AlphaZero with Connect4

Tor Saxberg

September 2019

1 Definition

1.1 Project overview

In 2016, Google DeepMind did the unthinkable: they defeated Lee Sedol, the reigning champion of the ancient board game, Go, decades before people would have believed it possible. while Go itself may seem like a somewhat insignificant achievement given all the other happenings lately, it was a huge step for machine learning, opening doors to all kinds of brand new techniques that could solve all kings of problems once thought impossible.

But what was even more amazing was Deepmind's next generation, Alphazero, easily defeating Alphago. whereas Alphago required substantiel work building datasets from historic records of professional games of Go, and a lot of training time and hardware, Alphazero required no datasets at all, and surpassed human players in less than a day. the most remarkable part of Alphazero is it's ability to matering any game from only the basic rules, 'Tabula Rasa;' it has also easily defeated the famous reigning Stockfish at chess.

1.2 Problem Statement

in this Capston project, I'm giong to show how DeepMind's algorithms work with a much simpler game, Connect4. Since it has a much smaller action space than Go, with only 7 possible moves in each state, it own't require Google's expensive hardware to train. still, the number of possible states grows exponentially with each turn, so it will make for a good demonstration of Alphazero's potential.

1.3 Metrics

I will be measuring success on two metrics: win rate against a random bot, and win rate against the previous agent (before the last training epoch). I expect that as the agent trains, it will perform better against an opponent playing randomly, and this will show readily on a win-rate vs epoch graph. once learning levels off, we should see a horizontal line somewhere above 70%, showing that the agent has stopped improving, but did learn to play the game.

similarly, we will see improvement from learning as the agent learns to beat it's previous version. each time it's win rate exceeds 55%, the opponent will be updated with the newest network weights, so we will see a cyclical graph between 50% and 55% win rates. once learning levels off, the graph will level out somewhere between those rates.

2 Analysis

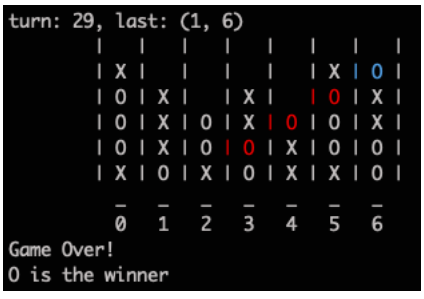
2.1 Data Exploration

the beauty of AlphaZero is that it does not rely on human-generated training examples, like AlphaGo did with data on historic games between Go masters. instead, it generates all of it's examples through self-play by first simulating hundreds of moves through a Monte Carlo Tree Search (MCTS) algorithm (discussed later), and then picking the best move according to an optimization function discussed later. each move invokes it's own MCTS, until the game is over. we use the chosen moves, the winning player, and an updated policy (discussed later) to train a nueral network to improve its game.

In a game like Go, where the action space is very large, MCTS can be very slow, so the DeepMind team at Google used advanced hardware, multiprocessing, and algorithmic adjustments to speed up the process. since Conenct4 is a much smaller game, these modifications won't be necessary, except for one: while MCTS usually involves a "rollout" phase where the game is simulated until completion, our algorithm will only simulate moves until a new state is reached, then it backpropogates the results to update the tree (discussed later). this will allow us to generate enough training examples for our agent to learn meaningful strategies.

2.2 Exploratory Visualization

the game of connect4 is simple. players drop pieces from the top row which land on top of another piece or on the bottom row. the first player to get 4-in-a-row diagonally, vertically, or horizontally wins. if the board fills up with neither player winning, it's a draw. an example of a finished game is shown below, wher each board state that player 'O' played last would get a value of 1, and each turn player 'X' played would get -1.

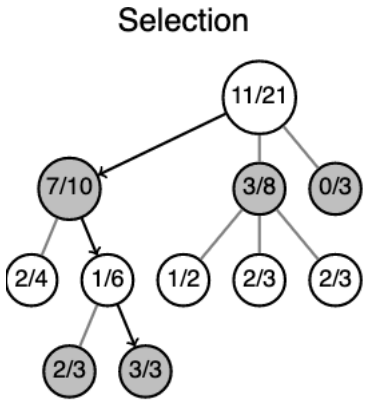


centering

2.3 Algorithms and Techniques

the MCTS algorithm is the backbone of AlphaZero's approach, since it produces as many training examples as the neural network needs to learn.

MCTS has three main components: finding a a leaf node, selecting the next child, and backpropogation to update the number of times each node has been selected $N(s, a)$ and the improved policy. the search tree is remembered throughout the game, so with each new move and its corresponding MCTS, the nodes higher up the tree are updated and become more accurate.

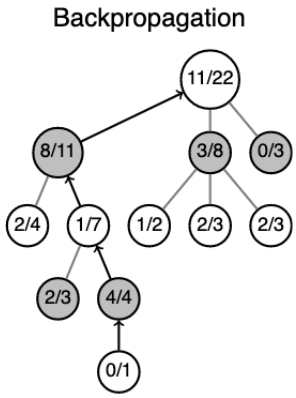


child nodes are selected according to the upper confidence bound

$$U(s,a) = Q(s,a) + c \cdot p(s,a) \cdot \sqrt{\frac{\sum_b N(s,b)}{(1+N(s,a))}}$$

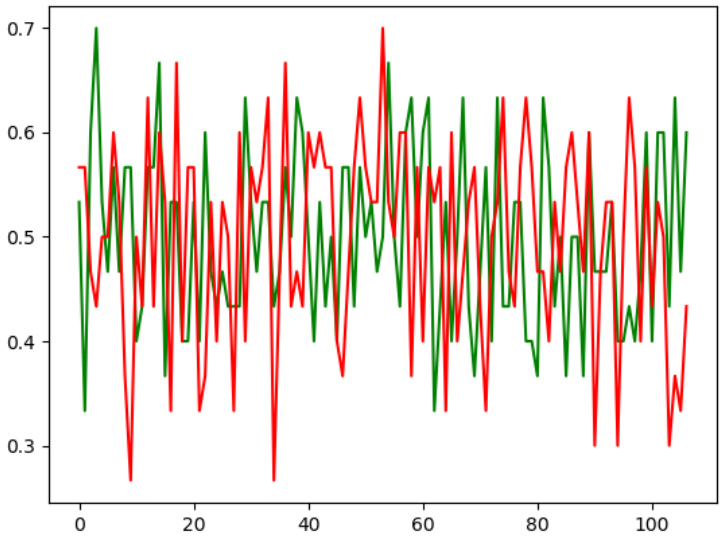
where $Q(s, a)$ is the predicted value [-1,1] from the network for state s and action a, $p(s, a)$ is the probability of selecting action a in state s (from the network), and $N(s, a)$ is the number of times action a has been chosen from state s. this equation balances the predicted value of an action against the reliability of that prediction (how many times this node has been chosen) according to some constant c, which was set to $\frac{1}{winrate}$ against the previous epoch agent.

when a new new state is discovered, it's added to the tree with $N(s, a)$ and $Q(s, a)$ initialized to 0. then it backpropogates a value either from the network if the game hasn't ended, or from 0,1 if it has. each node updates the number of times it's been selected, and computes an improved policy, $\pi(s) = \frac{N(s,:)^{1/t}}{\sum_b N(s,b)^{1/t}}$, where $t = \frac{N(s,a)-turns}{N(s,a)}$ is a 'temperature' variable to ignore predicted values for later states that are unlikely to have updated values. each parent node also updates its value $Q(s, a) = \frac{\sum_b Q(s,:)}{\sum_b N(s,:)}$ to improve future searches for the game.



2.4 Benchmark

I will be measuring success on two metrics: win rate against a random bot, and win rate against an agent using the previous network epoch. below is a graph for an untrained agent (not updating network weights). the red line displays the win rate against an agent playing randomly, while the green displays the win rate against an agent using the previous network weights. with a working network, the red line should move upwards, while the green should cycle between 50% and 55%



3 Methodology

3.1 Implementation

the neural network will take as input a batch of examples of the form (state, value, policy), where each batch contains one example for each board state in a game. some games will be longer than others, but this will have negligible effect on learning performance.

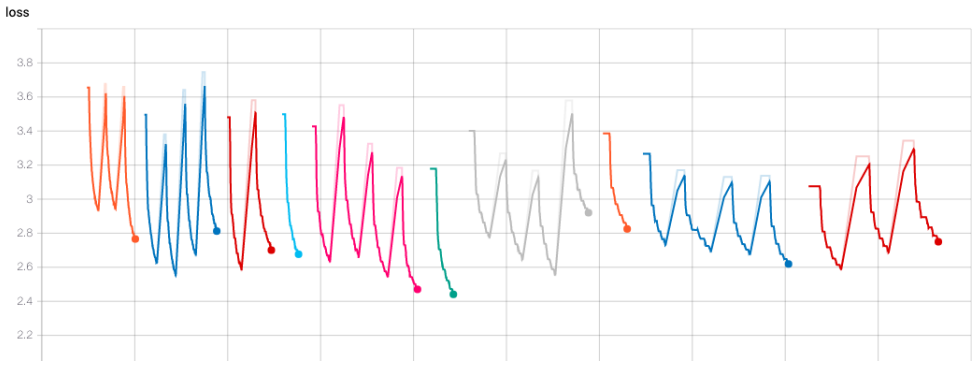
the state is just the 6x7 board of Connect4, with 0 for 'X', 1 for 'O' and '2' for empty. the value is 1 if that player ends up winning, 0 if it ends in a tie, and -1 if it loses.

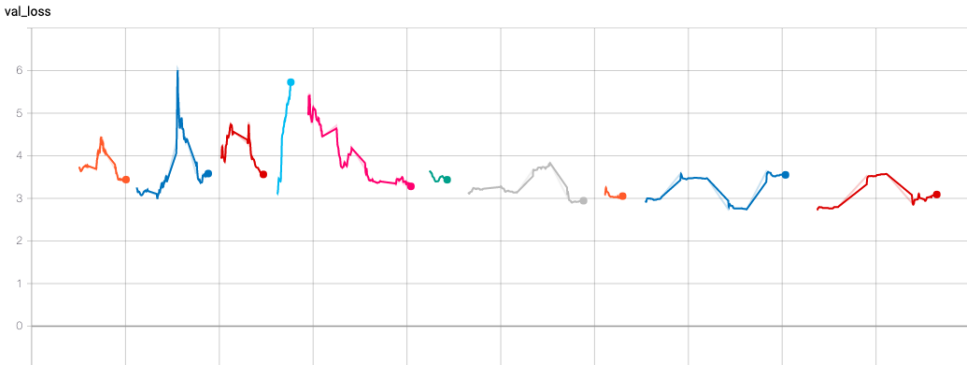
the value is filled out after the game has ended, but assigning a 1 (or 0) to the last board state (since the last player either won or drew), a -1 for the state before that, then 1, etc. until all board states of that game have a value.

the policy used to play out the MCTS and decide which move to make comes from the neural network's prediction, but the policy used to train the network comes from an improved policy that is simply based on the frequency of visits to each node (since nodes that lead to a victory will be chosen more often in MCTS), so it becomes a better approximation of an optimal policy than the network's initial predictions.

3.2 Refinement

originally a CNN network model was used, but since CNN's are more appropriate for large matrices like images and not so much for smaller matrices like the connect4 board, it was not able to reduce the validation loss across epochs. the below graphs show the loss and validation loss for each epoch; the jaggedness comes from training each epoch on different training sets until it beats it's predecessor 55% of the time.

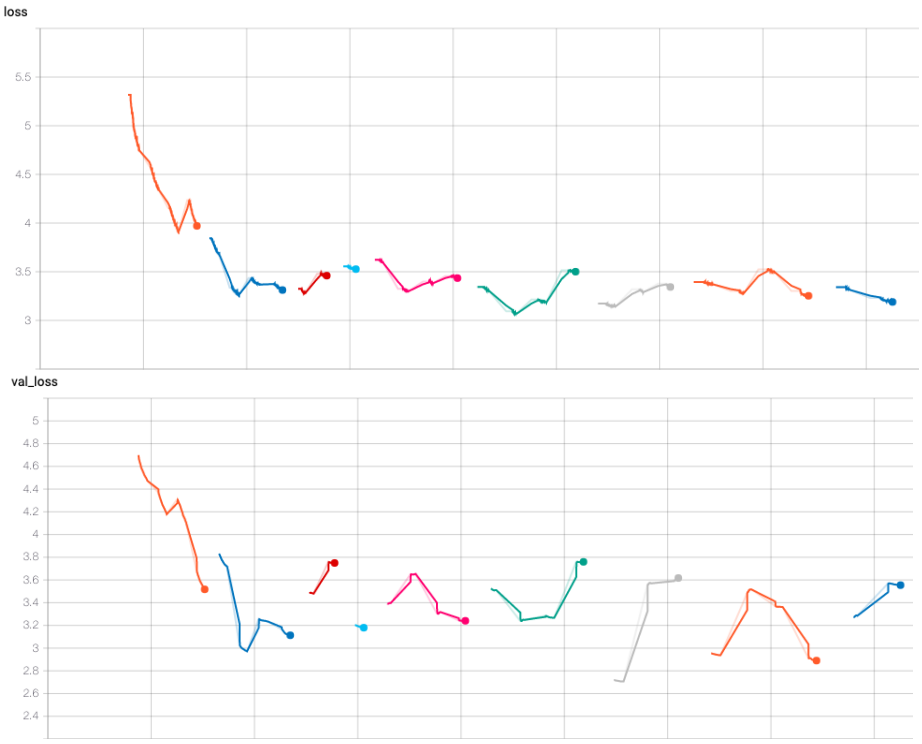




to address this issue, the network was simplified to a simple feed-forward Densely connected network. the number of dense layers was limited to $board \times action$ to avoid over-fitting.

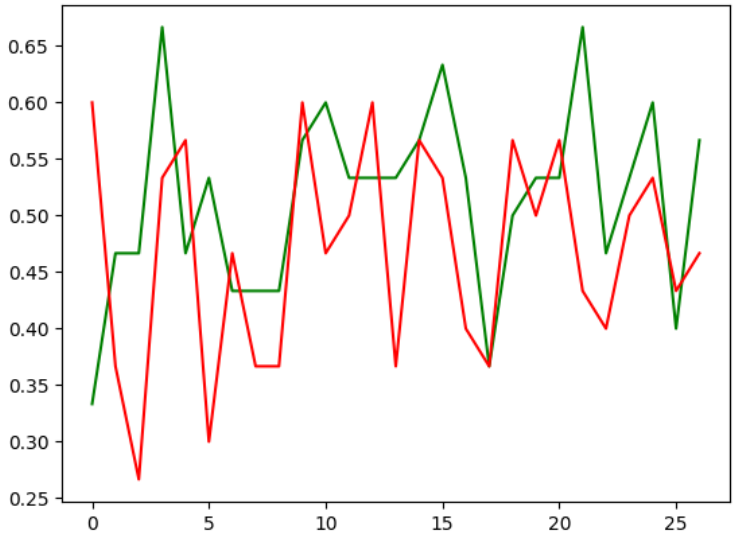
4 Results

4.1 Model Evaluation and Validation



the loss graph above shows that the agent made progress for the first two epochs, but learning leveled off quickly. the validation loss graph shows that learning from one epoch did transfer to another; the jaggedness of some epochs come form training on a new set of boards with an emptied tree, since the next epoch only starts when the new model beats the old 55% of the time.

4.2 Justification



the graph above shows that the agent did not make much progress in learning to play the game, and certainly did not approach human-level playing. the red line displays the win rate against an agent playing randomly, and the blue line displays the win rate against the previous epoch. we can see that the agent does not learn because the red line remains near 50%.

5 Conclusion

5.1 Reflection

DeepMind's AlphaZero was an incredible leap forward for machine learning, demonstrating that some problems can be solved much more effectively without human-provided datasets. in this project, I recreated AlphaZero for the much smaller game of Conenct4, reducing the training time required and simplifying the model to a feedfoward dense network.

Monte Carlo Tree Search was used to pick the best moves according to the network and to provide an improved policy for training. the real value of the

board -1,0,1 was assigned after each game, and a set of (board, value, policy) was used to train the network.

the agent was not able to perform at human level, but graphs of the loss and validation loss show some improvement early on but leveling off quickly. this result suggests another network model would be more suitable, but I was not able to find a model that performed better.

5.2 Improvement

since the network appears to stop learning too early, an improvement would involve testing out other network models. DeepMind used a deep RNN with convolution to train and make predictions on the Go board. since connect4 uses a much smaller board, it should not require as complex a network, but it appears still to require one more complex than mine.

since several games are played using the same network weights, parallel processing may significantly reduce the time spent generating training examples.

I've made the program work for any 2D game, so it may be interesting to test connect5, connect6, etc. to see how different models perform on different board sizes.

6 Sources

the complete source code for this project can be found at:
https://github.com/Tor-Saxberg/Alpha-Zero