Digitizing Chess Games

Aditya Pillai, Advait Patel, Dhruv Agarwal
CS 543 Computer Vision at the University of Illinois Urbana – Champaign
Thursday, December 19, 2024

This project addresses the challenge of analyzing in-person chess games without requiring manual tracking of the board. Variations in lighting, board designs, and piece appearances complicate this task, but we aim to solve these challenges by designing a fully automated pipeline capable of digitizing the game state.

Our approach employs computer vision techniques to identify and track chess pieces in real-time, converting the detected game states into FEN (Forsyth-Edwards Notation) strings for each move. These FEN strings enable accurate game analysis and replay. The method is designed to generalize across various standard chess boards with minimal setup requirements.
Key Findings:

- Robust Game State Detection: The system achieved high accuracy under varying lighting conditions and board designs.
- Automation of FEN String Generation: Our pipeline efficiently converts visual board states into FEN strings, streamlining game analysis.
- Scalable Approach: The generalizability of the model allows it to handle diverse chess board configurations without manual intervention.

Table of Contents

Introduction 2
Methodology 2
Technical Challenges 5
Results 6
Conclusion 6
Connection to Course Material 6
Statement of Individual Contribution 7

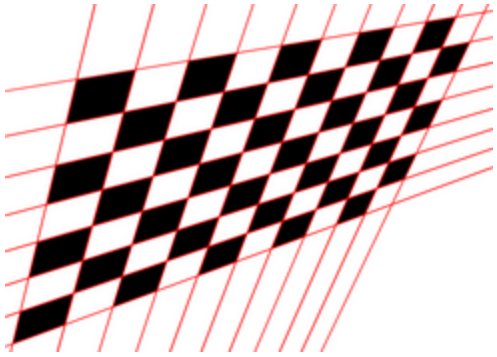
Introduction

The process of analyzing an in-person chess game without the necessity of an individual to manually track the board remains a challenging task. It is especially difficult when considering the variations in lighting, board designs, etc. The purpose of this project is to bridge that gap by digitizing the game state using a fully automated pipeline. We hope to generalize this solution to multiple chess boards by using a broad-level approach to identify pieces. Our approach leverages the fundamentals of computer vision to automate move detection and it is designed to handle standard chess boards with minimal setup constraints. Our method also outputs the FEN strings (strings that represent the game state) for each move, which is then used to get an analysis of the particular game state. The biggest challenge is being able to accurately identify the board state.

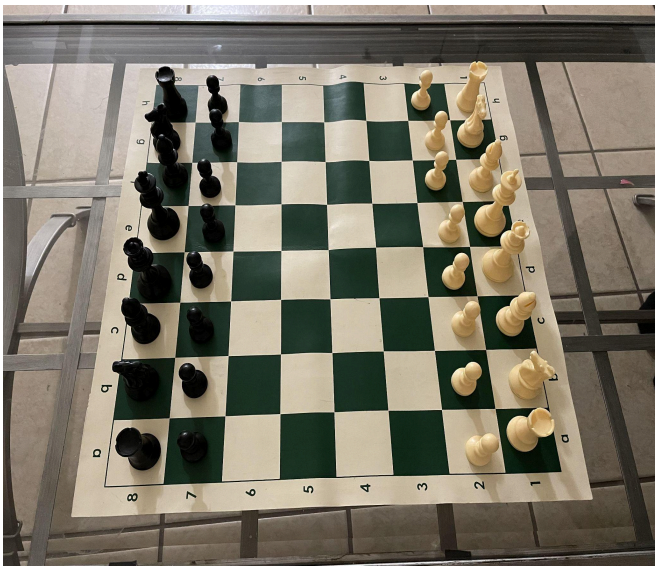
Methodology

Our computer vision algorithm can be broken down into 3 main steps. Board detection and perspective transform, followed by board segmentation into squares, finished by piece detection. Steps 2 and 3 are arguably the most important and complex steps. Below, further analysis for each step is detailed.

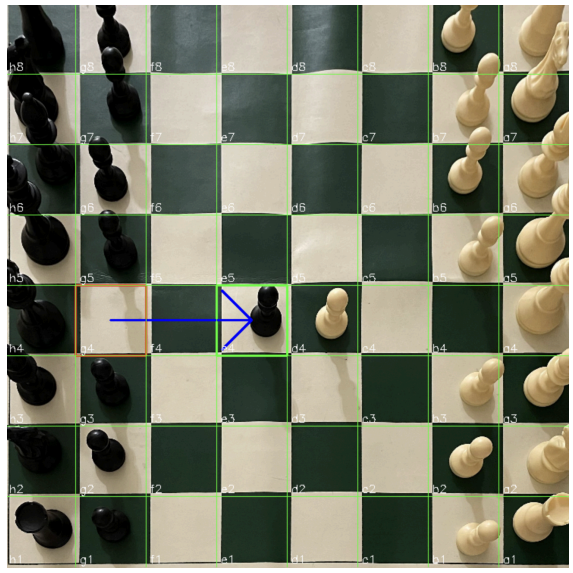
1. Board Detection and Perspective Transform
    a. Get chessboard corners
        i. In order to segment the board into 64 individual squares, we first need to detect the corners of the chessboard. Although cv2 has a function findChessboardCorners(), our testing proved this function to be fairly unreliable. Our approach uses edge detection on the furthest line near the border of the mat. Loosely, we can define a corner as the intersection of two edges. Using a hough transform, we can find the intersections we are looking for. However, this does not seem to work consistently when we introduce noise and other features into the detection frame. Additionally, upon more extremely angled views away from the birds-eye view, the edges/lines hide behind pieces more often, hiding corners. Thus, we recommend a >60° angle above the ground plane when recording games. Upon encountering outliers, this can generally be solved by manually setting the chessboard corners.
    b. Perspective Transform
        i. Getting a birds-eye view of the chessboard is important to both segment the board into 64 even squares, as well as detect pieces more accurately. We use cv2.getPerspectiveTransform() to transform the board as shown below.



Tile lines parameterized (source)



Original Chess Board



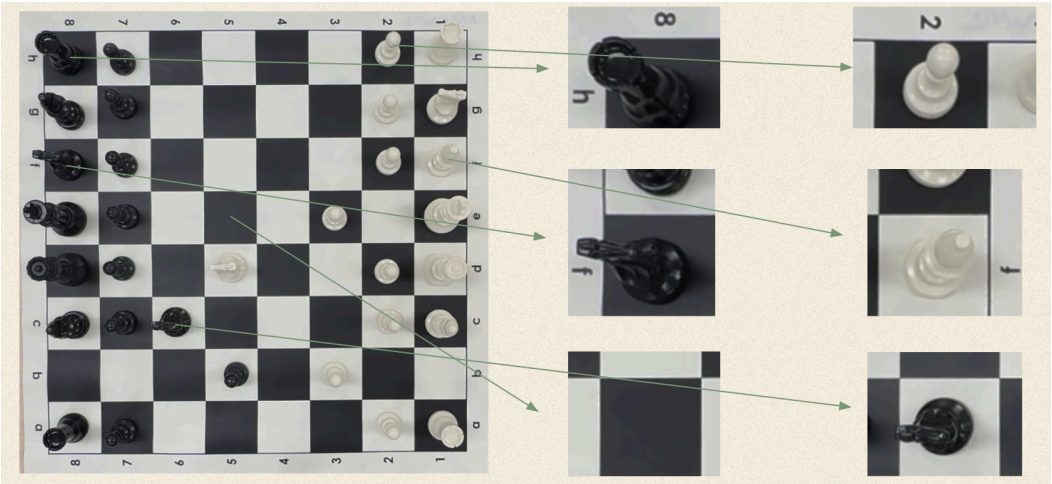
Perspective Transformed Chess Board

2. Board Segmentation into Squares
    a. Segmenting the board by dividing our perspective transformed chess board into an 8x8 grid is not the most efficient way to segment our board. The problem with this approach is that due to the perspective transform and piece heights, the tips of pieces can often be present in other squares as shown in the example above. Additionally, the edges of border squares tend to get cut off on the side they border. Thus, we use a dynamic smart segmentation algorithm to try and capture as much of every piece as possible:
        i. Border tiles
            1. We apply a different perspective transform to the original image for any border tiles. We apply a constant offset outwards to every corner coordinate, in order to capture views outside of the border, and then divide the image into 64 tiles.
            2. We also add additional padding to each tile based on what side of the board it borders. For example, if we border the bottom only, we apply a small top padding, and normal horizontal padding. If we border the top right, we apply

a small bottom and left padding. The top and right of this square will already be accounted for because of the second perspective transform.

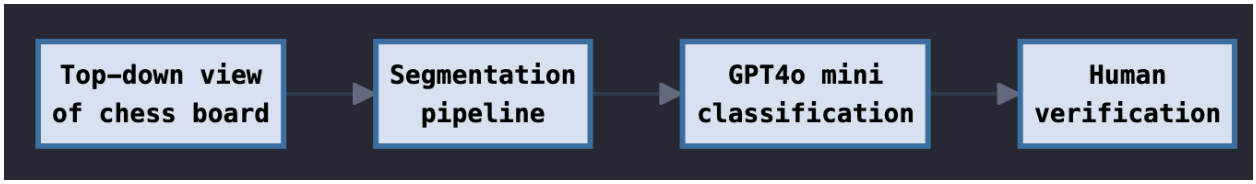
ii. Middle tiles

1. Each middle piece receives the same type of padding: constant horizontal and vertical padding, but slightly more top padding than bottom padding because pieces tend to appear more in the square above it.



Board segmentation examples

3. Piece Detection
   a. Piece classification
      i. We tried several different methods for piece classification, including using open-vocabulary object detectors, open-source and proprietary large vision-language models, and training our own neural networks on a collected dataset. We experimented with OWL-ViT, LLaMA 3.2 11B, LLaVA 1.6, Moondream 0.5B, GPT-4o mini, and GPT 4o. Unfortunately, none of them were reliable enough for our purposes. GPT 4o came the closest with an accuracy of ~60% on the dataset that we used for qualitative testing. Thus, we chose to train our own model.
      ii. We initially started with a Convolutional Neural Network with 2 convolutional layers and 2 fully connected layers. This barely learned anything at all, capping out at a validation accuracy of 25%. We then moved on to finetuning a Resnet-18 model pretrained on ImageNet. We froze all layers except for the final fully connected layers. This model trained to ~50% validation accuracy but especially had trouble differentiating between the black pieces. We hypothesized that this was because the samples in our dataset were fairly similar to each other, even across classes. Thus, the model needed to learn more fine-grained features that it was unable to with the encoder layers frozen. Thus, we unfroze the encoder layers as well, and saw significant accuracy improvements, up to 85%.
      iii. Due to our limited dataset, we also employed a number of data augmentation techniques. including color jitter, random flips, and random 90, 180, and 270 degree rotations with a 5 degree tolerance. We wanted our model to be agnostic to all of these perturbations, as this sort of information isn't relevant for the classification of the pieces.
   b. Dataset labeling



i. The figure above details our dataset labelling system. First, the segmentation pipeline extracted the relevant images from a top-down view of the board. Next, each image was fed through GPT-4o mini, which was told to classify the image according to which piece was visible, if any were visible. This was all fed into a JSONL file, which was then verified by a human using a basic TKinter GUI.

c. Detecting sides

i. Detecting which side is black or white is important, as this directly affects the notation of our FEN strings. We found the most reliable way to do this was to just identify what half of the board the first move was made on and classify that side as white, as the first move a player makes can only be on their half of the board.

4. Visualization and Analysis

a. Virtual Chessboard

i. We create a graphical representation of a chess board using the detected FEN string. As any chessboard, each square is set with a light/dark theme. We use OpenCV to render an 8x8 grid where the underlying color follows the chessboard theme and the icon of the piece is placed over top of it. We've taken the icons of the pieces from Chess.com and cached them to prevent hitting the API multiple times to replicate a given state. The board's orientation adjusts based on the detected side or white.

b. Eval Bar

i. We use a bar that is filled based on the returned evaluation score from the position (provided by the API we are hitting). The bar is displayed as a vertical strip beside the board. It is divided into two segments, the top is the white's evaluation and the bottom is the black's evaluation. We normalize the evaluation score within a range (-10 to +10 for black to white dominance). If the game is neutral, the bar is filled equally for both sides, while extreme evaluations result in a nearly full bar that favors a given side.

c. Future Move Display

i. We have a mechanism for displaying the next best move (provided by the API we are hitting). The move is visualized with an arrow that connects the "from" and "to" squares. We do this using OpenCV's cv2.arrowedLine function.

Virtual Chess Board Viewer

Technical Challenges

The development of this project presented several technical challenges that affected the team throughout its lifecycle, from the early stages of data preparation to model evaluation and refinement.

1. **Failed Approaches:**
    - **a.** Mean Intensity Difference
        - i. Using 2 game states (state 1 and state 2 for example), we find the mean intensity of all squares in both game states. We then find the difference between the intensities for each square. The 2 squares with the biggest intensity differences are the "moved to" and "moved from" squares. Using state 1, we are able to figure out the new state.
        - ii. Problem(s):
            1. More than 2 squares can have large mean intensity difference, especially tiles that border the actual 2 squares since tiles are not guaranteed to be segmented perfectly.
            2. Sensitive to added noise.
            3. Does not handle edge cases at all (castling, promotions, en passant)
    - **b.** Hough Circles
        - i. We can use the assumption that the base of a piece is always circular to narrow down our search for finding tiles with pieces on them.
        - ii. Problem(s):
            1. This worked extremely well for most pieces, but problems arose with detecting bases of black pieces on black tiles. The base seemed to blend in with the tile, and there did not seem to be any consistent workarounds.
    - **c.** Open vocabulary object detectors
        - i. There are several open-source open-vocabulary object detectors that theoretically should work out of the box, such as OWL-ViT. There are also several open-source and proprietary large vision language models, such as GPT-4, LLaMA 3.2 11B, and LLaVA, that can be instructed to act as an open-vocabulary object detector.
        - ii. Problem(s):
            1. Unfortunately, none of them could reliably detect the pieces. These did come in handy when creating our dataset, however, as we could use them to make an initial guess for the label of each square and then quickly go through each sample to verify and make corrections.
2. **Dataset:** In order to train a model to detect the chess pieces, we needed to gather lots of data. At first we attempted to use an existing model to detect our pieces. When this did not work, we shifted towards training our own model on our specific chess board and pieces. We simply manually took pictures of various different game states and ran our board segmentation algorithm on these various game states. This yielded a much better result. However, the process to collect the data required a human to label 1.5k images by hand, which would have taken tens of hours. To make this easier, we created an automatic data labelling system that leveraged GPT-4o Mini to guess the labels for each image. We also created a basic system that displayed the images and the proposed labels so that we could quickly go through the dataset and hand-correct any mistakes. Once the dataset was created, we filtered out some of the pawn and "none" classes in order to even out their prevalence compared to the other pieces in the dataset.
3. **Model Training:** We ran into many issues when training our own model for piece detection.
    - a. We first attempted to train our own model, using 2 convolutional layers and 2 fully connected layers, from scratch, but quickly found that this did not perform well at all.

b. Then, we experimented with using a pretrained Resnet-18 model and fine-tuning the last layers. The first problem we ran into with this was a lack of data. To solve this, we collected more data across various board positions. We also found problems with freezing the first layers and only finetuning the last ones. However, when doing this, the model would stop improving at around 50% train/validation accuracy. Our hypothesis for this is that because the samples were fairly similar, and the model needed to learn more fine-grained features to find the decision boundaries between samples. When we unfroze the pretrained layers, we found significant gains in accuracy (from 50% to ~80%). This is still not precise enough, but we believe we can increase this by adding more data.

c. Another thing that we found helped significantly due to our still relatively limited dataset was doing significant data augmentation. We employed a variety of data augmentation techniques, including color jitter, random flips, and random 90, 180, and 270 degree rotations, as piece detection should be agnostic to all of these.

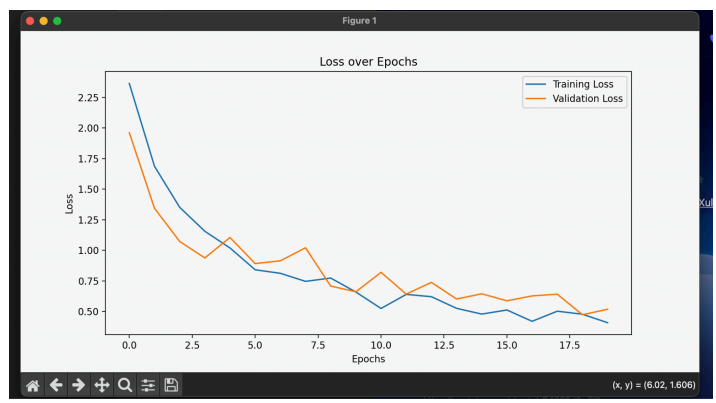
Results

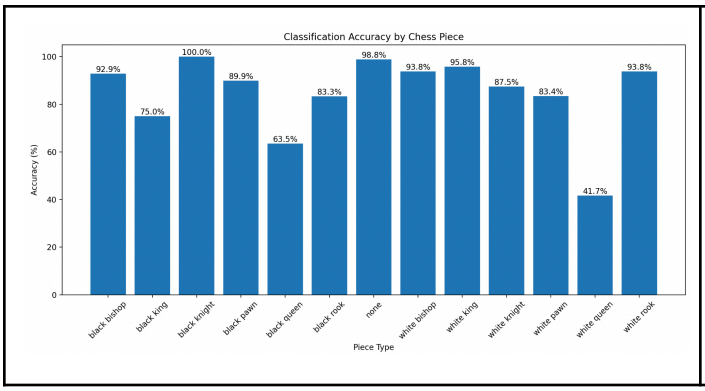
Demo: https://www.youtube.com/watch?v=9WMdKxHMAhI

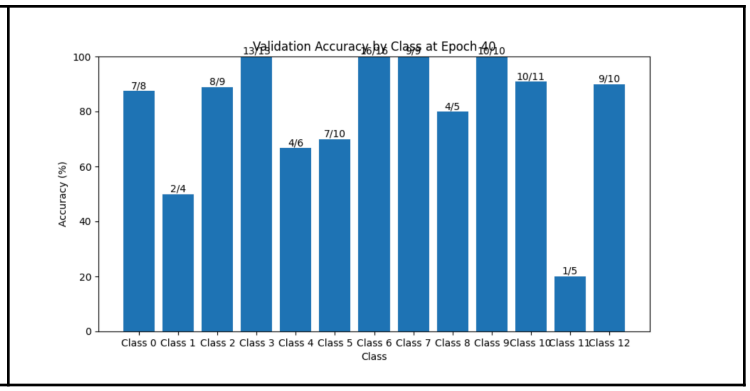
- We trained our detection model on an 80/20 train-test split. We had 1.5k segmented images of pieces on a board in our dataset. Of this, we filtered the data down to 577 images, since we didn't want the relatively more common pawns and empty squares to bias our model.
- Our best results came with our use of a fine-tuned Resnet-18 that was pretrained.

```
Detailed Results:
----------------
black bishop: 39/42 (92.9%)
black king: 15/20 (75.0%)
black knight: 47/47 (100.0%)
black pawn: 160/178 (89.9%)
black queen: 33/52 (63.5%)
black rook: 40/48 (83.3%)
none: 755/764 (98.8%)
white bishop: 45/48 (93.8%)
white king: 23/24 (95.8%)
white knight: 42/48 (87.5%)
white pawn: 161/193 (83.4%)
white queen: 10/24 (41.7%)
white rook: 45/48 (93.8%)

Overall Accuracy: 1415/1536 (92.1%)
```







- As visualized above, we reached a validation accuracy of 80% demonstrating the ability to handle piece differentiation across different board configurations. We also reached an accuracy of ~92% on the overall 1.5k image dataset, which included both the train and validation set, as well as images that were not used in the training and model tuning process at all. However, this is heavily skewed by the "none" label, as this makes up nearly half of the dataset. Excluding the "none" squares, we get a dataset-wide accuracy of 87.6%.

Label mapping: {'black bishop': 0, 'black king': 1, 'black knight': 2, 'black pawn': 3, 'black queen': 4, 'black rook': 5, 'none': 6, 'white bishop': 7, 'white king': 8, 'white knight': 9, 'white pawn': 10, 'white queen': 11, 'white rook': 12}

Conclusion

Strengths

- Game State Detection
  - With our model being of relatively high and consistent accuracy, we can firmly say that we have good detection capabilities.
- Game State → FEN
  - Our automated translation pipeline of image detection to FEN strings significantly streamlines the process of game analysis and provides a foundation for further evaluation and other state analysis capabilities.
- Board Segmentation
  - We were able to consistently segment the board to individually analyze each tile.

Pitfalls

- We were unable to see our initial attempts to make the project scalable and adaptable fully realized. Our model fell short of the precision needed for reliable piece detection under particularly challenging scenarios. We did not test for variable lighting conditions or generally noisy conditions. Moreover, our dataset was our own chess board and pieces and is not adaptable to other boards/pieces. We also did not do much testing for our edge cases (en peasant, castling, promotions, etc).

Potential Future Implementations

- Despite these downfalls, we made significant progress through refining our approach. Future work could consist of expanding our dataset, implementing a stereo imaging model for better feature detection, or advanced architectures for new learning algorithms to detect edge detections and capturing complex patterns.

Connection to Course Material

As aforementioned, many of our methodologies were closely related to the concepts covered throughout the course.

- Board Detection
    - Camera Calibration
        - We use perspective transform once we identify the corners of the board for a bird's-eye-view. In the course, we covered camera calibration and perspective projection matrices, where these types of transformations between world and image coordinates are fundamental.
        - Our method of using a homography for warping the chessboard before further processing is a practical application of concepts like projective geometry
    - Single-View Geometry
        - Extracting a planar chessboard and handling the perspective effects ties into the concepts covered about planar homographies and single-view metrology. This normalizes views for analysis.
- Image (Pre)processing
    - We played around with techniques such as gaussian blur to reduce noise as well as bilateral filtering to both reduce noise while preserving edges
        - Worked very well for mean intensity difference and hough circles algorithms
- Board Segmentation
    - Both our generation of individual tiles and "sliding window" algorithm for traversing said generated tiles were inspired by course material covering object detection
        - Dynamic board segmentation focused on novel techniques

Statement of Individual Contribution

- Advait
    - Trained the object detection algorithm
    - Dataset labeling and labelling pipeline creation
    - Corner detection algorithm
    - Integration
- Dhruv
    - Image processing
    - Board segmentation
    - Corner detection algorithm
    - Dataset generation
    - Integration
- Adi
    - Chess GUI
    - Evaluation functionality
    - Dataset generation
    - Integration