Chessboard Image to Position Notation Using Logistic Regression and Cross Validation

Ben Esposito, Colin Kohli, Laryssa Abdala, Rajeev Rajendran

November 22, 2020

The problem

When introducing their reinforcement learning algorithm AlphaZero in 2018, Silver et al remarked that "The game of chess is the longest-studied domain in the history of artificial intelligence...the study of computer chess is as old as computer science itself". Indeed, chess is a familiar field of exploration with many subdomains to explore computational concepts. Our group is creating a model which can receive an input of an image of a computer chessboard of variable color, size, and piece style, and generate as output a file in Forsyth-Edwards Notation (FEN). This notation assigns integers based on the number of empty board squares, and characters based on chess pieces. Said characters are in upper or lower case depending on what side of the board they came from, or in traditional chess, what color piece they are.

Chess is played on a square board with 64 spaces on it of alternating light and dark squares. In a full chess set, there are 32 pieces, 16 white and 16 black, with 6 distinct piece types: pawn, knight, bishop, rook, queen, and king. Piece design differs from set to set, but in nearly every set, these elements are consistent.

The model learns by training on individual board squares extracted from a dataset of digital chess boards. Figure 1 shows a row by row example of FEN notation on a digital chess board. The first row is a rook (r) and a king (k), separated by 6 empty spaces.

Data set

The dataset used in the present work was created by Pavel Koryakin and was posted to Kaggle under a CC0 Public Domain license. The data set consists of $100,000$ randomly generated chess boards of 5-15 pieces, with a training set of $80,000$ images of chess boards and a test set of $20,000$ images of chess boards. However, these boards are not all in the same exact format. Many vary in color and have stylistic differences between pieces. See Figures 2 and 3.

The images were compressed to handle the amount of data we were processing through the pipeline. As a reference, Figure 4 shows the compressed version of the original image in Figure 5. Note that these are not real game positions, as the purpose of this analysis is not related to gameplay or strategy, but is exclusively aimed at identifying 6 different piece types.

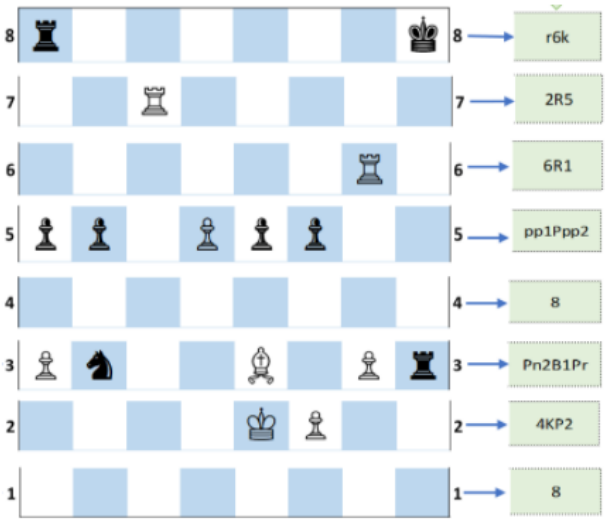

Figure 1: FEN notation on a digital chess board

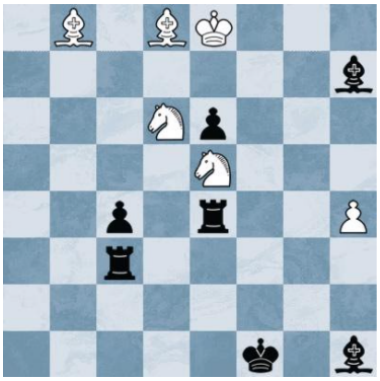

Figure 2: Example of board 1.

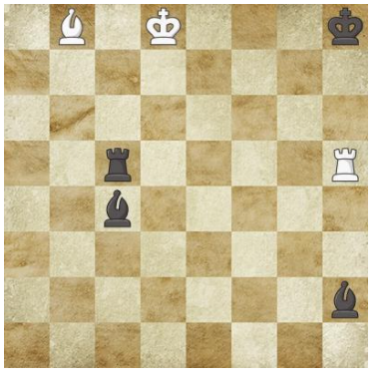

Figure 3: Example of board 2.

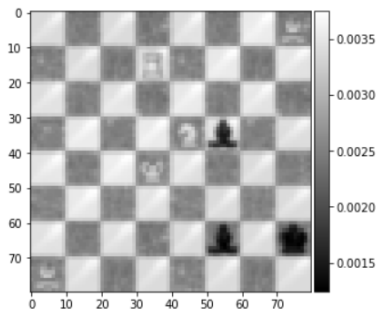

Figure 4: Compressed image.

Figure 5: Original image.

Parameters

```
IMAGE_LOAD_SIZE = 1000 #The amount of board files to load
DROP_BLANKS_PCT = .8 #When learning squares, drop this percent of blank squares from fitting
SAVE_CLASSIFIER = True #Saves squares classifier after fit
PIECE_LIST = 'prnbqkPRNBQK' #FEN piece codes. Case represents color.
PIECE_DICT = {} #dictionary from piece code to piece list
for ii in range(1, len(PIECE_LIST)+1):
    PIECE_DICT[PIECE_LIST[ii-1]] = ii
```

As you can see above there are several parameters that can be used to tune the accuracy of the model on the given data set. Image_Load_Size simply lets the model know how many of the 80,000 images to load. Of course, the model performs higher with all $80,000$ images (over 99.5% accuracy). However, there is a certain threshold where there is diminishing returns based on the time taken and the processing power (especially when you are honing in on parameters to tune). One can reach a rate of over 97% accuracy at just $2,000$ samples.

The Drop_Blanks_PCT (or drop blanks percentage) parameter, lets the user control the percentage of blank chess spaces or spaces without pieces occupying. By decreasing the number of blank pieces the model has to learn on, we speed up the process and waste less energy looking at spaces without any piece features to learn from. This parameter shifts the accuracy by smaller amounts than the total image parameters (multiples of 0.1% or 0.01%).

Save Classifier saves the classifier that was previously run so the user can make further adjustments if desired. The piece list is simply a character list of all FEN abbreviations.

We achieved 99.94% accuracy in the test set and the confusion matrix:

$$\begin{bmatrix}
172414 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 2888 & 3 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 2843 & 1 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 2 & 4 & 2812 & 0 & 6 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 3 & 2 & 2902 & 3 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 1 & 2 & 1 & 1460 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 2 & 1 & 4 & 5 & 3181 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 3 & 0 & 0 & 0 & 0 & 0 & 2863 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2908 & 0 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2893 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2827 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1505 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3200
\end{bmatrix}$$

Solution

We start our solution pipeline by preprocessing the images. We do this first by importing a series of functions from the Scikit-Learn machine learning/data analytics library, along with numpy, and matplotlib (to visualize the data).

In our image processing function, a gaussian filter is added to each image. Gaussian filters typically transform the image by reducing grain and smoothing edges. This removes unwanted detail and noise, and is especially useful when one compresses an image and wants to get rid of pixelation. The gaussian distribution is used to calculate the changes in each pixel of a given image. In one dimension, a gaussian filter simply uses figure g1 below for the transformation. When done in multiple dimensions, as for the present case that is bi-dimensional, it transforms each picture using a product of gaussian functions as in the expression for G_2 in Equation (1).

$$G_1(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}; \qquad G_2(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}; \tag{1}$$

The rescale function then ensures that all images are of the same size without becoming distorted. Subsequently each transformed image is returned.

The load images function next takes all of the preprocessed images and puts them into an array where they can be accessed and operated on by the functions that will classify them. Instead of working with the original 80,000 samples that we had available, we selected at random to work with 20,000 of them, assuming this would be sufficient for the current work. From that, we split the data into train (80%) and test (20%) sets. We used 5-fold cross validation to validate our problem.

Each of the images of the training were split into 64 squares with 10 by 10 pixels. Those 1,024,000 squares were split again into train (80%) and test sets (20%) in order to train and test classification of individual squares. The strategy of splitting each board into small squares, predicting the pieces and putting pieces together to make a prediction for the entire board is somewhat similar to pooling layers. Our goal was to use a multiclass classifier to learn the features of the chess pieces, allowing us to identify whether a square was occupied and by what piece. Since the blank squares vastly outnumbered squares with pieces, for the sake of distributing classes more evenly we used the variable Drop_Blanks_PCT at 0.8. In the following we used softmax to classify the squares. The implementation was done using the sklearn.linear_model.LogisticRegression from the SciKit package with the configuration multi_class='multinomial', solver='newton-cg', fit_intercept = True, verbose = 10.

What We Tried First and What We Changed

Our image processing algorithm originally tried to learn by viewing the chessboard as a whole. However, It became easier, more efficient, and more accurate to train the model to look at each board square to recognize the specific pieces. Once the squares are split one can more clearly decipher the distinctions between pieces, build the connections between pieces, and later assign the correct FEN values.

Another useful feature we added was the parameters menu described above. This lets us see how each parameter affects the accuracy of the model, and which parameters have the highest impact. It also lets us find the threshold for each parameter that yields the most accurate result without using too much processing power.

Related Work and Applications

There are many papers which explore the chessboard as a field for studying vision and classification. In 2005, Martin's presentation, Finding a Chessboard, introduced fundamental concepts of early computer vision with this motivating question in mind. A few of the publications we found tackled the problem of identifying physical pieces. In Kory and Sumer's 2016 paper, they tracked full sequences of moves in a chess game by comparing image frames before, during, and after moves with an RGB webcam and a series of processing steps implemented in MATLAB.

Image filtering is a vital tool for processing large datasets like the one we used. We were able to cut down on file sizes, while still maintaining the necessary feature components for our classification. Without this, each trial could take hours to run, or not run at all. Additionally, in 2018, Marzi *et al.* (2018) showed that preprocessing medical images for higher information sparsity can help defend against adversarial attacks.

This would be particularly relevant if we applied these techniques towards real chessboard images or images that may have distortions or irregularities.

Though our project focused on chess boards, one could hypothetically split any image up into pieces to identify desired components. Not only is this easily applicable to identify pieces/characteristics of other game boards but also for forms that follow consistent patterns. For example, this same pipeline could be modified to identify components of tax forms and QR codes. Also, the chess specific application can be applied as the first step of a puzzle solving algorithm. After we've identified the pieces and converted the board to FEN notation we can then use a strategic algorithm, such as Stockfish or Leela Chess Zero, to determine the next move.

Conclusions

Overall, our method of synthesizing various methods (logistic regression, cross validation and splitting our board squares to be processed) successfully labeled and classified the chess pieces. We consistently displayed levels of accuracy over 99% when parameters were optimized. Even when the image load size and drop blanks percentage were less than the optimal solution, our classifier consistently yielded results of over 95% accuracy. From there, we were able to smoothly label each piece position with the desired FEN character.

References

[1] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, Demis Hassabis: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. CoRR abs/1712.01815 (2017).

[2] Z. Marzi, S. Gopalakrishnan, U. Madhow and R. Pedarsani, "Sparsity-based Defense Against Adversarial Attacks on Linear Classifiers," IEEE International Symposium on Information Theory (ISIT), Vail, CO, 2018, pp. 31-35, doi: 10.1109/ISIT.2018.8437638. (2018) .

[3] Martin, Martin C. Finding a Chessboard. Presentation, (2009).

[4] Koray, Can and Emre Sümer. "A Computer Vision System for Chess Game Tracking." (2016).

[5] D. A. Christie, T. M. Kusuma and P. Musa, "Chess piece movement detection and tracking, a vision system framework for autonomous chess playing robot," Second International Conference on Informatics and Computing (ICIC) (2017).

[6] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, (2011).