FROM NOVICE TO GRANDMASTER

A midway report

Enrique A. Marrero Torres
Alejandro Rodriguez Natal
Julian D. Cuevas Paniagua

ICOM6025- High Performance Computing

FROM NOVICE TO GRANDMASTER

Problem

Chess is considered to be a *Zero-Sum Game*, that is to say that every loss(of state) has an equal gain that comes from said interaction. However, this is not reflected by a mere loss or acquisition of a piece, it can also be represented by a strategic advantage created by board position, sacrifices, and inherent advantages for certain colored pieces in any given game. Therefore, we can assign any given position and piece a numerical representation based on these factors. For the purposes of this paper the following assumptions will be made: 1 will represent a winning position, 0 will represent a neutral position(better known as a draw / stalemate) and -1 will represent an absolute loss in a position.

Following such assumptions, one must define the biggest problem (computationally speaking) in chess, that is, all possible states in which a given chessboard can take. To define its magnitude we must take into consideration the following: the pieces, the board itself, and the possible moves the pieces can make on a given turn.

- Chess board(dimensions): 8 X 8 (64 squares)
- Pieces (per player):
  Knights (x2), Rooks (x2), Bishops (x2), Queen (x1), King (x1), Pawns (x8)
- Moves

Where:

m is the amount of total moves
pieces is the amount of pieces for the players
players is the amount of players

$$Possibilities\,(\,m\,) \;=\; \frac{(chess\ board)!}{(m-pieces)!(piecess^2)!(players)^5!}$$

Assuming the average length of any given game at a ***GM*** level is around 40 moves:

$$P(40) \;=\; \frac{(64)!}{(40-8)!(8^2)!(2)^5!}$$

$$P(40) \;\approx 10^{43}$$

This is known as "*Shannon's Number"* named after the man, who first calculated the possible states of a 40 move chess board, Cloude E. Shannon.

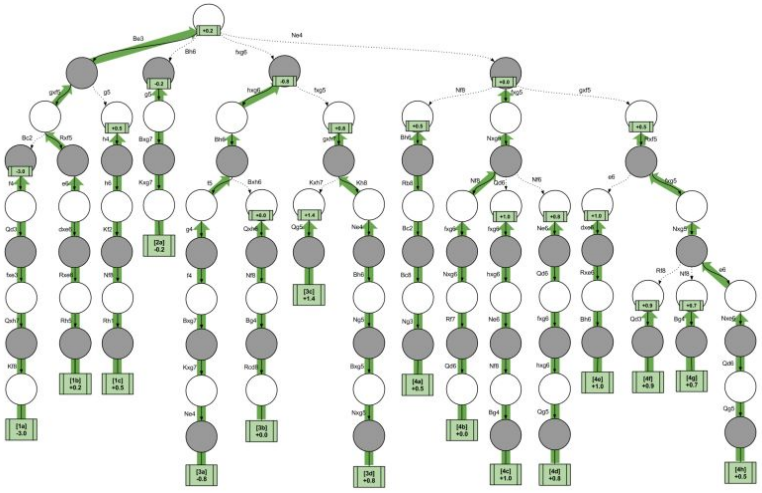
Thus, proving to be an infinitely complex task to find the optimal move 100% of the time given a finite time span. At best, the runtime of any algorithm that accurately chooses the optimal move in a position would be *O(n)*. This would be because, given any arbitrary tree ***T***

whose nodes represent the board position (lets call it p), would require you visit at minimum the *nth* node(where *n* would be the total number of moves *m*) of the aforementioned structure. Furthermore, this assumption presumes the existence of infinite memory on a given computational system without any regard for overflows(see *Figure 1*).

Figure 1. Tablebase Sizes Vs Number of Chess Pieces

Number Of Pieces	Tablebase Size (Bytes)	
3	62,000	62 kB
4	30,000,000	30 MB
5	7,100,000,000	7.1 GB
6	1,200,000,000,000	1.2 TB
7	140,000,000,000,000	140 TB

Figure 2. Weighted Chess Decision Tree



Credits: Kevin Binz

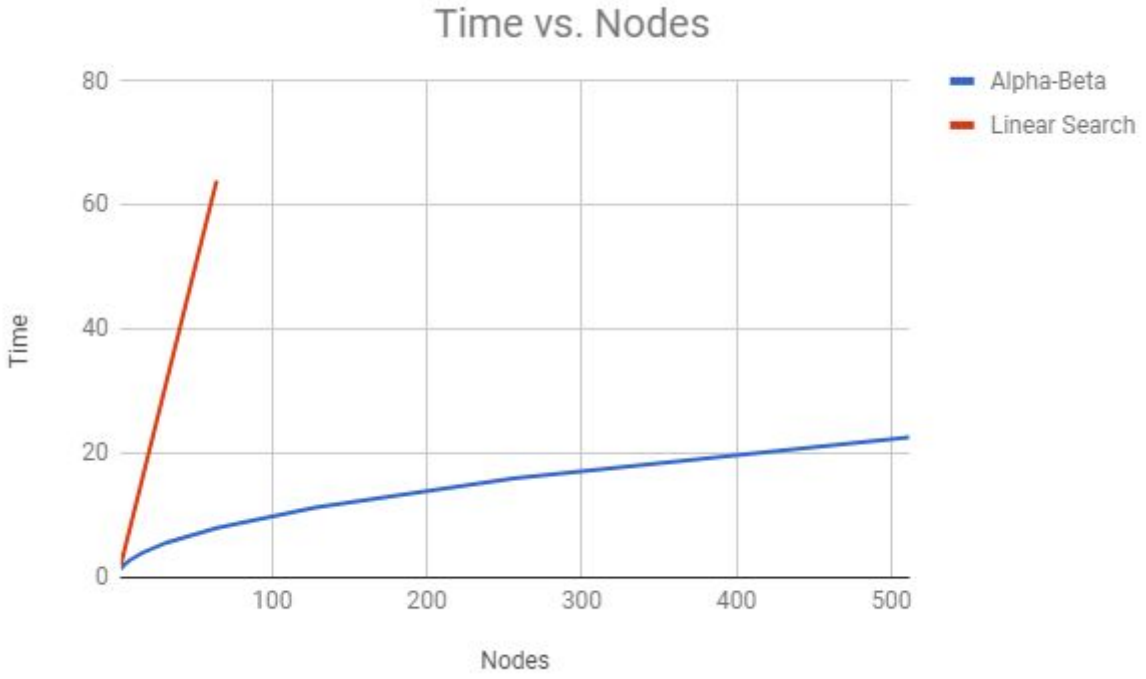
To overcome this flaw one must resort to a method known as *pruning*, with hopes of reducing, not only the computational time but also, the space occupied by the nodes of the tree. One such pruning method is known as *Alpha-Beta Pruning*, whose general idea revolves around keeping track of a *minimum(β)* and a *maximum(A)*. Wherein, the minimum represents a losing position (initially denoted by a -1) and the *maximum* a winning position (denoted as a *checkmate*). From which, the method begins to create subtrees while discarding those that do not supercede the current *A-β* pair.

This algorithm has the Best-Case performance of O($\sqrt{b^d}$)

Where:

b is the average branching factor
d is the depth of the tree at a given node

Figure 3. Graph: Time vs. Nodes Comparing Linear Search to Alpha-Beta Pruning



This assumes an average branching factor of 2 which is not at all accurate but is used for simplicity.

Theoretically, the pruning algorithm will always trump the linear search algorithm as long as the move chosen isn't always the most optimal one in the linear search. However, even further enhancements could be made such as, parallelization of tree traversals and further pruning. This could be achieved by assigning any given processor ***P*** a node *n* representing the root of a subtree pertaining ***T***. Each processor would theoretically prune their respective trees and upon completion, they would return a floating point number representing the state of the game after the remaining positions have been visited. Said number would then propagate back to their respective parent and execute the appropriate pruning action.

Although, there would be a theoretical speed-up based on the nature of parallelization, there also exists a cost of communication. Therefore, in some cases it is better to not parallelize the pruning and just perform a sequential execution of it. To determine under which circumstances it is favorable, we must take into account the size of the data and the amount of available processors.

FROM NOVICE TO GRANDMASTER

A midway report

Proposition

Thus, we must accept that we can't guarantee that the most optimal move will always be calculated by a computer without proper chess experience or knowledge. If we are to presume that any human player can't possibly calculate the best move at all moments in a finite amount of time then, the logical conclusion is that a player is driven by both, experience and preparation. Hence, we decided to employ the use of Neural Networks and backpropagation to aide in our endeavours.

Now we've raised another concern in our experiment, data consistency and reliability. It is safe to assume that depending on the data we were to use in the training of our AI it would dramatically affect the results of the ensuing games. This is called bias in the Neural Network and Artificial Intelligence world, which is much like how humans think. Parting from this point, we decided that to obtain better results than those of the original creator of *deep-pink* we'd use Grandmaster Level chess games. These constraints were chosen due to the lower likelihood of unfavorable moves being chosen by grandmasters which, in turn would likely yield higher winning odds.

Notice that the Neural Network does not completely replace the *Alpha-Beta pruning*, it is merely an aide. This is because it is much easier to slowly correct any mistaken move done by our AI than it is to completely calculate a decision tree. Granted, there will be fluctuations with the results depending on the games that are shown to our AI and how long it is exposed to them.

Neural Network Constraints

Input Layer : 768 wide layer (8 * 8 * 12) [size of board * types of pieces]
Hidden Layers : 2048 layers
Output: 1 scalar(piece to move)

Training Algorithm

Sigmoid Function

$$S(X) = \frac{1}{1 + e^{-x}}$$

Custom Function

$$\sum_{p,q,r} log\, S(f(q) - f(r)) + \kappa\, log(f(p) + f(q)) + \kappa\, log(-f(p) - f(q))$$

where:

f is the recursive function that determines the current board
κ *is an arbitrary constant*
q is the previous move

p is the current board's position

Note:

The function is not aware of the game's rules, it only takes input and generates output.

If the output results in a 1, and it is a mate position, then White wins.
If the output results in a 0, then we have a draw or stalemate position.
If the output results in a -1, and it is a mate position, then Black wins.

Technical approach

Our first approach was to utilize CuPy[1] as a means to accelerate Theano Tensor[2]'s functions, as they force the program to free up some of the overhead work on the GPU instead of basing it all on the CPU. This leads to a better runtime of the algorithm in a hypothetical environment, but by using chainer to relay computed data to the CPU it ends up being slower than the proposed execution speed.

Another approach(although discarded) was to directly use CUDA acceleration without any other modules to import. This would be a more explicit addition and might only allow the GPU to take the workload, as opposed to sharing CPU and GPU workload like with CuPy. Said approach might include some overhead as the GPU will do unnecessary amounts of work.

Experimental settings

Using a CUDA instance at the TACC computing system[3], a cupy version of the deep-pink AI was run. After running for 3 days, the AI was subjected to a set of matches against Sunfish[4], where the results were saved.

Experimental results

After running each AI against Sunfish, the results are as follows:

- Non-parallelized:
  Winner: B | A r/t: 10.690766 s B r/t: 1.844340 s
  Winner: B | A r/t: 0.343347 s B r/t: 16.545769 s
  Winner: B | A r/t: 10.591422 s B r/t: 1.969786 s
  Winner: B | A r/t: 0.406239 s B r/t: 8.887815 s
  Winner: B | A r/t: 20.545958 s B r/t: 30.680983 s

[1] https://cupy.chainer.org/
[2] http://deeplearning.net/software/theano/library/tensor/index.html
[3] https://chi.tacc.chameleoncloud.org/
[4] https://github.com/thomasahle/sunfish

Winner: B | A r/t: 20.547802 s B r/t: 30.554087 s
Winner: B | A r/t: 0.347552 s B r/t: 10.135159 s
Winner: B | A r/t: 0.335643 s B r/t: 15.871970 s
Winner: B | A r/t: 0.335476 s B r/t: 18.224965 s

Table 2. Deep-pink pre-Cupy optimization

Winner	A execution time (s)	B execution time (s)
B	10.69	1.844
B	0.343	16.54
B	10.59	1.969
B	0.406	8.887
B	20.54	30.68
B	20.54	30.55
B	0.347	10.13
B	0.335	15.87
B	0.335	18.22

- Parallelized:
  Winner: B | A r/t: 0.279521 s B r/t: 16.618723 s
  Winner: B | A r/t: 6.048960 s B r/t: 16.893157 s
  Winner: B | A r/t: 6.295330 s B r/t: 17.057679 s
  Winner: B | A r/t: 0.301726 s B r/t: 1.148115 s
  Winner: B | A r/t: 0.305870 s B r/t: 17.427192 s
  Winner: B | A r/t: 0.268470 s B r/t: 5.589318 s
  Winner: B | A r/t: 6.210891 s B r/t: 17.570688 s
  Winner: B | A r/t: 6.706699 s B r/t: 17.647201 s
  Winner: B | A r/t: 6.745099 s B r/t: 19.584320 s
  Winner: B | A r/t: 6.275125 s B r/t: 17.528655 s

Table 3. CuPy Enhanced Deep-Pink Execution

Winner	A execution time (s)	B execution time (s)
B	0.279	16.61
B	6.048	16.89
B	6.295	17.05
B	0.301	1.148
B	0.305	17.42
B	0.268	5.589
B	6.210	17.57
B	6.706	17.64
B	6.745	19.58
B	6.275	17.52

B is the Sunfish AI and A is the trained model. Player B can be seen winning all games in the taken data set, but this might be possible due to running a small amount of games in order to collect data (10 rounds of games).

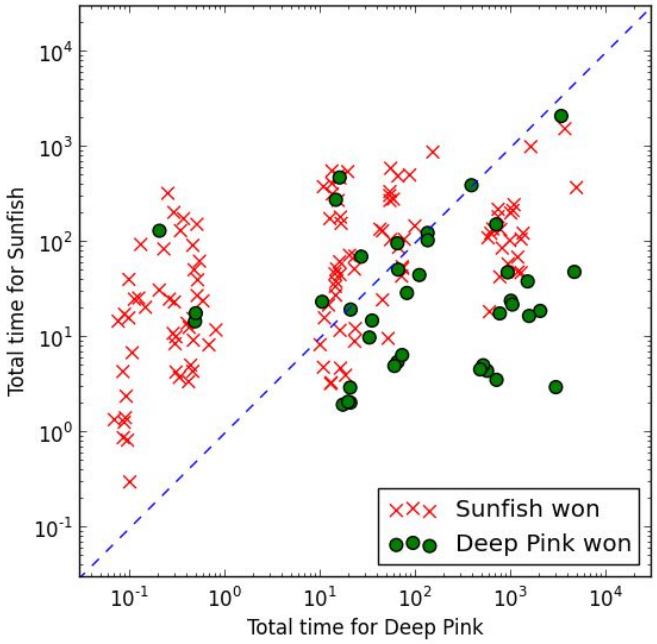
Aside from this, a noticeable difference in execution time could be seen between the non-parallelized and parallelized versions of the AI's training code. The non-parallelized revision ran 10,000 iterations at approximately 30 minutes, while the parallelized revision ran approximately 1,000 iterations in the same amount of time. This is most likely a bottleneck imposed by the use of chainer in order to transfer the GPU-created data back into the CPU.

It is of worth note that the original code randomized the overall depth search of the *Sunfish* AI. That is, *Sunfish* on each iteration would randomly be assigned a time-limit by the provided *Python* Random Number Generator. In contrast, our revision made sure we had a consistent time-limit (we assigned a default of 5 seconds for depth search)to avoid any random variance in our results.

While our results of winning as *deep-pink* were not what we'd hoped, we did however improve the execution time of our decisions when worked on a GPU.

Results:

Figure 4. Accuracy of Sunfish vs. Deep-Pink



Deep pink wins only a third of the time with respect to *Sunfish.*

Conclusions and Future work

The CuPy version of this project was trained with little time and low amounts of iterations, which ultimately leads to a less reliable AI. As is with the non-parallelized model, it would benefit greatly from a longer training time.

Due to the possible bottleneck imposed by chainer, our first goal would be to improve upon it. If possible, a good idea would be to increase the win percentage of the AI. Other approaches to the AI's branch prediction would be considered, as the Negamax algorithm with Alpha-Beta pruning is not well optimized. This seems to be due to the unmeasured use of recursion, which in every iteration creates new alpha & beta variables(wasting memory) and creating newer subtrees.

References

https://en.wikipedia.org/wiki/Solving_chess
https://en.wikipedia.org/wiki/Shannon_number
http://archive.computerhistory.org/projects/chess/related_materials/text/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon.062303002.pdf
https://kevinbinz.com/2015/02/26/decision-trees-in-chess/
https://chess-db.com/public/pinfo.jsp?id=4100018
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
https://en.wikipedia.org/wiki/Endgame_tablebase
https://en.wikipedia.org/wiki/Artificial_neural_network