Over the Board Chess Digitization

Kyle Coon
kylecoon@umich.edu

Abstract

As a result of the COVID-19 pandemic, the online chess community saw a massive surge in player numbers. Most of these players learned to play chess exclusively online and have continued to do so, both for its convenience in finding opponents but also for the immediate feedback it can give you in the analysis of the effectiveness of your play and predictions on which player has the upperhand at any given moment in a game. Chess that's played in real life, also known as Over the Board (OTB), offers none of these benefits and, oftentimes, players who learned to play online will experience a notable decrease in aptitude when switching to OTB due to the unfamiliar environment of pattern recognition. Many applications and websites allow for the layout of the pieces on a board to be inputted manually, but this is a tedious task. To bridge the gap between these two styles of play, I am proposing an application that can take an image of a chess board at any point in the game and digitize it into a format suitable for input into the famous chess engine Stockfish for analysis.

1. Introduction

An implemented solution to this problem essentially boils down to identifying what piece lies on any given square of a chess board, as well as identifying the color of that piece, with the options for both white and black being pawn, bishop, knight, rook, queen, king, or nothing. If that can be achieved, all that needs to be done is repeating that task of identification 63 more times, once for each square on an 8x8 chess board.

If the type and color of each square can be identified, the 2D-array-like structure of a chess board grants great ease at interpreting which pieces lie on which squares, which can then be translated into Forsyth–Edwards Notation (FEN). The specifics of FEN will be addressed later on, but it is essentially a way to format the positions of any layout of chess pieces into a standard string that chess engines, like Stockfish, can interpret. Conveniently, the popular online chess website lichess.org has a handy feature where if you have a FEN string, you can attach it to the end of the url https://lichess.org/analysis/standard/ and it will input that given chess board layout into Stockfish. Therefore, the final objective of this will be to translate a picture into FEN and lichess.org can handle the task of analysis.

To identify the piece on a given square, I will use a Visual Geometry Group convolutional neural network that is trained on a data set of several thousand individual squares of a chess board from real-life chess scenarios. I can then use this model to take an image of any chess board and translate it into FEN. No widely available dataset of this type exists to my knowledge, so I will be collecting it myself with my own chess board. To avoid the need for a hyper-specific and sensitive setup where the camera and board must be in the exact same position each and every time, I will additionally be creating a program to preprocess and normalize the input so that both the data being used to train the neural network and the images being guessed by the trained model will have the same formatting, no matter how the board and camera are oriented.

2. Related Work

2.1 Heatmap Approach

Several implementations exist for the digitization of OTB chess, and just about all of them include the use of a Visual Geometry Group convolutional neural network on a data set of many individual squares. But where their methods diverge is in how they preprocess the data. The main problem lies in how you are going to recognize and identify where a chess board is in an image and what lies on that chess board.

One solution to this is a heatmap approach[1]. This process involves looking at sub-areas of an image and comparing them to preexisting images of chess boards to determine if a chess board exists there. While this approach is accurate at determining *if* a chess board exists, it is not as effective at pinpointing precisely *where* a chess board exists. It can tell you the general area, but I need pixel-precision to create an effective neural network. I already know a chessboard will exist in the input I am giving the neural network, so more sensitive image collection will allow me to sacrifice sophistication of the chess board detection.

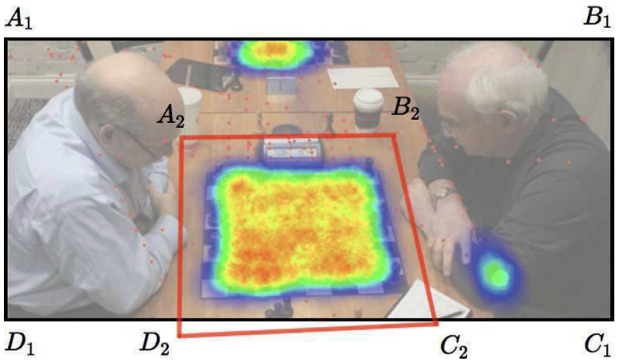

Figure 1: Use of a heat map to identify the location of a chessboard in an image. Notice a second, smaller chess board is also being detected at the top of the image.

2.2 Neural Network for Preprocessing

Another approach is to use a neural network to identify where the corners of a chess board are in a given image[2]. While this methodology is very accurate if implemented well, it involves hand-labeling the exact corners of dozens to hundreds of images of chess boards so the model can be effectively trained. I do not feel like hand labeling that much data, nor do I want this project to just be two neural networks, so I will take a different approach.

2.3 Straight Line Detection

A third approach to identifying the precise location of a chess board in an image is to take advantage of the grid-like layout of the chessboard to identify where straight lines lie in the image, then take the outermost intersections of those lines to locate where the corners of the chessboard are[3]. This is also a fairly effective approach, and is most similar to the approach I'll be taking, but I have come up with one that is much more mathematically simple.

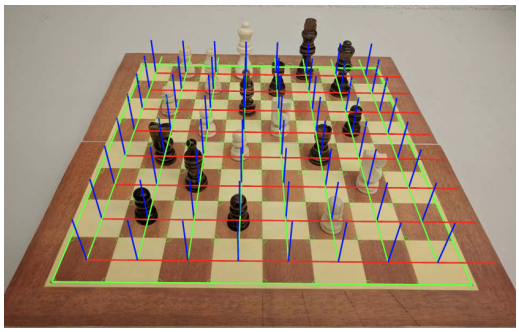

Figure 2: Use of straight line detection that also factors in the 3D-dimensional angularity of the input image with the normal vectors of each square.

3. Method

3.1 Chess Board Detection

Since I'm collecting all of the data, I can ensure that a chess board will exist in the images and that the input will be relatively consistent. The bare minimum amount of information that is needed to preprocess this data is where the corners of the chess board lie. If this can be found, I can easily create an orthoimage of the chess board using the warpPerspective() function from the Open Source Computer Vision Library.

I began by blurring the image with a simple blur kernel. This reduces the chance of small details like the wood grain in the chess board being recognized as a significant location in the image. Then, I shifted the image to a grayscale representation so it could be efficiently binarized, resulting in an image that is effectively an outline of the chess board. This can be fed into the Open Source Computer Vision Library's goodFeaturesToTrack() function. Using a sliding window, this function will locate all the points of interest where a corner is likely to be in the image. We can take this list of notable points and find the 4 coordinates that have the closest euclidean distance to the corners of the entire image and this will give us the corners of the board. After using these to create an orthonormalized version of the chess board, I zoomed in by about 6% since my chess board has a frame around it that is irrelevant. With this, we can easily divide the rows and columns by 8 to isolate each square on the chess board. Now, one image has given us 64 data points to use on our neural network.

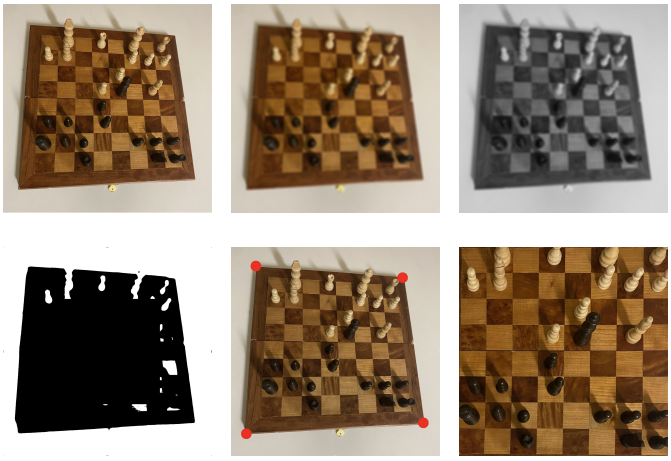

Figure 3: From left to right, a visualization of the steps taken to locate the corners of the chess board and create an orthogonal representation of that image.

3.2 Neural Network Model

Now that we have a way to generate a dataset, we can construct a neural network that will train with that data. For this, I have made a Visual Geometry Group convolutional neural network that takes in images of individual chess squares and assigns one of 13 labels to the image (6 kinds of white pieces, 6 kinds of black pieces, and an empty square). Because there is not a lot of variance in what an input image may look like, I was unsure if batch normalization would be needed so I decided to construct two neural networks, one that has batch normalization and one that does not and will eventually compare which one is more accurate.

Similarly on the note that input images of different labels still may look relatively similar, I want the images being used as input to have a relatively high image quality so as much distinction could be made between them as possible. I went with 256x256 images which, when scaled by the eventual dataset, was far too large for Google Colab to want to handle. Thankfully, I own a fairly powerful MacBook Pro so I transferred the project to run locally and was able to train the models.

The neural network with batch normalization and without batch normalization had the following architectures respectively:

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3),
          stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2,
          padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3),
          stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2,
          padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 128, kernel_size=(3, 3),
          stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(128, 128, kernel_size=(3, 3),
          stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2,
          padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(5, 5))
  (classifier): Sequential(
    (0): Linear(in_features=3200,
          out_features=512, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.3, inplace=False)
    (3): Linear(in_features=512,
          out_features=256, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.3, inplace=False)
    (6): Linear(in_features=256,
          þut_features=13, bias=True)
```

```
VGG-BN(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3),
          stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1,
          affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2,
          padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 128, kernel_size=(3, 3),
          stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1,
          affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2,
          padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 128, kernel_size=(3, 3),
          stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(128, eps=1e-05, momentum=0.1,
          affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): Conv2d(128, 128, kernel_size=(3, 3),
          stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(128, eps=1e-05, momentum=0.1,
          affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2,
          padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(5, 5))
  (classifier): Sequential(
    (0): Linear(in_features=3200,
          out_features=512, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.3, inplace=False)
    (3): Linear(in_features=512,
          out_features=256, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.3, inplace=False)
    (6): Linear(in_features=256,
          out_features=13, bias=True)
```

Figure 4: Left is the architecture for the neural network without batch normalization, right is the architecture for the neural network with batch normalization.

3.3 Solving a Board

Now that we both can format an image into 64 uniform squares and have a trained neural network, we can pass each of the 64 squares into the trained neural network to make its prediction of that squares piece type. Based on its prediction, we can append a character to a string that will eventually be in FEN. In FEN, white pieces are denoted with uppercase characters (‘P’ = white pawn, ‘B’ = white bishop, ‘N’ = white knight, ‘R’ = white rook, ‘Q’ = white queen, ‘K’ = white king), black pieces are denoted with lowercase characters, (‘p’ = black pawn, ‘b’ = black bishop, ‘n’ = black knight, ‘r’ = black rook, ‘q’ = black queen, ‘k’ = black king), empty squares are denoted with an integer 1-8 where the integer is how many consecutive empty squares are in a row, and each row is broken up with a ‘/’ character. Since I was already traversing each square on the chess board to predict its label, translating the predictions to FEN was a very intuitive process.

4. Experiments

4.1 Board Recognition

I did not conduct a formal experiment on how well the image preprocessing algorithm performed due to the fact that it is so straightforward and as long as it receives rational input, it behaves exactly as expected. However, it is quite sensitive to the conditions surrounding the chess board. When collecting input images, a few of them had the bag that the chess pieces go into in the frame and it then detected that as a corner, which makes the image unusable after it attempts to orthonormalize it. However, as long as I had a decent environment for the framing of the image, the preprocessing worked flawlessly.

Figure 5: Preprocessing algorithm’s attempt at orthonormalizing an image when other objects are present in frame.

4.2 Piece Recognition

To create my dataset, I used 32 images of chess boards with different setups of how the pieces would be realistically organized based on games played by chess professionals. Each chess board has 64 squares on it, so with only 32 pictures I was able to generate a dataset with 2048 images. However, in a real game of chess, there are only ever at most 32 pieces on a board meaning at least half the squares are empty. Using a dataset where over half of the input is the easiest input type to recognize is not a practical use of resources, so I deleted over half of the input of empty squares and was left with a data set that contained 1415 training samples, 303 validation samples, and 303 test samples. This isn't a massive amount of data, but the limited number of labels and conformity of input formatting balances that out. For each type of neural network, I experimented with different configurations of hyperparameters and eventually landed on this:

Learning Rate: 0.01
Momentum: 0.9
Batch Size: 16
Num Epochs: 20

These hyperparameters produced these results for each neural network:

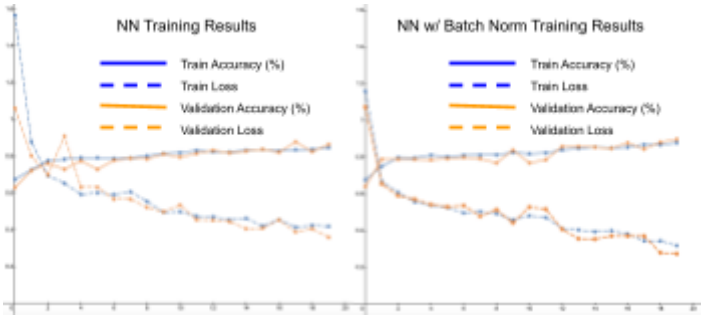

Figure 6: Accuracy-Loss graphs of training and validation sets.

Using a nearest neighbor test to determine accuracy, the neural network without batch normalization was able to achieve a top-1 average accuracy of 87.458% whereas the one with batch normalization was able to achieve a top-1 average accuracy of 89.768%. Interpretation of these results implies the neural network is able to correctly guess what piece type a given image is just shy of 9 out of 10 times. Furthermore, the top-5 average accuracy of the non-batch normalized model was 99.66% and the batch normalized model was an astounding 100%. This means that both neural networks practically always had the correct piece at least in the top-5 guesses. Evidently, the batch normalization did not have a drastic effect on the accuracy, but we'll take any percentage point benefits we can get. Therefore, when running the final image solver, we will use the model trained with batch normalization.

Figure 7: Input image for final test.

With Figure 7 as our input image, the application finally produces the url:
https://lichess.org/analysis/standard/8/3P3k/n2K3p/2p3n1/1b4N1/2p1p1P1/8/R2B4
On relatively sparse input like this one, the model is quite successful. However, using input where not many pieces have been captured yet and the board is very cluttered creates consistently flawed output. Where it struggles the most is when tall pieces towards the top of the board exceed the frame of the square they are in and when shorter pieces are covered by tall pieces. I was very pleased to see that the model correctly detected the pawn that's partially covered by the king in Figure 7, but it is typically not able to do so with great success.

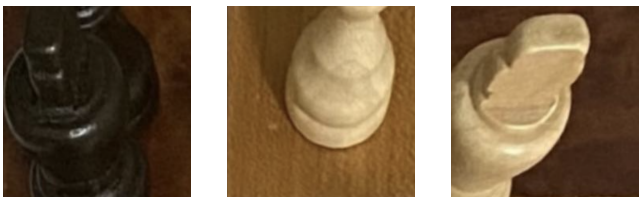

Figure 8: Examples of difficult input. Left should be labeled a black bishop despite the king blocking it, middle should be labeled as a king despite being mostly out of frame, and right should be labeled as empty despite the king blocking it.

5. Conclusions

In all, I'm very pleased with the success of this project. The neural network saw many tweaks and improvements that brought it from an 8% accuracy when I first started to over 89%! Ideally with something like this, you would want something at almost 100%, and I think the main way to do that would be to overhaul the way I was preprocessing data. Tall pieces that exceed the bounds of their square are very difficult to distinguish with my design. If I instead used a shape-recognition design where entire pieces are considered, I believe the accuracy would greatly increase. However, I realized this fault in the design when I was already well past the point of no return, but I'm pleased the approach was still able to achieve moderate success. Thanks for a great semester!

References

[1] Maciej A. Czyzewski, Artur Laskowski, and Szymon Wasik. Chessboard and Chess Piece Recognition With the Support of Neural Networks. *arXiv:1708.03898v3*, 2020.

[2] James Gallagher. Represent Chess Boards Digitally with Computer Vision. *roboflow.com*, 2023

[3] Youye Xie, Gongguo Tang, and William Hoff. Chess Piece Recognition Using Oriented Chamfer Matching with a Comparison to CNN. *Proc. of Winter Conference on Applications of Computer Vision*, 2018