Project Report

Devesh Marwah
Github Repo

November 2021

Contents

Part I
Game AI

1 Introduction

1.1 Motivation behind the project

I have always been motivated and fascinated by Board games and ability to design a perfect board game AI. This got me more interested after the release of AlphaGo which defeated the Go world champion in 2016 and the game of Go has more possibilities i.e. 10^80 than atoms in the universe which was an impressive feat. After learning about Neural networking independently, I tried my hands on this project. Few points which I have not understood completely have been linked to proper sources for reader to understand.

1.2 Why Connect only?

Any other game like Othello or chess or Go could be chosen but training the model requires a lot of computation power and more the branching, the difficult to enumerate the game can be. Connect although can be brute forced might not be a good example but it has a decent branching factor of 7 and required 30 hours of training to reach the Level 4. A game like Chess would have a branching factor of 35 or Go with factor greater than 200 which would require a massive amount of computing power, hence it was not possible to train it completely in the current model.

1.3 What I give to the society?

There is no publicly available code of Alpha-Zero online. The only available thing is the reference paper of the Alpha-Zero released by the Deep Mind team which has a bit of code and the exact idea and desgin of Alpha-Zero which I have tried to replicate in a smaller game here. I have linked the original doc in the references below.

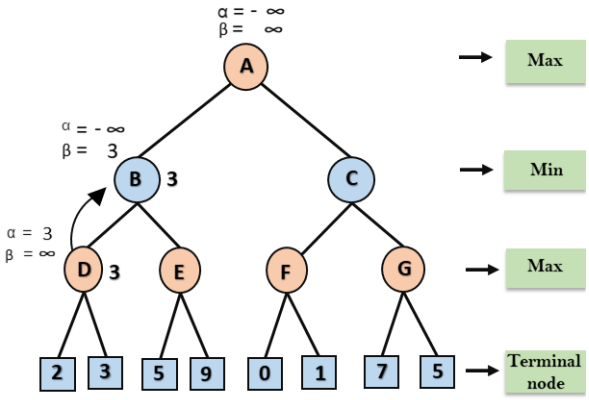
2 Classical Evaluation

2.1 Introduction

The idea was to make a Self Playing AI which would play games efficiently and analysing new ways to play games with AI. Currently the AI's which exist just modify the Brute force approach of the minimax algorithm and methods are made based on the evaluation function which can be difficult to make and are obviously not perfect always. In this project, instead of using the classic minimax, I have tried to do something new and make AI play the game without

knowing any basic rules. But before discussing the approach, let's discuss the classical approach and a bit of code.

2.2 MiniMax Algorithm



MiniMax Tree

In the classical MiniMax tree we 2 players, One player tries to maximise the value whereas the minimizer tries to minimise the value of the tree. In the above example consider red to be the Maximiser and Blue to be the minimiser. Now after a certain depth, we perform evaluation of the tree and then select the best possible state for the person playing.
For example in the figure when Red is the maximiser he will select 3 9 as his value. Now blue will take the minimum of the 2 values provided to him i.e. 3 and then in the right child, the min value is 1 from which the Red player selects 3 to be final evaluation

Note that such trees work in case of games of Complete information i.e. both the sides exactly know information about the other team and it's evaluation like Chess,Go,Shogi.Connect4 etc.

2.2.1 Alpha Beta Pruning

This is a way of modifying the current brute force way of checking all nodes and can be done to check only a part of the tree is needed to be searched and there is no need to search the rest of the tree. Like for the above example after searching the left tree, I already have found the value of the the minimizer i.e. β to be 3 and hence we know that the selection from right child is not needed.

In general terms α is used for the maxisimer player and β for the minimiser player. This has been implemented by me in Connect4 visualise_minimax.py.

3 Alpha-Zero-AI

3.1 Introduction

The idea behind the AI is to use reinforcement learning and teach AI based on previous games. The idea is pretty counter intuitive because often reinforcement learning experts have argued that the learning requires some games from top players and data set is needed, but DeepBrain came up with a nice idea to generate self play data on the own by using Monte Carlo Trees and the idea in whole would be to optimise the tree search. A lot of deep learning models like this fail but the ability of it to work relied on the fact that it can be easily simulated and is a "perfect knowledge game"

3.2 Node Design

3.2.1 Tree Design

Before understanding the algorithm, we need to know what the tree represents. A single MCT represents a complete game played which is guided by various policy networks which tell the AI how to play, the better the policy value function implies better the AI would be. In the MCT each node stores its state,value and probability of it being pick represented in the paper as (s,π,p). Also a node stores the number of times it has been visited (more the visit show it to be a better move) and another value named U. Each edge of the MCT represents the move being played. However, to store states at each node can be memory intensive hence we store the initial state only and from then can make the moves simply as suggested by the edges. The nodes also perform various functions which can be described as below:-

Select and Expand This is the function which plays the game to the end while evaluating each move probability according to the function given below.
Here s is the board state and a the move played

$$U(s,a) = c_p * P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \tag{1}$$

here c_p is a constant which is equal to 5 after experimentation, N is visit count and P is prior probability. Hence we can see it favours actions with high probability and low visits.
This along with another constant named U i.e. Q is the mean of previous actions and Q+U in total is used to evaluate the probability.

Backup After reaching the end of the tree it returns the result as -1,0,1 and updates the value of Q and increments the number of games by 1. After this note that while returning and updating the values, we have to return the negative of the previous at each step.

3.2.2 Move selection

Now to select the move we have to select the move with the best score of Q+U.

3.3 Neural Network

3.3.1 Introduction

After seeing the working of the MCT, now it's time to improve the MCT and guide it via a network in order to enforce it to play a better move. In the beginning it was mentioned that training data is needed in order to play the game. This data can be generated via playing with the MCT's and then updating the neural network via the loss function in order to help it understand which solution is the best.

3.3.2 Self play

This can be improved via our understanding of the policy value network and it has been divided in 2 halves via self play.

First 30 moves A constant named temperature or τ is added in order for deeper exploration in the first 30 moves. Here τ=1 causes more exploration.

Post 30 moves Here τ=0 hence exploration is not encouraged i.e. if N(s,a)¿N(s,a') then p=1 .Also dirichilet noise is added which was something I found difficult to understand but is explained here. What I could understand was that a constant of 0.3 basically assumes we have visited the position 10 times in a 35 move game and is added for better exploration.

3.3.3 Supervised learning

The team of Alphago found the combination of Supervised learning + the evaluation of the leaf nodes to be the most efficient one, Hence it combined both to build the function. Supervised learning is basically implementing the entire neural network by keeping the learning rate i.e. $l_r = 0.9$ and $l_2 = 10^-4$.

3.3.4 Description of network Implementation Details

The network is similar to the alpha-zero network :-

1. A convolution of 32 filters of kernel size 3 with stride 1
2. A convolution of 64 filters of kernel size 3 with stride 1

3. A convolution of 128 filters of kernel size 3 with stride 1
4. A dense layer of output filter size 64, 128 and 1
5. Activation used as relu for conv2d layers
6. A tanh non-linearity outputting a scalar in the range [1, 1]

The code is further updated according to the following code:-

```
def compute_loss():
    predicted_prob,predicted_arr=neural_network(new_state)
    loss_1=tf.reduce_mean((new_arr-predicted_arr)**2)
    loss_2=-tf.reduce_mean(tf.reduce_sum(new_prob*predicted_prob,axis=None))
    loss=loss_1+loss_2
    return loss

optimizer.minimize(compute_loss,neural_network.trainable_variables)
```

4 More Learnings

1. Award winning documentry for beginners here
2. Minimax explanation here
3. Video explanations which dive a bit deep in explanations 1 and 2
4. Rigorous analysis for interested here
5. Official Document for Researchers here

Part II
New ideas learnt

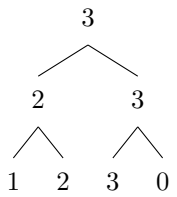
5 Segement Trees

5.1 Introduction

This data structure is of great use as it can be used to perform Queries on a certain set in log(n) time and update the values also in log(n) time against the standard O(n) time. The implementation can be sometimes memory intensive in case of very large inputs but it is of O(n) complexity only (the constant is a bit high).

5.2 Details

On seeing the O(log(n)) it can be intuitive that some structure of tree is definitely required. Before making range queries on an array we build it first in such a way that range queries can be performed. For example we have array 1,2,3 and we need to find the maximum, then the Algorithm will be



1. Fill the array with values like 0 or ∞ or -∞ i.e. values which do not affect the output, here it would be 0.

2. Define an array 4 times the size of the original array and fill it in such a way that the parent node stores the value obtained from the 2 child nodes i.e. in case of 1,2,3 as we can see, we need to keep adding extra nodes in it to perform the operation.

5.3 Range Minimum Query

After building the segement tree, now is the main part on trying to perform the range minimum query. Say we want to find max in a specific array then there can be 3 cases

1. No overlap
   If it's this case then we don't need to go any further searching for values and just return -∞ so that the ans is not affected and the search is stopped.

2. Complete overlap
   This is the case when the searched part lies **completely between** the specified array. If it's the case then return the value of the node then only.

3. Partial overlap
   In this case go further into the node and keep checking until we reach a leaf or above of the 2 conditions.

As we could clearly observe that we did not have to complete the complete array and recursive calls are being made as that of the height of the segtree hence the total number would be O($\log_2$ (n))

5.4 Update

Now if we want to update a given member in the array, this can be done simply by traversing upto the node in the array and then while backing up, we will recursively update the value of nodes. Since again the traversal was only the height of the tree, the total time it would take is ($\log_2$ (n)).

5.5 Conclusion

We can here observe the strength of the datastructure and it has massive use especially in the CP in order to get better time complexity, although the implementation can be a bit heavy. The implementation for a min segtree can be found here and the max segtree can also be implemented in a similar fashion by just reversing the signs.

6 Linear Increasing Sub sequence

6.1 Introduction

Given an array of numbers. we need to find the linearly increasing longest sub sequence in it. The problem has a dynamic programming approach discussed in class but here I would like to present something more than that and optimise from n^2 to O(nlogn).

6.2 Program

We will maintain lexicographically minimum array for each size and check if the new value added is greater or less than the last one, if it is greater we replace it and discard that array if it is greater then we need to simply add it at the end.

We can simply use the lower_bound function of C++ to find the minimum boundary and hence find the answer. The link for my implementation can be found here

7 Hashing

7.1 Introduction:-

While coding on Codeforces my recent solution got "hacked" because of using an unordered map. By hack I mean that a problem's solution can be easily tested against a strong custom test case designed to fail the code. But I was using the C++ standard STL library hash function and still it was hacked. Why and if it has been hacked earlier why are the C++ developers not revising the function? To know more we need to study about unordered map in a C++ module.

7.2 Unordered Map

Unordered map in a C++ module is a set of hash tables defined to work properly for storing a multiple values for a single key. The time complexity for insertion and search ideally should be $O(1)$ but due to collisions can be upto $O(\text{n})$ where n is the size of the input. The C++ hash function is quite well tested and is publicly avaiable so if inputs are designed specifically against the function, it can be easily hacked as the time complexity increases and although the output would not be wrong, the time complexity will go beyond the allotted limits in the question of Codeforces.

7.3 Ways to resolve the issue

The idea to resolve would be to make a hash function which cannot be guessed, hence has a sense of randomisation to it. Personally making one would be quite difficult as it has to avoid collision but for usage in CP we can use the splitmix. This is a nice hash function derives specifically for the use of CP.

7.4 How does this Custom Hash Work ?

We see the number $0x9e3779b9$ in the function and the result is rather an experimental one. This "magic number" is a part of the golden ratio's fractional part

$$(\frac{\sqrt{5}-1}{2}) * 2^{32} = 0.61803398875 \tag{2}$$

This number has special scaterring properties observed over an experiment when analysing the equation of line . The randomisation has been added by the line

```
chrono::steady_clock::now().time_since_epoch().count();
```

which counts the time in milliseconds since the beginning.

$$y = x \cdot c - floor\left((x \cdot c)\right) \tag{3}$$

Change the values of c and we will observe that this the Golden Ratio value has the best uniformity over the scattering. This observation was shown by Donald Knuth in his book The Art of Computer Programming: Volume 3: Sorting and Searching. This experimental result is also known as the **Fibonacci Hashing**.

7.5 Why does C++ not change the model ?

C++ language is made for general use and emphasises more on speed than on making a strong hash function which cannot be guessed. With the use of C++ in High Trade Banking Firms where codes are written in C++ for quicker speed and every second is counted, they need more speed hence have provided the option to change the hash function in our code. The custom hash leads to a slower speed due to a bit more collisions hence might not be preferable.

7.6 Conclusion

Hence in the end I was able to get an AC using the custom hash function link, though a bit slow as the hash function is still not the best. The code for the same is

```
unordered_map<int, int,custom_hash> <unorderd_map_name>;https://www.overleaf.com/pro
```

7.7 References

Codeforces Blog by neal

8 Sieve of erathoneses

8.1 Introduction

This is a very basic algorithm which discusses to get prime numbers in range 1 to n. This can be done simply by first selecting 2 and discarding all the multiples of it, then we encounter 3 and discard it's next divisible elements present and so on. The reason for it's popularity is it's simple implementation.

8.2 Time Complexity Analysis

The reason of it's working is pretty straightforward but a further analysis of it's time complexity can be done.

1. The number of prime numbers approx $\frac{n}{ln(n)}$
2. The k'th prime number approximately k ln(k)

This can be written as

$\sum_p \frac{1}{p} = 0.5 + \sum \frac{1}{kln(k)}$ (4)
This summation can be simply changed to an integral for large values and finally gives log(log(n)) as the answer hence the complexity is O(log(log(n))

9 Implementations

Following are various implementations of different algorithms taught in class, strictly in C++.

1. Merge Sort
2. DSU
3. Kruskals
4. Task Scheduling
5. DP Problem

10 Github Repo

The link for the same is here