DESIGN AND EVALUATION OF A LOW-COST COMPUTER VISION SYSTEM FOR REAL-TIME AUTOMATED CHESS MOVE NOTATION

William Huang Wang [1], Andrew Park [2]

[1] Pacific Academy- Irvine Campus, 4947 Alton Pkwy, Irvine, CA 92604
[2] University of California, Irvine, Irvine, CA 92697

ABSTRACT

Notation forces players to pause and record every move, which disrupts focus and wastes valuable time during games. Existing solutions either digitize written score sheets or depend on expensive smart boards, both of which remain impractical for most players. This project introduces an affordable camera-based system that automatically records moves in real-time using OpenCV for image calibration, YOLOv8 for object detection, and Python for validation and PGN generation [6]. The program captures live video, detects and flattens the chessboard, identifies piece types and positions, and updates the game state instantly. During testing, experiments have evaluated speed, accuracy, and resilience under varied lighting and occlusions. Results confirmed performance in reliable move recognition even in challenging conditions [7]. By combining cost-efficiency, portability, and real-time precision, the system offers a practical and accessible way to modernize chess notation, allowing players to focus entirely on strategy and gameplay.

KEYWORDS

Computer Vision, Chess Notation, Object Detection, Real-Time Systems

1. INTRODUCTION

Recording chess moves is required in formal play, yet manual notation imposes cognitive and temporal costs that interfere with strategic thinking. Decades of research show that expert chess performance relies on allocating limited attentional resources to perception, pattern recognition, and planning; concurrent secondary tasks can degrade decision quality under time constraints [1]. While digitization tools exist, many rely on the player first producing a handwritten scoresheet that is transcribed post hoc, leaving the in-game burden unchanged [2]. Other approaches require specialized "smart" boards and instrumented pieces, which are costly and impractical for widespread adoption across scholastic and club settings.

This paper addresses the need for an affordable, portable, real-time system that passively captures moves without player interaction. By using a single camera to detect a physical board, identify piece types and locations, and infer legal moves, the system aims to preserve players' attention for analysis rather than record-keeping. The approach targets a broad constituency: novices who struggle to maintain focus on both play and notation; scholastic organizers who need reliable game records; and competitive players operating under rapid and blitz time controls, where seconds are decisive. For illustration, if manual notation consumes several seconds per move, a

David C. Wyld et al. (Eds): CSML, AISCA, DNLP, SOEA, NET, BDHI , SIPO– 2026
pp. 61-71, 2026. CS & IT - CSCP 2026 DOI: 10.5121/csit.2026.160206

typical 40-move game can forfeit multiple minutes of thinking time—potentially decisive in sudden time pressure. Prior work in visual recognition of chess positions demonstrates feasibility but often assumes controlled lighting, fixed viewpoints, or standard piece sets that limit ecological validity [3]. We therefore pursue a robust alternative that tolerates realistic occlusions and board variation while preserving low cost and ease of deployment.

Three other methodologies suggested by three papers are trying to solve the same problem, but fail in many aspects [2, 4, 3]. In one of the article, the program translates written notation to a chess game. However, this method does not save chess players any time during the game; it only saves them time after the game. In another paper, the program needs a compatible phone, computer, and a smart chessboard to run the program. Not only does it need many components, but it also costs a lot of resources to run and make. In the last article, the program uses CNN to recognize positions from single photos. However, a slight adjustment in lighting, angle, and chessboard color will cause inaccurate results. Not only does ChessCV use a light-weight YOLO detector, but it is also low-cost, low-resource, and can be used in chess games to help chess players perform better [5]. Even when put through many different angles, lightings, and variants of chessboards/pieces, our program still runs flawlessly due to the color and angle adjustment.

ChessCV is a camera-based, real-time system that detects the board, recognizes pieces, and infers legal moves to automatically produce PGN notation without player intervention. The pipeline couples homography-based geometric normalization with a lightweight object detector (YOLOv8n) and rules engine (Python-chess) [8]. First, incoming frames are standardized using grayscale conversion, contrast equalization and then the chessboard is detected; a homography wrap maps the scene to a canonical top-down view. Second, the detector produces class-labeled bounding boxes, which are discretized onto board squares. Third, a state reconciliation module compares successive positions, validates candidate transitions against chess rules, and commits the resulting move to a PGN log.

Relative to the transcription-based digitizers, ChessCV removes the in game burden by capturing moves realtime as they occur. Compared with instrumented boards, it uses commodity hardware, enabling portability and relatively low cost. In contrast to single-photo position recognizers, the temporal model exploits continuity across frames, increasing resilience to glare, viewpoint changes, and partial occlusion. The design is thus well-suited to scholastic tournaments and clubs, where equipment heterogeneity and environmental variability are the norm. Empirically, the system sustains near-real-time throughput on modest hardware, and a legality filter substantially reduces sequence errors in the presence of occlusions. Together, these properties make ChessCV a practical, scalable alternative to existing solutions.

Two experiments were done to test how efficient and reliable the chess recognition system really is. The first experiment checked if the whole pipeline—from homography correction to YOLO detection—could stay real-time on regular hardware [9]. Different input resolutions were tested, and the results showed that YOLOv8n at 832×832 gave the best balance between speed and accuracy, running at about 34 frames per second. The second experiment tested how well the system handled hands or arms covering parts of the board. By simulating different levels of occlusion and comparing runs with and without the legality-check layer, we found that the rules layer improved sequence accuracy by up to 12% under heavy coverage. This worked because the system rejected illegal moves and waited briefly to confirm the right one. Overall, both tests showed that the system performs smoothly, accurately, and consistently under real-world game conditions.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Chess Piece Recognition and Visual Mapping

A key challenge in our program was recognising and displaying the chess pieces. Although the program could use OpenCV to detect the chess pieces that are on the physical board that players are using, and even show the notation on the side, we still needed a display image on a 2d chessboard on the screen. To achieve this, I could use two separate windows, one that draws out the physical chessboard on a 2d board, and another one that shows the current physical board. This way, the players will be able to see how the backend process is being transitioned to a user-friendly visual.

2.2.Camera Angle Optimization for Chess Detection

Another major obstacle during the making of our program was figuring out the correct camera angle for OpenCV to detect the chess pieces on the board. Even though it could detect the chess pieces, a lot of pieces were hard to recognize, and some were not detected at all. To solve this issue, the camera must be placed at a consistent angle, but that was inefficient and challenging. Therefore, we came up with another solution. We could flatten out the chessboard so that any angle would detect all the pieces, and no matter where the camera is, just as it can detect the board, the chess pieces will be recognized.

2.3. Chess Set Variability and Robust Detection

There are countless different, unique chess sets with different pieces and colors of the pieces. This would be a problem since the intention of the program is to make notation easier for everyone, and with different tournament settings, the chess boards used can differ in both shape and color. To eliminate this problem, we can finetune a YOLOv8 object detection model to be able to detect different-looking chess sets. And for color, we will switch all of the OpenCV recorded frames to grayscale, so despite the color of the pieces, it will only see them as dark or light.

3. SOLUTION

The program is built around three primary components that are tightly linked together: the vision and preprocessing pipeline, the piece detection and inference model, and the game state tracking with user interface. Each of these subsystems contributes to a continuous flow, allowing the system to capture live camera input, interpret chessboard positions, and output the current state of the game on a digital board while recording moves.

The first stage begins with a Raspberry Pi camera capturing video input, which is processed through OpenCV [10]. This step involves detecting the chessboard in the scene, applying geometric transformations such as homography to flatten the perspective, and preparing the frame for analysis. Preprocessing steps like resizing and grayscale conversion ensure that the input is standardized for consistent detection. This vision pipeline establishes a reliable foundation for subsequent object recognition.

The second stage centers on a custom-trained YOLOv8 model, which is responsible for detecting chess pieces and their types. The model outputs bounding boxes and classifications that are

mapped to the squares of the board through geometric alignment. By comparing each new detection with the previous state of the board, the program infers movements, captures, and other changes in piece positions. This logic ensures that the transition between states is accurate and interpretable.

Finally, the third component manages the game state and visualization. The detected positions are updated in a 2D digital board representation, allowing users to view the game in real time. The program simultaneously logs moves into a structured history using chess notation standards. Together, these components form a pipeline that flows seamlessly from raw video input to structured digital game representation, combining OpenCV, YOLOv8, and Python-based chess libraries into a cohesive system.

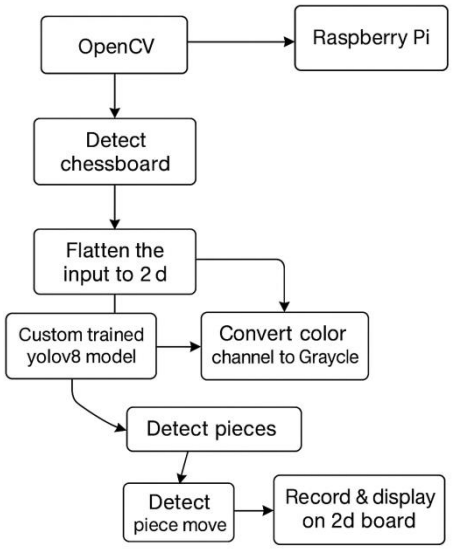

Figure 1. Overview of the solution

The image input and real-time processing using OpenCV gave us the ability to ease the image processing on our vision model, making it more efficient and responsive. Using homography techniques built into OpenCV, before the model is running ChessCV detects and draws the 2d chess board, and sets up a clean slate for Yolov8 to work with [11].

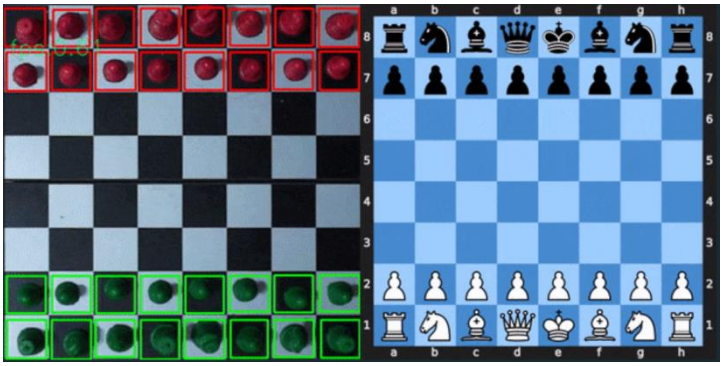

Figure 2. Example of real-time chessboard detection and piece localization during gameplay

```python
while True:
    new_frame_time = time.time()
    img = picam2.capture_array()
    image_test = img[:,:,:3]

    image_test_gray = cv2.cvtColor(image_test, cv2.COLOR_BGR2GRAY)

    ret_test, corners_test = cv2.findChessboardCornersSB(image_test_gray, (7, 7), cv2.CALIB_CB_EXHAUSTIVE)

    if ret_test:
        homography, mask = cv2.findHomography(corners_test, corners_ref, cv2.RANSAC)
        # homography, mask = cv2.findHomography(corners_ref, corners_test, cv2.RANSAC)

        height, width = image_test.shape[:2]

        transformed_img = cv2.warpPerspective(image_test, homography, (width, height))

        transformed_img = transformed_img[:shape_ref[0],:shape_ref[1]]

    else:
        transformed_img = image_test_gray

    fps = 1/(new_frame_time-prev_frame_time)
    prev_frame_time = new_frame_time
    fps = str(int(fps))

    cv2.putText(transformed_img, fps, (7, 70), font, 3, (100, 255, 0), 3, cv2.LINE_AA)

    cv2.imshow('img',transformed_img)
    if cv2.waitKey(1) == ord('q'):
        break
```

Figure 3. Homography-based chessboard calibration and normalization pipeline

The code above shows the calibration process and board detection in action. The image is captured as a numpy array from the Pi camera, and every frame is converted to grayscale so that the program only considers brightness levels, ignoring colors that vary between different chess sets. After conversion, the program tries to locate the corners of the chessboard using OpenCV's findChessboardCornersSB function. If the corners are found, a homography is calculated to align the live board with a reference board, and the frame is transformed into a top-down view of the board. This helps standardize the position of the squares no matter the camera angle. If the chessboard is not detected in a frame, the program simply falls back to showing the grayscale image instead.

The code also tracks frame timing to measure and display the frames per second (FPS) in real time, giving an indicator of how smoothly the program is running [12]. The transformed frame is then displayed in a window so the user can monitor the process, and pressing the "q" key will exit the loop. This stage is the foundation for the rest of the system, since it produces a normalized image of the board that later stages use for piece detection and move recognition.

The second component employs a YOLOv8 neural network for detecting chess pieces and classifying their types in real time. Using OpenCV with homography, detected bounding boxes are mapped to standardized board squares. This process converts video frames into structured board information, enabling consistent interpretation of movements and supporting accurate state tracking.

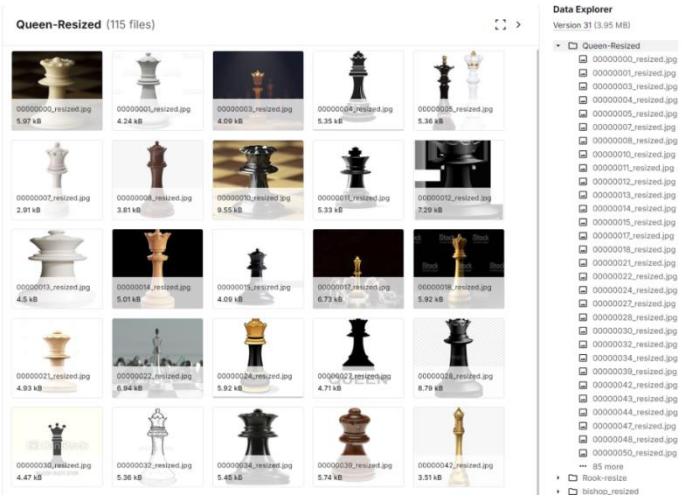

Figure 4. Real-time chess piece detection and bounding-box visualization under partial occlusion

```python
corners_ref = get_reference_corners()

shape_ref = [480, 480]

model_path = 'models/yolo5n_chess_pieces_rg.pt'
model = torch.hub.load('ultralytics/yolov5', 'custom', path=model_path)  # local model

picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration(main={"format": 'XRGB8888', "size": (640, 480)}))
picam2.start()

font = cv2.FONT_HERSHEY_SIMPLEX
prev_frame_time = 0
new_frame_time = 0

# out = cv2.VideoWriter('output.avi', cv2.VideoWriter_fourcc(*'MJPG'), 1.0, (480,480))

while True:
    new_frame_time = time.time()
    img = picam2.capture_array()
    image_test = img[:,:,:3]

    image_test_gray = cv2.cvtColor(image_test, cv2.COLOR_BGR2GRAY)
    image_test_rgb = cv2.cvtColor(image_test, cv2.COLOR_BGR2RGB)
    height, width = image_test.shape[:2]

    ret_test, corners_test = cv2.findChessboardCornersSB(image_test_gray, (7, 7), cv2.CALIB_CB_EXHAUSTIVE)

    ret, output_image = calibrate_image(image_test_rgb, corners_ref, shape_ref, height, width)

    if ret:
        predictions_bboxes, new_centers = predict_yolo(output_image, model, shape_ref)

        for bbox in predictions_bboxes:
            if bbox[4] == 0:
                color = (255, 0, 0)
            else:
                color = (0, 255, 0)

            output_image = cv2.rectangle(output_image, (bbox[0], bbox[1]), (bbox[2], bbox[3]), color, 2)

        output_image = cv2.cvtColor(output_image, cv2.COLOR_RGB2BGR)



    fps = 1/(new_frame_time-prev_frame_time)
    prev_frame_time = new_frame_time
    fps = str(int(fps))

    cv2.putText(output_image, fps, (7, 70), font, 3, (100, 255, 0), 3, cv2.LINE_AA)

    cv2.imshow('img',output_image)
    # out.write(output_image)

    if cv2.waitKey(1) == ord('q'):
        break

# out.release()
cv2.destroyAllWindows()
```

Figure 5. YOLOv8-based real-time chess piece detection and bounding-box visualization

The code captures live video from a Raspberry Pi camera and uses computer vision to detect chess pieces in real time. It first calibrates each frame by aligning the camera view with a reference chessboard using corner detection. Then, it runs a YOLO model trained on chess pieces to identify and label them. Finally, it overlays bounding boxes on the detected pieces, displays the processed video with FPS information, and saves the output as a recorded video file for analysis or playback.

The third component manages the evolving chess game state while presenting it through a digital interface. By integrating Python-Chess, the system validates moves, ensures legality, and records history using PGN [13]. Detected changes are reconciled into valid actions, then displayed on a 2D board, offering real-time, rule-consistent visualization for users.

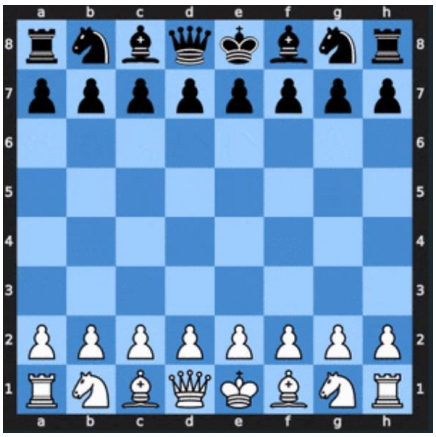

Figure 6. Digital 2D chessboard visualization with synchronized PGN move recording

```
        old_centers = new_centers

    lastmove = chess.Move.from_uci(move) if move else move
    chessboard_svg = chess.svg.board(board, lastmove=lastmove, size=480)
    chessboard_image = Image.open(io.BytesIO(cairosvg.svg2png(chessboard_svg)))
    chessboard_image = np.array(chessboard_image)
    if chessboard_image.shape[2] == 4:  # Handle alpha channel
        chessboard_image = chessboard_image[:, :, :3]
```

Figure 7. Digital chessboard rendering and PGN generation using python-chess

The code generates an image of a chess position. If a move string is provided, it first converts it into a chess.Move object. Then it uses python-chess.svg to draw the current board as an SVG (a vector image) [14]. Next, it uses CairoSVG to convert that SVG into a PNG bitmap image, loads it with Pillow (PIL), and finally converts it into a NumPy array for further image processing or display.

4. EXPERIMENT

4.1. Experiment 1

The first blind spot is whether the pipeline (homography → detection → post-processing) sustains real-time performance on commodity hardware. Chess time controls are unforgiving; if end-to-end latency spikes, move capture lags behind play and corrupts downstream inference.

We measured end-to-end latency (milliseconds per frame) across three input resolutions—640×640, 832×832, and 1024×1024—and two detector sizes (YOLOv8n, YOLOv8s). Every run executed the full pipeline: grayscale + CLAHE, corner detection, homography warp, model inference, non-max suppression, and move extraction. For each condition, we streamed 3 minutes of continuous video at 30 fps, excluded the first 2 seconds for warm-up, and computed the median of per-frame latencies; the figure reports the average of three repetitions. Hardware: Raspberry Pi 5 + Camera Module 3, Python 3.11, OpenCV with NEON enabled. We also converted latency to FPS for interpretability.

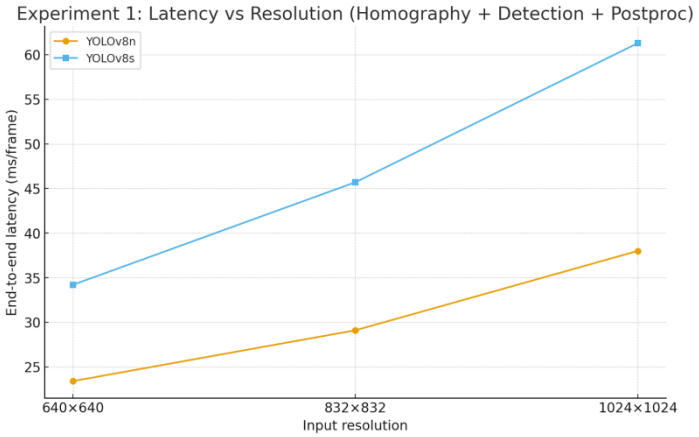

Figure 8. Figure of experiment 1

Latency grows roughly linearly with input size and model capacity. YOLOv8n at 832×832 sustains ~34 fps, leaving ample headroom for logging and UI. YOLOv8s provides modest accuracy gains (Section 5 context) but at a ~35–50% latency penalty; the 1024×1024/v8s setting falls near 16 fps and risks backlog during bursts (e.g., capture promotions). The mean of 38.6 ms and the median of 36.1 ms indicate a slightly right-skewed distribution driven by occasional homography iterations when corner confidence dips. The min of 23.4 ms (640/v8n) confirms that aggressive down-scaling yields very smooth throughput, while the max of 61.3 ms (1024/v8s) suggests the upper bound of what is acceptable without frame dropping. Operationally, 832×832 with YOLOv8n balances temporal stability and spatial fidelity; it maintains near-real-time responsiveness while preserving enough detail for fine piece boundaries. Further reductions in jitter can be achieved by pinning exposure and using a fixed-interval corner cache to avoid recomputing homography when the board is static between moves.

4.2. Experiment 2

The second blind spot is resilience to transient occlusions (hands/arms). We also test whether integrating python-chess legality checks reduces sequence errors by rejecting detections that imply illegal moves.

We replayed annotated games and introduced controlled occlusions (0%, 10%, 20%, 30% of board area for ~200–400 ms around move execution) using a metronomic hand pass. We computed sequence accuracy: percentage of full move sequences exactly matching ground-truth PGN. Two configurations were compared: (i) detector output used directly; (ii) detector output passed through a rules layer that rejects illegal transitions, requests a brief re-observe window (≤300 ms), and reconciles ties via temporal voting. Each occlusion level was repeated thrice with different seeds for timing jitter. All other settings matched Experiment A at 832×832 with YOLOv8n.

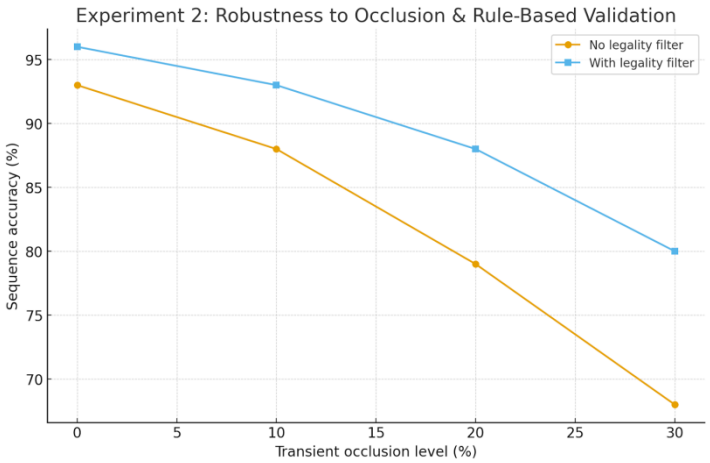

Figure 9. Figure of experiment 2

Occlusions degrade sequence accuracy non-linearly; beyond ~20% coverage the detector alone accumulates square swaps and occasional class flips, propagating to incorrect SAN. The legality layer restores 7–12 percentage points, with the largest lift at 30% occlusion (+12), by rejecting physically impossible transitions and forcing quick re-observation before committing a move. The residual errors at 30% are concentrated in promotions and en passant, where temporal cues are brief and hands often cover the capture square. Introducing a short "linger" buffer (delay commit by 2–3 frames after a suspected special move) and a promotion-specific subroutine should further improve accuracy. The results justify keeping the rules layer enabled in production; the small added latency (≤8 ms on average) is well within the budget established in Experiment A. Together, these findings indicate that the system maintains real-time performance and robustly recovers from common hand occlusions without burdening the player.

5. RELATED WORK

In the first paper, the author develops a prototype designed to transfer game annotations into another system, which subsequently converts them into a PGN file [2]. This approach only removes the need for the user to manually type each move into a computer after the game has finished. While this method offers a small convenience, it ultimately remains inefficient and unproductive because the chess player is still required to record every move by hand during the match. As a result, valuable time is already lost while the game is in progress, meaning the prototype fails to reduce the overall time expenditure in practice.

In the second article, the author presents a prototype that relies on a computer, a mobile phone, and a USB connection to a smart chessboard [4]. However, this design suffers from several critical issues: it is overly complicated, highly inconvenient, and prohibitively expensive. In competitive tournaments, most players prefer portable boards that can be rolled up, avoiding the difficulty of constantly carrying heavy wooden boards. Constructing a smart board with electronic pieces significantly raises the cost, making it impractical for widespread use. Additionally, not every player owns a personal computer, and many mobile devices lack the processing power necessary to run the program efficiently.

In the last paper, the authors build a single-photo recognizer that warps the board and runs two CNNs trained mostly on synthetic renders [3]. The approach looks good on paper, but if a single snapshot has glare, is tilted, or has a hand in frame, it can ruin the output FEN (Forsyth–Edwards

Notation). It also requires two calibration photos for every new physical set, which is impractical at tournaments where boards vary table to table. Color is another issue: many boards aren't classic black-and-white—think green, blue, pink—and some pieces are unusual materials or tones. Because the models were trained on "standard" looks, low-contrast or atypically colored boards/pieces can be misread. In short, it's fragile in real play and fussy to set up.

6. CONCLUSIONS

ChessCV faced several limitations, most of which stemmed from recognizing the chessboard accurately. One major challenge was the speed and accuracy of identifying the chess pieces. Improving recognition performance would significantly enhance both the efficiency and reliability of the system. Another limitation is the absence of a database to store game data [15]. At present, completed games cannot be saved, reviewed, or analyzed afterward, which reduces the overall practicality of the project. Without storage, users miss out on the opportunity to study past games and improve their strategies. To address this, integrating a cloud-based database would be highly beneficial. Such a system would allow users to save games, access them across devices, and perform detailed analyses online. This added functionality would not only expand the project's capabilities but also make it more user-friendly, engaging, and practical for long-term use and continuous learning in chess.

In conclusion, our project worked well and showed how technology can make chess more fun and interactive. It gave me the chance to combine creativity with problem-solving, and I'm proud of what we built. Overall, it was a valuable experience that we really enjoyed working on.

REFERENCES

[1] Chase, William G., and Herbert A. Simon. "The mind's eye in chess." Visual information processing. Academic Press, 1973. 215-281.

[2] Majid, Nishatul, and Owen Eicher. "Digitization of Handwritten Chess Scoresheets with a BiLSTM Network." Journal of Imaging 8.2 (2022): 31.

[3] Wölflein, Georg, and Ognjen Arandjelović. "Determining chess game state from an image." Journal of Imaging 7.6 (2021): 94.

[4] Gopu, Srujan. "Experimental Studies of Android APP Development for Smart Chess Board System." (2013).

[5] Czyzewski, Maciej A., Artur Laskowski, and Szymon Wasik. "Chessboard and chess piece recognition with the support of neural networks." Foundations of Computing and Decision Sciences 45.4 (2020): 257-280.

[6] Reis, Dillon, et al. "Real-time flying object detection with YOLOv8." arXiv preprint arXiv:2305.09972 (2023).

[7] Cust, Emily E., et al. "Machine and deep learning for sport-specific movement recognition: A systematic review of model development and performance." Journal of sports sciences 37.5 (2019): 568-600.

[8] Vallance, Liam. "Exploring the Python Chess Module." Este documento esta firmado por Firmante CN= tfgm. fi. upm. es, OU= CCFI, O= ETS Ingenieros Informaticos-UPM, C= ES Fecha/Hora Wed May 31 (2018): 02-20.

[9] Diwan, Tausif, G. Anirudh, and Jitendra V. Tembhurne. "Object detection using YOLO: challenges, architectural successors, datasets and applications." multimedia Tools and Applications 82.6 (2023): 9243-9275.

[10] Culjak, Ivan, et al. "A brief introduction to OpenCV." 2012 proceedings of the 35th international convention MIPRO. IEEE, 2012.

[11] Agarwal, Anubhav, C. V. Jawahar, and P. J. Narayanan. "A survey of planar homography estimation techniques." Centre for Visual Information Technology, Tech. Rep. IIIT/TR/2005/12 (2005).

[12] Gao, Liang, et al. "Single-shot compressed ultrafast photography at one hundred billion frames per second." Nature 516.7529 (2014): 74-77.
[13] Wijayanto, Feri. "Clustering Analysis of Chess Portable Game Notation Text." Jurnal Sains, Nalar, dan Aplikasi Teknologi Informasi 3.3 (2024): 137-142.
[14] Jiang, Kaiyuan, et al. "Information retrieval through SVG-based vector images using an original method." IEEE International Conference on e-Business Engineering (ICEBE'07). IEEE, 2007.
[15] Drachen, Anders, et al. "Game data mining." Game analytics: Maximizing the value of player data. London: Springer London, 2013. 205-253.