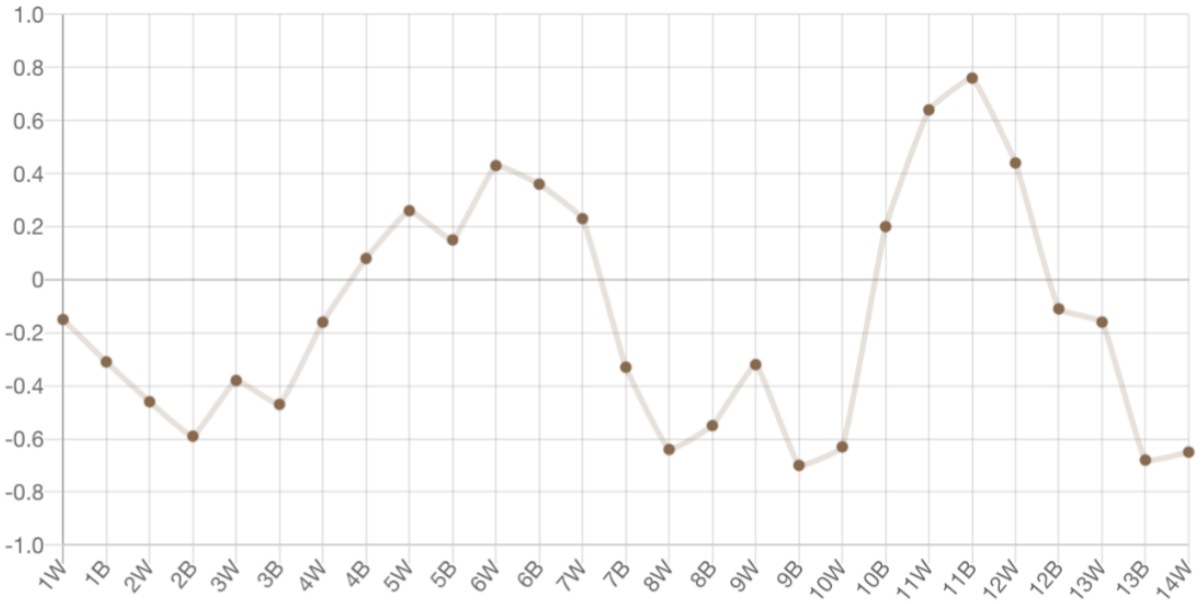
Chess Winner Prediction

This project uses data science to evaluate a sequence of chess moves in the form of a [PGN (Portable Game Notation)]. Each move in the sequence is rated to be more favorable to a white win or to a black win. This can be used with partially played games to predict the likelihood of a white win or a black win. Or it can be used to analyze already played games to find which sequence of moves turned the board to be more favorable to a white win or a black win. This project is made publicly available at [www.kohlenz.com/chess]



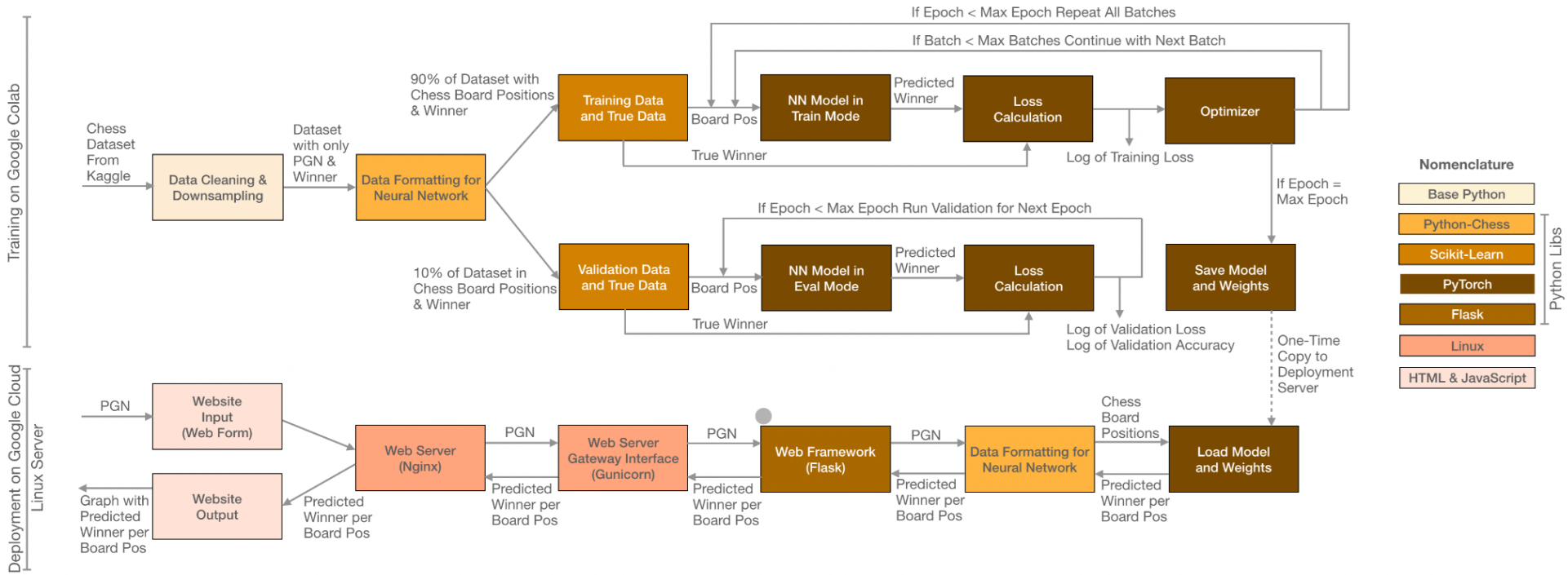
Moritz Kohlenz

10/3/2023

TABLE OF CONTENTS

METHODOLOGY

The project is split into two parts, Training and Deployment, which are both running independently on different compute platforms. Both parts are connected through a trained model, which forms the output of Training and backbone for all computations in Deployment, shown with a dashed line in the below diagram.



Like the project, this document is structured into Training and Deployment sections explaining each of the steps in the above diagram. This is followed by a section on Model Tuning, which explains how varying parameters affect model performance and thereby prediction quality.

TRAINING

The input to training is a dataset and the output is a trained model. During training, the dataset is reduced to the parameters relevant data cleaning and for training. For this project, the training data is a PGN (Portable Game Notation). The true data that the model aims to predict is the winner of the game, either black, white or a tie.

DATA CLEANING & DOWNSAMPLING

The first step of training is data cleaning, which refers to removing invalid and unnecessary information from the dataset. The dataset chosen for can be found in the REFERENCES section at (Kaggle, 2019).

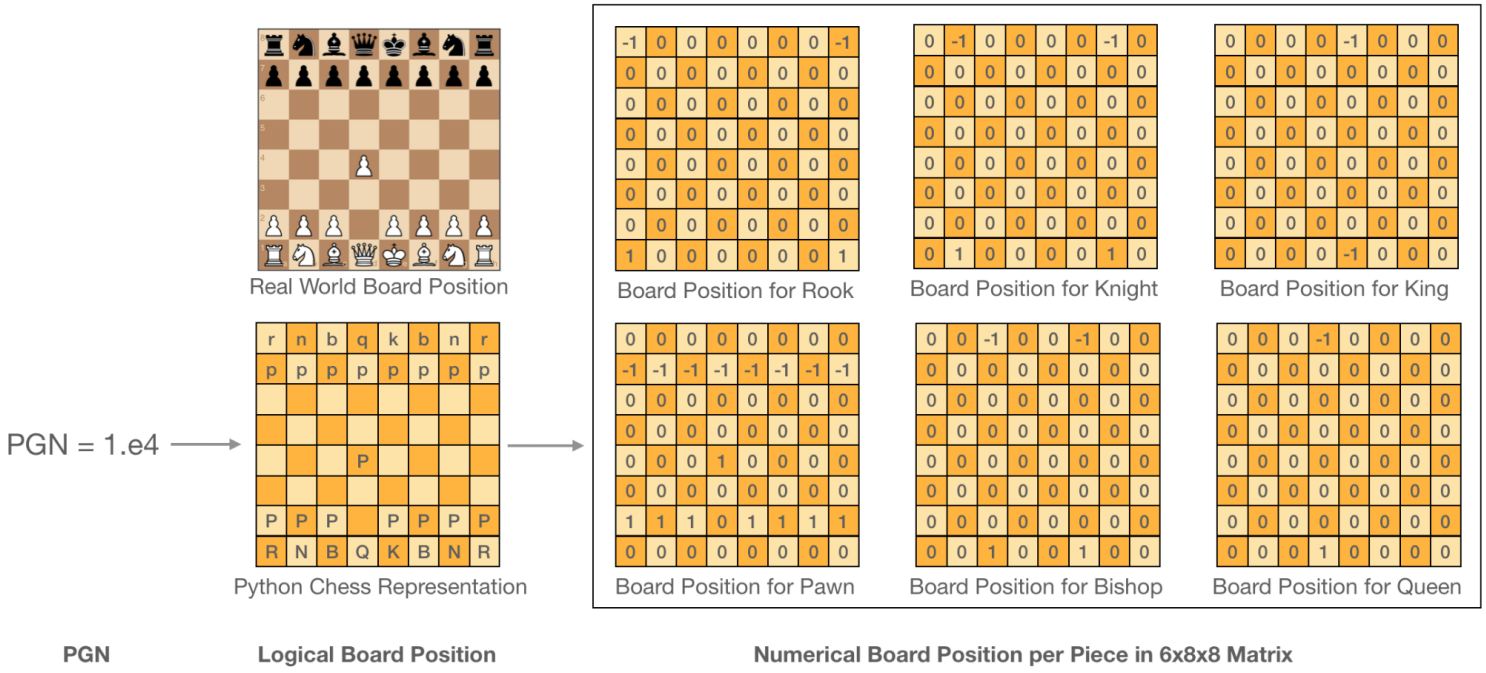
This dataset contains a move sequence for each of the 3.5 million games in the dataset that is close to PGN format but not exactly matching. It also contains a large amount of meta information for each game. Data cleaning converts the move sequences to a proper PGN and removes all metainformation except for the “Winner”, “White ELO” and “Black ELO”.

ELO is a rating for chess players and for this project only chess games played by players with an ELO larger than 2500 were chosen. This was done to ensure a consistency in the quality of moves and a high variation in move patterns, assuming that high rated players play unique moves instead of repeating standard moves.

Since the dataset with 3.5 million entries is much larger than what can be processed with an interactive Google Colab session, downsampling is performed while filtering entries for ELO until a configurable number of samples has been selected.

DATA FORMATTING FOR NEURAL NETWORK (NN)

The PGN needs to be converted into a numerical format that can be processed by a neural network. The below diagram shows how this can be achieved by the example of one of the simplest PGNs possible, “1.e4”. This means that a white pawn (a chess game always starts with a white move) is moving to position e4. The view as we see it is translated by standard Python-Chess functions into the view bottom left, where upper-case letters mean white pieces and lower-case letters mean black pieces and the letters themselves identify the piece types. For consumption in a neural network, the Python-Chess representation is then converted to a custom numerical format, first by splitting the board into six views for six piece types. The color of a piece is represented with +1 for white and -1 for black. Empty positions on the board are filled with zeroes. This forms a 6x8x8 array for every board position generated from the PGN move sequence.



TRAINING AND VALIDATION DATA SPLIT

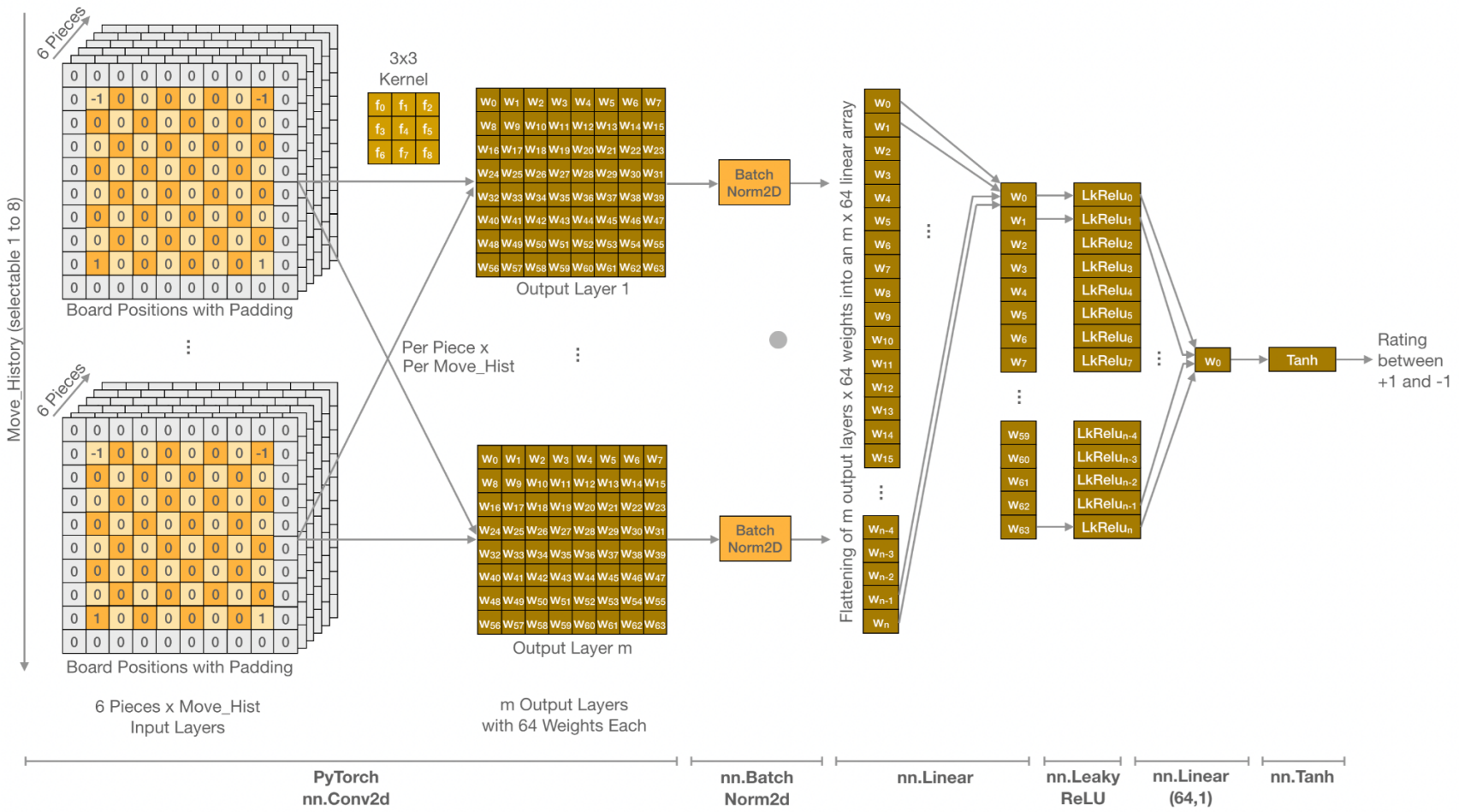
After data has been cleaned and downsampled, it is common practice in data science to split the dataset into training and validation datasets.

The training dataset is used to update model parameters by comparing predicted output against the true output. For neural networks, this is done iteratively in an inner-loop over batches, where all batches comprise the complete training dataset. This is repeated in an outer loop over epochs where the whole training data set is provided to the neural network over a configurable number of epochs. Each epoch the model parameters are refined and should lead to a better prediction.

The validation dataset is never used to update model parameters. It is solely used for prediction at each epoch. This separation allows one to spot overfitting, which can occur if the prediction accuracy on the training dataset keeps increasing while the same trend cannot be seen for the validation dataset.

NEURAL NETWORK (NN) MODEL

The below diagram shows the structure of the neural network used in this project. On the left side are the 6x8x8 matrices described in section DATA FORMATTING FOR NEURAL NETWORK (NN). On the right side is a single move rating output represented by a value between +1 for a white win and -1 for a black win. To keep the diagram simple, biases are not shown in the diagram.



The bottom of the diagram in section NEURAL NETWORK (NN) MODEL shows the various steps to get from inputs to this single output.

nn.Conv2D

The 2D convolution starts with padding the six 8x8 matrices with a ring of zeroes. This is done to create an even multiple for when the 3x3 Kernel of nn.Conv2d scans over each resulting 9x9 matrix. This padding also puts the edges of the chess board positions into the center of the 3x3 kernel.

One step of nn.Conv2D takes into account multiple moves, configurable with parameter "Move_History". If Move_History is set to 8, the 2D convolution takes 8 moves x 6 pieces x 9x9 board positions with padding into account. We call 8 moves x 6 pieces the 48 input channels.

The input channels are mapped to a configurable number of output channels by scanning the input channels with a 3x3 kernel and cross-correlating the results across all input channels. Each output channel is an 8x8 matrix of weights.

If this mapping from input to output channels is hard to visualize, please find a link to a video guide in the REFERENCES section at (ML Explained, 2022).

BatchNorm2d and nn.Linear(m x 64, 64)

After normalization, of each output layer, all matrices are flattened into a one-dimensional array, which is then further reduced to a smaller 64 deep one-dimensional array in a brute-force way, where each of the 64 output values is a function of all m x 64 input values and a bias for each output.

LeakyRELU

Each of the 64 output values is passed through a LeakyRELU activation function. An activation function acts as a filter which inputs to let through. In this case positive values will be let through and due to the "leaky" function, negative values will be let through with attenuation. This choice for LeakyRELU in this project was through trial-and-error. At this position in the neural network, LeakyRELU worked better than Tanh based on validation accuracy results.

nn.Linear(64,1) and nn.tanh

The final steps in the neural network model are a reduction from the 64 activation function filtered values to one value using nn.Linear and constraining the value to be anywhere within +1 for a white win and -1 for a black win.

For model creation, it is important to keep an eye on the computational complexity of the algorithms used in the neural network. Unsurprisingly, the 2D convolution has the highest computational complexity and therefore has the biggest influence on runtime. If training is runtime constrained, computational complexity can be reduced by reducing the number of input and output channels or by grouping input channels onf the 2D convolution. It is a tradeoff that must be done carefully between the gain in accuracy from processing more samples with a less complex algorithm and loss in accuracy from a less complex algorithm.

Model	Complexity In General Formulas	Complexity In Model Parameters
nn.Conv2d	(Number_of_Input_Channels x Number_of_Output_Channels x Kernel_Size + Number_of_Output_Bias) x Number_of_Outputs per Output_Channel	(6 Pieces x 8 Move_History x 6 Output_Channel x 3x3 Kernel + 6 Bias per Output_Channel) x 8x8 Outputs per Output_Channel = **2598 x 64 Parameters**
nn.BatchNorm2d	Number_of_Outputs x BatchNormDimension	6 Output_Layer x 2 Dimensions = **12 Parameters**
nn.Linear(384,64)	Number_of_Inputs x Number_of_Outputs + Number_of_Output_Bias	(6 Output_Layer x 8x8 Parameters per Output_Channel) x 64 Outputs + 64 Bias per Output = **24640 Parameters**
nn.Linear(64,1)	Number_of_Inputs x Number_of_Outputs + Number_of_Output_Bias	64 Inputs x 1 Output + 1 Bias per Output = **65 Parameters**

LOSS CALCULATION AND OPTIMIZER

Loss calculation is a standard step in neural network training. It computes the difference between predicted value and true value. In this project, the loss function is a standard MSE (mean squared error) function.

The optimizer goes hand-in-hand with loss calculation. It adjusts the weights and learning rates of the neural network model to minimize the loss calculated in the previous step. In this project, the optimizer chosen is Adam (Adaptive Moment Estimation).

SAVE MODEL AND WEIGHTS

The save model and weights step entered when training has run through all configured epochs. This step is important because its output is generated on one platform, Google Colab, and consumed on another platform, Google Cloud. The compute environment is different, where the model trained using a GPU, whereas for deployment, the model will be run on a standard CPU. These conditions are not difficult to handle, but need to be considered when saving the model and its weights for export.

DEPLOYMENT

The goal of deployment is to capture a PGN from a user on a webpage, pass it through the model created during TRAINING, and return the ratings per move in the PGN to the user on the same webpage.

WEBSITE INPUT

The website input is as easy as creating an HTML file with a textbox to capture the PGN.

WEB SERVER

The website input needs to be hosted on a web server. Because this will be a dynamic website returning output to the user based on the user's input, this will require a custom, configurable web server, which can connect to a Python framework. A web server's tasks is to hold multiple user's requests in parallel and running with very few compute resources per user. The webserver chosen for this project is NGINX.

WEB SERVER GATEWAY INTERFACE (WSGI)

WSGI provides a bridge to communicate between the Web Server and Web Application. WSGI is a set of rules which allow a WSGI compliant server like NGINX to work with a WSGI compliant Python application. Gunicorn is the implementation of a WSGI server for Python applications chosen for this project.

WEB FRAMEWORK

The web framework is a set of functions that facilitates the calls from WSGI to the python application. The web framework chosen for this project is Flask. Flask functions are embedded in WSGI code.

DATA FORMATTING FOR NEURAL NETWORK (NN)

This is the same formatting that had to happen during training, the conversion of a PGN to a numerical value that is understood by the neural network, see DATA FORMATTING FOR NEURAL NETWORK (NN)

LOAD MODEL AND WEIGHTS

The formatted data is now passed to the model created and saved during training, see SAVE MODEL AND WEIGHTS.

WEBSITE OUTPUT

Once the model generated the move ratings based on teh PGN provided, most previous steps are taken backward: The model returns data to Flask functions in the WSGI, WSGI serves the output to the Webserver, which then communicates to the dynamic website that requested this information.

It is the task of the website to visualize the move ratings. This is done using custom javascript code alongside library functions for graph generation from https://www.chartjs.org/ and for chessboard generation from https://chessboardjs.com/.

MODEL TUNING

EVALUATION PARAMETERS

The first step in tuning the performance of a model is determining which parameters to optimize. In this project, like in most other data science projects, the main parameter is validation accuracy. Validation accuracy measures how close the prediction matches the true data for data that the model has never seen during training. There are two more secondary indicators that can be taken into account: (1) Validation loss is the output of the loss function. Prediction is run validation data that the model has not used for training. (2) Training loss is the output of the loss function. Prediction is run on the data that it is trained on. Counterintuitive why this parameter is used. Used for spotting overfitting

HYPER PARAMETERS

Hyper parameters are changed in order to increase validation accuracy. For this project, hyper parameters are classified into data preparation parameters, data selection parameters and model parameters.

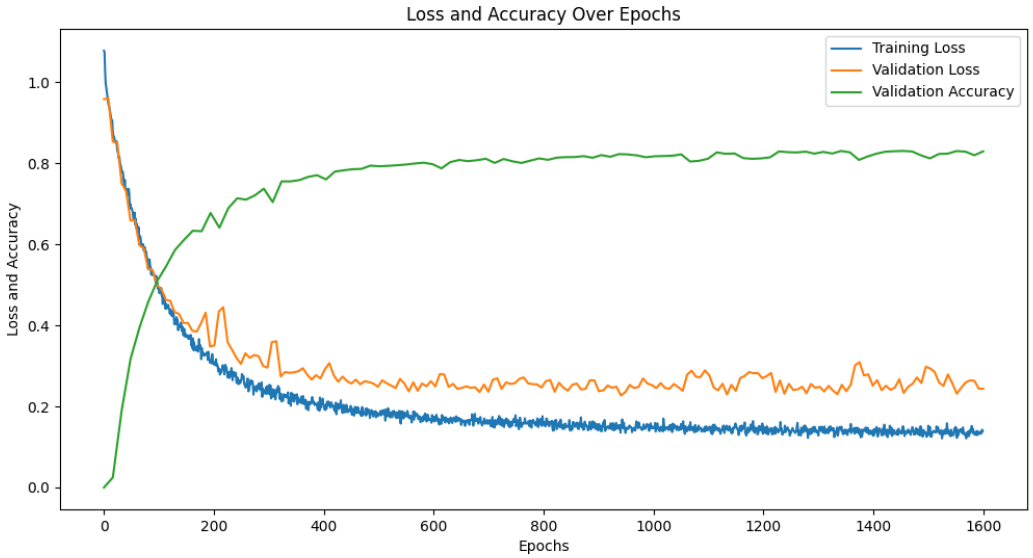
DATA PREPARATION PARAMETER EVAL

The below table shows the influence of Move_History, which is briefly described in section NEURAL NETWORK (NN) MODEL. Looking at the validation accuracy results, it be seen that the increasing Move_History has a positive effect, but the effect plateaus at around 87% validation accuracy.

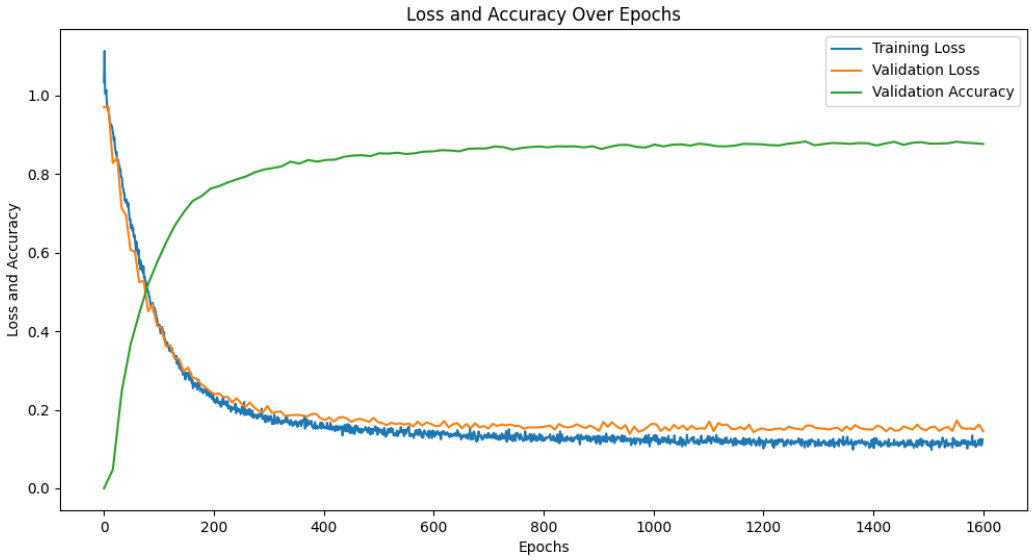
Model	Number of Games	Move History	Remove Ties	Max Epochs	Validation Accuracy
Baseline	500	1	Yes	100	**82.9%**
Baseline	500	2	Yes	100	**84.5%**
Baseline	500	4	Yes	100	**87.3%**
Baseline	500	6	Yes	100	**87.5%**

Baseline	500	8	Yes	100	**87.7%**

The below shows the graph for Move_History = 1, Training_Loss = 13.5%, Validation_Loss = 24.3%. It can be seen that not only the validation accuracy is suboptimal as shown in the table above, but also that the spread between training loss and validation loss is large, pointing towards overfitting of model parameters to training data.



The below graph for Move_History = 8, Training_Loss = 11.5%, Validation_Loss = 15.4% shows a much smoother graph, indicating stability in model weight optimization. It also shows a much smaller spread between training and validation loss.



DATA SELECTION PARAMETER EVAL

From a data selection perspective, there is a decision to be made whether to remove tied games from training or not. The model can predict a tie because its output ranges from +1 for a white win to -1 for a black win. A value at or close to zero would indicate a tie. However, ties in training data are hard to predict. Tied games contain many moves that in other games would have resulted in a white win or a black win. Therefore the expectation is that having ties in training reduces the validation accuracy. This is confirmed with the below results from test runs with and without ties during training.

Model	Number of Games	Move History	Remove Ties	Max Epochs	Validation Accuracy
Baseline	500	4	No	50	**81.3%**
Baseline	500	4	Yes	50	**85.7%**

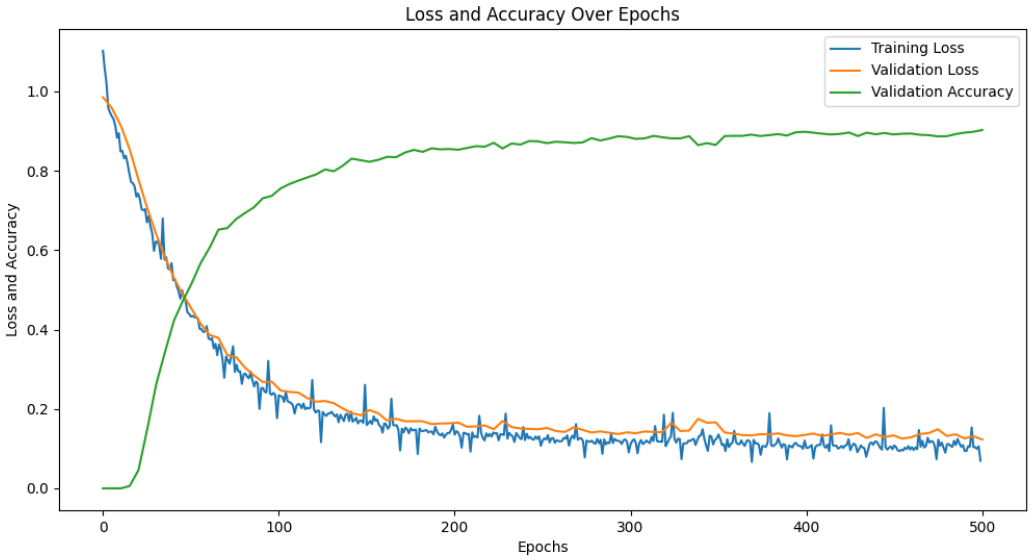
The number of games parameter presents a similar condition like ties. The more moves are provided to a neural network the more chance there will be that some moves have little to no influence towards the outcome of a game. For instance, moves in the

beginning of a game have little to no influence on the outcome of the game. The model processes them correctly in adjusting the model weights towards a move rating closer to zero. However, in validation, a move rating close to zero will result in wrong predictions when compared to true data that is either a white win or a black win. Predictably, the more games are evaluated, the lower the validation accuracy becomes, although the actual prediction result is improving.

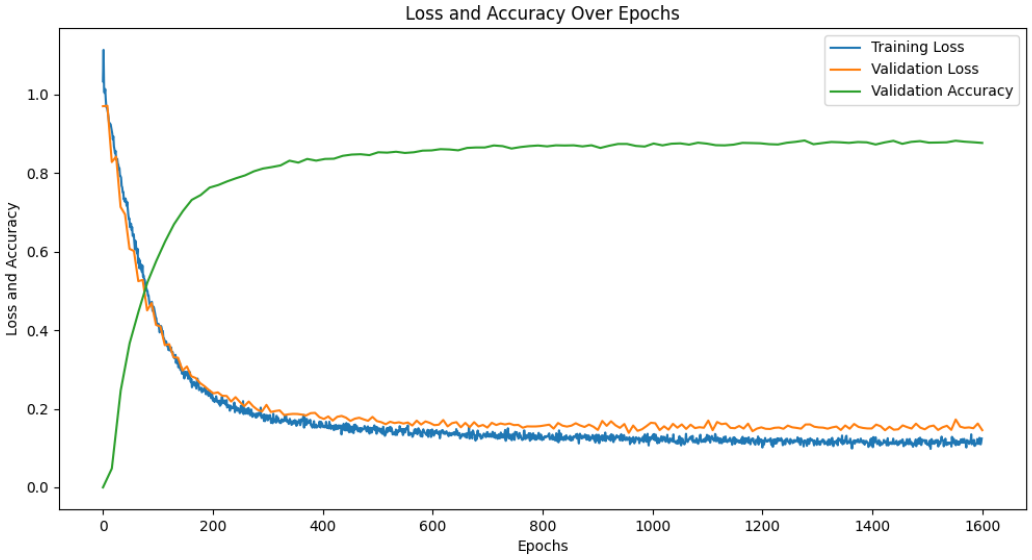
Model	Number of Games	Move History	Remove Ties	Max Epochs	Validation Accuracy
Baseline	125	8	Yes	100	**90.2%**
Baseline	250	8	Yes	100	**89.0%**
Baseline	500	8	Yes	100	**87.7%**
Baseline	1000	8	Yes	100	**85.1%**
Baseline	2000	8	Yes	100	**80.5%**

The below graphs show that the validation accuracy, training loss and validation loss curves are most stable with 500 samples, meaning 500 games that were evaluated in the model. However, for deployment, still the model created from 2000 samples was chosen due to aforementioned shortcomings in the validation.

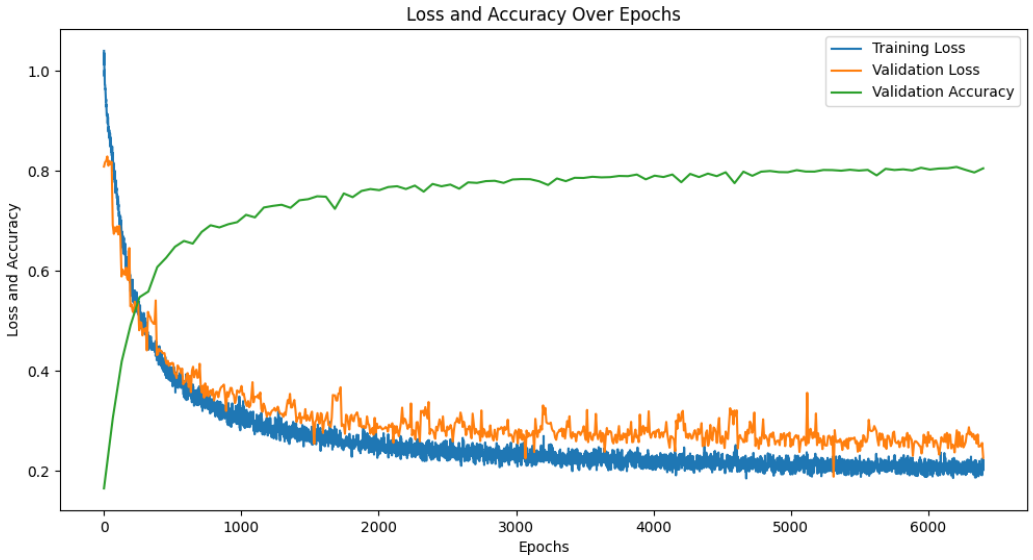
Number_of_Games = 125, Training_Loss = 9.6%, Validation_Loss = 12.3%



Number_of_Games = 500, Training_Loss = 11.5%, Validation_Loss = 15.4%



Number_of_Games = 2000, Training_Loss = 20.4%, Validation_Loss = 25.4%



MODEL PARAMETER EVAL

The model parameter evaluation is an extension of the model introduced in section NEURAL NETWORK (NN) MODEL. The question was whether model complexity can influence validation accuracy in a meaningful way. For model complexity, the number of model parameters was taken as the measure. Adding an additional 2D convolution layer increases complexity roughly by a factor of 7.

Grouping of inputs reduces complexity by a factor of 4 for grouping by piece type. This reduction of complexity is achieved by collapsing 6 input layers for 6 pieces into one input layer. Similarly, grouping by Move_History reduces complexity by a factor of 3. For grouping, the number of input_channels need to be divisible by the number of output_channels, hence adjustment in the number output channels for the two grouping options.

Model	Complexity In Model Parameters
Baseline	2598 x 64 for Conv2D + 12 for BatchNorm2D + 20,640 for Linear(384,64) + 65 for Linear(64,1) = **186,989 Parameters**
Additional Conv2D Layer	(6 Pieces x 8 Move_History x 6x8 Output_Layer x 3x3 Kernel + 6x8 Bias per Output_Channel) x

	8x8 Outputs per Output_Channel for Conv2D + 96 for BatchNorm2D + 27,315 Parameters for Baseline = **1,357,587 Parameters**
Group by Piece	(8 Move_History x 6 Output_Channel x 3x3 Kernel + 6 Bias per Output_Channel) x 8x8 Outputs per Output_Channel for Conv2D + 12 for BatchNorm2D + 20,640 for Linear(384,64) + 65 for Linear(64,1) = **48,749 Parameters**
Group by Move_History	(6 Pieces x 8 Output_Channel x 3x3 Kernel + 8 Bias per Output_Channel) + 8x8 Outputs per Output_Channel for Conv2D + 16 for BatchNorm2D + 32,832 for Linear(512,64) + 65 for Linear(64,1) = **61,073 Parameters**

For this project, the complexity of the model is not a reliable indicator for predicted validation accuracy. In fact, the Baseline model performs the best. The second best is the least complex model, with a complexity reduction by a factor of 4 and within less than 1% of the validation accuracy of the best performing baseline model.

Model	Number of Games	Move History	Remove Ties	Max Epochs	Validation Accuracy
Baseline	125	8	Yes	100	**90.1%**
Additional Conv2D Layer	125	8	Yes	100	**88.9%**
Group by Piece	125	8	Yes	100	**89.2%**
Group by Move_History	125	8	Yes	100	**88.3%**

CONCLUSION

This was an ambitious project that resulted in solid prediction quality towards qualifying moves as either predicting a white win or a black win. This result can be validated by everyone with a PGN on the website provided.

A topic for further study will be the measurement of validation accuracy for an increasing number of games evaluated. The more moves are provided to a neural network the more chance there will be that some moves have little to no influence towards the outcome of a game. The model processes these moves correctly in adjusting the model weights towards a move rating closer to zero. However, in validation, a move rating close to zero will result in wrong predictions when compared to true data that is either a white win or a black win. Maybe predictions closer to zero could be reduced in their significance towards validation accuracy.

REFERENCES

Grg, P. (n.d.). *Build a Web App using Python's Flask*. Pema Grg. Retrieved October 29, 2023, from https://pemagrg.medium.com/build-a-web-app-using-pythons-flask-for-beginners-f28315256893

Kaggle. (2019, March 9). *3.5 Million Chess Games*. Retrieved October 8, 2023, from https://www.kaggle.com/datasets/milesh1/35-million-chess-games

ML Explained. (2022, September 7). *PyTorch 2D Convolution*. YouTube. Retrieved October 29, 2023, from https://www.youtube.com/watch?app=desktop&v=n8Mey4o8gLc

Volkov, A. (n.d.). *How to deploy Flask on Google Cloud GCP Compute Engine with Gunicorn + Nginx + systemd*. Alex Volkov. Retrieved October 29, 2023, from https://alex-volkov.medium.com/how-to-deploy-flask-on-google-cloud-gcp-compute-engine-with-gunicorn-nginx-systemd-96da1f32a11a