MAIS 202 Deliverable 3

Aurélien BÜCK-KAEFFER, 261032581

Final training results

We have tried different sets of parameters to improve the performances of the model. Mostly, the architecture: the parameter that seems to have the most effect when decreasing the mean average error is the number of filters per convolutional layer.

Drastically upsizing the model from 32 filters per layer to 256, making the model go from 179 201 to 3,095,569 total parameters, has interesting results, as the validation mean absolute error got a very significant improvement, from about 2300 to about 1500 (about 34% improvement). However, this has massive consequences on the length of training and on the number of positions the model is able to consider before making a move.

The biggest model considers approximately 200 positions per second on my machine, which is excruciatingly slow. For reference, the model with 180k parameters can consider about 1450 per second, and a full search algorithm with no neural network can consider about 4700 per second.

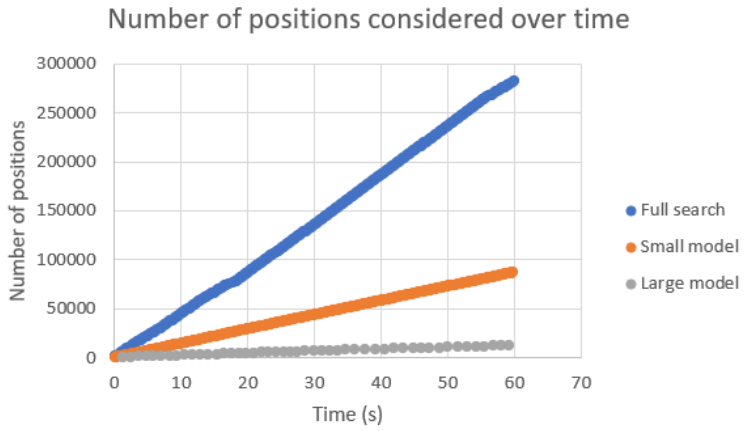

Figure 1: Number of positions considered over time by the three algorithms

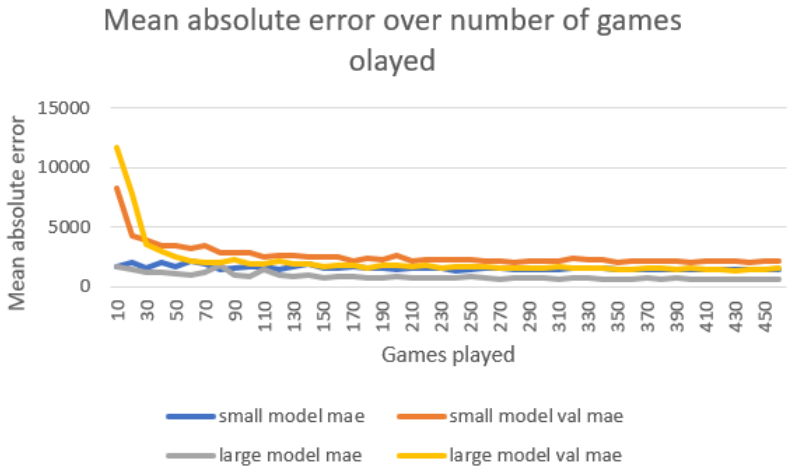

Figure 2: Mean absolute error in the large and small models

To test whether it was better to have a more precise model or a model that can consider a lot of positions, we made both play a match of 12 games, with 6 different book openings (this is important since the algorithm is deterministic. It will always play the same moves if in the same position). Each model played 6 games as white and 6 as black. Each model had about 20s per move to calculate what to play. In total, there were 4 wins by the faster model (2 as black, 2 as white), 2 by the slower (1 as white, 1 as black) and 6 draws, making the faster model winner 7 to 5. We therefore will go with this one for further testing of the results.

A middle point between 180k and 3M parameters could undoubtedly be found, but it is extremely long to do so, especially since training gets slower the more parameters we have, and proper testing takes hours just to compare two models.

It is important to note that the deterministic nature of these algorithms makes them prone to repeating moves, as if a previous position reappears, the exact same sequence of moves that led to it will be repeated. This is a problem that does not exist when playing against non-deterministic bots or humans (though they could exploit this to get an easy draw in a losing position). Since this issue mostly occurs when comparing two models and isn't significant outside of this scenario, it has been neglected.

In the previous deliverable, multiple problems have been mentioned:

1- The model tends to play moves that have not been thoroughly considered since they appear better at face value but were not worth diving deeper into as they were too bad.
2- The model tends to play optimally when the position is deemed equal. If a side is clearly winning, it will either consider too many moves per node in the tree or too little and hyperfocus on one. This was due to the inconsistent nature of the priority function in our tree search: linear when a position is good and 1/x when it is not. Combined with the depth penalty, this created the previously described behavior.

Both problems appear to be fixed to some extend by the implementation of normalization in the tree search: We take the average of the model's predictions for each possible very next move from the root (prediction done without depth, supposed to orient the tree search), and subtract it from every prediction in the tree search.

This way, the algorithm will always behave as if the positions were equal for its best move search, which we observed to be the optimal behavior. This removes the problem of considering too many or too little moves if winning or losing, making the model significantly better at holding disadvantageous positions, and keep its advantage in advantageous ones. This had the involuntary consequence of making a more uniformly distributed tree search among the moves that do get considered by the model, making rare moves that are only considered at depth 1, and consequently effectively removing problem 1.

There still are problems though, mainly difficulties to convert a winning position into a checkmate, which resulted in a few unnecessary draws against low rated opponents. This was improved by the implementation of a special piece-square table for pawns in the naïve evaluation function. By dividing the piece-square table into two parts, one for the early/middle game, and one for the endgame, we can tune the behavior of the algorithm depending on which phase of the game it currently is in. So, by introducing an endgame piece-square table for the pawns that rewards more advanced pawn, we made the model more likely to promote and win a game. This is an interesting path for improvement, since similar things can be introduced for each kind of piece. Note that the

model has not been trained using these improved piece-square tables due to the training and testing time implied, so that is also something to possibly investigate.

To test the performance of our model, we made it play against the chess.com bots rated 250 to 2000. Since they are bots, we can assume they play consistently at the same level, and their regular Elo spacing allows for great benchmarking. In total, our model played 32 games against 16 different bots, 2 games against each, one as each color, with no opening book, 20s per move. Full results are in the following table.

Opponent rating	Win as white	Win as black	Loss as white	Loss as black	Draws	Total score (Model-Opponent)
250	1	0	0	0	1	1.5-0.5
400	1	1	0	0	0	2-0
550	1	1	0	0	0	2-0
700	1	0	0	0	1	1.5-0.5
850	0	1	0	0	1	1.5-0.5
1000	1	1	0	0	0	2-0
1100	1	0	0	1	0	1-1
1200	1	0	0	1	0	1-1
1300	0	0	0	1	1	0.5-1.5
1400	0	0	1	1	0	0-2
1500	0	0	1	1	0	0-2
1600	0	0	1	1	0	0-2
1700	0	0	1	1	0	0-2
1800	0	0	1	1	0	0-2
1900	0	0	1	1	0	0-2
2000	0	0	1	1	0	0-2
Total:	7	4	7	10	4	13-19

We can make a few interesting observations: The model did not lose a single game until the 1100 Elo opponent. However, it did draw a few due to the previously mentioned inability to convert a winning position into checkmate. The turning point between reliably winning and consistently losing appears to be between 1100 and 1200, opponents that both had one win and one loss against our model. The model drew one last game against the 1300 bot before losing every subsequent game against higher rated opponents.

The average opponent rating was about 1203. Using the formula for calculating performance rating, we can get an Elo rating estimation for our final model:

$$R_p = R_a + 400 * \frac{Wins - Losses}{n}$$

With R_p the performance rating, R_a the average opponent rating, and n the total number of games played.

$$R_p = 1203 + 400 * \frac{13 - 19}{32}$$
$$R_p = 1128$$

If we assume that the bots' Elo is a good benchmark and that they are properly rated, this puts our model approximately in the 84th percentile of chess.com players. Still, note that chess.com Elo is considerably inflated compared to FIDE rating.

We are exceeding the 1000 performance rating we were hoping for as a baseline objective in deliverable 1. This should hopefully mean that our model can reliably beat new players. However, it is to be expected that humans play vastly differently than bots, making hazardous any prediction on performance against real players.

Final demonstration proposal

I will make a webapp allowing users to play against my model on a chessboard similar to lichess or chess.com.

I would be using gcloud and flask, for the simple reason that I have previous experience in developing webapp with it in a limited amount of time due to previous hackathons. More specifically, I have experience implementing a machine learning model in a website due to my participation in the last edition of the MAIS hackathon.