Static Evaluation of Chess Positions with Convolutional Neural Networks

DD2424 Project Report

Bröndum, Eric brondum@kth.se

Torgilsman, Christoffer torgils@kth.se

Ågren, Wilhelm wagren@kth.se

Abstract

This paper proposes a CNN model for static evaluation of different chess positions. Both a classification model and a regression model were implemented. The classification model was implemented in order to validate that the architecture was suitable given the generated data, and that the training procedure was stable. The regression model was then used to evaluate and give centipawn-scores on the different chess positions. After fine-tuning the CNN model, the final results of the best model was 3.76 Mean Absolute Error (MAE). The final model was compared to the 'Stockfish 13' engine, and the results were satisfying. The main drawback of the CNN model was its ability to evaluate more complicated positions involving tactics. To improve this one can train the model more on these rare positions, since it mainly has been trained on early- or mid-game positions with centipawn-scores close to a draw.

1 Introduction

The field of deep learning has made a huge impact on todays state of the art chess engines. Limited look-ahead and Monte Carlo approaches to minimax has greatly increased chess engines' abilities to evaluate positions through optimized search algorithms. Human chess grandmasters have a great ability to look ahead at every move, much like the chess engines, but what they also possess is an ability of *static evaluation*. The aforementioned chess engines heavily rely on looking at future moves in order to evaluate current positions, but how well can you create a model for static evaluation? This report presents an approach to static evaluation by utilizing a Convolutional Neural Network (CNN) with no prior knowledge of the rules of chess.

The problem of static evaluation can be defined as a regression problem, but an easier starting point is to create a model suited for classification. The reasoning behind this approach was that it would be easier to determine the correctness of the proposed model. Both binary- and categorical classification was tested in order to determine the capabilities and correctness of the model. An initial model was experimented on but results indicated that it was too complex and demanded both too much time and data to train. The second model proposed was less complex, had a more stable training/evaluation procedure, and was much faster to train. This model achieved a testing accuracy of 83.80% on the binary classification task, and a 67.66% testing accuracy on the categorical classification task (7 classes). The second proposed model was deemed suitable and we moved on to tackling the initial problem of static evaluation, using a regression model.

The regression model performs well, although it might be slightly underfit on certain positions of chess due to an imbalance in the generated data. The model is in general bad at noticing the features of tactics in positions, but does a pretty good job at giving a fair centipawn evalution. The model was not expected to give an accurate centipawn score compared to the state-of-the-art chess engines whenever it is evaluating a less common position.

Project work (group 70) for course DD2424 Deep Learning in Data Science at KTH, Stockholm, Sweden.

2 Background

Chess is a zero-sum game where two players utilize different pieces on an 8x8 board in order to ultimately capture the adversaries king, or end the game in a draw. Since it is a zero-sum game one can apply many different game theory algorithms to find optimal strategies and moves; such as minimax, iterative deepening depth-first search, alpha-beta pruning and more. The minimax algorithm is reliant on a heuristic function, γ, which gives a numerical evaluation to the current position. This is the case of *static evaluation* which means that the function takes the current state/position and outputs a single value [1].

2.1 Problem

The task in this project was to perform *static evaluation* and model the γ function by using a CNN trained on a large variety of recorded chess games. The problem was defined as a regression problem; where the target values are the position evaluations found by the chess engine 'Stockfish 13' [2].

2.2 Prior work

Multiple previous studies has been performed evaluating the performance of neural networks in chess. Such as deep models for position evaluation and classification, as well as using it to predict the next move. One such study was made by B. Oshri and N. Khandwala [3] were they trained a three layer CNN to predict the next move using high ELO players games as training data. The training data contained 8x8x12 features and the end result was that it allowed them to achieve a validation accuracy of 38.3%. They also tested their network against the Sunfish Chess Engine where their network managed to draw 26 games out of a hundred.

Another study that trained a CNN that could predict the next move was made by S. Bhattacharjee [4]. He also preprocessed his training data so that each position was represented by 8x8x12 features, but his architecture only used two Convolutional layers. The accuracies that his model managed to achieve was lower in comparison to the model created by [3], only managing to achieve a validation accuracy of 34.8%.

More relevant work for this project has been done in the article [5] where they made a classifying two layer CNN but with pooling layers. They also preprocessed their training data differently and transformed every position so that it could be one-hot encoded into 8x8x7 features. Using this network for binary classification they managed to achieve a validation accuracy of 73.5% and a test accuracy of 71.8%.

The most relevant study for this project is the one performed by M. Sabatelli et al. [6] where they trained a classification CNN and a classification Multi Layer Perceptron (MLP) and compared their performances. Afterwards they also trained a regression CNN and a regression MLP and compared the performance here as well. They preproccessed their training data so that every chess position was represented as 8x8x12 features. They tested both algebraic inputs assigning values to each piece and bitmap input representing each piece as one-hot encoding and found that bitmap inputs achieved higher accuracies.

3 Implementation, method and experiments

The project implementation is divided into different parts corresponding to the part of the project of which they impacted. All of the code during the project was written in `Python 3` and utilized some of the most well known ML libraries, e.g. `numpy`, `pandas`, `TensorFlow/Keras` etc.

3.1 Data generation and pre-processing

To acquire a large number of positions on which the model would be trained with, publicly played chess games were downloaded from the FICS Games Database [7]. The data was supplied in Portable Game Notation (PGN) and thus had to be pre-processed and converted to an internal representation.

The data was modeled such that for each square on the 8x8 chess board there was a corresponding feature vector. The feature vector was of length 7, where the first 6 integers is a one-hot encoded representation of what type of piece is present on the square. The sign of the one-hot encoding determines the piece color, where negative is black and positive is white. The last integer represents which players turn it is to make a move in the position, where a -1 means it is blacks turn to move and a 1 corresponds to whites turn to move. This ultimately leads to each chess position being represented as a 8x8x7 feature vector only containing discrete numbers from $\{-1, 0, 1\}$.

A python script was written which would output a compressed pandas DataFrame that contain all of the extracted positions from the downloaded PGN file. In order to clamp the corresponding label to each data sample the python library 'python-chess' [8] was used together with the open source chess engine 'Stockfish 13' [2]. python-chess handles the communication with the external chess engine and also keeps track of the parsed games whilst supplying a Forsyth–Edwards Notation (FEN) representation of the position. The FEN string was then supplied to a handcrafted parser which translates the string to the correct 8x8x7 data representation. Due to not having a streamlined parser publicly available, the parsing of the data is not optimized and thus takes some time to perform. The amount of data available in the project is thus limited to a realistic limit of ~ 1 million positions.

3.2 Data imbalance

For the different problems, categorical classification and regression, there is exists the problem of data imbalance. See Appendix A for statistics.

This is not as notable for the binary classification task, and there is approximately a 45/65 split between the data. For the categorical problem the data is centered around 0, i.e. a draw position, which means there is clear under-representation for the outer classes. There is a $\sim 30\%$ baseline acc for the draw class.

The same thing can be noted for the regression problem, where the data distribution is similar to an extreme Gaussian distribution with extra outliers. The reason for this is the general distribution of positions in a chess game. On average the majority of positions will be close to drawn and often whenever one players starts to lead, that lead only becomes larger. Normalizing the data, with Studentized residiual [9], does not necessarily alter the distribution of the data; but simply scales the loss and predictions instead.

Synthetic Minority Over-sampling TEchniques (SMOTE) were attempted in order to deal with the imbalance in the categorical data. These attempts were not successful due to the sheer size of the data, and it was expected to take ~ 200 hours in order to accurately apply SMOTE to the data [10].

3.3 Classification problem

The first step towards modeling the proposed CNN was to simplify the problem and treat it as a classification problem instead of a regression problem. Rather than finding an exact numerical value for the chess position, the classifier's aim was to predict which player that is currently winning.

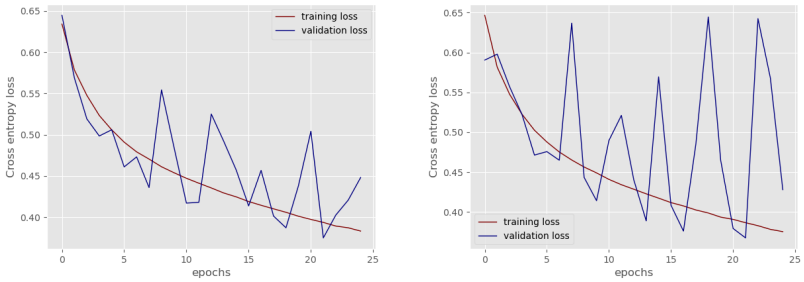
3.3.1 First model

To have a starting point and frame of reference for the CNN model, the article "*Chess position evaluation with convolutional neural network in Julia*" [5] was followed thoroughly. The author in the paper proposes a large scaled model with ~ 500 thousand parameters to tune, and 20 million samples of data available for training, validation and testing. The architecture for the model was of the form [conv-relu-pooling]x2-[affine-sigmoid-dropout]-sigmoid and the convolutional layers had respectively 400- and 200-filters and kernel sizes (4, 4) and (2, 2). The authors achieved a final testing accuracy of 71.80% and deemed the model okay for classifying chess positions but also mentions the room for obvious improvements.

One immediate limitation for this project was the amount of data that could be generated and pre-processed in a reasonable time frame, and also the time which we could spend training the models. The attempts at training such a large model as [5] with the available data for this project were reasonably satisfactory. The model performed well on the final testing data but it took way too long to train and the model performed inconsistently on the validation data. One proposed change to the original model was to introduce sensitive weight initialization on all layers with tunable parameters. This change proved somewhat helpful in promoting a slightly more stable training procedure, as can be seen in figure 1.

The author of [5] used Stochastic Gradient Descent (SGD) as optimization technique but this was changed to AdaGrad for our model. This change was motivated by the fact that the generated data was rather unbalanced, which meant that some features were more frequent than others and as a result the parameters associated with that feature would otherwise get updated more often/more.

Training the model with Adagrad and hyperparameters: learning rate $\eta = 0.01$, $\epsilon =$1e-7, and batch size $= 128$, yielded a final testing accuracy of 86.10% after training for 25 epochs. The evolution of the loss and accuracy can be seen in Appendix B.



(a) He Normal weight initialization.

(b) No sensitive weight initialization.

Figure 1: CNN model [5] weight initialization comparison.

Our model with some proposed changes outperforms the initial one which was followed, but the highly increased testing accuracy might be caused by a relatively smaller testing batch. There were also some other concerns which were discussed, regarding the use of Pooling layers, dropout, and the stability of the training process.

3.3.2 Issues with the first model

As can be seen in the loss and accuracy evolution figures for the first model, the validation results are extremely inconsistent and unstable. This is most likely an issue caused by the fact that the model has half a million parameters to tune and we only have $\sim$ 700 thousand positions in our training set. To both deal with this issue of instability and the large amount of time it takes to train, we instead propose a smaller CNN model similar to the one proposed by B. Oshri et al. in "*Predicting Moves in Chess using Convolutional Neural Networks*" [3].

Using Pooling layers in a CNN assumes independence of locality. This works great for image related problems, but a pawn in the middle of the chess board is not equal to a pawn on the side of the board. Transformations of the position is not interesting since this would yield a different state of the board, and thus a different final outcome of the position [6] [4] [3]. It was therefore decided that the new model would not feature any pooling layers, in order to preserve the locality of the features, and to decrease the size of the model.

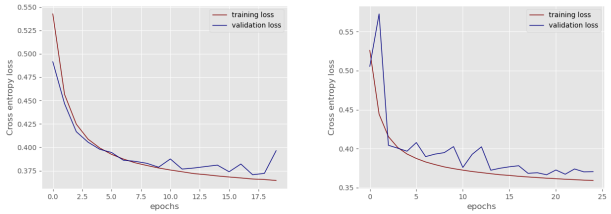
Dropout has the perks of promoting a stable training procedure and a robust and general final model, but this comes at a cost of dropping some potential important activations in the affine layers. This could lead to the model dropping certain activations which leads to the model missing potential key features of positions [3]. It was decided to try the next model both with and without dropout in order to determine its importance, and impact on training, for the given problem.

3.3.3 Second model

The second CNN model proposed was of the form [conv-relu]-[affine-dropout-affine]-softmax and is based on the architecture proposed by B. Oshri et al. in [3]. The convolutional layer had 32 filters with kernel size (5, 5) and the two affine layers both had 128 fully connected neurons. This three layered CNN contains $\sim$ 88 thousand tunable parameters. It was also trained using AdaGrad with the same hyperparameters as before.

Initial results with this model were surprisingly good for such as small network. The training procedure was significantly more stable and the model was much faster to train. The testing results both with and without dropout was similar to the first model, i.e. around 83% accuracy. Even though both we and other sources, e.g. [3], discuss the effects of dropout and come to the conclusion that perhaps it shouldn't be used in order to preserve all activations; the results from our experiments clearly showed the importance of dropout in order to promote a more robust and stable training procedure. It was therefore decided to keep dropout in the model, even though it may decrease the general amount of features learned during training. See `figure 2` for results on the effect of dropout.

The second model achieves a final testing accuracy of 83.80% after training for 40 epochs on the binary classification problem. The loss and accuracy evolution can be seen in `Appendix B`. To truly test the capabilities of the model, a second experiment was performed. We expanded the classification



(a) Second model loss, with dropout. (b) Second model loss, NO dropout.

Figure 2: The effects of dropout on second model, a comparison.

task to 7 classes, instead of 2, where the classes were `[black---, black--, black-, draw, white+, white++, white+++]` and indicates whether its a draw or any of the two players is leading by a certain amount. This model achieves a testing accuracy of 67.55% when trained for 40 epochs. See `Appendix B` for results.

These results clearly indicate that the second proposed model is more suitable for the problem at hand given the generated data. It was now time to tackle the intended problem, static evaluation of a chess position as regression.

3.4 Regression problem

When transforming the CNN model for the regression task the final layer is modified such that the output is now one single continuous value and there is a linear activation function instead of softmax. Thus the model is of the form [conv-relu]-[affine-dropout-affine]-linear. Cross Entropy loss is no longer a valid loss function to use and the problem now is to minimize the Mean Square Error (MSE) loss for the network. Since there are clear outliers in the data the MSE, or Mean Absolute Error (MAE), is going to be rather skewed which leads to potentially larger gradient updates than wanted. To deal with this one can instead use the **Huber loss** L_δ,

$$L_\delta(\mathbf{y}, \hat{\mathbf{y}}) = \begin{cases} \frac{1}{2}(\mathbf{y} - \hat{\mathbf{y}})^2 & \text{for } |\mathbf{y} - \hat{\mathbf{y}}| \leq \delta \\ \delta|\mathbf{y} - \hat{\mathbf{y}}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

which defines the loss to be MSE whenever the absolute difference is less than or equal to the threshold δ and MAE otherwise. The choice of δ is extremely important since it more or less determines what the loss function will consider to be an outlier in the data. δ was determined to initially be set to 1 since the majority of targets lie in the range $[-1, 1]$.

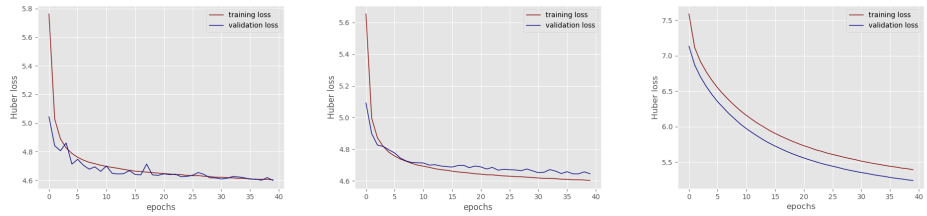
3.4.1 Hyperparameter tuning

Given that we find the model architecture suitable for the problem at hand, an assortment of regularization techniques and hyperparameters were tested in order to conclude the optimal settings for training the regression model.

A comparison experiment between `L1` and `L2` regularization was performed. This kernel regularizer was added to the two affine layers in the model. The results clearly showed that `L1` regularization is not suitable for the given data, and that `L2` regularization shows similar results to not using any regularization. See `Appendix C` for results. It was therefore decided to only use dropout.

We tried to fine-tune the δ parameter for the huber loss. Since the majority of data points lie in the range $[-1, 1]$, we tried the values $\delta \in \{0.5, 1, 2, 5\}$. The results indicated that smaller values for δ has no greater impact on the testing performance, but is instead just a scaling factor of the loss during training. It was concluded to keep $\delta = 1$.

The learning rate η was optimized. Three different values $\eta \in \{0.1, 0.05, 0.001\}$ were experimented with. Results were that a lower learning rate leads to more stable training but evidently needs more epochs to train in order to reach convergence. Because of this we favoured a slightly larger learning rate, $\eta = 0.05$, which gave us a good trade-off between the time to converge and stability. See `figure 3` for results.

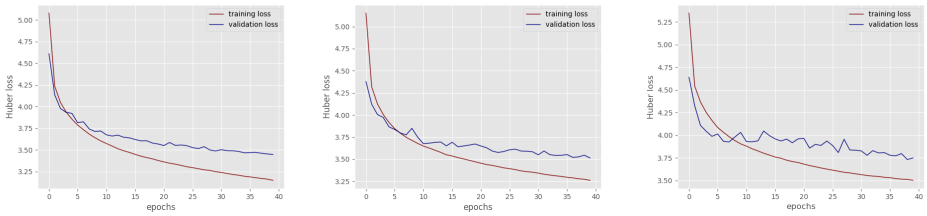


(a) $\eta = 0.1$. (b) $\eta = 0.05$. (c) $\eta = 0.001$.

Figure 3: Learning rate comparison. Dropout=0.3, $\delta = 1$, 40 epochs training, linear activations.

We experimented with the activation function in the affine layers. The functions were: ReLU, ELU, and tanh. ReLU performed poorly, while ELU and tanh performed similarly. Using ELU and tanh made the validation loss to be increased compared to using linear activation, but the models achieved lower MAE on the testing data. ELU achieved 3.8 MAE and tanh 3.88 MAE. It was therefore concluded to use Exponential Linear Unit, ELU, in the affine layers. See `Appendix C` for results.

In order to find a suitable parameter value for the dropout layer, we tried three different values $\{0.1, 0.2, 0.3, 0.5\}$. The results showed that a larger dropout % leads to higher MAE and not as stable training. Having a dropout rate $\{0.1, 0.2\}$ yielded quite similar MAE but dropout=0.1 achieved the most stable training so it was concluded to use this dropout rate.



(a) Dropout rate 0.1, MAE=3.87 (b) Dropout rate 0.2, MAE=3.92 (c) Dropout rate 0.5, MAE=4.16

Figure 4: Dropout rate comparison, 40 epochs training, $\eta = 0.05$ and $\delta = 1$, activation function=ELU.

The final model is thus of the form [conv-relu]-[affine-elu-dropout-affine-elu]-linear, with hyperparameters: $\eta = 0.05$, $\delta = 1$, dropout rate $= 0.1$, batch size = 128. After training with AdaGrad for 60 epochs it achieves a final testing result of 3.76 MAE.

4 Conclusions

We are satisfied with the results of our regression CNN for *static evaluation* of chess positions. As can be seen in `Appendix D`, the CNN performs reasonably similar to 'Stockfish 13'. Of course it is not perfect in any way, and it is rather clear that it gives skewed evaluations for less common positions. Since the training/validation/testing data mainly consists of equal positions without too much complexity to them, the model is heavily trained on these types of positions. To improve the model one can generate more data with un-common positions such that the CNN model is taught those types of features as well. Ultimately, we believe that it is more a problem of quantity and not quality. The proposed model architecture and data dimensionality works good for both us and others who have performed similar experiments.

Finally, we believe that we have learned a great deal of practical skills regarding large data-sets, modelling, training and evaluating a CNN. Even though there are clear room for improvements in the project, we declare it successful since it performs sufficiently good compared to 'Stockfish 13'.

References

[1] *Minimax*. Wikipedia. May 2021. URL: https://en.wikipedia.org/wiki/Minimax.

[2] Tord Romstad, Marco Costalba, and Joona Kiiski. *Stockfish 13*. May 2021. URL: https://stockfishchess.org/about/.

[3] Barak Oshri and Nishith Khandwala. "Predicting Moves in Chess using Convolutional Neural Networks". In: 2015.

[4] Sayon Bhattacharjee. *Predicting Professional Players' Chess Moves with Deep Learning*. Nov. 2018. URL: https://towardsdatascience.com/predicting-professional-players-chess-moves-with-deep-learning-9de6e305109e.

[5] *Chess position evaluation with convolutional neural network in Julia*. int8. 2021. URL: https://int8.io/chess-position-evaluation-with-convolutional-neural-networks-in-julia/.

[6] Matthia Sabatelli et al. "Learning to Evaluate Chess Positions with Deep Neural Networks and Limited Lookahead". In: Jan. 2018. DOI: 10.5220/0006535502760283.

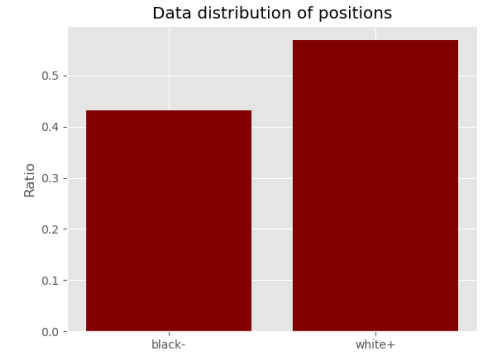
[7] *FICS Games Database*. Free Internet Chess Server FICS. Apr. 2021. URL: https://www.ficsgames.org/.

[8] Niklas Fiekas. *python-chess*. May 2021. URL: https://python-chess.readthedocs.io/en/latest/.

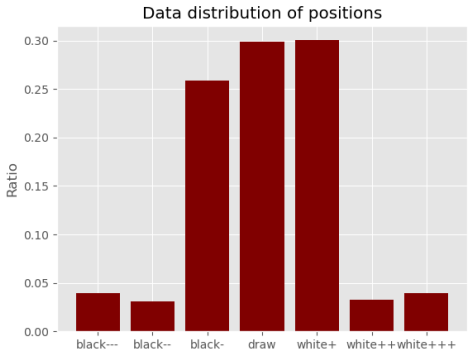
[9] *Studentized residual*. Wikipedia. Dec. 2020. URL: https://en.wikipedia.org/wiki/Studentized_residual.

[10] Tuanfei Zhu, Yaping Lin, and Yonghe Liu. "Synthetic minority oversampling technique for multiclass imbalance problems". In: *Pattern Recognition* 72 (2017), pp. 327–340. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2017.07.024. URL: https://www.sciencedirect.com/science/article/pii/S0031320317302947.

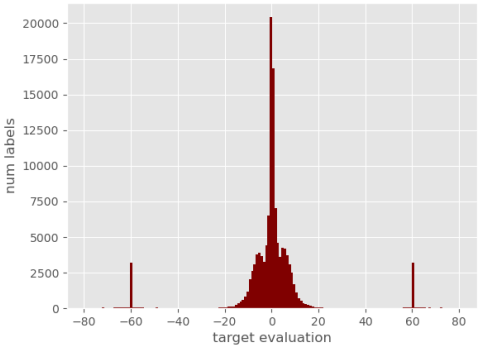
A Data distribution



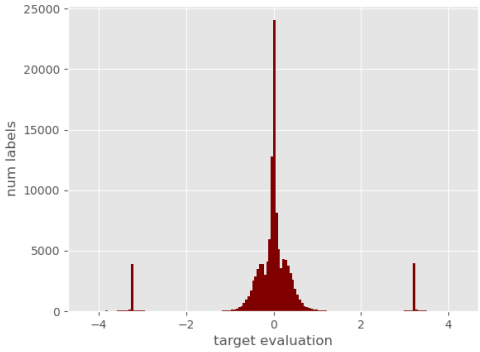
(a) Binary classification.



(b) Categorical classification.



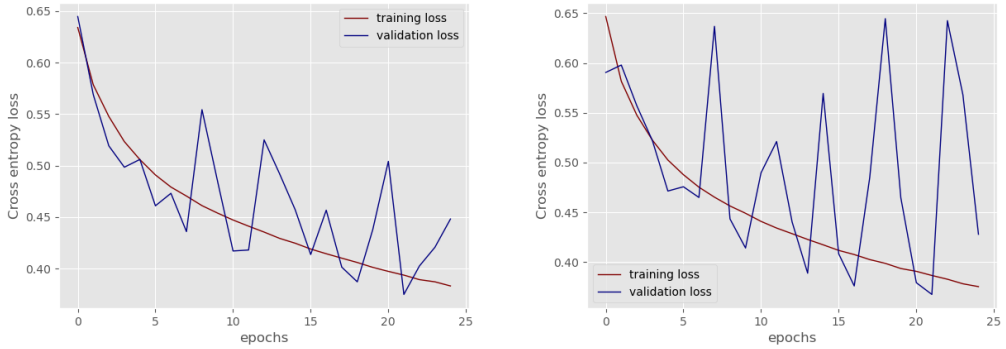
(c) Regression target values.



(d) Regression target values studentized.

Figure 5: Data distribution for the different problem formulations.

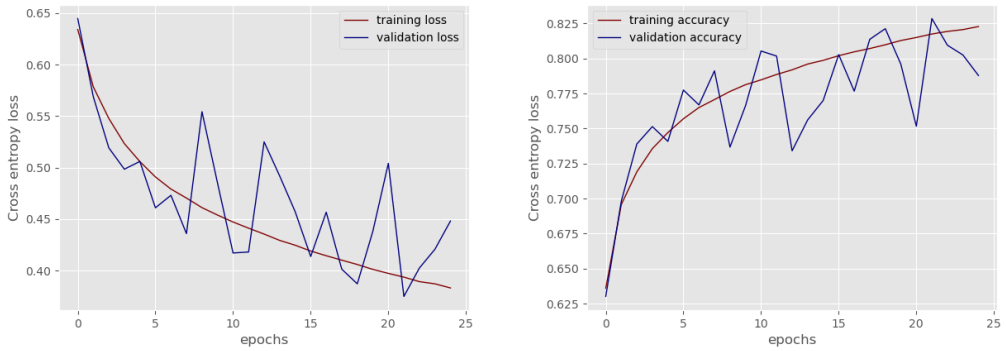
B Classification results



(a) He Normal weight initialization.

(b) No sensitive weight initialization.

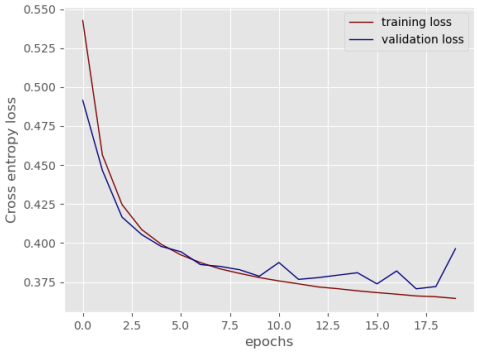
Figure 6: CNN model [5] weight initialization comparison.



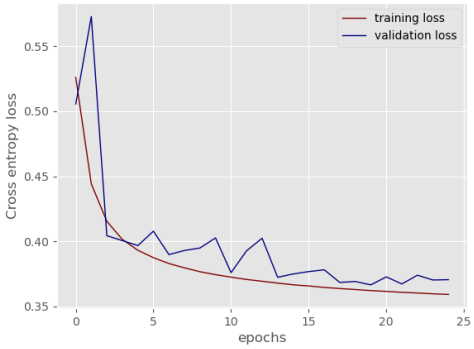
(a) Training and validation loss.

(b) Training and validation accuracy.

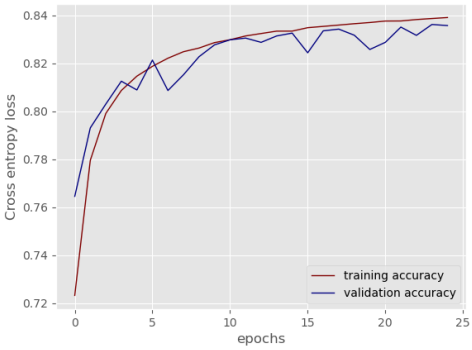
Figure 7: CNN model [5] loss and accuracy evolution.



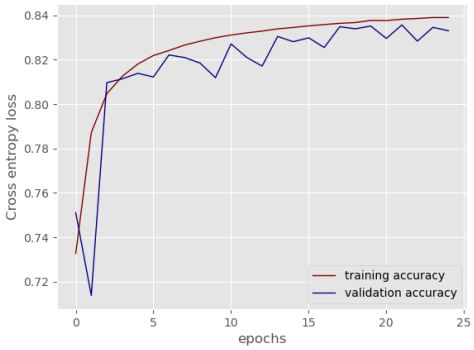
(a) Second model loss, with dropout.



(b) Second model loss, NO dropout.

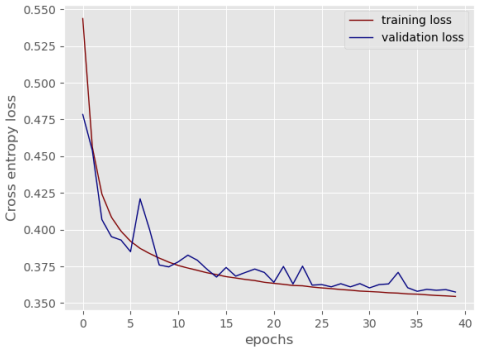


(c) Second model acc, with dropout.

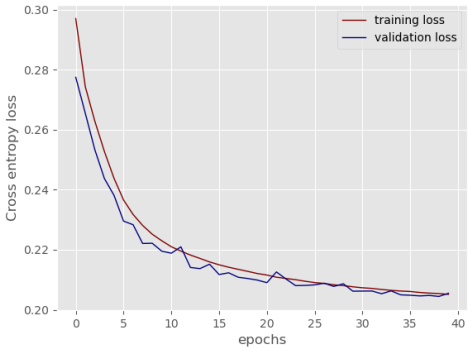


(d) Second model acc, NO dropout.

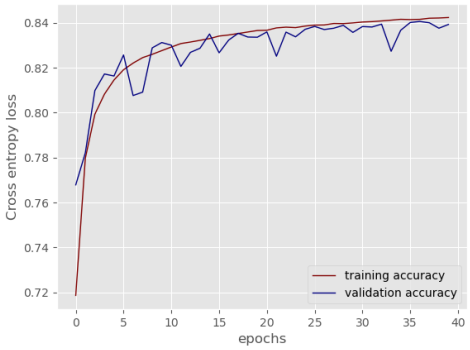
Figure 8: The effects of dropout on second model, a comparison.



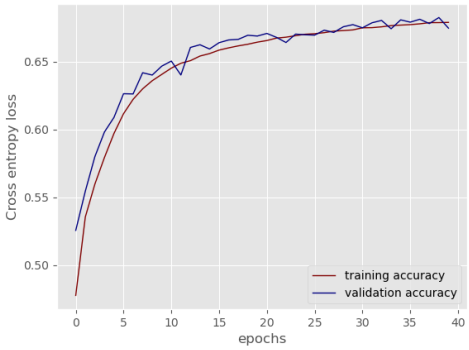
(a) Binary classification loss.



(b) Categorical classification loss (7 classes).



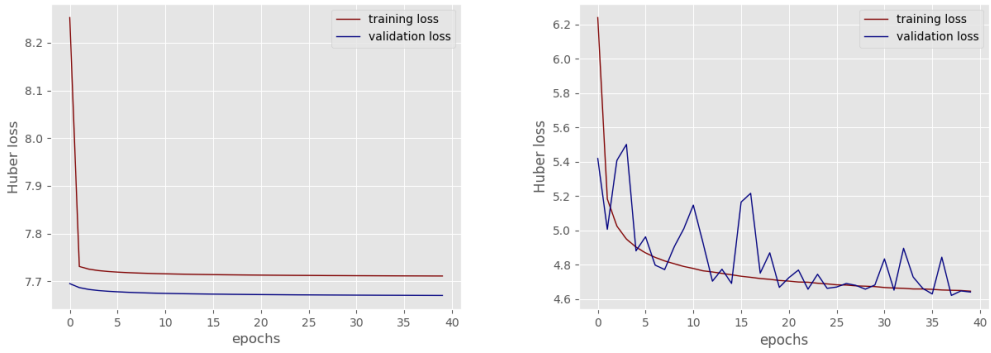
(c) Binary classification acc.



(d) Categorical classification acc (7 classes).

Figure 9: Binary and categorical classification results for second model.

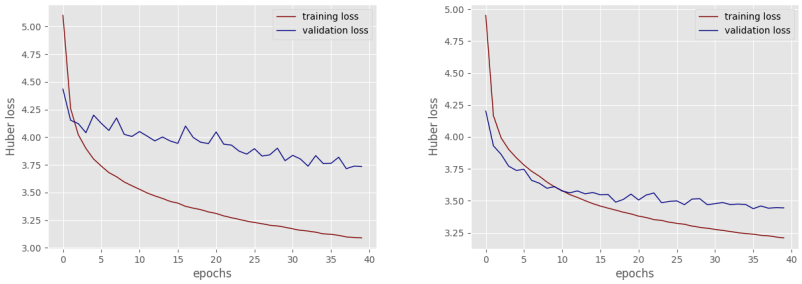
C Regression results



(a) L1 regularization, loss.

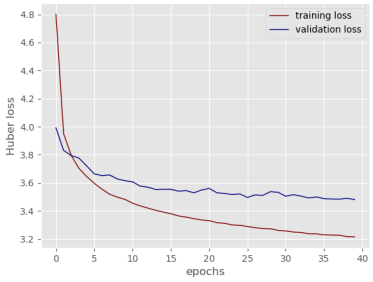
(b) L2 regularization, loss.

Figure 10: Regularization comparison in affine layers.



(a) Rectified Linear Unit, `ReLU`.

(b) Exponential Linear Unit, `ELU`.



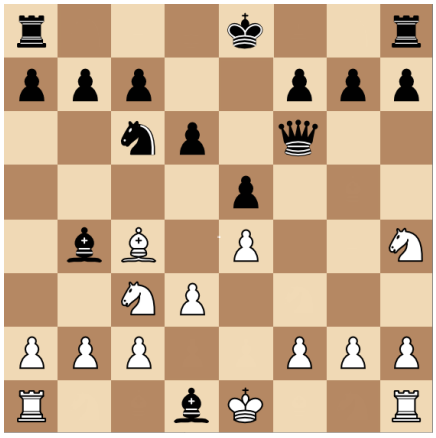
(c) Hyperbolic tangent, `tanh`.

Figure 11: Affine layer(s) activation function comparison.

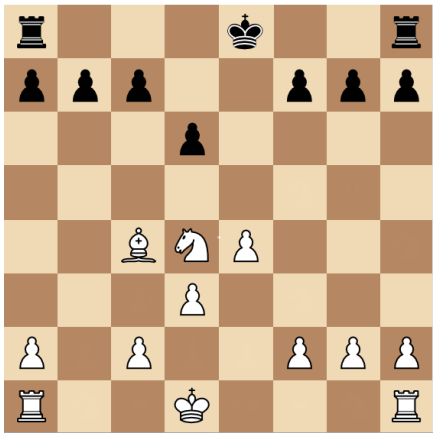
D Final model results

(a) CNN static evaluation $= 0.24$, stockfish $= 0.31$.

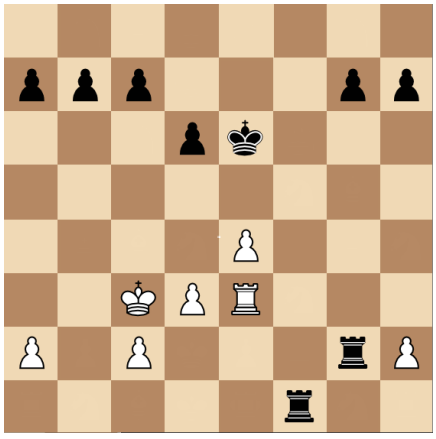
(b) CNN static evaluation $= 0.47$, stockfish $= 0.21$.



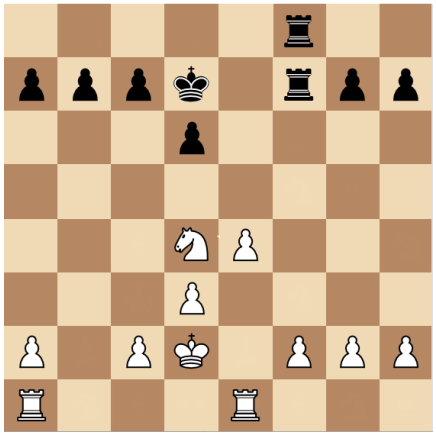
(c) CNN static evaluation $= -7.61$, stockfish $= -13.3$.



(d) CNN static evaluation $= 6.48$, stockfish $= 9.82$.



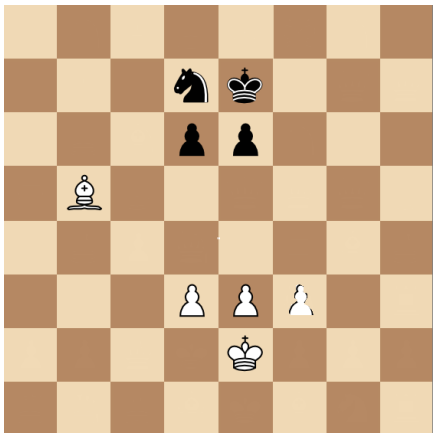
(e) CNN static evaluation $= -7.81$, stockfish $= -10.1$.



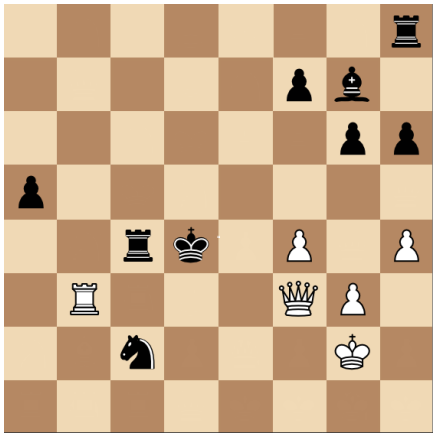
(f) CNN static evaluation $= 7.59$, stockfish $= 7.04$.

Figure 12: Evaluation comparison.

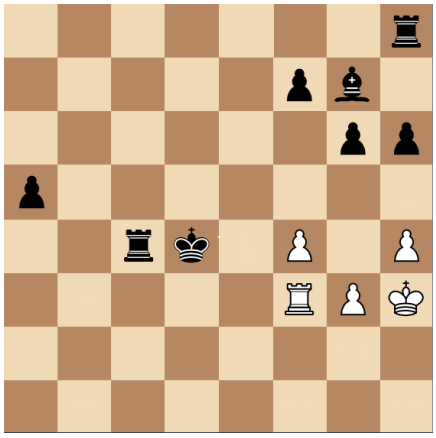
(a) CNN static evaluation $= 0.13$, stockfish $= 4.41$.



(b) CNN static evaluation $= 1.80$, stockfish $= 2.24$.

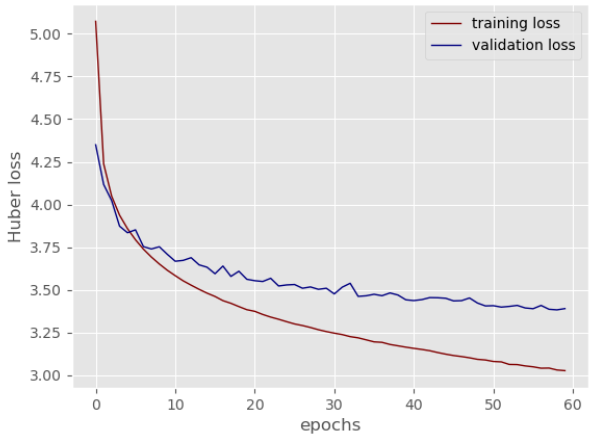


(c) CNN static evaluation $= -8.53$, stockfish $= -13.4$.



(d) CNN static evaluation $= -14.3$, stockfish $= -14.0$.

Figure 13: Tactical positions evaluation comparison.



(a) Training and validation loss.

Figure 14: Final model loss evolution, training for 60 epochs.