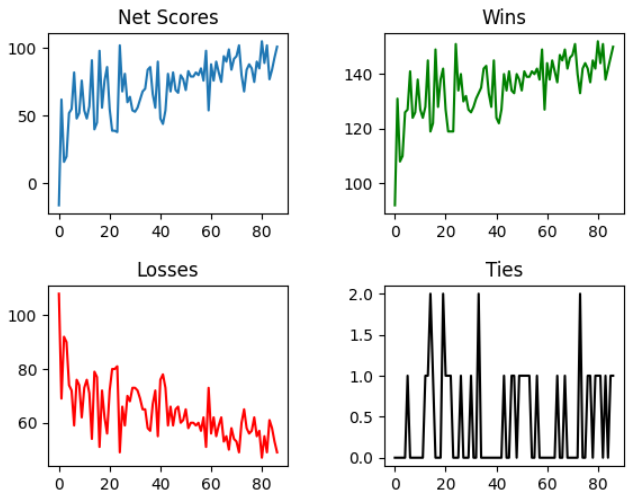

Fig. 4 These graphs show the record our AlphaGo Zero based Connect Four model without residual layers had against a random agent following each training cycle

The chief way this project can be improved is through optimization of the MCTS algorithm. Running MCTS took up most of the runtime when generating an episode. In AlphaZero, researchers were able to parallelize tree exploration in MCTS through batching NN calls made during MCTS and by preventing evaluations of the same node with virtual loss. Either of these strategies would be a significant improvement over our technique, which calls the NN seperately for each node, resulting in tensors needing to be regularly shifted from gpu to cpu on top of needing to evaluate each board separately. An optimized MCTS means faster episode generation, higher quality data, this means that more complex models can be trained.

Of course, this project could also have been improved with more raw compute. These models were trained on low-end desktop hardware, so utilization of larger clusters for training would also help with training better models. Still, MCTS would first need to be changed to be parallelizable to make use of the increased compute resources.

5 Contributions

John worked on the Connect 4 neural net architecture, training loop/episode generation/model comparison, and the Connect Four environment.

Alexander worked on Monte Carlo Tree Search, rendering of the Connect Four environment, graphing of results, and the script that lets you play against the pre-trained models.

References

[1] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2018). A general reinforcement learning algorithm that Masters Chess, Shogi, and go through self-play. Science, 362(6419), 1140–1144. https://doi.org/10.1126/science.aar6404

[2] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of go without human knowledge. Nature, 550(7676), 354–359. https://doi.org/10.1038/nature24270