CSC 486B Final Project Report
Alpha-Go Zero

Matthew Paletta, Eric Hedlin, Jarred Hawkins
University of Victoria
Spring 2018

Abstract

We chose to implement AlphaGo Zero. AlphaGo Zero is a program created by a team of researchers at DeepMind which is capable of mastering the game of Go without human knowledge [8]. We based our implementation off of an existing implementation of the original AlphaGo paper, Alpha Zero General [1]. Our implementation provided a few additional improvements. We deployed a scaled down version compared to the AlphaGo Paper on the Compute-Canada HPC network. We showed our networks beginning to self-learn over 10 runs of self play and retraining. We compared 5 different architectures shown in the AlphaGo paper.

1. Introduction

In this project we looked to gain insight into how deep learning projects are structured, how to read and interpret an academic paper, and how to recreate the results gathered from the paper. This would not only involve reading and understanding an academic paper, but working on neural networks and a high performance computing cluster.

We chose a difficult project because of the relevance and significance of the contributions made by the paper. An opportunity to learn more about cutting edge research during undergrad is very exciting.

Our end goal for this project was to have a fully functioning version of AlphaGo Zero up and running, along with some minor tweaks and improvements for accessibility.

2. Contributions of the original paper

AlphaGo Zero, and its predecessor AlphaGo were both groundbreaking papers. Go has remained a challenge for AI to play in due to the size of the search space required to know how to learn it. The search space consists of 3^{361} possible moves (although not all of them are legal)[5]. Exploring a search space this large to find the optimal move is not computationally feasible. The lack of ability to brute force the search space means an AI or other less expensive method must be used to predict the next best move for an AI to take.

AlphaGo solved this problem in October of 2016 at a superhuman level. By training a deep neural network on the results of professional Go matches, the neural network was able to defeat top ranked professional player Lee Sedol 4-1 in a tournament [2]. This far surpassed what had previous efforts to teach AI's how to play Go had accomplished.

One year later AlphaGo Zero was published. This version of the algorithm did not learn from previous professional matches, but instead learned from knowing nothing about Go except for the rules of the game. It performed iterations of self play and retraining, and surpassed what had been achieved in previous versions of the algorithm.

Not only did AlphaGo Zero learn the rules of the game itself, it managed to learn a fundamental understanding of different playing styles of Go, as well as discovering new moves. In the paper [8], extended data, figure 2, it is shown how the algorithm prefers certain moves at different points of time. There is also an examination into how it taught itself to perform new actions that human players have not yet discovered. As opposed to learning from previous masters, AlphaGo Zero taught itself how to play Go, discovering new playing styles as well as surpassing the previous versions in less time than before.

2.1. Monte Carlo Tree Search Algorithm

There are more board configurations of Go that there are atoms in the universe [5] [3]. As a result evaluating all of these positions would be unfeasible. The Monte Carlo Tree Search Algorithm (MCTS) is a method for lowering the amount of board positions that need to be evaluated. MCTS takes a very human approach to evaluating board positions; it only considers board positions based on a combination of how good the move is and how likely it is to be chosen. The basic search part of the algorithm is pretty straight forward. MCTS uses selection, expansion, simulation, and backpropagation. [7]

A leaf is selected by traversing through the tree and at

each iteration the child I with the highest score for U for the function below is selected.

$$U = Q + C * \frac{\sqrt{N_P}}{1 + N_I}$$

where Q is the value of the board position, C is an exploration constant, and N_P and N_I are the number of times the parent and current nodes have been visited respectively.

Once at the leaf, expansion is performed. This adds all possible children (possible next moves) from the current board state to the tree and for each the value of U is calculated. In classic cases, the value of Q is found through simulation. This involves rollout in which random moves are played from the current board position until a terminal state is reached. The value for Q reflects whether or not rollout in this case lead to a win or loss. Backpropagation is the process of recursively passing this value to the node's parents and updating their value of Q until the root is reached. The more this is carried out the more accurate the values of Q becomes.

This method, however, is very computationally inefficient and requires rollout to be performed many times to get an accurate estimate for Q. AlphaGo Zero replaced rollout with a neural network that estimates the value of Q. Using this technique they were also able to better estimate the probability that a move would be chosen at each step, adding P to the equation. [1]

$$U = Q + C * P * \frac{\sqrt{N_P}}{1 + N_I}$$

The networks determining the values of Q and P are more commonly known as the value and policy networks respectively. When it comes time to make a move, the child of the current board state with the highest visit count gets chosen. Although the longer you look the better your search gets, MCTS can be performed any arbitrary amount of time and the move returned at any point represents the best next move found so far.

2.2. Policy and Value Networks

The neural network trained for making game predictions was an improvement over previous versions. It had a long trunk network, which then branched into two heads. These two heads were called the policy head and value head.

The purpose of the policy head is to estimate which move to take at a given point will give the best odds of winning. It serves itself as a 1-1 representation of the game board, but with probabilities at each tile. In the Monte Carlo search, the output of the policy network is the prior probabilities, P. The value head estimates the likelihood that the current player at the current board state will win the game. It returns a scalar value.

Using the estimates given by these network heads, one can speed up the MCST substantially. Instead of performing a full rollout, one can simply get the return values from these networks as the next choice. As the trained network improves, the accuracy of these predictions improves. In the AlphaGo Zero paper towards the end of training, at a given point the network could predict the next move a professional would play with nearly 50% accuracy as well as predicting the winner of the match with a mean squared error of around 18%.

3. Contributions of this project

Due to the difficulty of implementing this system from scratch, we chose to base ours off of an existing implementation [1]. This implementation allowed for generic perfect information games to be trained as well. It was not completed to the full details of the paper, so this is where our work came in. The next section of this report details our contributions.

Since we were interested only in Go, we de-abstracted the game to only contain the code related to playing Go. From this we examined it for any improvements that could be made, or details missing from the AlphaGo Zero paper in [8].

3.1. ELO

In AlphaGo Zero, they make evaluations of the algorithms performance by comparing it's ELO against known opponents [8]. ELO is a way to compare multiple competitors in zero sum games, or games where improved results in one agent leads to worse results for their opponent [6]. A benefit of the ELO rating is that it can be applied to any zero sum game, such as Go, Chess, or Shogi (a variant of chess). We implemented an ELO system in order to rank how subsequent iterations of our implementation perform relative to each other.

AlphaZero had a specific ELO rating based on known competitors and other AI's. In chess they were able to compare versus programs like Stockfish whose ELO is known. In Go, they competed against the current world best, Lee Sedol, whose ELO is known. For every iteration, they were able to test their ELO versus the version that played Lee Sedol.

For our implementation, our initial ELO level is unknown. It represents a relative score versus other training iterations than compared to the broader world and other competitors. We could standardize all our games if we were able to test our system against someone with a known ELO, however we were unable to find a standardized opponent. Our ELO rating is showing the rate of growth compared to each iteration and is therefore viable heuristic for comparing various architectures.

The equations for finding the elo are as follows. [4]

$$E(1) = \frac{R(1)}{R(1) + R(2)}$$

$$E(2) = \frac{R(2)}{R(1) + R(2)}$$

$$R(1)' = R(1) + K * (S(1) - (E(1)))$$

$$R(2)' = R(2) + K * (S(2) - (E(2)))$$

R refers to the player's rating. E refers to the expected likelihood of that player winning the next game. S is whether the player won or not (respectively 1 or 0). K is a multiplier controlling the amount each players' ELO's change. In our experiments K was set to 32.

3.2. Architectures

The existing implementation we used did not have a neural network architecture comparable to the published ones in AlphaGo. DeepMind tested AlphaGo Zero with 5 different architectures [8]:

- separate 12-convolutional-block policy and value networks
- single 12-convolutional-block network with dual policy and value outputs
- separate 20-residual-block networks
- single 20-residual-block network with dual policy and value outputs
- single 40-residual-block network with dual policy and value outputs

We tested similar architectures but had scaled down versions of each. When a single network was used, the final layer of the network was fully connected to each the policy and value networks. They have been numbered for future reference.

- architecture 0: 4 layer convolutional architecture with 2 fully connected layers, combined policy and value networks
- architecture 1: 4 layer convolutional architecture with 2 fully connected layers, separate policy and value networks
- architecture 2: 2 residual blocks each with 2 convolutional layers within, combined policy and value networks
- architecture 3: 2 residual blocks each with 2 convolutional layers within, separate policy and value networks
- architecture 4: 4 residual blocks each with 2 convolutional layers within, combined policy and value networks

The performance of these architectures is discussed in the results section.

3.3. Dirichlet Noise

Dirichlet noise was a missing contribution in the implementation we used. This was a way of increasing the algorithms exploration from the root node of the Monte Carlo search tree. It added noise to the probabilities of each child node; however a property of this weighted noise is that it will still pick better choices more often, but occasionally will pick what the network thinks is a suboptimal choice.

$$P(s, a) = (1 - \epsilon)p_a + \epsilon\eta$$

where $\eta \sim Dir(0.03)$, $\epsilon = 0.25$, and p_a is the prior probabilities of the children. We implemented this to reassign prior probabilities in the root node of the search tree. As mentioned in the future directions section, we would like to look more into the impacts this makes in the networks ability to learn.

3.4. Deployment Configuration

We deployed our system on ComputeCanada. Due to time and resource constraints, we chose to deploy a scaled down version compared to the original paper. These parameters can be seen in 1. From deploying on ComputeCanada, we learned how using limited HPC resources changes development styles. Working on traditional code, you can make changes and deploy/test in a matter of minutes. Using HPC resources requires submitting jobs to a queue, waiting for the queue, and then waiting for the job to execute. This requires making sure that your code is correct and will execute in time. If any errors cause your program to crash, it will slow your development pipeline significantly. We found this to be an issues with development, especially due to the complexity of our code.

4. Discussion and future directions

In this paper we showed an implementation of AlphaGo Zero that shows initial signs of learning. Due to training time restrictions, we were not able to demonstrate these over long spans of time. In the future we would like to show how the models learn over longer time periods. Now that our training pipeline is working as expected, it is now restricted by access to HPC resources.

In addition to running our models for longer, we would also like to collect more data that comes from it. We are able to show convergences and compare across models, however we were not able to examine how the moves the AI chooses

Hyperparameter	DeepMind	Ours
Temperature Threshold[0]	30	15
Number of Monte-Carlo Tree Simulations per action	1600	100
Dropout rate		30%
Batch Size	2048	16
Learning rate	dynamic	10^{-4}, static
# of training Epochs	$6.35 * 10^6$	10
Self Play Games[0]	$2.9 * 10^7$	400
Required Winning %	55	55
Training examples[1]	500 000	10 000
Board Size	16x16	9x9
# of GPU's[2]	64,1	1,1

Table 1. Our deployment configuration vs the configuration specified by DeepMind in [8]. Ours is considerably scaled down. [0]: Number of iterations before temperature is set to 0. [1]: Total amount of games played over training. [2]: It is not specified how many moves were selected, but they mention they draw from the last 500 000 matches. We simply draw from the last 10 000 examples. [3]: DeepMind used 64 TPUs for training, and 1 for playing.

to take over time change. Seeing the intuition that our model learns over time could provide insight as to how it is training and learning.

From the data we did manage to gather in time, we could a noticeable upwards trends in the ELO ratings across runs. In as little as 10 runs, we were able to achieve as maximum ELO score of 1615 using architecture 2 (2 residual blocks each with 2 convolutional layers within, combined policy and value networks). Comparing the differing architecture results to those shown in the AlphaGo paper ([8], fig 4.a), they also received their best ELO rating with the combined policy and value networks, using residual networks. This is the same result that we obtained. Although we did not run our experiments for long enough to see the ELO rating converge to a point, it is exciting to see that our initial results confirmed what is said in the paper.

While interpreting the paper, there were no weaknesses with the actual papers content. We did find it difficult to gather all of the specific details of implementation from the paper, but they were present. Some details were mentioned briefly in the bottom of the paper for only one sentence so ensuring those were implemented, along with finding the details was challenging. For many of the other details, fortunately there was a wealth of resources available (Due to the significance of this paper). The main issue we ran into while developing is the difficulty in debugging software while we have to wait for HPC resources. Even when running the software on smaller hyperparameters, infrequently occurring bugs did not often happen enough that we could

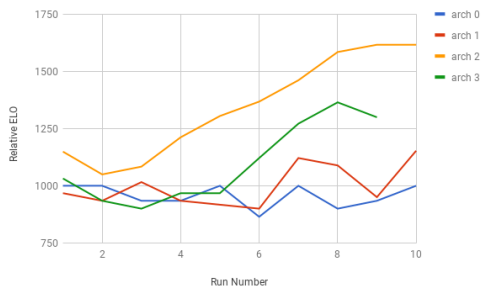

Figure 1. Performance of different architectures over a training run. Arch 3 is missing a data point due to computation job being cut off. Arch 4 is missing due to us underestimating how long the ComputeCanada queue is.

reliably debug them.

Despite not being able to gather as many results as we had hoped, we learned a lot in this project. We learned a lot about how to adjust our workflows for working on deep learning projects, which is much different than traditional programming. We gained experience reading and interpreting results from an academic paper, and attempting to reproduce them.

References

[1] Alpha zero general. https://github.com/suragnair/alpha-zero-general. Accessed: 2018-04-09.

[2] Alphago seals 4-1 victory over go grandmaster lee sedol. https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals-4-1-victory-over-grandmaster-le Accessed: 2018-04-11.

[3] How many atoms are there in the universe. https://www.universetoday.com/36302/atoms-in-the-universe/. Accessed: 2018-04-10.

[4] How to calculate the elo-rating. https://metinmediamath.wordpress.com/2013/11/27/how-to-calculate-the-elo-rating-including-example/. Accessed: 2018-04-13.

[5] Number of legal go positions. https://tromp.github.io/go/legal.html. Accessed: 2018-04-09.

[6] What is an elo rating? https://www.thechesspiece.com/what_is_an_elo_rating.asp. Accessed: 2018-04-11.

[7] G. M.-B. Chaslot, M. H. Winands, H. J. V. D. Herik, and J. W. Uiterwijk. Progressive strategies for monte-carlo tree search. *Nature*, June 2008. Article.

[8] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, 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 with-

out human knowledge. *Nature*, 550:354 EP –, Oct 2017. Article.