Advances in Computer Science and Information Technology (ACSIT)
p-ISSN: 2393-9907; e-ISSN: 2393-9915; Volume 7, Issue 1; January-March, 2020, pp. 1-5

http://www.krishisanskriti.org/Publication.html

Artificial Intelligence or not? *The Best way to use Limited Computing Power*

Pranay Mundhra

Cathedral and John Connon Senior School, Mumbai
Class XII, IBDP

Abstract—*Artificial Intelligence (AI) is widely considered to be the technology that will define future development and growth. Despite organizations such as Deepmind making headlines with their impressively powerful AI systems, the fundamental weakness of artificially intelligent software remains unaddressed, i.e. the vast reams of computing power required to reach such strength.*

Traditional systems can be evolved in an AI-based approach through the use of evolutionary algorithms, which also experience rapid initial growth rates. If artificially intelligent systems lacking adequate computing power are used, they can be reinforced with certain hard-coded algorithms to greatly enhance their performance. When there is incomplete information and nearly infinite choices, AI struggles to make the optimal choice. Testing AI against expert humans in such a space is overwhelmingly difficult since AI invariably has mechanical advantage over a human. Thus, despite wins, AI's decisions may yet be sub-optimal. To extend AI to other tasks, therefore, the evolutionary approach, with its logistic growth, may be a preferred alternative in certain cases.

This paper takes an empirical approach to analyze some typical methods of using software-based methods to accomplish a task, partly through the framework of Chess and StarCraft 2. Based on observations made within this framework, this research attempts to extrapolate the findings on the efficacy of the evolutionary approach and optimization of systems with minimal computing power to other fields, and derive generalizations.

Introduction:

The use of Artificial Intelligence (AI) is growing exponentially of late. Deepmind has defeated top Chess engines, including Go players and StarCraft 2[1] pros with variants of its now-famous "Alphazero" program. However, Deepmind developers have glossed over exactly how much computing power was used to train it. Although information about the computing power used during match play (4 first generation TPUs[2] and 44 CPU cores) is available on the Deepmind blog, it's considerably harder to ascertain the number of TPUs used to train Alphazero, and for what period. The blogs reveal that Deepmind utilized 5,000 first generation TPUs and 64 second generation TPUs to train the neural network part of Alpha Zero for four hours before the match with Stockfish (Silver et.al, 4). Whereas, prima facie, this appears to be an incredibly short period of time, it's important to consider the vast amount of processing power provided by each TPU. Each first generation TPU offered 34 GB/s bandwidth, and each of the 5000 deployed were used to generate games that the neural network could then learn from. If one starts from a tabula rasa,[3]a massive amount of computing power is necessary. Such power, however, is unavailable to most. Therefore, feasibly creating an algorithm that comes close to replicating Alphazero's success is singularly challenging. In recent years, however, an alternative has surfaced. Leela Chess Zero, a new AI similar in operation to Alpha Zero but open sourced, was created by Gary Linscott in 2018 and distributed. Users and enthusiasts download Leela'slatest version and let it play itself as a workable substitute for the 5000 TPUs used by Alphazero. For most others, however, inadequate computing power precludes using the brute force approach, which is why it is necessary instead to increase the effectiveness of the algorithm through other means.

Section 1: The traditional Chess engine

Artificial Intelligence is admittedly revolutionary, but if computing power is limited, traditional engines would actually be equally good, perhaps better. Stockfish, the winner of TCEC 2016 (Top Chess Engine Competition) has continued adding power

[1] Starcraft 2 is a popular online RTS (Real Time Strategy) computer game, that is seen by many as one of the new major challenges for AI controlled players to overcome
[2] TPU or Tensor Processing Unit, is a circuit developed by Google specifically for neural network machine learning.
[3] Literally "blank slate"

through what could be allegorized as a Darwinian process of evolution. Being open source, any programmer can suggest improvements to the code. The "improved" version and the original version play a number of games and, based on the results, the suggestion is either adopted or discarded. This approach could well be adapted to a Chess AI. It is prudent to note that the MCTS[4] portion of a Chess AI would be unaffected by any of these techniques but rather speed up the neural network that predicts the most promising lines.

One way to incorporate Stockfish's improvement system is through an evolutionary algorithm. A simple model for such a model could be as follows:

Each agent is assigned a random weight for pre-decided factors, including, but not limited, to:

- Material (pieces)
- Control of squares, with different values for different squares based on positional importance
- Open files
- Passed pawns
- Outposts for pieces
- King defensibility and attacking potential, based on a separate algorithm
- Any other relevant positional factors such as potential for zugzwang, pawn structure weaknesses, etc. The more the additional factors, the more efficient the algorithm (as demonstrated by the success of Stockfish, which essentially applies the same technique, but in a human-driven mode.)

Note: Based on preliminary testing for this particular type of algorithm, the addition of a separate algorithm for calculating forcing moves and attacking combinations beyond the final depth markedly improves the performance of the engine.

The agents then play against each other in matches comprising several games. Assigned weights, depending on their rate of success or failure, determine the probability for each of their variable values to be passed on to the next generation, and the degree to which one value dominates the other. Of course, there are also a proportion of random values or "genes," created by "mutation." In our experiments, we used a 3% rate of mutation. This method works wellsince it bypasses traditional problems of computing the "fitness" of an individual by merely having agents compete against each other. This is explored in-depth later in this paper.

Experimentally, it was found that algorithms based on the process of genetic selection, as mentioned above, experience something akin to logistic growth, although this was not entirely measurable. To actually view the progress, the game between the two agents with the highest win rates was printed after each generation had completed the internal competition. The algorithm reached the level where it beat an online Chess engine (chess.com's engine was used, with a difficulty level of seven, representing the play level of a strong amateur player) after a hundred generations (the number of generations was hard coded). In comparison, a neural network (without MCTS or any other method of calculation) could only beat the fourth level of the engine after a four-hour training period on a 3.1 gigahertz laptop. An engine based purely on computation could only beat level five. When the evolutionary algorithm was used with 150 generations, it could just defeat level seven of the computer.

Note: This was not a true evolutionary algorithm but one that was intended to simulate the way that Stockfish came to be.
Nevertheless, making an algorithm this powerful necessitates an important attribute: the ability to calculate moves to great depths, as discussed earlier. The question then is: How to maximize the strength of an engine with finite/constrained computing power? Of course, the specific technique will vary based on the maximum depth to which the engine calculates moves. The next section, therefore, discusses something similar to an edge case where there is very limited computing power.

Section 2: Minimalisation using statistics

Note: This section is purely academic as the level of computing power used is so low that the engine would lose to any reasonably competent human. This does, however, yield insight into some behaviors of preset programs, and their subsequent evolution as computing power is increased.

Stockfish, when played on a laptop, can convincingly defeat accomplished human players. However, what if the computing power of a modern laptop was not used; if instead, an absolute bare minimum of computing power was deployed? For one, AI training techniques would be off the table immediately. MCTS would also be highly impractical, leading to the conclusion that

[4] MCTS, or Monte Carlo Tree Search, is a probability-based method used in Chess to find optimal combinations in a given board position.

the only remaining option is a shallow-depth calculation along with an evaluation based on more statistical anticipatory techniques that account for the shallower depth.

We tested several programs, while keeping the following two maxims:

- Simplicity- no particularly complicated evaluation systems were used
- A maximum depth for calculation of 3

As expected, when these algorithms played against each other, they performed quite like a poor player, making blunders and playing in a predictable fashion. So, in a way, making this algorithm as efficient as possible also provides insights on how poor players could increase their level of play at a very short notice.

Following are some quick changes that substantially increased the strength of the engine:

- Valuing knights at 0.1 weight higher than a bishop. It appears that despite bishops being powerful for skilled players or engines in general, knights perform better when search depth is restricted to 3.
- Disproportionately valuing certain squares- d5, f7 and f2 were some of the squares that, when focused on by the controlling engines, significantly increased their win rate.
- Always castling when feasible- this simple change stopped some games where the engines had left their kings in the center and lost either due to attacking play by the opponent or due to compromised development as a result of the king being in the center of the board. This was a quick fix for the removal of the king safety value generation.
- Increasing the proportional contribution of the variable measuring attacking potential which was generated by the proximity of the pieces to the opponent's king.

It was found that, empirically, this modified engine had an 80%+ win-rate against the un-modified engine. However, tellingly, both engines were incredibly weak. This section is a good comparison to segue into what is discussed next. Much like how an engine with limited computing depth cannot objectively pick strong moves, an engine in a game with near infinite options and imperfect information is also strategically hobbled. However, as demonstrated in this section, there are steps that can be taken to optimize strength.

Section 3: The weakness of an AI-based approach when there is imperfect information

Predictably, the Alpha Zero approach works optimally while playing games with perfect information and when there are a finite number of moves to choose from. It doesn't, however, perform quite as well when playing a game with imperfect information and a nearly infinite subset of possible actions. Alpha Star defeated both MaNa and TLO[5], but there were several problems with its matches that raise questions as to just how effectively the algorithm achieved its purpose.

First, although the fog of war was enabled, Alpha Star was able to play to an effectively zoomed outversion of the map. Frequently, in human play, "cheese" early rush strategies[6]can secure a win purely because the opponent was looking at another part of the map. In some cases, as soon as the human opponent sent a unit in on the fringes of Alpha Star's range of vision, it reacted and began queuingtroops to effectively counterit. When Alpha Star was confined to a system that limited its camera to a certain region, it was defeated decisively by MaNa.

Second, Alpha Star played with perfect control. Although this may seem insignificant, in human playmost actions are somewhat imprecise due to the speed involved; precise gameplay therefore, often compensated for strategic misplays by AlphaStar. In particular, although Alpha Star's intuition of which buildings and units were optimal in a given situation was excellent, analysis by human pros revealed that it often did not place them in the best possible locations, as a human pro would undoubtedly have done. It usually failed to ascertain how the enemy troops would path towards a target. Additionally, although Alpha Star's APM (Actions per Minute) were lower than those of its human opponents, the actions taken were not uniformly distributed over time. After carefully reviewing the footage released by Deepmind, it is evident that Alpha Star's APS (Actions per Second) spiked for very short time periods during critical confrontations to reach superhuman levels. Alpha Star makes up for this lacuna during more peaceful times by executing fewer actions per second. Undoubtedly, this enhanced its performance to a large degree, albeit incidentally. Also, in human playmany actions are "double clicks," or clicks that serve no purpose, but are there as a consequence of the speeds involved. We conjecture that if an EPM (Effective Moves per Minute, or APM minus the redundant moves) graph was generated, Alpha Star's would be at least as high as that of the average professional player's, if not more, and

[5]MaNa and TLO are two professional Starcraft 2 players that Deepmind invited to test Alphastar.
[6] "Cheese" strategies involve attempting to catch the other player off guard to ensure a quick win

during confrontations would far exceed it. To analogize, it is akin to comparing a man to a car in a footrace and arguing that the car won because of superior technique.

Lastly, it is important to keep in consideration that the human players that Alpha Star was initially tested on, TLO and MaNa, may not have been performing at peak during the event for various reasons. Perhaps the most obvious one is that the matches were played on an older version of Starcraft 2, which the pros were understandably out of practice with. MaNa also stated in a video recording that he had been kept in the dark about Alpha star's details and was consequently nervous. He also isolated an incident in his first game that caused him to lose in a winning position, merely because he forgot to deploy a building that was required to be placed in the older version but not in the newer one. Often enough, Alpha star was observed to be making obvious grave strategic misplays, like preparing units that were specifically weak against the opponent's main army.

The question then is "why are Alpha Star's decisions subpar to those of professional players inmany niche situations, which certainly isn't the case in Chess?"

I would argue that this omission lies in the way that it was trained. AlphaStarhad an initial observation based learning process and rapidly discarded seemingly bad strategies like quick rushes with Dark Templars and Photon Cannons[7], as these are rarely used in professional play. I conjecture that this is because a successful "cheese" rush requires considerable precision and skill to execute perfectly, something rarely present at lower levels. It also was an "all or nothing" style of play, so dominant versions of Alpha Star preferred to avoid the risks associated with it.

In addition, as the neural net and MCTS system adapted for Chess and Go would obviously not work in a game like Starcraft 2, Alpha Star used only the neural network, with some adaptations. The problem with this approach, however, was that neural networks fundamentally have a propensity to repeat certain actions, evident in the rematch against MaNa when AlphaStarcould not maintain control over its economy. Over fitting was also a problem. Since Alpha Startended to try and end games early on, it never gainedthe experience to change its army once the game had progressed and more "late game" units were viable.

Finally, for truly optimal performance, the algorithm should be well versed to minimize losses. In Alpha Zero games, it was observed that when Alpha Zero was in a losing position, it often "gave up" and made irrational moves, rather than try and win. We conjecture this it is because Alpha Zero was trained only by playing against itself, and thus unused to fightingfrom a losing position; opposing agent would undoubtedly know how to convert it to a win. For remedial action, we recommend that weaker versions of an AIranging from a few hundred ELO points[8] to over a thousand points under the strongest version should occasionally be brought to play against the latter in the training process in "odds" games (games played from a worse position). This would ensure that the algorithm learns that it is possible to win or draw from a disadvantageous position if facing weaker players, for fewer immediate losses in a bad position.

Having examined the strengths and drawbacks of a neural net powered AI, the next section examinesextending the originally discussed evolutionary system to tasks beyond simple games.

Section 4: The extension of the evolutionary approach to other tasks

Generally, the evolutionary approach is an excellent for creating an AI-powered solution to a problem. Its main drawback, however, is the difficulty in generating the fitness function for each individual or instance.

One way to generate a value could be through a separate evaluation algorithm. For example, if an algorithm was made to play a one-player game like Super Mario, or Brick Breaker, the algorithm could have a fitness value[9] based on the percent of the game completed in terms of levels, and the amount of time taken to get that far. The question, though, is how will that apply to real-life problems or tasks, like perhaps driving a car? Evaluating an agent[10]on the basis of performance on a set course, or number of courses, would not work, as it would rely excessively on predefined courses. An approach, therefore, may be a combination of the evolutionary approach with a neural network for image recognition. A certain set of hardcoded procedures with variable values would be required in order to make this work effectively, like Turn x degrees to the left, and sensor functions that could detect pedestrians, or other obstacles.

A full breakdown of a sample pseudo code that could achieve this is given below:

Function pick Random Course: This function picks a random course for the agents to test on.

[7] Troops typically utilized for "cheese" strategies in Starcraft 2

[8] ELO is a convenient unit to measure a player's strength; a difference in ELO indicates that one player is more likely to win a given game

[9] A fitness value essentially measures the competence of the individual at a given task, specifically according to preset criteria

[10] An agent is a single instance of the algorithm that is undergoing training through the evolutionary process. Many agents compete against each other to procreate and pass on their genes and characteristics to future agents.

Function evaluate Performance (course, individual): This function tests agents on the course by giving them a start and end point that is specific to the course, and assigns them a relative fitness value based on the number of collisions or collateral damage incurred, whether they reached the end, and the time taken to complete the course.

All individual Car functions eg accelerate, decelerate, turn a certain amount left or right, etc.

All individual scanner functions detect obstacles, traffic lights, etc.

Function create Generation(number): This function creates an entirely new generation with each individual having random policies such as random ifs, loops, and policies based on those with random values.

Function next Generation: This function runs the evaluate Performance function for each agent in the generation, and then removes the bottom 50%. A new batch with completely random values, 20% the size of the previous batch is created and appended to the generation. Each member of the generation then "breeds" with another member, causing a random half of each member's code to be passed on to the progenitors. It is desirable that the top 10% have a higher probability of selecting other elite individuals. There is a small percent chance of mutation, changing characteristics of a certain functions or the value of an individual randomly. The new generation then conducts the same process again with a new course until a progenitor reaches a certain minimum "score" threshold on the evaluation function, or the maximum number of generations is reached.

Ultimately, such an approach is generally viable so long as the performance of an agent is easily measurable. In addition, in the above algorithm one might notice that the assumption is that all detection is done correctly, the function of the neural network mentioned earlier.

It is important to keep in mind is that possibly, contingent on the weightage of the fitness function, agents may attempt to min-max their score by, for example, cutting corners in the example above to reduce the time taken. If that is observed, the fitness evaluation algorithm will need to be adjusted accordingly.

Conclusion:

Ultimately, one of the major problems with any AI-based issue is overfitting. Almost all inaccuracies can, in some way, be traced back to it. Solving this problem is however, singularly challenging without massive computational resources or manual code. The evolutionary approach, therefore, is a good compromise that statistically requires less computing power, but can still obtain a reasonably powerful performance.

To take this forward, experimental conditions can be set up in two ways:

- Those with preset invariant conditions to enable the algorithm manually learn the best possible configuration, including optimizing subways, in which over fitting would not be a problem.
- Those with changing conditions, so that the algorithm's learnings are based on a set of protocols rather than steps such as controlling software for a self-driving car in which over fitting would largely mar future performance.

Broadly, if a problem can be defined in the above two ways, experiments can be conducted to compare the efficiency of the evolutionary approach with the efficiency of a traditional AI approach such as neural networking or a manually crafted solution.

References

[1] Silver, David, et al. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play." *Science*, vol. 362, no. 6419, 2018, pp. 1140–1144., doi:10.1126/science.aar6404.

[2] "Blog." *Blog - Leela Chess Zero*, 2018-20, lczero.org/blog/.

[3] "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II." *Deepmind*, Google, 2019, deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii.

[4] DeepMind Starcraft 2 demonstration - MaNa's personal experience- video posted online.

[5] Wong, Ka-Chun. "Evolutionary Algorithms." *Nature-Inspired Computing*, 2015, pp. 111–137., doi:10.4018/978-1-5225-0788-8.ch006.