CSCI 1430 Final Project Report: Chessboard Recognition Without Deep Learning

Video Games: Joshua Hill, Mehek Jethani, Joel Manasseh, Justin Zhu
Brown University

1. Introduction

The goal of this project is to develop an algorithm that takes in images of a chess game from start to finish, and outputs the chess board graphically. This would help chess tournaments and other major chess events because organizers would not need to manually display chess graphics; they would be able to easily and automatically create a graphical representation of the board, just from players hitting a button (which can be easily attached to a chess clock).

We have chosen to limit ourselves by not using deep learning/neural networks for piece classification as this is a task that has been solved many times already. The problem at hand is solvable without using deep learning, and thus we have chosen to avoid it to be able to explain more in-depth how the task is done. By using simpler computer vision techniques for feature detection such as Canny Edge Detection and Hough Line Transforms and tracking the board state with our own game logic class, we can accomplish the task without convolutional neural networks.

Though there is no definitive quantitative measurement of success, we were able to produce fairly consistent results in recognizing the locations of pieces on the board. We faced some limitations due to natural and hardware constraints as well as the fact that we did not fully implement all forms of error checking to ensure legality of moves and account for special cases such as castling and en passant. However, we were able to achieve our fundamental goal of building a program that can digitize a physical game of chess without relying on deep learning methods.

2. Related Work

A lot of prior work exists in the area of chess detection [3] [4]. Though there are various ways to approach the board detection, one common method that appeared throughout the sources we consulted was using Hough Line Transforms to find the grid lines of the board and use the intersection points to find the corners of each square, which is something that we carried over into our project. The common factor in every one of these methods was the use of deep learning strategies, typically convolutional neural networks, for piece detection and classification to determine the board state. In order to try and deviate from a task that has already been accomplished many times in the same way over and over again, we came up with our own solution unlike anything we found in any of the sources that we surveyed.

As for external tools utilized for this project, we relied heavily on OpenCV for reading camera input, image preprocessing such as color conversion and blurring, and line and edge detection algorithms. We used the python chess and chess-board packages for displaying the digital GUI of the board state as predicted by our algorithm. Lastly, we did use modified versions of some code that we found online through the help of Stack Overflow for smaller helper functions involving the filtering of overlapping Hough lines [1] and the conversion of a matrix of piece representations into a FEN string to be read by the GUI package [2].

3. Method

In order to explain our methodology, we will start from the desired end-product, namely a chess board display that shows the current chess board based on an input of one image taken every turn. Based on the work that other people have done, they chose to do piece detection (i.e. telling the difference between a rook or a knight or a queen or etc.) using deep learning, which is outside of the constraints we set for ourselves with this project. Since there is no easy way to do piece detection otherwise, we decided to try to detect whether a square is filled or not, as well as if the piece is white or black to update the game state (though in practice, the latter task did not achieve the best accuracy). Doing this allows for us to have full control over the entire process without the black box logic of CNNs and even the reliance on the continuous capture of moves to update the game state using logic fits the use case of clocking moves in a chess game.

For the computer vision aspect of the project a lot of our work focused on being able to identify the grid of the board and separate the individual squares to check if each one is filled and store a representation of the boards filled and empty squares as 1s and 0s respectively. Based on research described in the earlier Related Work, we settled on using

Hough Line Transforms. When the program is run the live video feed of the webcam suspended above the chessboad is read using OpenCV and upon pressing space, an image is captured. This image goes through some basic preprocessing such as conversion to grayscale and blurring, before we perform Canny Edge Detection and then Hough Lines on that output. We found this method to be very effective for the task as it is pretty much always able to detect all of the grid lines as long as the full board is in clear view of the camera. This method also proves to be strong where other comparable algorithms might fail such as Harris Corner Detection because it is very robust to various levels of partial occlusion of the grid lines. Even when the camera is at an angle and the pieces may overlap the boundaries between the squares, it is able to detect strong enough edges to figure out where the lines are.

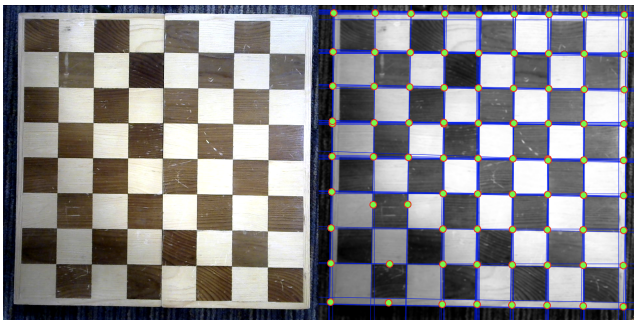

Figure 1. One example of the result of running KMeans on the Hough line intersection points to try and get the grid corners.

OpenCV's Hough Lines detects multiple lines for every edge in the image, so we could not simply take the intersections of all the lines as the corners of the grid squares or we would end up with many duplicates. Our first approach to remedy this was to use KMeans clustering to get 81 cluster centers for the points, but due to variations in the number of lines found for different edges, the intersections were very unevenly distributed throughout the grid leading to some cluster centers doubled up around a specific corner and other areas where the center would end up averaging out to the middle of two corners. Clustering the lines themselves produced a similar predicament. The end solution we went with was to iterate through all of the lines and take the first full set of grid lines found where none of them are within a certain threshold of each other so that no lines that are too close together and correspond to the same edge are chosen.

To account for some final issues we faced with this, we made sure to filter out any stray diagonal lines that don't fit the alignment of the grid and to separate the lines into vertical and horizontal before choosing our distinct edges. This ensured that we would always get 9 horizontal and 9 vertical lines for the grid and that we wouldn't need to waste time finding intersections between lines that are parallel on the board.

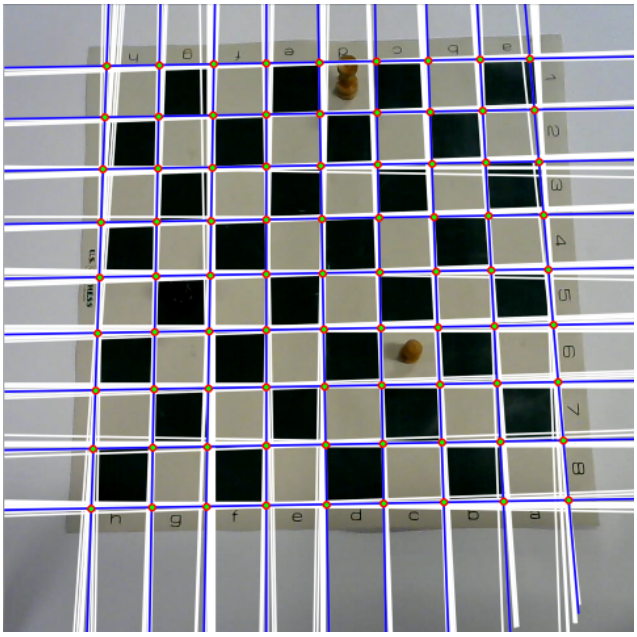

Figure 2. The white lines represent the output of OpenCV's Hough Lines. The blue shows the lines that we isolate through our filtration process. The red outlined circles are the final grid corners determined by finding the intersections of the blue lines.

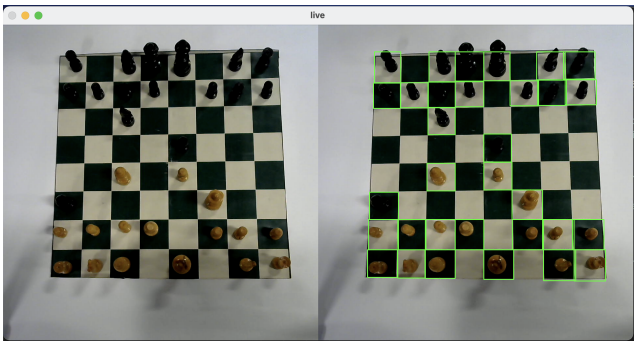

Figure 3. Here the squares identified as being filled are marked with green bounding boxes.

We could then take the grid coordinates and order them into a matrix that fits the grid. The corners could be used to perspective warp the image and look at the central area of each square to detect Canny Edge features indicative of a piece filling the spot. We would also store the average intensities of the features to be compared to the previous image so that in the event of a piece capture, we could check each filled square for changes in intensity and rank them to figure out where the color of a piece in a square changed indicating a capture.

(As an aside, our initial approach to classifying filled versus empty was to have users take a calibration image of the empty board and store the features of this grid, then take an image of the board with the starting setup and rank the

differences between the features. We could then take the top 32 differences and identify this as being our cutoff for what the difference between an empty and full square should be. Then on each move we could compare the new features to the calibration ones and use our determined threshold to choose filled squares. However, we found that the calibration features were pretty much always all 0 since no edges were detected within the plain squares so we hard-coded this threshold to be 0.)

Thus, our computer-vision side of the project would output a 2D array of 1s and 0s, where 1 represents a position on the chessboard filled by a piece, and 0 represents a position that is empty. Assuming we set up the board correctly at the beginning of the game, we can figure out which pieces went where based on these outputted 2d arrays of 1s and 0s. We save the state of the board (both its filled or not representation and its actual representation) in the game.py file. Each turn/time we get a 2d array, we first compare the number of pieces remaining on the board (calculated as the number of filled in squares) with the saved board state.

If there's the same number of pieces, there is no capture, and we can just compare the two filled in grids to find the change. We find the coordinates of that change and then update the board with the piece representation with the same coordinates.

If there is a different number of pieces, there is a capture, and we try to find out where the capturing piece went. Note that in certain cases in chess, a piece has the option of capturing multiple different opposing pieces, but if we were to only use this filled in versus not grid, we would be unable to tell the difference between the piece taking one or another options.

Therefore, we decided to have an extra element of our image output, where we take all of the filled squares and select the one with the greatest absolute change in intensity. This would theoretically be the capturing piece (i.e. the white queen taking the bishop, or the white queen taking the pawn), as the chess pieces differ in color significantly. Then we return the coordinates of that piece for the game logic.

We imposed some constraints on the project in order to make it more feasible. We chose one chess board with one particular set of pieces, and set up a webcam attached to a computer such that the angle was somewhat consistent over all of our testing. We also imposed that players of the chess game only make legal moves, and they are also not allowed to castle. We assumed the camera and chess board would remain consistent. We attempted to adjust for lighting as much as possible, avoiding glare-filled environments.

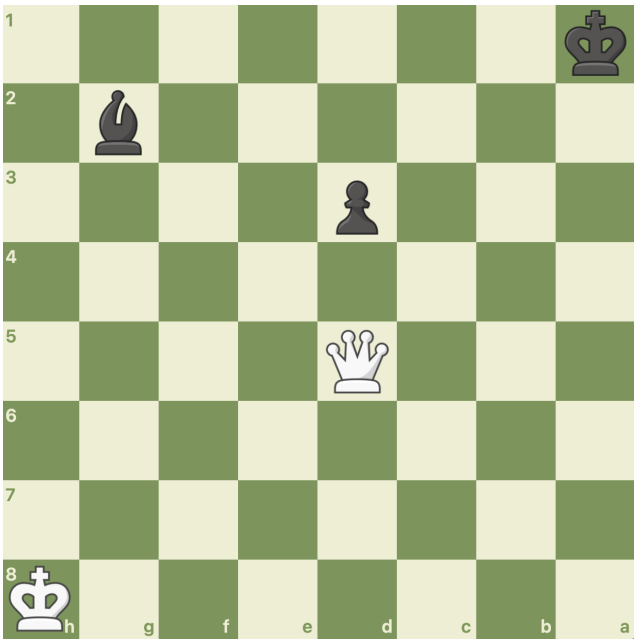

Figure 4. Suppose it is white's turn to move. The white queen can take either the bishop at 2g or the pawn at 3d, which the program would output the same filled/not filled grid for both. This leads to an ambiguous result.

Figure 5. Digitization of board from figure 6 image. We convert the right image of figure 6 to this image via the game logic and a sourced board display

4. Results

Our project ended up doing fairly well overall. Because we did not do deep learning, we didn't truly output a training accuracy, instead relying on tests and the demo to give us an idea of how we are doing.

For the most part, we were able to accurately show the board. We had some difficulty with piece capture, namely

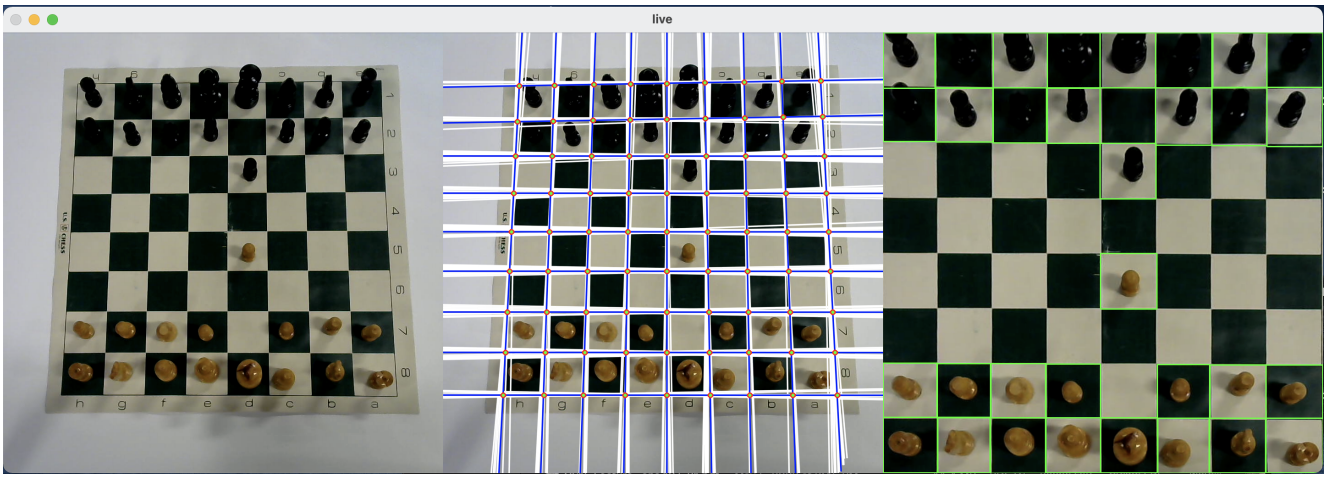

Figure 6. Full sequence of detection before figure 5 digitization *Left:* Live feed of board from webcam. *Middle:* Line detection and intersections. *Right:* Warped board and piece detection

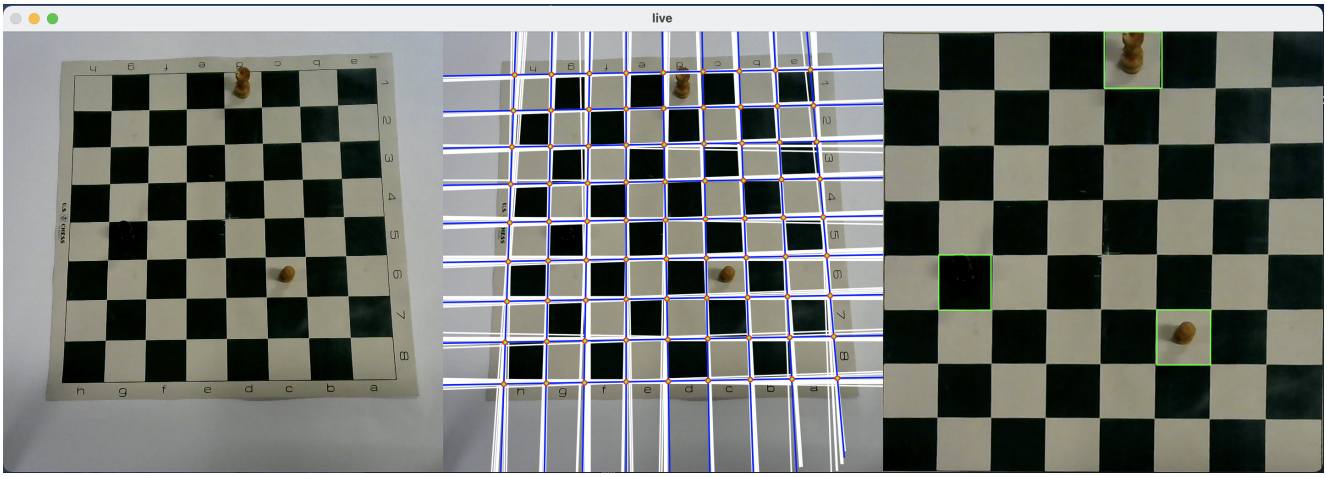

Figure 7. An example of a situation where the algorithmn was able to detect a piece despite how difficult it is to see through the webcam lighting.

sometimes the wrong piece would show up as captured.

Our computer vision techniques mostly worked fine, but we encountered issues with finding pieces given unfavorable circumstances due to glare from lighting or the board not showing up as lit in the live feed from the webcam. Therefore, in demos and testing, we saw issues detecting dark pieces in dark squares. Even in these cases, it was often hard for us to be able to see the piece on the board through the webcam using the human eye so this was just an unfortunate limitation of our hardware and situational lighting. Due to the nature of our time-based approach to tracking moves, this meant that if the program was unable to find all the black pieces, it would often mess up the logical state of the board and incorrectly identify future moves.

Because our game logic centered around the filled board state, we were unable to implement full chess logic such as checking for illegal moves, and more advanced plays such as castling and en passant (however, the latter might be able to be detected in some edge cases due to our capture logic).

One thing we noticed in testing was that our algorithm was able to correctly detect filled squares with surprising accuracy, sometimes finding pieces in situations where we couldn't even identify the pieces in the webcam feed with the human eye like with black pieces on black squares, however the game logic could sometimes get messed up, and the digitized version does not always match. In practice, we have low accuracy on piece captures when it comes to visualizing it on the screen. Even when the boxes are drawn correctly outlining the filled squares, the digital board is wrong. This suggests problems with our capture game logic or our method of checking the change in piece color to find the capture location.

Here is a link to a short video demo of our project / results. Unfortunately we forgot to capture a longer sample when

demo-ing it on the presentation day but it is possible to play a longer sequence of successfully identified, this one was cut short by the aformentioned capture issues.

4.1. Technical Discussion

When we originally started the project, we were convinced that we would need to use deep learning methods somewhere in the process, whether that be for piece detection or for moves. However, after reviewing our goals for the project, we realized we could complete the tasks using computer vision methods alone.

While this method definitely helped with the pace of the project and ease of understanding for others, it can be argued that a trained mode, might be able to improve the robustness of the project by giving us new methods for piece and move detection. In our project, we had to rely on the game logic to help define pieces which meant some moves on the board could end up incorrectly in the digitized version of the game because it mistook one piece for another.

Nevertheless, we were able to find methods, including comparing board states and setting a start board, to bypass the need for deep learning with fairly accurate results. This decision also meant we had to trial various methods for singular piece detection in each of the squares. We discussed and shifted between comparing intensity changes and simply performing edge detection on small boxes within each square.

4.2. Societal Discussion

We were given five ethical concerns about our proposed project. Here are the concerns and our answers to them.

1. Video constraints: a video of a board game can encompass a very broad scope of possibilities. A video of a chess game could be a virtual match between players which could be fairly easy and standardized for an algorithm to read, but it could also be an in-person match between players, in which the board may not be easily visible or it may be difficult to detect pieces. For games with even more varied mechanics like Monopoly and its cards/buildings, this is an even bigger problem for being able to detect moves. As such, is a required standardized video format necessary? If that is the case, then how can you enforce all users to record in a certain way?

   Given the increased constraints of our problem, we see this as fairly reasonable to implement. Chess games (especially those in tournaments) often require players to use a clock, which they hit a button every time they make a move. We would just hook up a camera to there and just take a picture every time the clock is hit. Photos would most likely be standardized in some meaningful way, though the way our code is written it doesn't really matter too much the exact calibration.

2. Chance of error: Suppose that your project were used in an actual tournament, let us assume for chess. The judges use the algorithm to track previous moves, and use the data to better stream the match as well as check for inconsistencies and cheating. However, what if an error occurred? Consider the situation where your project incorrectly documented a chess move, and the move that your model recorded was an illegal move. In this case, the judges who review the case will incorrectly disqualify the contestant.

   Given our test accuracies as well as nature of the product, it seems highly irresponsible to even suggest using this product as the lone judge of an illegal move or not. Instead, judges/legal game states should be entirely separate from the technology, as the technology is meant to display the chess board to an audience, not 100% accurately record it.

3. Abuse: Similar to above, your project most likely has a margin of error larger than that of a human. There is a nonzero chance that a player can come to understand the inner mechanics of your program and how to consistently make it misreport moves, and thus use this exploit to their advantage. This player may thus abuse your algorithm by doing moves that will trip up the program into either reporting the opponent as cheating or miss an illegal move the abuser did, unfairly giving them the win.

   The technology is meant to help visualize the board state, not report the correct board state to some machine. Potential abusers will have little reason to take advantage of the system, save for causing their potential audiences headaches when the chess board is displayed inaccurately.

4. Liability: In the case that either one of the two cases above occurred, who is liable for this? Should it be you, for making the algorithm? Or should it be the judges who used it despite the fact that models are prone to error?

   It seems reasonable that liabilities would fall on judges or tournament organizers, as this technology is not meant to have a significant impact on the outcome of games or tournaments. Instead, it is meant to display the chess board to an audience without the need of a human who can make mistakes. Therefore, any liabilities that may occur happen as the result of poor choice of usage of the technology, and thus fall on the users of said technology.

5. Consent: What happens if one player does not consent to being recorded?

   We will only be taking pictures of the board state, and no physical aspect of any players will be documented. In addition, the board state is visible to any passerby, and thus is not of ethical concern to us if the players wish to keep their chess game private.

5. Conclusion

We set out to do this project with the goal of recognizing and digitizing a chess game for ease of visualization. In the end, apart from a few fixable bugs, we achieved that goal. One of the good aspects of this project is that it has much potential to be extended, in the game of chess and other board games as well. In a fairly uncomplicated way, we could print out a log of moves that occur throughout a chess match like is done in online chess platforms. We can also change parameters to play other board games such as checkers or Othello since we do not use deep learning algorithms to make it chess specific.

References

[1] Choosing lines from hough lines, 2014 Accesssed: 2022. https://stackoverflow.com/questions/25303396/choosing-lines-from-hough-lines. 1

[2] Generate chess board diagram from an array of positions in python?, 2019 Accesssed: 2022. https://stackoverflow.com/questions/56754543/generate-chess-board-diagram-from-an-array-of-positions-in-python. 1

[3] U. Andrew. Board game image recognition using neural networks, 2020. https://towardsdatascience.com/board-game-image-recognition-using-neural-networks-116fc876dafa. 1

[4] B. Saurahb. Convert a physical chessboard into a digital one, 2019. https://tech.bakkenbaeck.com/post/chessvision. 1

Appendix

Team contributions

Please describe in one paragraph per team member what each of you contributed to the project.

Justin Zhu I wrote the game logic! This mostly involves the logistics of how to take a 2d array of filled vs not-filled (1s and 0s) and read it into a chess board format. I focused on creating the logic that compares each state to its past state and using logical deduction to figure out what pieces are which filled squares. I also tried to handle piece capture, though it seems that is quite inconsistent. I additionally helped with the physical set up as well as being a (metaphorical) rubber duck.

Mehek Jethani I worked heavily on the board square detection, so getting the Hough Lines and filtering them out to make sure that we only ended up with the distinct lines of the chessboard grid to get the correct corners. I also worked on the detection of filled versus non-filled squares using Canny Edge Detection. Lastly, I implemented the OpenCV part of the demo interface so reading and displaying the live camera feed plus the captured image with the plotted grid and the warped image of detected squares.

Joel Manasseh I helped with testing and debugging of the game logic. I also assisted with the process of ideas and techniques for some of the board square detection, including the feature detection of the pieces. I also helped with the poster content and design.

Josh Hill I worked on the board detection and feature recognition. I mostly was involved in using our detected lines to get a matrix of intersection points. I additionally worked on using this matrix to warp the perspective of the board into a more readable, flat, 2D format; then ultimately storing the features for each square on this image.