Efficient Chess Vision – A Computer Vision Application

Allen Wu Stanford University – CS231N

Abstract

This report presents an application of computer vision to identifying the pieces on a chess board from an online screenshot of a chess board. The model is meant for screen captures from popular chess websites, or videos of chess games. The method proposed is not meant to be used on actual pictures of chess pieces, i.e., pictures of chess books, or pictures of real-life chess boards themselves. Therefore, since the data set is so narrow, the model is optimized for speed and performance.

The method proposed consists of 3 main portions. The first portion is to crop the image using summed gradients such that only the board remains. The second portion is to identify the correct piece on every square, which was done using various classifiers, though the best was a convolutional neural network (CNN). Predictive adaptions were added to the final layer of the CNN based on what positions in chess are common/logical to improve the prediction.

1. Introduction

Chess as a game, and especially online chess, has exploded in popularity over the past 2 years, partially due to the pandemic. Chess streamers such as Hikaru Nakamura pull in tens of thousands of viewers each stream while popular tournaments such as PogChamps feature very famous personalities such as Mr. Beast and have upwards of 100,000 live viewers, rivaling the largest esports. Chess videos have also exploded on YouTube, with there now being 3 YouTube channels with over 1 million subscribers that upload chess content daily.

Due to all of these livestreams, videos and online chess events, tools that can translate an image of a chess board to Forsyth-Edwards Notation (FEN) have become very important as they allow viewers to study the games they see online, review different variations, and analyze them with chess engines. Chess vision also can help viewers find videos to help explain the games that they watch.

The issue is that many existing chess-vision implementations are computationally expensive, meaning that it is impractical for viewers to run such models on videos or livestreams, and expensive for companies to build databases. If a model is light enough to run on every frame, or every few frames, it then becomes practical to extract entire chess games in Portable Game Notation (PGN) form from videos.

> FEN example:
>
> rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1

Such a model will also allow the creation of a database of chess videos related to famous games or games from top level players. Chess at the top level is extremely hard to understand for casual fans, and as such, the popularity of videos on YouTube of high-level players explaining chess games is very high. A feature that finds chess videos given a specific game could be highly marketable and highly profitable to the casual chess audience.

Problem Statement

The problem being addressed in this paper is to convert a screenshot of a chess board into a chess FEN depicting the same chess board using the least computationally expensive method available while still maintaining high accuracy.

2. Related Work

There are some existing works relating to chess vision. Most aim to convert images from the camera view of a real-life chessboard into a Chess FEN.

One of the first papers of this kind [1] uses a discrete Fourier transform and various corner-based detection algorithms to find the location of a chessboard in 3D space from 2 overhead cameras.

Later on, another project [2] utilizes a single overhead camera and a SIFT based descriptor classifier on Harris Corners to detect the chess board. A heatmap based approach is then used to track moves and classify the pieces on the board.

A more recent implementation [3] uses a Canny edge detector, Hough transform and DBSCAN clustering algorithms to accurately identify the location of the chess board. Then, a projective transformation is used to find the pixels of the pieces on each square, and a convolutional neural network (CNN) models that were pre-trained on ImageNet was further refined to classify chess piece in question.

Several others have attempted similar work [4-6] and generally use SIFT based on Harris corners to find the board, and then either a histogram of gradients, or template matching on piece outlines to perform piece classifications. Most error rates on tests sets for these types of applications are under 5% for identifying the board but are much high when classifying the pieces.

For chessboards from online screenshots, there several online hobbyist implementations, though many of these are overly complicated. One implementation [7] uses a CNN to produce a heatmap of which pixels contain the chess board, and another CNN to classify the pieces. The unet CNN architecture was used, resulting in around 35 million parameters total, which is far to heavy to be practical outside of single frames.

Another implementation [8] uses the Canny detection algorithm to find the edges of the image and extrapolates them to identify the boundaries of the chess board. Then, template matching is used to remove the background from the pieces, and a small CNN is used to classify the pieces. [9] uses a harris corner detection algorithm and a support vector machine (SVM) instead of CNN to classify the pieces themselves.

3. Methodology

Dataset

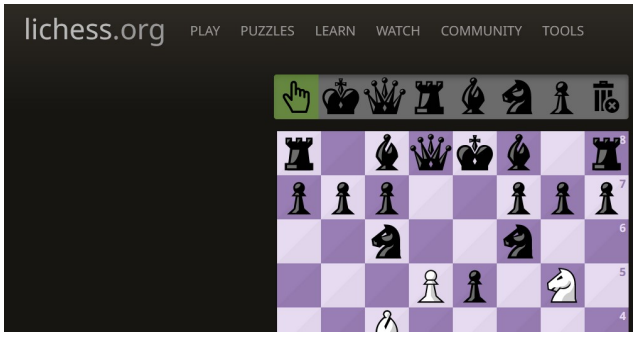
The data in this approach will be from lichess.org's open database [10]. Lichess is the second most popular website for playing chess online, and their April database contains 87 million games, though only 10000 were used for this paper. For each game, each position was inputted into Lichess's opening editor, where a random piece set and theme was applied.

Then a screenshot was taken using python's selenium library and some random cropping was done to ensure that not all inputs had the same dimensions. The more obscure themes (such as the anarchychess theme) were not used, as they do not appear practically in chess videos or screenshots. Each position was weighted to appear with probability.

$$1 - \frac{2}{i}$$

This was done to avoid the opening positions, which are very similar, from appearing too much in the data and potentially distorting the model. Different zoom windows settings also used in the Firefox browser to simulate different board sizes in the images. 20% of the data was also randomly resized using interpolation in OpenCV's resize() function, which would help account for distortion/image compression, as the borders of the chess squares don't always lie neatly between pixels.

The castling privileges were removed from the FEN as they are undetectable without seeing the entire game. No other processing was done, as Lichess already contains a large amount of variety in piece sets and board textures. The screenshot is used as input and the output is the FEN from the given position. An example of the data is shown below:

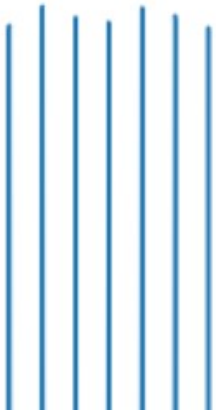


The data was split 50% - 25% - 25% for training, validation and testing.

Board Extraction

To extract the board, the image was first converted to grayscale, then the gradients of the image were taken in the x and y directions. It is assumed that the chessboard is fully rectangular, and that it is upright at 90 degrees. These assumptions make sense when considering that the application of this method is for screenshots and chess videos, where the board is almost always upright. Hence, many shortcuts can be taken to save time and speed up the model. Below are the gradients in the x and y directions, respectively.

Next, the absolute values of the gradients are summed across the entire axis. This is effective in detecting the chess squares because significant changes in color occur between different squares on the chess board which create a large gradient in the x and y directions. The absolute value is needed to ensure that opposite sign gradients do not cancel out. Below are the gradients plotted to their respective axis.



Next the top 7 peaks are chosen and tested to check whether they are equidistant from each other, within a margin of 90%.

If they are not, then more peaks are chosen until there are 7 that are roughly equidistant from each other. To do this, every pairwise distance between adjacent peaks is calculated, and the most common distance is chosen as the length of the chess square, we can refer to this as the stride length. Then, stride length steps are taken to the right from each peak starting from the left, stopping when there is a starting peak where 6 strides each result in another peak.

One special case is where there are 8 or 9 adjacent tall peaks. This happens quite often due to the border of the chess board being present. When there are 9 peaks, then the middle 7 are taken. When there are 8 peaks, a stride is taken outwards from both outermost peaks, and the resultant location that includes the peak is considered as the other edge of the chessboard. Then, the 7 inner lines are used to decide the location of the chess board.

Piece Classification

First, the board is scaled to a 256x256 image using the resize() function from OpenCV [14]. The bilinear interpolation is used, as it was determined to be most accurate when used with the SVM classifier.

Type of interpolation	Test acc
Nearest Neighbor Interpolation	98.12%
Bilinear Interpolation	98.3%
Cubic Interpolation	98.26%
Lanczos Interpolation	97.66%

The reason that the area-based interpolations failed was likely because the squares themselves are already quite small, and there are not many pixels for the pieces to begin with. The squares are 32x32, and Lanczos Interpolation looks at an 8x8 pixel neighborhood, which may lead to the pieces being blurry.

Next, various classifiers were used to classify the pieces themselves. Several different architectures were tried, including a linear SVM, linear SoftMax, and various CNN architectures. Here is one CNN architecture attempted:

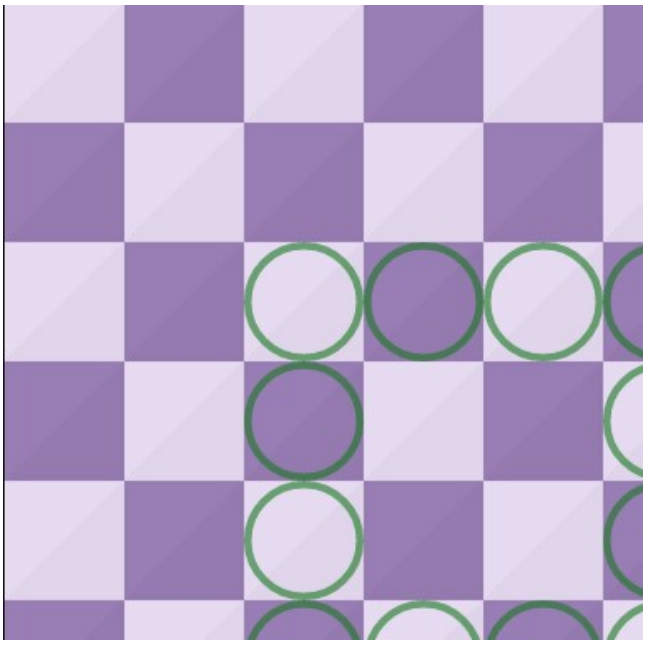
Layer (type)	Output Shape
conv2d_55 (Conv2D)	(None, 32, 32, 16)
max_pooling2d_22 (MaxPooling2D)	(None, 15, 15, 16)
dropout_28 (Dropout)	(None, 15, 15, 16)
conv2d_56 (Conv2D)	(None, 15, 15, 32)
max_pooling2d_23 (MaxPooling2D)	(None, 7, 7, 32)
dropout_29 (Dropout)	(None, 7, 7, 32)
conv2d_57 (Conv2D)	(None, 7, 7, 64)
conv2d_58 (Conv2D)	(None, 7, 7, 128)
conv2d_59 (Conv2D)	(None, 7, 7, 13)

For the other CNNs, the structure was very similar, but the small CNN only had one conv-pool-dropout layer while the large CNN had 3.

The linear classifiers were trained using stochastic gradient descent and the Adam optimizer. A random search was used to tune the hyperparameters with the validation set. The linear classifiers were trained for 20 epochs. The convolutional neural networks were set up and trained using PyTorch for 20,000 steps. Due to the image resolutions (each piece is 32x32), there was no need to use larger models, or a large pretrained model.

Predictive Adjustments

Since each piece was classified individually, the model receives no information about the context of that piece on the board. As such, a final manual tuning was done right before the SoftMax layer and right before the classification was done. In chess, it is not very likely for the king to be near the center of the board, so a constant was subtracted from the output value corresponding to the King piece based on which ring of the chessboard the square was located on. This would constitute as the second ring of the chess board:



Another manual edit was that pawns are not allowed to be on the first or last rank, so a large constant was subtracted from the output value corresponding to the pawn.

The third adjustment is related opposite-colored bishops. If there were two bishops of the same side (white or black) on the same-colored squares on the chessboard, a negative penalty is applied to both bishops. The same is done when there are too many pieces on the board. In this case, if there are ever more than 3 queens, rooks, bishops, or knights of the same side, a penalty is applied scaling linearly by the number of pieces.

All these parameters were manually chosen so that that were proportional to the average output values given by the models without the predictive adjustments.

Evaluation

The results in this paper were compared to that of ChessVisionAI, the commercial leading AI for chess vision. ChessVisionAI does not have an open-source code release, so it was run on the same testing set using their Discord bot. Therefore, it is impossible to compare runtime, as Discord has significant overhead times.

4. Results

Here are the results when there were no predictive adjustments. The accuracy is calculated per board, so the accuracy per piece/square is in fact much higher.

Model	Test acc
SVM	98.3%
SoftMax	97.94%
CNN – Small (10k features)	99.26%
CNN – Medium (40k features)	99.44%
CNN – Large (90k features)	99.14%
ChessvisionAI	99.64%

Here are the results with the predictive adjustments

Model	Test acc
SVM	98.32%
SoftMax	97.94%
CNN – Small (10k features)	99.24%
CNN – Medium (40k features)	99.5%
CNN – Large (90k features)	99.16%
ChessvisionAI	99.64%

Overall, the medium sized CNN performed best. This was likely due to the fact that the training time was standardized for all 3 CNNs, so the largest one was likely not fully trained yet. Regardless, this shows that a medium sized CNN is a good compromise between speed and accuracy.

The predictive adjustments had a net positive impact on the test accuracy, though the results were very minor. Since the adjustments were done at the level above where the CNNs were functioning (on the board as a whole instead of by square), it was impossible to perform backpropagation on these predictive parameters, so the parameters needed to be manually chosen, which meant they were not very well optimized. A random search may have performed better, but the space to search them over was simply too large to be practical, especially when these parameters should be optimized precisely.

The results achieved were similar to that of the industry standard. However, the results in this paper were more impressive, as ChessVisionAI has been trained for far longer on a far larger and more robust data set. This demonstrates that training a

ChessVisionAI for a more concentrated purpose that is only centered around videos/screenshots leads to better results at the cost of flexibility, though flexibility isn't really needed in the applications mentioned above.

Second, it is shown that using predictive adjustments to affect the model can be beneficial, especially when considering that the parameters for the predictive adjustments in this model were not chosen optimally at all. There isa lot of potential possible for optimizing chess vision AI's by using the information of what positions are most likely to appear in a chess game.

It was also very surprising how well the linear classifiers performed. These classifiers are generally not well suited to computer vision tasks, but the combination of a small amount of easily recognizable categories and the small size of each piece meant that the linear classifiers still performed quite well. If there was no variation in piece/board sets, it may be possible for the linear classifiers to outperform the CNNs.

Other Failed Approaches

One approach which was tried that was not as effective was using Harris corners to find the chess board. Due to the different piece sets and board themes, it was very difficult to find the right parameters such that only the chess board corners were detected and nothing else. Often, there would be more than one pixel detected as the corner, so a clustering algorithm would have been necessary to find the exact location of the corners. Another step would then be necessary to find the horizontal and vertical lines of the chessboard, and this then is prone to error due to the margins of error in the clustering.

Another failed attempt was using the other corner/edge detection algorithms built into OpenCV such as ORB, SIFT, and findchessboardcorners(). The reason behind this is that the chessboard is often a small portion of the input images, and these algorithms are much more generalized and meant for a context far greater than just chess boards. Hence, tuning the parameters for these functions was very difficult and locating the chessboard was prone to error.

Limitations/Future Work

The main limitation of this result is the lacking dataset. While Lichess has many piece sets and board themes, it has a very uniform background which it makes it easy to identify straight lines using gradients. The same result may not be replicable on other chess sites or chess videos where there may be scenes of people talking in the background. Livestream chess overlays may also have other easily identifiable straight lines.

Future work could include expanding the dataset to include a wider diversity of images containing chess boards, or perhaps applying the result of this paper to chess videos.

Another area of future work would be expanding on the predictive adjustments at the end of the piece classifier. Perhaps a fully connected layer could be added at the end that incorporates all 64 pieces on the chess board, so that the final layer could learn what positions are most likely to appear in a chess game, instead of having to manually code in those parameters.

5. References

[1] Stuart Bennett and Joan Lasenby. ChESS – Quick and Robust Detection of Chess--board Features. 2012.

[2] Jay Hack and Prithvi Ramakrishnan. CVChess: Computer Vision Chess Analytics. 2014

[3] Georg Wölflein and Ognjen Arandjelovic. Determining Chess Game State from an Image. 2021

[4] Ding, J. ChessVision: Chess Board and Piece Recognition. 2016

[5] Danner, C.; Kafafy, M. Visual Chess Recognition; 2015

[6] Xie, Y Tang, G.; Hoff, W. Chess Piece Recognition Using Oriented Chamfer Matching with a Comparison to CNN. In Proceedings of the IEEE Winter Conference on Applications of Computer Vision, Lake Tahoe, NV, USA, 12–15 March 2012

[7] Gudbrand Tandberg. ChessVision. 2020
[8] Abhishek Roy. ChessPutzer. 2018
[9] Chessvision. Jialing Ding. 2016
[10] https://database.lichess.org/
[11] Code library: Selenium. https://selenium-python.readthedocs.io/
[12] Code library: Pytorch. https://pytorch.org/
[13] Code library: Tensorflow. https://www.tensorflow.org/
[14] Code library: OpenCV. https://opencv.org/