MAIS 202 Deliverable 2

Aurélien Bück-Kaeffer, 261032581

Problem Statement

We are creating a chess engine using reinforcement learning. Our model's goal is to learn through self-play how to correctly estimate a position's value based on material advantage, positional advantage, and predicted future positions and outcome. We then use this model to orient a tree search algorithm by prioritizing some game variations over others in our search for good moves, and finally we pick the one that has the best expected result to play.

Data preprocessing

We are working with games of chess generated through self-play by our model. It was originally suggested to use grandmaster's games from databases in the training as well, but this has not been done so far. This, however, remains an option, as a similar method was used in AlphaGo [1]. We are using a convolutional neural network, and therefore treating the chess board as an (8×8) image.

We are using one-hot encoding for each case of the chess board: Every case in represented by a vector of dimension 6, with one element for each type of piece, positive being white and negative black. Each vector is of the format $[pawn, knight, bishop, rook, queen, king]$. So, a white pawn would be represented as $[1, 0, 0, 0, 0, 0]$ while a black queen is $[0, 0, 0, 0, -1, 0]$. Therefore, our data's complete dimensions are $(8 \times 8 \times 6)$.

The number of samples increases as time goes by, as it is generated through self-play. We let the model play 10 games, then generate the Q-value corresponding with each position S using the formula

$$Q(S) = naiveEval(S) + \gamma Q(S + 1)$$

With $naiveEval(S)$ being an evaluation of the value of the current position based on material imbalance, piece positions and if this board is a checkmate/stalemate. It does not consider future positions, hence the name "naïve" (see *static_board_evaluation.py* for the exact implementation).

$Q(S + 1)$ is the Q-value of the next position that was played in the game.

γ is a hyperparameter between 0 and 1 that we use to decrease the impact of the value of future positions the further away they are. We set it at 0.9.

Machine Learning Model:

As for Alpha Zero [3], we are using a convolutional neural network as our model for this task. However, this choice was originally made to play Go, since its rules are translationally invariant, which is not the case for chess (castling isn't the same on both sides, pawns promote once they reach the other side of the board…). This was because the point was to make a general-purpose algorithm and changing the architecture for each problem defeats this purpose. We, however, are looking into chess exclusively, and can therefore tune our architecture to fit it better.

As seen in lectures, convolutional neural networks have pooling layers in order to obtain the translational invariant important in image processing. We, however, do not want this invariant, and therefore removed the pooling layers from the CNN.

The model is implemented using Keras. It is composed of 6 convolutional layers with a kernel of 3×3 and 32 filters as well as padding and relu activation function, one 64 neuron wide dense layer with linear activation, and one output layer containing only one neuron with linear activation as well. Since modifying this architecture yields widely different results, changes in these specifications are to be expected.

Each iteration, training is done on 10% of the previously seen positions to limit catastrophic forgetting [2], on top of all the new positions.

A validation split is done to save 10% of the dataset for assessing the efficiency of the model on new data.

To increase the occurrences of checkmate in the training data for our model to learn to recognize it as the goal, we made playing any checkmate in 1 move that happens in a game mandatory to play. It appears to be sufficient at making non-drawn outcomes appear often.

As stated before, the full algorithm to play a game is a tree search algorithm that uses the model to know in which parts of its search it should focus. We have both a priority queue to determine which node is the next to explore, and a tree. The tree's root is the current position, and its children are the positions we can reach on the next move, and so on.

Every node is also in the min-heap with a priority value P. Finding a proper equation to calculate P is the key to our algorithm. The lower the P value for a given position, the more our algorithm needs to prioritize visiting it in the tree search. We want P to increase with the depth of the node in our search tree, as moves further away are less worth considering.

Multiplying the network's prediction by a depth penalty constant $\delta > 1$ to the power of the node's depth looks like a good way of doing this at first, but it runs into an issue: The model predicts real values, both positive and negative. So if it predicts a negative value of -500 for a board at depth 8 for $\delta = 1.05$, it will actually attribute it the priority $-500 * (1.05)^8 \approx -738.7$, which is lower than our original value. But we are in a min heap, which mean we actually increased how important this node is to consider, which is not what we want.

To get rid of this problem, we need a function f that turns the negative values positive but keeps the ordering between them as we still want values that used to have a lower priority to keep said lower priority.

$$f: \mathbb{R} \to \mathbb{R}^+$$
$$x > y \to f(x) > f(y)$$

Luckily for us, there exists a function that does exactly that, which is the exponential function. Unfortunately, it has another issue: it gets extremely big extremely quickly, as well as extremely close to 0 extremely fast. Our computer's precision being limited, this doesn't work for us. We need to find a way to modify it to fix this property.

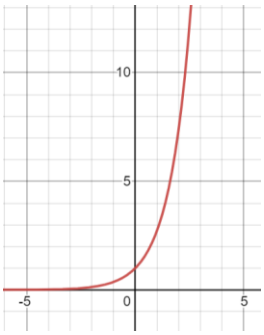

Fig 1: The exponential function

The natural logarithm does exactly that for us, but it is only defined on $[0, \infty)$. No worries, we simply need to make a piecewise function out of 2 logarithms: $\ln(x)$ and $-\ln(-x)$, which covers $\mathbb{R}^*$.

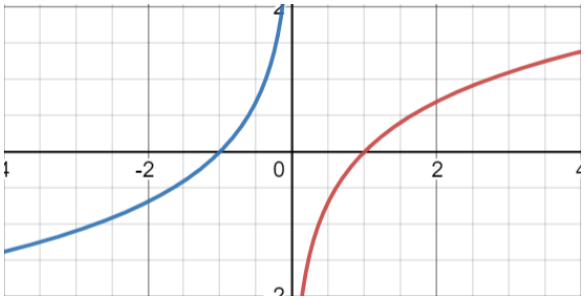

Fig 2: $ln(x)$ (red) and $-ln(-x)$ (blue)

We shift them to get a continuous function

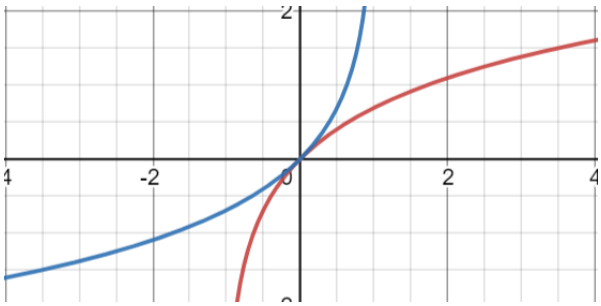

Fig 3: $ln(x+1)$ (red) and $-ln(-x+1)$ (blue)

We then define our piecewise continuous function h as

$$h: \mathbb{R} \to \mathbb{R}$$
$$h(x) = \begin{cases} -\ln(-x+1)\ if\ x < 0 \\ \ln(x+1)\ if\ x \geq 0 \end{cases}$$

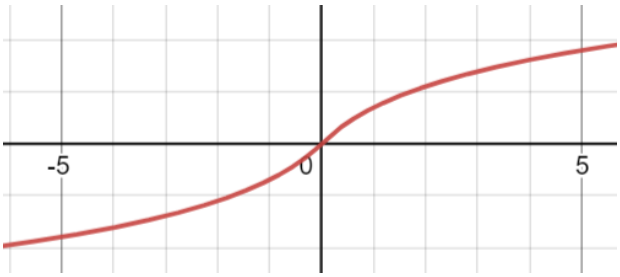

Fig 4: $h(x)$

Now since this function is defined using natural logarithms, we can input it into an exponential function and it will not go uncontrollably towards infinity or 0, but only at a steady pace in both directions, solving our limited precision problem. We can finally find our function f that gives the priority of each node to be:

$$f(x) = e^{h(x)}$$

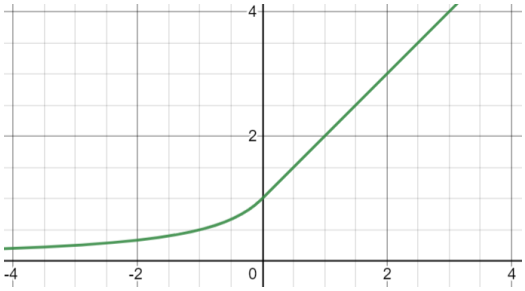

Fig 5: $f(x)$

Now this function should be good enough as is, but it still has a few tweaks we can give it. We have a min-heap, so we care more about the small values than about the big values. But we have all our small values cramped together between 0 and 1, while the big values have all the space between 0 and infinity. We can reverse this by having a few well-placed negative signs giving us our final function

$$f(x) = -e^{h(-x)}$$

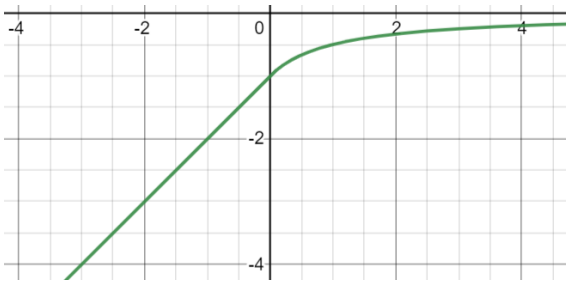

Fig 6: the new $f(x)$

Additionally, it has the advantage of making values we multiply by a number between 0 and 1 get a lesser priority in the min-heap, which is a lot more intuitive and simpler to work with.

Now getting back to our priority value P, the function we use to calculate it is

$$P = f(\hat{y} * (-1)^w) * \delta^d$$

With $\hat{y}$ the model's prediction, d the depth of the node in the search tree and $\delta \in [0, 1]$ the depth penalty that we chose to be 0.3. The deeper a node is, the less priority we give it. This parameter can be tuned: a depth penalty closer to 0 will result in more positions being considered at each step, and therefore a wider tree but that goes into less depth for each variation. On the other hand, a depth penalty closer to 1 will favorize going deep into a few variations, resulting in a narrower, deeper tree, but more likely to miss moves.

The variable w is equal to 1 if the side whose turn it is to play is white, and 0 if it is black. This is following a min-max algorithm logic: We want the network to prioritize moves that are good for itself when its his turn to play and consider the best opponent's moves when its not (meaning, the moves

that are worst for itself). The priority queue is a min-heap so the smaller the priority value, the more we want to consider that move.

We then apply a classical min-max algorithm on the tree we obtained after either a certain number of positions has been considered or a certain amount of time has elapsed and pick the move that had the highest value based on this. We only pick nodes that have at least 1 child (except if it is a checkmate), because we do not want to play moves that our tree search did not dive into.

It is important to note that we do not run the entire algorithm during training, as it is way too slow for it. Instead, we only play the move the model predicts the highest output for without diving any deeper into the tree. Considering games regularly go into the 200 moves before finishing, and with an average of 35 legal moves per position, this is already 7000 positions to run our algorithm on per game. More than this is simply too slow given our processing power.

To train our model, we let it play 10 games against itself, then for every encountered position we mirror the board in order to have both white and black perspective to double the amount of data we get, calculate the Q-value, and fit the model on it plus 10% of the positions seen in previous games for 5 epochs before repeating. To ensure the model plays different games each time, we add a random factor rf that decreases over time, and that makes it so that every prediction the model makes gets added a random value within $\pm rf\%$ of itself, to vary the move chosen and encourage confidence. The move chosen should still be among the top moves, but not always the top one, creating multiple game variations.

$$rf = \frac{1}{\ln(gameNumber)}$$

This function decreases very slowly and is between 20% and 10% for most of the training.

However, there is an issue: if the model plays multiple times very similar games during a batch, it will have multiples instances of the same data when training and start to overfit, which will make it play more confidently the same moves and diminish further the likelihood of playing a different move, and the model gets into a self-reinforcing spiral that makes it always play the same game. This happens sometimes during training, but not often.

Preliminary results

Training goes relatively well, but the model seems to stop learning relatively fast. After about 300 played games, the validation mean absolute error and loss stops decreasing and flatlines, letting us think that the model might be overfitting past this point.

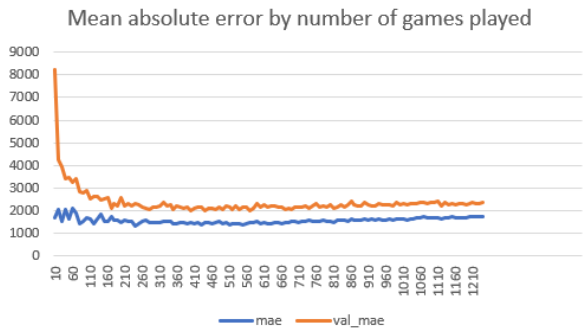

Fig 7: Evolution of the MAE along the training

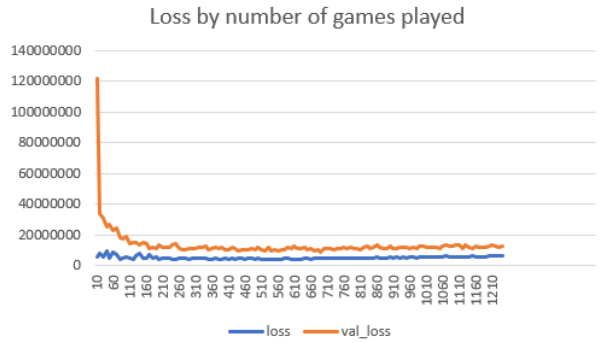

Fig 8: Evolution of the loss by number of games played

We did not have a lot of time to test the algorithm so far, however we can still already see some results.

We selected a model that trained for a total of 1200 games, and made it play a game against a 1000 rated chess.com engine, which it won convincingly in 40 moves by checkmate after finding a mate in 4. However, it did blunder at multiple points during the game. Considering that the objective was to perform at a reliable 1000 Elo level, this is very positive results. Further testing is needed to assess whether this was a fluke or not.

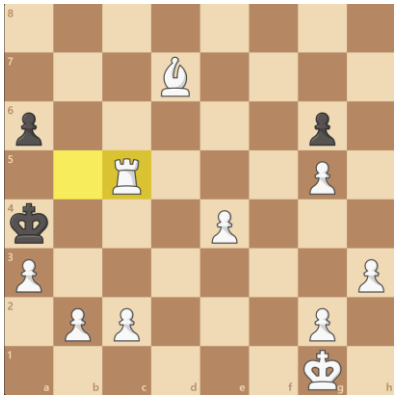

Fig 9: The checkmate found by our model against a 1000 rated bot

We also made it play against a 823 rated human player in a game that it lost in 31 moves. The model seems to perform better against other bots than against humans. Still, the chess.com evaluation attributed a 57.9% accuracy score to the model (and 70.0 to its opponent).

More time is needed to make it play more games against rated opponents and determine an actual Elo rating.

It calculates roughly 750 positions per second on our computer, which is an extremely low number, but the point of the algorithm is to privilege quality over quantity. This could probably be optimized. More rigorous testing is needed to plot this and compare it against a full search algorithm, but this will have to wait for later as we simply lack time to do this.

Next steps

It might be interesting to implement an A3C algorithm to distribute the chess-playing part of the algorithm across multiple CPU cores and consequently massively speed up training, as this is the

most time-consuming part of it. It would also allow for an increased dataset as it is generated by playing games. This, however, might prove to be too complicated and/or time consuming.

Some parameters-tuning needs to be done. Maybe through grid search, though this algorithm isn't necessarily preferred as it is extremely long to see results given how slow training is.

The network's architecture needs to be thoroughly tested and modified for optimal results too.

Modifying the way of selecting the best move resulting from our tree search would be very interesting too, as it currently is not necessarily among the nodes that have been the most explored, which defeats the purpose of the algorithm. When it blunders, it's because a move that the network did not consider because it was terrible was evaluated as good by the naïve evaluation algorithm (like when you can take a piece with your queen, but it results in your queen being immediately taken back). Since we do not give any advantage to moves that have been more deeply considered compared to others, it chooses this naïve move and blunders a piece in one. It also means that in situations where it is losing, every move that is more deeply considered will show just how bad the situation actually is and be worse than a move that has not been considered in depth, making the AI terrible at recovering from poor situations. A quick fix was to pick the best move that had been considered at more than face value by our algorithm, but it does not fully resolves the problems and is but a temporary fix.

References

[1] Silver, D., Huang, A., Maddison, C. *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature* **529,** 484–489 (2016). https://doi.org/10.1038/nature16961

[2] Andrew Cahill, Catastrophic Forgetting in Reinforcement-Learning Environments, University of Otago (2010) Catastrophic Forgetting in Reinforcement-Learning Environments (otago.ac.nz)

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis, A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play (2018), Science, 1140-1144, 362, 6419, doi:10.1126/science.aar6404