INFO8010: Application of convolutional neural network in chess evaluation

Quentin Lowette,[1] **Pierre Hockers**,[2] and **Hubar Julien**[3]

[1] *Quentin.Lowette@student.uliege.be (s150853)*
[2] *Pierre.Hockers@student.uliege.be (s152764)*
[3] *jhubar@student.uliege.be (s152485)*

I. ABSTRACT

In a game of chess, each move is made according to a strategy that aims to optimize the situation of the active player. The evaluation of the position of the different pieces is therefore a central issue that we will be tackling in this project. The goal we fixed ourselves is to use an evaluation of the board positions through a convolutional neural network (CNN). This evaluation aims to mimic the complex heuristic of the well known chess engine *Stockfish*. We will then use this learned evaluation as the heuristic in a `minimax` algorithm to implement our own chess AI.

II. INTRODUCTION

For several hundred years the game of chess has fascinated the greatest strategists. With this fascination came the desire to build machines able to master the game. In 1990 already, IBM built a supercomputer specialized in chess by adding specific circuits in order to beat the best players. Nowadays, it is no longer a question of beating the best players but of making different programs able to play this ancestral game, each of them better than the previous one.

The game of chess is seen as a zero-sum game. Indeed, a zero-sum game is a game where the sum of the gains and losses of all the players is equal to 0, which means that the gain of one player is necessarily a loss for the other. The goal is therefore to play moves that will improve your situation and worsen the one of the opponent.

The first step was to create a chess game with a graphical interface allowing the player to measure to the algorithm `minimax`. We implement it with a first intelligent evaluation of the board: the *Simplified Evaluation Function* [1]. However, even if this type of architecture is already good at the game, it suffers from disadvantages. Indeed the chess board evaluation is very basic and based on hard-coded values tables and on the number of pieces on both sides. This static scoring can be improved by using a CNN.

III. RELATED WORK

The game of chess has for a long time been a playground for AI builders. Its reputation of intellectual game brought many people to try to bring machines to perform at superhuman level, and some of them succeeded. This means that there is a lot of previous work covering this specific subject.

The simplest way to build a chess AI is through the use of a simple evaluation function. The AI is presented with the different actions it can take, and uses its evaluation function to determine which action is best. It can be based on very simple heuristics, such as how many pieces are left on the board, or attributing values to all pieces and estimate the winner according to the summed values of the pieces remaining.

The problem with such systems is that they are blind to the potential consequences of the action they choose to do. In order to prevent this, many chess AI choose to implement an algorithm called `minimax`, which explores the tree of all subsequent possible states after an action, and tries to find the best action based on which future scenario is evaluated as best [2]. This algorithm can be sped up through the use of alpha beta pruning. In this scenario, an evaluation function is used to evaluate states that are deep in the explored tree Henk Mannen 2003 [3].

Of course, the better the evaluation function, the better the AI. There exist many heuristics using some important chess features Henk Mannen 2003 [3], but it is also possible to make another AI learn this evaluation function, and notably, through Deep Learning, since neural networks are universal functions approximators [4].

In Sabatelli et al. 2018 [5] and Vikström 2019 [6], the authors try and succeed in using NN to approximate the board evaluation of the famous Stockfish engine, a strong chess AI known to attribute a very complex and accurate score to positions, predicting who is winning the game.

However it is also possible to get very creative on the manner to train such a heuristic. In Isaac Kamlish et al. 2019 [7], the authors use 2 networks to learn to evaluate board positions by extracting the sentiment analysis attributed to a move in a dataset of games commented by professionals and translating it into a move appreciation, they were able to match positions to an evaluation and train a NN on these position-evaluation pairs.

In a more classic approach, Sebastian Thrun 1995 [8] trains an evaluation function through neural network using the final outcome of the game, explanation based learning (EBL) and temporal difference (TD). EBL is done through another neural network.

It is also possible to create AI that not explicitly rely on an evaluation function, and therefore spare the use

of a minimax implementation: in this article from Sayon Bhattacharjee [9], 2 CNN are used, one to select which piece to move, and one to select where to move the selected piece. These NN were trained by using professional players moves, recording what piece the player moved and where from a given state.

Another possibility is to do as in David et al. 2017 [10]: use a network able to compare 2 positions and tell which one is better.

Finally, one could dive into deep reinforcement learning and use neural network in conjunctions with some powerful reinforcement learning algorithm, as in Henk Mannen 2003[3] where the `TD(λ)` algorithm is used. Some variation can be found in the aforementioned paper, where the powerful Giraffe engine is being described. It was implemented using deep learning with the `TD-Leaf` algorithm. This field culminates in the famous state of the art alpha zero deep reinforcement learning algorithm, which is able to learn and beat any human in chess [11].

For our project, we decided to take the first approach: we used CNN and MLP to learn an evaluation function based on Stockfish's scores that will be used in a minimax algorithm. We therefore took inspiration from [5] and [6].

IV. THEORETICAL BACKGROUND

In this part, we will describe theoretically the different concepts of deep learning we have used for this project. We will beforehand elaborate on the evaluation of a chess board.

A. Chess position evaluation

Chess is two-player game played on a square of 8 by 8 tiles. Each player starts with a set of pieces that are placed on that board. There are 6 different pieces, each piece has its own way of moving which gives it a different strength. The final goal of the game is to take the opponent's king. There are two types of color pieces on the chessboard, the white one that starts the game and the black one. Players take turns until one of them has lost his king. Different board positions will therefore represent different strength balances between the white and black players. These balances depend on the pieces still alive on both sides, on their positions, on the safety of the king and some other parameters. *Stockfish* has a complex evaluation function it uses to assess a position. This is the function our network will try to mimic.

B. Convolutional Neural Network

A CNN consists of multiple convolutional and pooling layers followed by some fully connected ones.

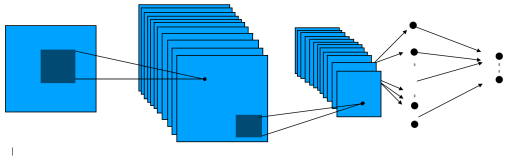

FIG. 1. Illustration of a CNN architecture

C. Convolutional Layer

A convolutional layer is a neural network layer in which an input tensor underwent a discrete convolution based on a given kernel. This kernel will slide across the input tensor and the sum of the element-wise product will be computed at each step.

This kind of layer contains many hyperparameters that allow to tune the model to the task at hand. The main ones are the following:

- **Kernel size**: This parameter indicates how many elements are considered at once in each convolution.
- **Stride**: The stride is the step with which the kernel will be moved along the input tensor. It is useful to reduce the spatial dimension of the output feature map by a constant factor.
- **Padding**: The padding refers to the amount of pixels added to an image when it is being processed by a convolutional layer. For example, if the padding is set to one, a border of one pixel of value 0 will be added to the image. In contrast, if the padding is set to zero, then no additional pixel is added on the border. This parameter allows us to also control the size of the resulting feature maps.

D. Pooling Layer

The pooling layer is another type of layer often coupled with convolutional ones. This layer is used to reduce the feature maps size while preserving their global structure. This reduction is done by summarizing the information contained in the kernel by one of its property, for example taking the average or maximum value of the elements of the tensor encompassed by the pooling kernel.

In general, pooling progressively reduces the spatial size of the representation to reduce the amount of parameters and calculations in the network. The most common approach to pooling is max pooling (see Figure 2). This is extremely useful for images that are typically large in size.

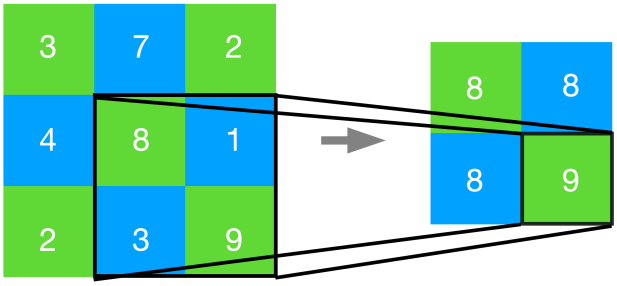

FIG. 2. Max Pooling with a kernel 2×2 over a 3×3 matrix

E. Fully Connected Layer

In CNN architectures, the network generally ends with some fully connected layers. This type of layer is composed of a stack of neurons that gathers some information and some weights and computes an output. All the outputs of the neurons form a new vector that can be used as the inputs of an other layer. An example of fully connected structure is given in the Figure 3. As a fully connected layer only works with vectors, CNN architectures also contain a kind of flattening functionality. Indeed, the tensor output of the last convolutional layer may need to be turned into a vector in order to be fed as input of the first fully connected layer.

Depending of the task at hand, the output of the last fully connected layer may vary. In the case of chess board evaluation, the problem is a regression task so the final layer contains only one neuron that outputs the computed evaluation.

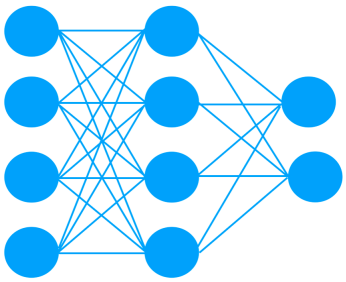

FIG. 3. illustration of the fully connected part of a convolutional network

F. Multi-layer perceptron

A multi-layer perceptron (MLP) is a class of forward-acting artificial neural network (ANN). A MLP consists of at least three layers of nodes: an input layer, a hidden layer, and an output layer. With the exception of the input nodes, each node is a neuron that uses a non-linear activation function. In addition, each neuron in one layer has connections to all neurons in the next layer.

G. Feedforward propagation

A feedforward neural network is an artificial neural network in which the connections between nodes do not form a cycle. Each hidden layer accepts input data, processes it according to the activation function used, and sends the data to the next layer. The Figure 4 shows the propagation of information along the network.

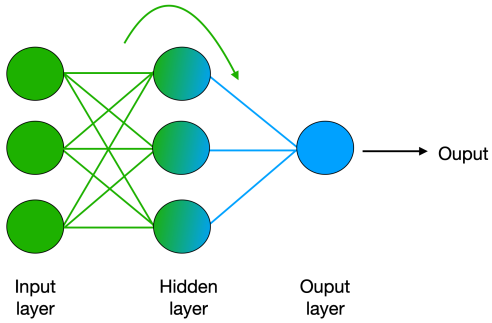

FIG. 4. Illustration of feedforward propagation

H. Backpropagation

Backpropagation is an algorithm used in the formation of early-action neural networks for supervised learning. When fitting a neural network, the backpropagation algorithm works by calculating the gradient of the loss function with respect to each weight according to the chain rule. It calculates the gradient one layer at a time, iterating backwards from the output layer to avoid redundant calculations of intermediate terms in the chain rule. This efficiency makes it possible to use gradient methods to form multi-layer networks, updating the weights to minimize loss; gradient descent, or variants such as stochastic gradient descent, are commonly used. The Figure 5 shows the modification of the weights along the network [12].

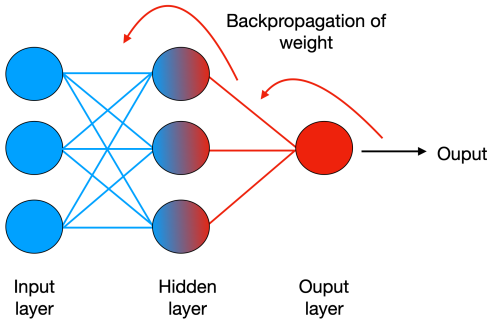

FIG. 5. Illustration of backpropagation

I. Activation functions

Inside neural networks, non-linear activation function are used on the output of a layer. These are useful to introduce non-linear complexities to the model. In this project, we used two types of activation functions: *ReLU* and *ELU*.

Rectified Linear Unit (ReLU): Activation function defined as the positive part of its argument:

$$ReLU(x) = max(0, x)$$

Visually, it looks like the Figure 6

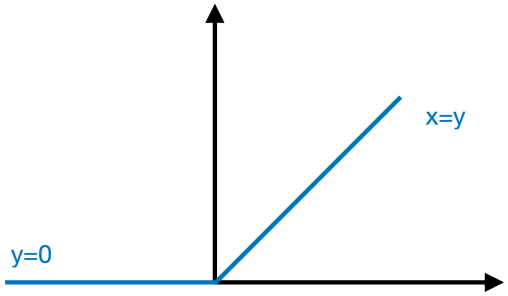

FIG. 6. ReLU graphical representation

Exponential Linear Unit (ELU): This rectifier looks very much like its neighbour ReLU. However this one has negative values, this will allow it to bring the average activation closer to zero. Consequently, it will allow to reduce the problem of the vanishing gradient. In addition, ELU lead to faster learning than ReLU.[13] The graph of this function is given in the Figure 7.

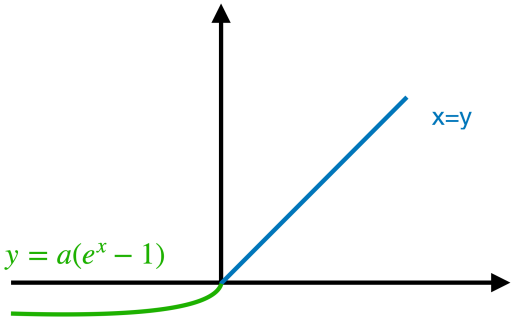

FIG. 7. ELU graphical representation

J. Batch normalisation

Batch normalization can be used inside CNN to maintain good activation and derivative statistics. At training, it will shift and re-scale the data according to the estimated mean and variance of the batch.

Moreover, on a simpler intuition, normalizing inputs that have very different natural orders of magnitudes speeds up learning. It also makes every layer a bit more independent from the others.

K. Dropout

Dropout is a technique that is designed to prevent overfitting the training data by dropping units in a neural network. In practice, the neurons are either dropped with a probability p or kept with a probability $1 - p$. This forces the network to learn in a more robust way, as it can no longer rely solely on a few "powerful" links and has to learn on a more global scale. The Figure 8 illustrates learning with and without dropout.

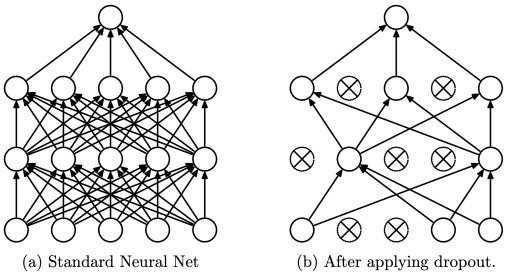

FIG. 8. Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped [14].

L. Vanishing gradient

The vanishing gradient problem is a fairly drastic decrease in the gradients during back-propagation leading to the cancellation of the latter and a halt in the learning of the network.

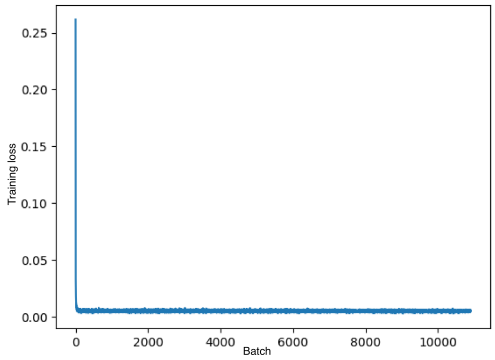

FIG. 9. Training loss of DS600K over 50 epochs

We suspect we might have encountered this problem since our networks stop learning almost immediately after one or two epochs. In the example shown in the Figure 9, the learning is carried out over our DS600K dataset during 50 epochs and with a batch size of 128. The graph is generated by plotting the training loss every 20 batches. It can be noticed that the final value of 0.006 is reached very quickly. A more in-depth study shown that this value was reached after 600 batches from the approximately 200,000 batches.

M. Weight initialization

The internal weights of an MLP or CNN can be initialize in several ways. To mitigate the vanishing gradient issue, we used the Kaiming normal initialization of `PyTorch`. This one is particularly suited when using ReLU activation functions.

V. METHODS

A. Chess position representation

Since the final goal is to train a convolutional neural network on a set of chess board positions, we need an adequate representation of this one. It must contain all the information about a given position. The idea is to use separate channels for each type of piece. Each position is represented by $12 \times 8 \times 8$ grids. Each of the 12 channels encodes only the information related to the positions of one type of pieces (color and category). For example, the 1st channel would represent an 8×8 board, where all cases are filled with 0s except where white pawns can be found, those tiles take a value of 1. The second channel would be the same except the 1s represent the presence of a black pawn. And so on for all pieces types. The Figure 10 represents the 6 white channels, the same representation is used for the black ones but reversed . Each position is thus represented by a $12 \times 8 \times 8$ tensor which is the input for the convolutional neural network.

The evaluation of a chess board during a game is heavily dependant on which players plays next, black or white. Therefore, in the datasets where we take this information into account, we only represented states where it was white's turn. To do this, each time it was Black's turn a mirror operation was performed on the chess board, and the corresponding evaluation was inverted as well.

B. The datasets

Eight datasets were used in the project. This section will detail the construction of each one of them, their advantages and their weaknesses. Each dataset was generated by extracting games from the FICS games database.

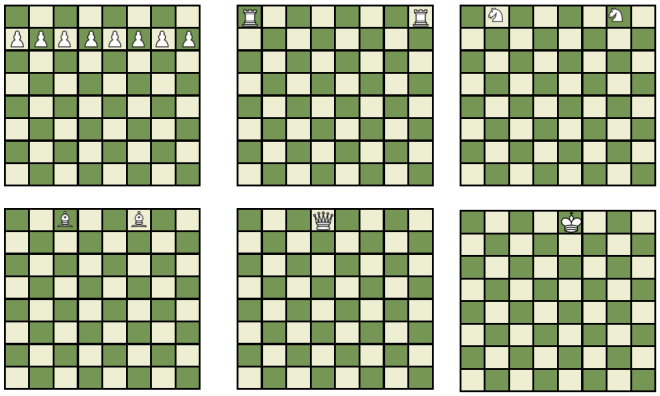

FIG. 10. Illustration of CNN in chess evaluation

Each game is then segmented into states. A state represents a board position at a moment in the game when a player must move a piece and make a decision that will impact the next moves. Note that each file is in *portable game notation* (PGN) format designating a standard chess game encoding format. Thus each PGN file underwent a transformation to extract all the states of the various games and convert them to the tensor form we mentioned above.

1. *DS3000K & DS600K*

The two datasets DS300K and DS600K are the first two datasets that were generated. Both of them suffer from a major problem as, during the state generation process, it was not taken into account that chess is a two-player game. Therefore, each generated state did not carry any sort of information as to whose turn it was. Moreover, although the generation of DS300K (300 000 states) and DS600K (600 000 states) is fast, their number of states is not high enough for a good learning curve.

2. *DS4200K & DS2800K*

The two datasets DS4200K (4,200,000 states) and DS2800K (2,800,000 states) are the two datasets that were generated to overcome the problems of the first two. Indeed, during the generation of these, information about the player's turn was included. Moreover, these are composed of many more states. Unfortunately, the DS4200K (15Go) dataset couldn't be used because it was too demanding on hardware resources. There certainly exists a solution to this problem (as some networks are trained with more data) but we didn't had much time to explore and find this kind of solution.

As a position doesn't vary much between 2 moves, it is difficult for the model to evaluate it. Indeed it can face a situation where *Stockfish* gives white a high score but,

after a bad move, it gives a very low score as black has now the advantage. It was thus crucial to include the turn in some way. This turn information was incorporated by reversing the chess board and the ground truth output when it is black's turn. The board is thus flipped and the colors are switched. This way the model only learns white's point of view. This thus requires to adapt the way we will use our model. Indeed at evaluation time, we need to flip the board whenever it is black perspective that we try to estimate.

3. *DS2800K10K, DS2800K32767 & DS2800K2048*

After multiple attempts, the network was still not learning significantly. It therefore seemed logical that the way the data was generated needed to be reviewed again. An analysis of the first 1 000 outputs of the DS2800K showed that it was very discontinuous. Indeed, as the Figure 11 shows, most of the results are not visible because there are some peaks of value around -100 000 and 100 000. These peaks are the checkmate or near checkmate configurations, which were arbitrarily put to these values to translate their very high value in game. We believe that these peaks could be at the source of some of our learning problems. Indeed, there is only one move from a "normal position" to a checkmate position, yet the associated values vary greatly. We suspected that these irregularities could harm the learning process.

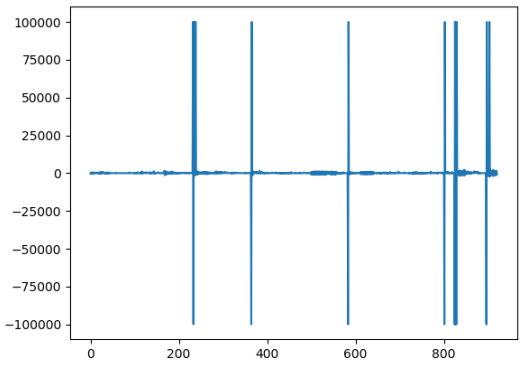

FIG. 11. 1000 first output of DS2800K

Therefore, to improve the linearity of scores between end game states, the datasets DS2800K32767, DS2800K10K and DS2800K2048 were generated by setting the values of a checkmate equal to 32 767, 10 000 and 2048 respectively. The value 32 767 comes from the *PythonChess* documentation [15]. The value 10 000 is an arbitrary value, which also attempts to increase linearity between states. Moreover, this one, although 3 times smaller than 32 7687, is generally already bigger than all other Stockfish board evaluations during a game. When the scores were analysed, it was found that they very rarely exceeded 2000. Therefore the value of a checkmate was chosen as equal to 2048. The purpose of this choice is to increase the linearity between the different states during a game and the end-game states.

4. *DS4200K2048*

Following the best results obtained with the DS2800K2048 dataset, the generation of a dataset containing 4 200 000 states with a mate score of 2048 was done. In order to be able to perform the tests on it the hardware problems had to be solved. After this, DS4200K2048 allowed to obtain the best results.

C. Network architecture experiments

During the realisation of this project, we experimented with many different and varied neural network architectures. More specifically, we organised ourselves as follow: we separated our resources to have a part of the team working on an implementation close to the one developed by Sabatelli et al. [5] and the other part of the team had to try as many different implementations of relevant neural networks as possible as well as studying the impact of several hyperparameters to try to isolate successful components. Apart from that we went through several stages of experimentation which we will be summarizing here.

The first part was devoted to the discovery of convolution network implementations. During this phase, several elements were studied. In particular, the effects of the number of layers in the network and how this influences the output results. The DS300K and DS600K datasets were used to perform these different tests. We implemented, amongst other, the following architectures:

- (Cov + BatchNorm + ReLU + MaxPool)×2 + FC Following this article [16]. This will be our **Model 1**.
- 5 conv2D layers + Pooling + 3 fully connected layers with ReLU. **Model 2**
- 8 conv2D + pooling layers + 4 FC with ReLU. **Model 3**
- (Cov + Dropout + BatchNorm + ELU)×2 + FC Following this paper. [5] **Model 4**

For all these implementations, we added dropout at all layers, with dropout probability ranging from 0 to 0.5, we tried with and without batch normalization, varied the size of kernels, changed our ReLU activation to an ELU activation function, toyed with the learning rate (from 0.001 to 0.1), tried to combine some of these hyper parameters, tried to train the model for 2 to 1000 epochs and so on.

No matter what network configuration or dataset was chosen, the resulting MSE was very poor, reaching

($\Theta 10^8$). We suspected that the following factors could explain these results :

- the board evaluations of the datasets used here are between 100 000 and -100 000, and we use a mean squared error, which obviously brings errors in high values.
- The two datasets used are not very large, only a few hundreds thousands elements.
- The datasets did not take into account whose turn it is, keeping the network blind to a very important information.

In order to make this output more interpretable by the NN, we tried to normalize the *Stockfish* evaluations between $[0; 1]$. Our loss then stabilized at around 0.05 for the **Model 4** and around 0.25 for our other experimental architectures. We therefore obtain the graph of the Figure 12.

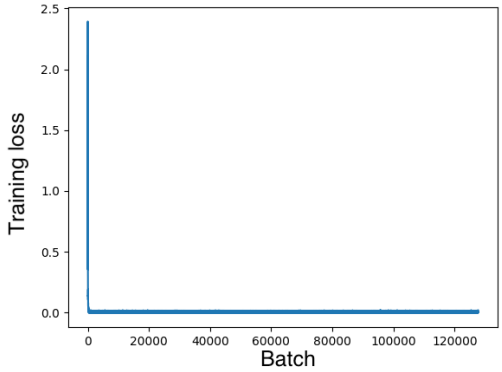

FIG. 12. Training loss after normalization

When confronted with these very static results we started to consider the data as the main problem source. That's when we realized the datasets did not take into account the notion of whose turn it was, information of prime importance. We also took this opportunity to generate much more training data. This is how we ended up with the dataset DS2800K.

This change in dataset lead to considerable changes: the **Model 2** and **3** begin to oscillating much more and the loss could evolve between 2 epochs by a factor of 1000. We also noticed that this time the dropout lead to a slight enhancement of the loss in our experimental architecture when the dropout probability was between 0.1 and 0.3. The rest of the architecture didn't seem to make the results vary much. In contrast, the **Model 4** outputs better losses as this time we reached 0.005 at training time. However we still had the same kind of graph with a quick learning and then no more evolution.

From there we also started to build other kind of experimental networks: we built MLP, of varying lengths. We also built convolutional networks without the pooling layers, as these were known in the literature to be a poor choice due to their local invariance properties, we created deep convolutional networks (10 convolution layers with pooling and 4 fully connected layers with ReLU) and we copied the implementation of Joel Vikström [6].

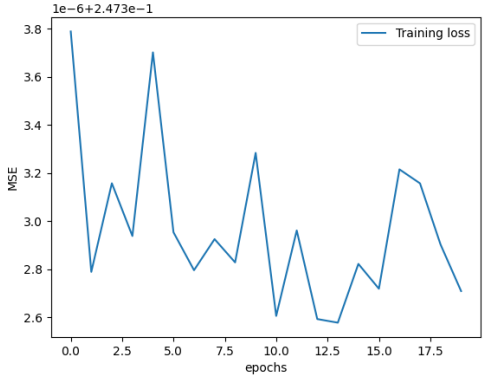

FIG. 13. Training loss from a [6] inspire architecture

None of these sounded to bring much if anything at all, we stayed at the same level of loss, and our network sounded to still not learn over epochs. We started to believe that it might be a case of vanishing gradient, and worked on solving this. We therefore decided to apply batch normalization and to initialize the weights of our networks. With a ReLU activation function, our research lead us to use a Kaiming initialization. These added implementations still wouldn't bring any noticeable results.

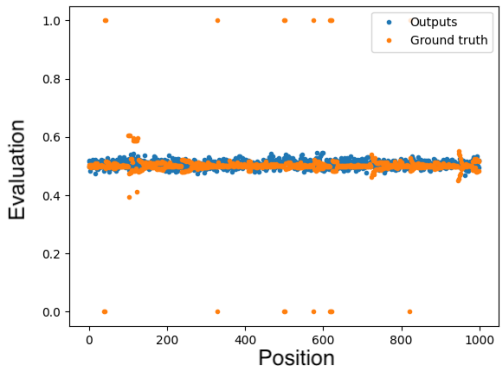

FIG. 14. Outputs and ground truth of the test set with discontinuity

We then again reconsidered our data. When analysing the graph of the Figure 14 We realized that the predicted results were staying very close to the median values, as some kind of safe bet for the network to minimize the loss. The graph of the ground truth also seemed strange, as most values were around 0.5 while a couple of them were extremely high. We realized that by setting the value of a checkmate to 100 000 we had created a real discontinuity

between the states values pairs of our dataset. We assume that this could hurt our learning process. We thus created 3 new datasets, where the value of a checkmate would be this time set to 32 767, 10 000 and 2048.

The resulting graph is the one shown in the Figure 15. It can be noticed that the ground truth we try to mimic is more continuous. In particular, we can observed some regular vertical alignments of points, those represent endgames that are highly disputed.

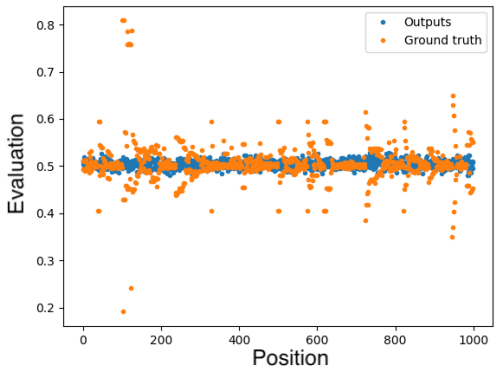

FIG. 15. Outputs and ground truth of the test set with less discontinuity

With the DS2800K2048 dataset, the improvement was noticeable: we reached a final training loss of 0.002 and an average test loss of 0.0015 with our **Model 4**. However it did not bring much on the other architectures.

Encouraged by these results, we generated a final dataset of over 4.2 million states, with a 2048 checkmate value. We this time obtained an average test loss of 0.0009 with the same architecture.

D. Best architecture

As described above, our best architecture is the one called **Model 4** and derived from the work of Sabatelli et al. [5]. More specifically this model composition is given in the Table I.

VI. RESULTS

We will now present and discuss our main results. In order to assess the quality of our models we have split our dataset to have 90% of it dedicated to the learning phase and 10% to the testing phase. This testing dataset is used to compute an average MSE loss over all its elements. This average is the score we based ourselves on to evaluate and compare our models.

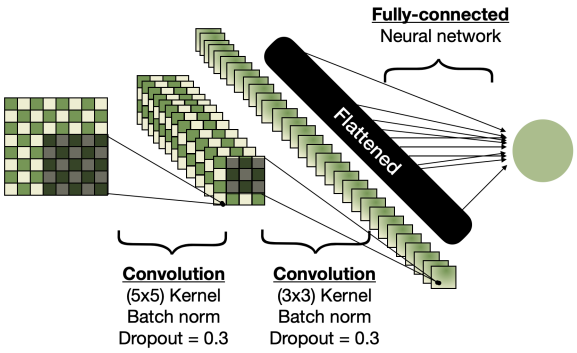

FIG. 16. Graphical representation of our **Model 4**

Layer type	Output shape
Conv2D	[-1, 20, 4, 4]
BatchNorm	[-1, 20, 4, 4]
ELU	[-1, 20, 4, 4]
Dropout	[-1, 20, 4, 4]
Conv2D	[-1, 50, 2, 2]
BatchNorm	[-1, 50, 2, 2]
ELU	[-1, 50, 2, 2]
Dropout	[-1, 50, 2, 2]
FC	[-1, 1]

TABLE I. Detailed view of the architecture of the **Model 4**

A. Qualitative analysis

As explained in the previous section, our best result was obtained with our **Model 4** over the DS4200K2048. We used this model to play with our GUI, as the heuristic of minimax. We noticed that the resulting AI was easily beaten. It tends to not foresee obvious counters to its moves that result in the loss of important pieces, such as the Queen. We however noted some positive elements:

- The opening game of **Model 4** is of rather good quality, making book openings and reacting smartly to opponents starting moves. These openings are reminiscent of the Gambit or Sicilian Defence. These last ones are very well known in the chess world See figure 17).
- The AI is perfectly able to capture a piece that represents an obvious opportunity.
- The AI played some smart defensive moves when put under pressure, including a well thought castling.
- When faced with a beginner player, the AI will be able to hold its ground fairly until late midgame, where it seems to crumble down and get confused on what to do

- This model behaves way better than our less good models, the improvement over **Model 4** with a dataset with checkmate values set to 32767 for example is massive.
- It also seems that the AI has trouble gauging piece values and is able to give a queen against a pawn (See figure 18).

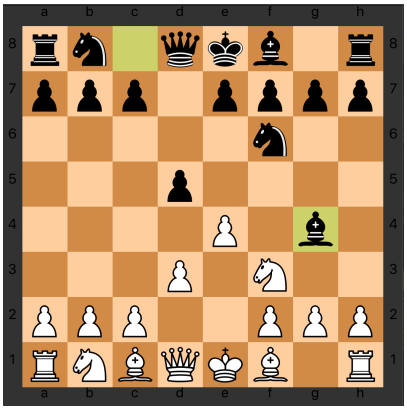

FIG. 17. Phase 1: Opening of a game

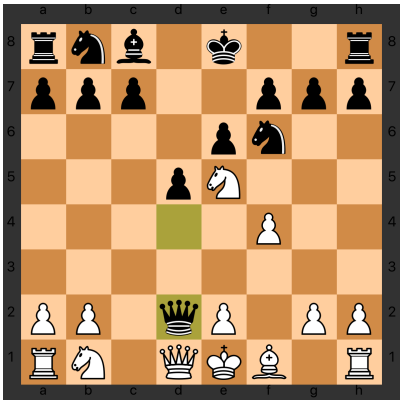

FIG. 18. The queen takes a pawn without any counterpart.

B. Quantitative analysis

As previously mentioned, our best model is the **Model 4** on the DS4200K2048 dataset. The Figure 19 shows the evolution of its training loss in blue and the average loss over the epochs in orange. It can be observed again that the model reaches quickly a plateau and doesn't learn much from there. As the orange curve illustrates, we could have stopped after 2 epochs as there isn't a significant improvement between the second and third epoch.

When we look at the results of this architecture over the test set, we get the Figure 20 which shows that our model stay in the safe zone of the mean value 0.5. However, in comparison with other models, it is more spread around this value. This might explain why the opening's moves, associated with more average scores, are better handled than the endgame's more diverse and unbalanced positions, which are therefore associated with more outstanding values.

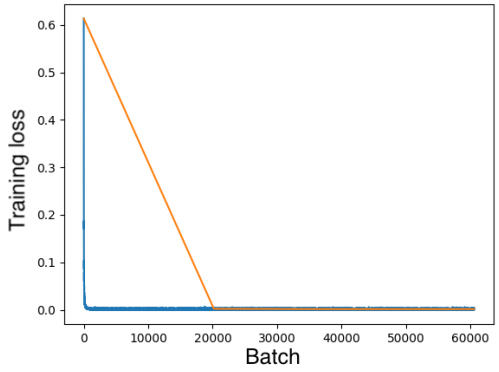

FIG. 19. Training loss of our best model

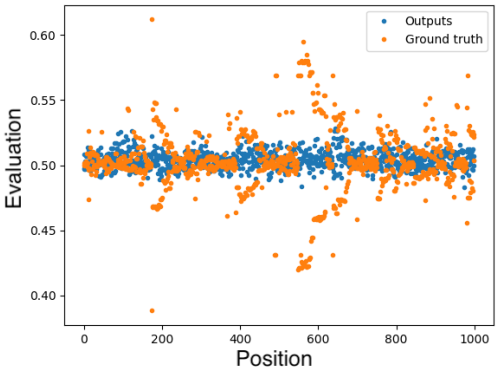

FIG. 20. Results on the test set for our best model

Architecture	Dataset	Average MSE
CNN Vikström	DS2800K2048	0.247394
Model 4	DS2800K2048	0.001550
Model 4	DS4200K2048	0.000900

TABLE II. Average MSE for different models and datasets

We have compiled the various end testing loss of the more relevant models in Table II. One can realise that the results are indeed better with **Model 4** when using a dataset of bigger size, and that this implementation gets far better results than the Vikström architecture [6].

VII. DISCUSSION

We have implemented a CNN that learned to approximate the evaluation function of Stockfish. We managed to reach a testing loss of 0.0009 with our final and best implementation, thanks to our dataset of 4.2 million states with checkmate values of 2048.

The result is an AI able to play chess but at a very low level, as discussed in the result section. We theorize that it could be due to the distribution of the predictions of the network. Indeed we can observe that most predictions still remain close to the mean of 0.5, while the ground values vary much more especially in unbalanced games situations, which can for example happen in end game. We believe that because the network still can't really understand fully what makes a really advantageous/dangerous situation, it is unable to select an action leading to these advantageous situations or to avoid falling in dire positions. Therefore the amateurism of the produced AI could boil down to its inability to distinguish an average situation from a good or bad one.

It is however surprising to realise that the MSE we obtain is quite low, which should translate to accurate evaluations. These two observations are therefore in contradiction.

Given the results we have, we conclude that despite some underwhelming results we managed to deliver an AI that has understood some basics of the game. We believe that using a bigger dataset could lead to improvements. The selection of the value associated to a checkmate could also be refined through more thorough optimization, potentially leading to better results as well.

We however do not exclude that some other cause that we would not have thought about could improve our learning process altogether, maybe some data representation element missing or needing a reformulation.

[1] Chess Programming. Simplified evaluation funciton.
[2] Wikipedia. Minimax.
[3] Henk Mannen. Learning to play chess using reinforcement learning with database games.
[4] Noboru Murata Sho Sonoda. Neural network with unbounded activation functions is universal approximator.
[5] Valeriu Codreanu Matthia Sabatelli, Francesco Bidoia and Marco Wiering. Learning to evaluate chess positions with deep neural networks and limited lookahead.
[6] Joel Vikström. Training a convolutional neural network to evaluate chess positions.
[7] Nicholas McCarthy Isaac Kamlish, Isaac Bentata Chocron. Sentimate: Learning to play chess through natural language processing.
[8] Sebastian Thrun. Learning to play the game of chess.
[9] Sayon Bhattacharjee. Predicting professional players chess moves with deep learning.
[10] Wolf David, Netanyahu. Deepchess: End-to-end deep neural network for automatic learning in chess.
[11] Arthur Guez Marc Lancto Julian Schrittwieser Thomas Hubert David Silver Matthew Lai, Ioannis Antonoglou. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play.
[12] Wikipedia. Backpropagation.
[13] Thomas Unterthiner & Sepp Hochreiter Djork-Arné Clevert. Fast and accurate deep network learning by exponential linear units (elus).
[14] Alex Krizhevsky Ilya Sutskever Ruslan Salakhutdinov Nitish Srivastava, Geoffrey Hinton. Dropout: A simple way to prevent neural networks from overfitting.
[15] Niklas Fiekas. Python-chess.
[16] Pukit Sharma. Build an image classification model using convolutional neural networks in pytorch.