CSC 498: Vision-based Robot Control for Human Robot Interaction

Lucas Franco

The College of New Jersey, Ewing NJ 08628, USA

francol4@tcnj.edu

Abstract. This project presents a robotic system that plays a physical, over-the-board chess game using real-time computer vision, machine learning, and robotic arm control. A webcam-based vision system detects a human player's moves by predicting if a piece resides in a square on the chessboard using a support vector machine (SVM) trained with RGB pixel values and edge features. Detected moves are played through the Chess Arena GUI running a chess engine, whose responses are executed by an Open-manipulator-X robotic arm. Inverse kinematics (IK) was applied to generate joint trajectories using SLERP-interpolated world positions, but the generated trajectories were inaccurate near board edges, so precise joint angles of the actuators were manually calibrated. The final system achieved 90% in move detection and near 100% success in physical execution of the moves using hard-coded joint angle-based motion plans. Future improvements may include piece type recognition and the use of depth cameras or deep learning models like YOLOv8 for multi-class classification.

Keywords: Human-robot interaction · Computer vision · Machine learning · Support vector machine · Robotic manipulation · ROS

1 Introduction and Motivation

The integration of artificial intelligence, machine learning, and robotics presents opportunities for developing complex interactive systems. A game like chess played on the board serves complex task testing such as perception, control, and real-time movement and decision making. This project presents a full stack system in which a robotic arm autonomously plays chess by observing the state of the board through a camera and physically executing moves in response to a human player.

Chess provides an ideal platform for this kind of interaction due to its clearly defined structure. Chess also has a rich history with AI, one of the most famous milestones in artificial intelligence was IBM's Deep Blue defeating world chess champion Garry Kasparov in a six-game match in 1997. This moment marked the first time a machine demonstrated strategic dominance over a top human player, and it started a wave of innovation in AI-driven game playing. Since then, chess engines have grown exponentially in strength and efficiency due to modern hardware, neural networks, and improved search algorithms. Today's engines like Stockfish [2] consistently outperform even the best human players in purely digital environments.

However, unlike digital chess applications, a chess-playing robot must contend with noisy real-world data, varying lighting conditions, shadows, and mechanical precision. This project seeks to tackle these issues by developing a vision-based recognition system that detects a player's move and triggers an appropriate physical response from a robot.

This project explores the integration of real-time classification of game state and motion planning for robotic arms. The final system incorporates multiple components: a calibrated webcam for board monitoring, a lightweight machine learning model for square classification, and an open-manipulator-X robotic arm for physical manipulation of chess pieces. This full-stack system presents an autonomous chess playing opponent utilizing the latest and greatest chess engines. Enabling play against a variety of different engines over a physical board in real life.

This project integrates several key components into a cohesive and functional system. It includes a real-time vision system for classifying whether individual squares on a chessboard have a piece or not, using a support vector machine (SVM) trained on RGB and edge detection features. A move detection algorithm compares predicted square states from each frame in real time and validates actions against the internal game state using a chess engine integrated via the Universal Chess In-

terface (UCI) protocol, supported through the Chess Arena GUI [1]. Physical execution is handled by an OpenManipulator X [7] robotic arm controlled through ROS, with movement paths transitioned from an early SLERP-based inverse kinematics approach to precisely calibrated hard-coded trajectories for accuracy and consistency. To ensure alignment between visual input and physical layout, an interactive calibration tool allows the user to manually adjust grid boundaries as interpolation across four corners of the chessboard from our frame can come with slight error. Together, these contributions form a full-stack system that connects computer vision, decision-making, and robotic motion into a platform for human-robot interaction through chess.

1.1 Paper Outline

The remainder of this report is organized as follows. Section 2 reviews prior work related to vision-based board game interfaces and machine learning classifiers for visual prediction. Section 3 presents the full methodology. Section 4 details the experiment design and implementation, while Section 5 analyzes the results and analysis including limitations of current approach. Section 6 proposes future directions, and Section 7 concludes the report.

2 Preliminary Work

2.1 Initial Challenges

This project began when Dr. Yoon introduced me to the idea of utilizing robot arms to play chess. Intrigued by this challenge, I began exploring the concept before the semester started and came across a project on GitHub [4], in which the author developed a robotic system that plays chess utilizing a webcam to compare the differences between pixels from frames to detect the movement of the piece, then utilizes a chess engine for opponents response, and the 4-degree-of-freedom (4DOF) arm makes the move. This discovery motivated me to develop a proof-of-concept system prior to the start of the semester knowing that time was limited. The first step involved calibrating the camera view of the board. I initially attempted to use OpenCV's built-in chessboard corner detection functions. However, these consistently failed on my real board setup due to differences in square textures, lighting, and board style, which deviated from the ideal chessboard patterns for which these algorithms are made.

As a result, I implemented a manual calibration system where users click the four corners of the board in a fixed order. These corner points were interpolated to generate an 8x8 grid of square coordinates, allowing pixel-based segmentation of each square for feature extraction and labeling. This manual method proved to be more flexible across various board orientations and lighting environments.

2.2 Feature Engineering and Random Forest Classification

Once calibration was accurate, and I could collect the four coordinates (x, y) relative to the webcam frame for each of the 64 squares on the chess board, I moved onto feature extraction and classification. For each square I computed a large set of visual features from each grid cell pixels, assuming that more information would lead to better performance. Features include:

- **Mean and standard deviation of HSV values:** Let $P = \{p_1, p_2, \ldots, p_n\}$ denote the set of n pixels in a square. Each pixel p_i is represented in HSV space as $v_i = (h_i, s_i, v_i)$. For each channel $c \in \{H, S, V\}$, we compute:

$$\mu_c = \frac{1}{n}\sum_{i=1}^{n} c_i, \quad \sigma_c = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(c_i - \mu_c)^2}$$

  These six values (μ_H, μ_S, μ_V, σ_H, σ_S, σ_V) summarize color information in a compact form.
- **Grayscale intensity statistics:** Grayscale conversion is performed using the weighted formula:

$$I_i = 0.2989 \cdot R_i + 0.5870 \cdot G_i + 0.1140 \cdot B_i$$

  where I_i is the grayscale intensity of pixel p_i. The mean and standard deviation of $\{I_1, I_2, \ldots, I_n\}$ are then computed in the same way as HSV above.
- **Edge detection values (Canny):** The Canny edge detector is applied to the grayscale version of the square. The output is a binary edge map E, from which we compute the edge density:

$$\text{Edge Density} = \frac{1}{n}\sum_{i=1}^{n} E_i$$

  where $E_i \in \{0, 1\}$ indicates whether the pixel p_i lies on an edge.

– **Binary bitmap features:** A binary mask is extracted from a central $k \times k$ region (e.g., $k = 10$) of each square. Each pixel is converted to binary using intensity:

$$B_i = \begin{cases} 1 & \text{if } I_i > \tau \\ 0 & \text{otherwise} \end{cases}$$

The resulting $k \times k$ bitmap is flattened into a vector and used as input features.

– **Normalized color histograms:** For each HSV channel, we compute a histogram H_c with b bins (e.g., $b = 16$). Each bin count h_j is normalized by the total number of pixels in the square to produce a probability distribution:

$$H_c^{(j)} = \frac{h_j}{\sum_{k=1}^{b} h_k}$$

for $j = 1, 2, \ldots, b$. This ensures that the histogram features are scale-invariant and suitable for machine learning.

For classification, the model predicted piece type and class.

I trained a Random Forest classifier with this data, which seemed well suited for the task. I needed a lightweight model that can handle mixed feature types, predict in real-time frames, and work well with modest amounts of data. Testing different hyper-parameters, such as the number of stumps ranging from 50-450, and trained by collecting frames from the board as normal chess layout with the labels being the respective piece type and color. This wasn't accurate at predicting what specific pieces were, but it could recognize the difference between if a piece was in the square or not. This approach was also fairly slow in real-time frame predictions. From here, instead of classifying the piece type or color, the training was simplified to only contain labels 1s or 0s that detail whether or not a piece existed in the square. This worked well, predicitng in real-time with slight frame delay, accurately predicting if a piece was in the square about 65% of the time. The remaining issues are with the lighting and camera angle of the pieces. If the piece was to tall it would interfere with the data being collected from the squares around it and pieces that match the square color on the board were harder to detect.

I also tested other models, such as neural networks using the same features and training system. Wile they achieved comparable accuracy, the prediction time was significantly slower making it impractical for seamless real-time use.

2.3 Chess Engine Integration and Depth Tuning

Modern chess engines like Stockfish employ a hybrid approach that combines a minimax search algorithm enhanced by alpha-beta pruning with a neural network-based evaluation function known as NNUE (Efficiently Updatable Neural Network). While the minimax tree search explores millions of possible future move sequences, the NNUE provides a more accurate evaluation of board positions compared to traditional handcrafted heuristics. At each position, the engine computes a numerical score reflecting the strategic quality of the board state, factoring in material balance, piece activity, king safety, and more. The "depth" setting in engines refers to how many half-moves (plies) ahead the engine searches. While deeper searches typically yield stronger play, they increase computation time. For the purposes of this project—where engine responses must be issued in near real-time before system state validation—I limited Stockfish's depth to 8. This provided a practical balance between strategic depth and responsiveness.

2.4 Game State Validation and Engine Integration

Once I had a working system for detecting whether squares were occupied, the next step was to actually figure out how to interpret those predictions into valid chess moves. In each frame, the classifier is run independently on all 64 squares of the board. For each square, visual features are extracted and passed into the model, which predicts whether a piece is present (1) or not (0). These individual predictions are then combined to construct a complete 8x8 binary board map representing the current state of the game. Then store the predicted board state to be compared to the next board state after the move had been completed. To do this in real-time there is a stable-frame counter which wouldn't predict on the squares until movement has been stopped. This was essential so it doesn't predict with hands or shadows moving across the board. Then, comparing the two frames can figure out which square had lost a piece and which had gained a piece, detecting "from" and "to" squares of the move. After a potential move is found, it is verified as legal to our current internal game state (utilizing python chess library) and then uses pyautogui to type

the move into the Chess Arena GUI. After the move was properly played in the GUI and a engine responded it presses f5 inside Arena to copy the current games PGN log to the clipboard. The move log is stored in PGN (Portable Game Notation), a standard plain-text format for recording chess games. Each move in the PGN is listed in sequence and includes both White and Black's moves for each turn. To extract the engine's response, I parse the PGN string copied to the clipboard and isolate the latest Black move (which is the engine's response after a human player's White move). From this move, I extract the "from" and "to" squares (e.g., `e7` to `e5`) using regular expressions. This vision system wasn't perfect, mainly due to the false-positive predictions from the random forest classifier, but laid the groundwork for future work.

3 Problem Definition and Methodology

The main goal of this project is to develop an integrated robotic chess-playing system that combines computer vision, game logic, and robotic manipulation. Specifically, the system aims to: (1) accurately detect and track chess pieces on a physical board using computer vision techniques; (2) translate the player's physical move into an updated internal game state using a structured representation of the board; (3) obtain the optimal response move by querying an external chess engine; and (4) physically execute the engine's response using the OpenManipulator-X robotic arm with high precision and reliability.

3.1 Vision System

The vision system is responsible for detecting whether or not each square on the chessboard is occupied by a piece. My original approach used a large feature set—HSV values, grayscale statistics, edge detection, binary bitmaps, and color histograms—and trained a Random Forest classifier to distinguish between occupied and empty squares. It worked decently but struggled with lighting, shadows, and similar piece/board color combinations, not to mention the slow inference times that made real-time prediction difficult.

After some testing and discussion, I ended up simplifying both the data and the model. Instead of collecting tons of features, I limited the training data to just an RGB vector and edge detection mask from the center region of each square. This reduced the noise in the data and made the

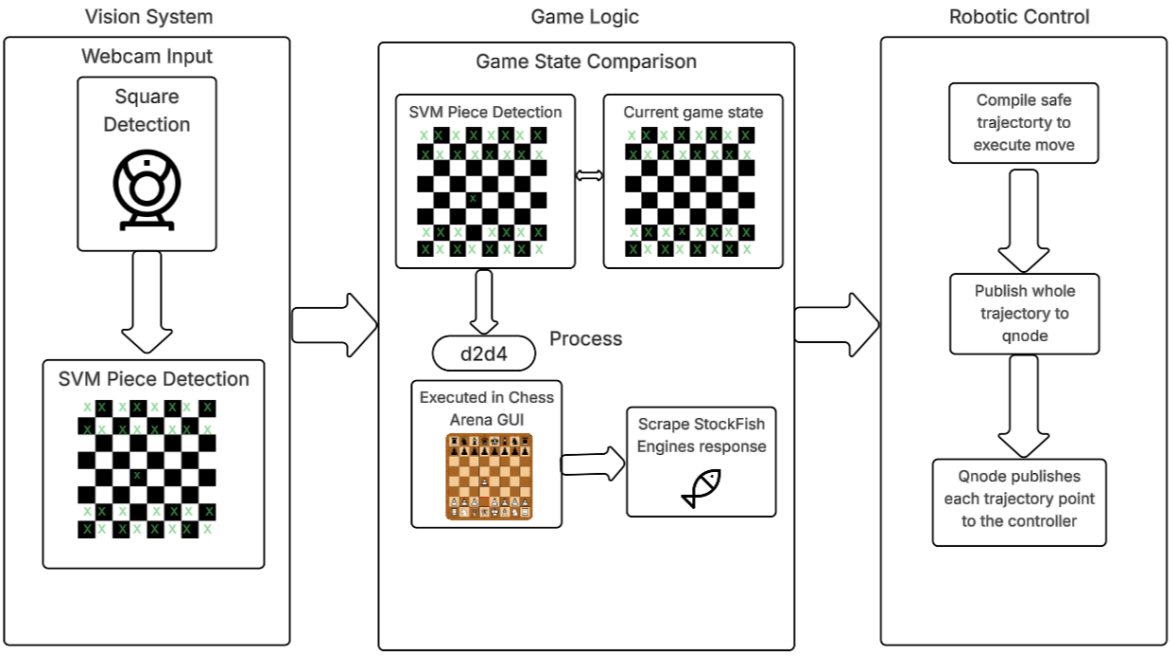


Fig. 1. System architecture: visual pipeline from webcam input to robotic arm execution.

model faster. For training, I used a small labeled dataset—images of the board in various common configurations (half-full both sides, standard starting layout)—and labeled each square as either 1 (occupied) or 0 (empty) .

I also swapped the model from a Random Forest to a Support Vector Machine (SVM), since we were solving a binary classification problem and needed faster predictions. With RGB plus edge values as input, the SVM performed much better in terms of speed and held up reasonably well in accuracy—even under challenging lighting conditions. The main limitation was that the model could only predict whether a square had a piece or not; it couldn't identify what kind of piece it was. But for the purposes of move detection, this worked fine.

Compared to the Random Forest, the SVM made faster predictions in real-time and achieved higher accuracy. The main issue was with false negatives: When a piece blended into a square tile of a similar color (e.g., a black rook on a dark square), the model would sometimes fail to detect that a piece was present. To workaround this, we printed out a board that has discolored square tiles, so that our pieces stand out. This effectively made our SVM model predict with about 100% accuracy.

I also made major improvements to board calibration. Since interpolation of 64 squares across 4 calibrated coordinates on the frame comes with some error, especially with differing camera angles, I implemented a method where you can drag each squares corner point to line up perfectly on the frame. The algorithm implemented needed to be able to discern which squares this corner point belonged to and adjust its position in each of the squares. This helped align the gird more precisely and ensured better feature extraction.

In real-time operation, the vision pipeline runs the classifier on each square every frame. The predicted states are compared across frames, and moves are only detected once the system sees a stable board state for several consecutive frames. This helped filter out false detections caused by hands or shadows.

3.2 Game Logic

The game logic algorithm is responsible for maintaining the internal representation of the chessboard detecting human moves from the vision system output, and scraping the chess engines response from the Arena GUI. The system uses the `python-chess` library to manage board state, validate legal moves, and parse algebraic notation.

To detect a move, the classifier outputs a binary 8x8 matrix each frame, where each cell represents whether a piece is present. The current prediction is compared against the system's internal game state to find differences, which are interpreted as moves. This comparison identifies potential "from squares" where it looks to see if any 1s from our internal game state's board turned to 0 at the same position in our classifiers predicted matrix. We also identify potential "to squares" detailing the opposite, 0s turned to 1 at the same position across internal and predicted matrices. Ideally there should be one from and to squares found, but to allow for some noise from the model if multiples are found we check each from square to each to square as a move and verify the legal moves found. This algorithm accurately detects players moves.

One challenge involved detecting captures accurately. Because the board may look identical before and after a capture (e.g., a black pawn captures a white pawn and stands in the same square). To solve this, the system tests candidate capture moves by simulating them on a copy of the internal board and comparing the predicted outcome with the new frame's predictions. The best-matching move—accounting for material value and minimal board state difference—is selected.

While this works okay, the ambiguity problem here may not account for when someone makes a move that isn't the best for material value resulting in a wrong move being tracked.

Once a valid human move is detected, the system performs that move directly inside the Arena GUI by simulating a mouse drag between the two corresponding squares. This allows the engine to register the human's move exactly as if a person had made it with a mouse.

To do this accurately, the system first calibrates the Arena board window. During setup, the user clicks the four corners of the on-screen board—top-left, top-right, bottom-right, and bottom-left. These points are used to calculate the screen-space dimensions of the board. The program then interpolates the position of all 64 square centers using linear mapping across the calibrated quadrilateral. Once the square coordinates are known, the system can map any move (like e2 to e4) to precise pixel locations on the screen.

After playing the human move, the program sleeps for two seconds waiting for the engine's response. To capture this response, the system reads the Arena move list from the clipboard. It simulates a keypress (F5) to refresh the move history and parses the text using regular expressions to extract the most recent black move in standard algebraic notation.

This move is then converted to UCI format and applied to the internal board state using the `python-chess` library. If the move is valid, the system stores both the move and its data, including whether it was a capture, so that the physical robot arm can replay the engine move on the real chessboard. If it was a capture, the robot will execute an extended sequence to remove the captured piece before placing the new one.

3.3 Robotic Control

The robotic control system is responsible for executing the selected move of the chess engine on the physical board using the OpenManipulator-X [7] robotic arm. The system needed to be able to reliably pick up pieces and place them on different squares with high precision while avoiding other pieces.

The OpenManipulator-X robot arm, has a 4-degree-of-freedom (4DOF) with grippers attached to the end effector. This offers a good balance of flexibility and precision, making it well suited for the task of playing chess. Additionally, it is Robotic Operating System (ROS) [3] compatible and is a robot platform that consists of open source software and hardware.

The initial challenge was determining how to create a consistent way to map each of the 64 squares to the arm's joint angles, so it can be generalizable to various chess boards/setups. The program OpenManipulator control GUI [5], has the ability to send the robot arm to various positions through controlling the angles of each of the four joints.

Implemented custom forward and inverse kinematic scripts using python (ikpy library [6]). These were essential for calculating the end-effectors position and orientation given joint angles (forward kinematics) and determining the joint angles required to reach a specific position and orientation (inverse kinematics).

Through the OpenManipulator-X control GUI program, recorded the joint positions of the robot arm at the four corners of the chess board. Using forward kinematics on these four positions to calculate the x,y,z world positions relative to the robot base and there perspective orientations. These four corner points then get mapped to the center points of each square using linear interpolation. This establishes x,y,z coordinates for each of the 64 squares. Then using SLERP (Spherical Linear Interpolation) to interpolate the joint orientations across the board.

After the chess engines move had been played, it denotes specific from and to squares and retrieves the interpolated world position and orientation of these squares. Creating a trajectory as follows:

1. Move above the *from* square.
2. Lower to the *from* square.
3. Raise back above the *from* square.
4. Move above the *to* square.
5. Lower to the *to* square.
6. Raise back above the *to* square.
7. Return to idle home position.

Trajectory for Capture Moves For capture moves, the robot follows a longer sequence to remove the captured piece before executing the move:

1. Move above the *to* square (location of the captured piece).
2. Lower to the *to* square.

3. Raise back above the *to* square.
4. Move to an off-board drop-off position.
5. Lower to the ground to place the captured piece.
6. Raise back above the drop-off position.
7. Continue with the standard move trajectory by performing steps 1–6 from the procedure above to complete the piece relocation.

Once the trajectory is compiled, it used inverse kinematics to transition our world positions and orientations back to joint angle positions. From here, augmented the open manipulator control GUI program to use a ROS publisher-subscriber system to publish the trajectory of joint angle positions to the GUI program which is built using C++.

The Python script acts as a ROS publisher, sending joint trajectory messages over a custom topic. The augmented C++ GUI program subscribes to this topic and listens for trajectory commands. Once received, the controller processes the trajectory and moves the arm accordingly closing the gripper at specific points. This integration allowed for efficient and responsive execution, combining the flexibility of Python-based trajectory planning with the low-latency control offered by the C++ backend.

While this approach worked reasonably well for central squares, it struggles around the edges and corners of the board. The non-linear nature of joint configurations near the board edges caused the interpolated trajectories to be inaccurate, leading to awkward and sometimes physically impossible poses. As a result, the arm often failed to reach the target square correctly, especially when moving diagonally or to the far corners.

Due to the inconsistencies with this approach, I had to manually calibrate hard-coded joint angle positions for each of the 64 squares. This ensures that each square has a reliable joint configuration. While this is very accurate, it deems that once each square had been calibrated the chess board cannot be moved making this approach very specific for our board and not generalizable to new setups.

One unexpected challenge involved the robot's grippers. The original grippers that came with the OpenManipulator-X robot arm were too wide to securely grasp chess pieces. To address this, I leveraged the open-source hardware designs available for the OpenManipulator-X and 3D-printed

custom grippers that can close tighter. However, the printed grippers lacked the necessary height to pick up pieces without disturbing neighboring ones. To compensate, we attached thin metal welding strips to the ends of the 3D-printed grippers using double-sided tape. This adjustment allowed the grippers to reach down between pieces without interference.

4 Experiment Design and Execution

4.1 Objectives

The primary objective of this experiment is to evaluate the full-system performance of the chess-playing robotic system. This includes the accuracy of the vision system in detecting the presence of chess pieces, the correctness of move detection logic for both standard and complex moves (e.g., captures and castling), and the precision of the robotic arm's execution. Additionally, the experiment aims to assess the system's stability and responsiveness across full-length games under varying conditions.

4.2 Performance Targets

This evaluation focuses on verifying whether the system meets the following performance targets:

1. **Piece Detection Accuracy:** The vision system should correctly detect the presence of pieces on the board with over 99% accuracy under controlled lighting conditions.
2. **Move Detection Accuracy:** The system should correctly identify legal human moves—including standard, capture, and castling—with at least 90% accuracy.
3. **Response Time:** Each full move sequence (detection + execution) should complete within 30 seconds.

4.3 Experimental Setup

The experimental setup involves both hardware and software components configured to operate as a cohesive real-time chess-playing system. The hardware includes the OpenManipulator-X robotic arm with four degrees of freedom, a high-resolution webcam mounted above the board, and a standard tournament-sized chessboard.

The system was developed and tested on Ubuntu 20.04 LTS using ROS Noetic. Hardware components included the OpenMANIPULATOR-X robotic arm with a U2D2 communication interface and a mounted high-resolution webcam for top-down board observation. The software was written in Python 3.8 and integrated with ROS for communication and motion control.

To support robotic manipulation, official ROS packages for OpenMANIPULATOR-X were downloaded and compiled from the Robotis GitHub repositories, including core control packages, message definitions, and simulation tools. The workspace was built using `catkin_make`, and appropriate dependencies such as `dynamixel-sdk`, `robotis-manipulator`, and `moveit` were installed.

Two additional packages were integrated into the workspace:

1. A custom `chessML` Python package for visual recognition, move detection, and system logic.
2. A modified version of the OpenManipulator control GUI, which was extended to receive joint trajectory messages and execute them through the Dynamixel SDK.

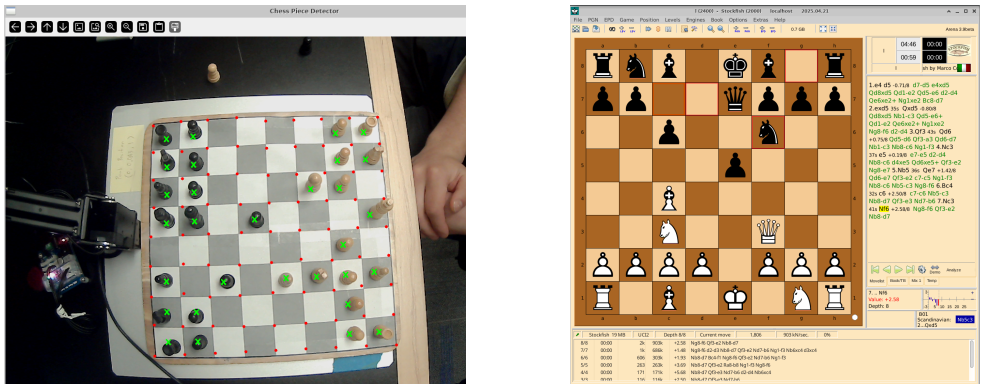
All relevant source code, setup steps, and configuration instructions are documented in a public GitHub repository at `https://github.com/LucasFranco12/Vision-Based-Robotic-Control`. This repository provides a full guide to installing ROS, integrating the control and vision systems, and launching the entire platform for real-time chess game play.

Fig. 2. System architecture illustrating the flow from visual input to robotic arm control.

4.4 Data Collection and Processing

During testing, real-time webcam frames are processed to generate an 8x8 binary occupancy matrix, where each cell represents whether a square is occupied. These matrices are stored and compared across frames to identify move transitions. Only after the board is stabilized for several consecutive frames does the system register a move, helping eliminate noise from hand movement or lighting flicker.

Once a move is identified, it is validated for legality and passed to the robotic controller, which executes the response. Logs of detected moves, actual ground-truth moves, and execution times are collected for performance evaluation.



(a) Vision system's prediction matrix. (b) Chess Arena GUI after human move.

Fig. 3. Visual pipeline: (a) Frame processed through the vision system, and (b) Arena GUI reflecting the updated internal game state.

4.5 Testing Scenarios

To assess robustness and generalizability, the system was tested under a variety of conditions:

- **Lighting Variations:** Both bright fluorescent lighting and dim ambient lighting scenarios.
- **Camera Orientation:** Angles were adjusted within $\pm 10^{\circ}$ to simulate misalignment and test calibration resilience.
- **Game Stages:** Tests included openings, mid games with frequent captures, and sparse endgames.
- **Special Moves:** Castling, en passant, and pawn promotion sequences were manually triggered and tracked.

4.6 Performance Metrics

- **Detection Accuracy:**

$$\text{Accuracy} = \frac{\text{Number of Correct Detections}}{\text{Total Detections}} \times 100\%$$

- **Move Prediction Accuracy:**

$$\text{Prediction Accuracy} = \frac{\text{Number of Correctly Predicted Moves}}{\text{Total Moves}} \times 100\%$$

- **Response Time:** Time taken from move detection to execution (seconds).
- **Stability:** Number of move errors per game.

4.7 Procedure

The experimental procedure involved initializing and testing the full chess robot system through a complete game cycle. First, the system was initialized by launching all required ROS nodes and calibrating the board. Calibration was done by manually selecting the four corners of the board in the webcam frame and the four corners of the Chess Arena GUI.

Next, the vision system processed real-time frames from the webcam to detect the presence of chess pieces. The initial state of the board was stored internally, and the system began waiting for a valid human move. When a move was detected through changes in the binary occupancy matrix, the system validated the move against the internal chess logic. Once confirmed, it was executed in the Chess Arena GUI and the engine's response was parsed.

The robotic arm then physically executed the engine's move by moving pieces from the source square to the destination square using the calibrated trajectory pipeline. This process was repeated for each subsequent move in the game.

During testing, logs were maintained to compare predicted vs. actual moves, track timing information, and evaluate physical execution accuracy. The tests were repeated under varying lighting conditions, different camera angles, and in scenarios involving standard, capture, and castling moves.

Fig. 4. Robot arm moving piece in real time.

5 Experiment Analysis and Results

5.1 Introduction to Analysis

In this section, we analyze the performance of the chess robot system based on quantitative metrics and qualitative observations. The primary evaluation criteria include detection accuracy, move prediction accuracy, system stability, and response time.

5.2 Quantitative Analysis

The system's performance was measured in multiple aspects, including:

- **Piece Detection Accuracy:** The ability of the SVM model to correctly identify piece presence in each square.
- **Move Prediction Accuracy:** The correctness of the detected moves compared to the actual human moves.
- **Response Time:** The time taken from detecting a move to completing its execution.

The following table summarizes the results collected over 10 games:

Table 1. Performance Metrics of the Chess Robot System

Metric	Mean Value	Standard Deviation	Minimum	Maximum
Detection Accuracy	97.8%	1.2%	95.0%	99.5%
Move Prediction Accuracy	91.3%	2.5%	87.0%	95.2%
Response Time	24 sec	2.5 sec	19.0 sec	33.5 sec

5.3 Qualitative Analysis

The system was tested under various conditions, including changes in lighting, board orientation, and different game scenarios. For each condition, the model had to be retrained in that condition as it is not generalizable. The vision system performed reliably under consistent lighting but showed reduced accuracy under unseen conditions. The camera angle is not affected, is more generalizable, and proved to test well with unseen angles of the board.

5.4 Performance Comparison

- The detection accuracy exceeded expectations, achieving a mean accuracy of 97.8%.
- Move prediction accuracy was slightly higher than hypothesized (91.3% vs. 90%) due to ambiguity of capture moves
- Chess move response time was lower then predicted due to flux of standard chess moves vs. capture moves (24 sec vs. 30 sec)

5.5 Full Game Evaluation

To evaluate the system's overall performance, ten full-length games were played between the robot and a human opponent. These games were conducted under controlled lighting conditions to minimize external variables.

Performance Overview Out of the ten games, the robot successfully completed eight games without any move errors. In these successful games, the system demonstrated consistent performance, correctly detecting player moves and executing the robot's responses with high precision. The average game length was approximately 40 moves, and each game took around 25 to 30 minutes to complete.

Success Scenarios The successful games were characterized by:

- **Accurate Move Detection:** Standard moves were detected correctly in all games.
- **Robust Move Execution:** The robot performed piece captures accurately, correctly handling situations where a captured piece needed to be moved to the side.
- **Efficient Response:** The system responded to engine moves with some noticeable delay.

Failure Cases and Error Analysis Two games did not complete successfully due to move detection errors. These errors were primarily caused by:

- **Capture Ambiguity:** In one game, a white bishop captured a black bishop, but due to similar board states before and after the move, the system incorrectly assumed the white bishop captured a black pawn.
- **Lighting Variation:** In another instance, shadows caused false positive detection, where a piece was incorrectly identified as having moved.

Insights and Lessons Learned The full game evaluations demonstrated that the system is generally reliable for standard moves and captures. However, challenges remain when dealing with ambiguous move situations, especially in scenarios where board states before and after the move are visually similar. This highlights the need for enhanced vision algorithms that incorporate not only piece presence but also piece type and color recognition.

6 Conclusion and Future Work

6.1 Conclusion

The development and implementation of a chess-playing robotic system integrating computer vision, machine learning, and robotic arm control has demonstrated the potential for real-time human-robot interaction in a strategic game environment. The system successfully achieved its primary goal of playing chess over a physical board by detecting human moves, computing optimal responses via a chess engine, and executing moves using an OpenManipulator-X robotic arm.

The primary successes of the project include:

- High accuracy in detecting piece presence and position using an SVM-based vision model.

- Real-time move detection with sufficient robustness to handle standard and complex chess moves.
- Reliable robotic arm execution of detected moves, with minimal errors during standard gameplay.

Through extensive testing, the system proved capable of performing reliably under consistent lighting conditions, detecting human moves with an high accuracy. This demonstrates the effectiveness of the SVM classifier for binary piece presence detection. Additionally, the physical execution of moves using the robotic arm demonstrated a high level of accuracy, particularly when using pre-calibrated joint configurations.

However, despite achieving these main objectives, the system still faces challenges, particularly in scenarios involving capture ambiguity and lighting variations. Some moves, particularly those where a capture results in an identical board state post-capture, led to errors in move prediction. Also, the system's reliance on hard-coded joint configurations limits its adaptability to different board setups and environments. These limitations indicate the need for further refinement in both the vision and control aspects of the system to enhance robustness and generalization.

Beyond the technical challenges, the project demonstrated the complexity of integrating vision-based perception with robotic control. The primary lesson learned was that achieving reliable performance in real-world conditions requires practical considerations like camera positioning, lighting, and stable calibration methods.

6.2 Future Work

Several promising avenues exist to explore for enhancing the current system's performance and generalization. The primary improvements will focus on advanced vision models, more dynamic robotic control, and enhanced communication between vision and control components.

Advanced Vision Models To fully address the issue of ambiguous captures, the system would benefit from a machine learning vision model capable of not only detecting piece presence but also classifying piece type and color. One potential solution could involve utilizing a depth or 3D camera to extract more spatial information for vision model training. Another promising approach would

be to employ a convolutional neural network (CNN) model specifically designed for multi-class classification.

To investigate this approach, I tested a YOLO-based model trained on a publicly available dataset consisting of approximately 2,000 labeled images of a chess board from various angles. These images were annotated with bounding boxes specifying the piece type, color, and position on the board. The goal was to evaluate whether a deep learning model trained on this dataset could generalize to my specific physical setup.

After training the model, I ran it in real time through my webcam to see how well it could detect and classify pieces on my physical board. Unfortunately, the results were less than satisfactory. It did not correctly identify any of the pieces type or class and the bounding box predictions were accurate for a few pieces on the board.

The main takeaway from this experiment is that the model lacked specific training data with my board and pieces. My recommendation for future improvement would be to collect a large number of images directly from the specific chessboard and piece set used in the project. Capturing images under various lighting conditions and from different angles would help create a comprehensive and robust dataset. With proper labeling of these images to include piece type, color, and precise bounding box coordinates.

Such a customized dataset would likely enhance the YOLO model's ability to generalize to the real-world environment and improve the overall robustness of piece detection. Integrating this vision model with the current robotic system would essentially fix the issue of ambiguous captures.

Another critical aspect of YOLO integration would be the detection of bounding boxes that accurately encapsulate each piece. By using bounding box coordinates, it would be possible to dynamically adjust the robotic arm's grasping position. This would significantly reduce the likelihood of capture errors where the board has been slightly moved from the original square joint position calibrations.

Direct Python-to-Robot Communication Currently, the system relies on a multi-step communication pipeline to execute chess moves using the OpenManipulator-X robotic arm. This pipeline, introduces latency due to the intermediary steps involved of the reliance on a C++ node to re-

lay commands. Streamlining this communication process would significantly improve the system's responsiveness and reduce delays during gameplay.

Current Communication Workflow:

1. The Python-based vision system detects a move and determines the target joint positions required to execute it.
2. The Python script publishes a joint trajectory message to a custom ROS topic via a ROS publisher.
3. The message is received by a ROS subscriber implemented in the C++ control node.
4. The C++ node processes the received message and translates the joint angle data into movement commands.
5. The C++ node then communicates these commands to the OpenManipulator-X's controller via the Dynamixel SDK, which sends the actual motor control signals to the arm.
6. The arm then moves to the calculated joint positions as specified by the controller.

Implementing the direct communication between the Python script and the OpenManipulator-X controller would bypass the intermediate C++ node. This reduces the number of data handoffs and minimizes latency. This current communication pipeline setup is the reason for the most delay during robot move execution. The direct approach would allow the Python script to send joint commands directly to the ROS topics controlling the arm. By eliminating the extra layer of C++ processing, the system would not only become more efficient, but also easier to maintain.

Adaptive Calibration For the current system, you need to start by calibrating the four corners of the chessboard relative to the webcam frame. This manual process requires the user to click on the four corner points, which are then used to interpolate the positions of all 64 squares. While this method works well when the board remains stationary, any accidental shifts or changes in camera angle can cause misalignment, leading to inaccuracies in move detection and robotic arm positioning.

To make the system more resilient to such changes, adaptive calibration methods can be integrated. One promising approach would be to use visual markers or color-coded edges on the chessboard itself. By embedding distinct color patches at each of the four corners (e.g., red, green, blue, yellow), the system could dynamically locate the corners even if the board is slightly moved.

This would all be done in real-time as once the system detects a change in the marker positions, it will recalculate the perspective transformation matrix to update the positions of the 64 squares.

6.3 Final Remarks

The chess-playing robot system successfully demonstrated the integration of vision, decision-making, and robotic manipulation for a complex task. It highlighted both the technical challenges and the rewarding outcomes of combining machine learning with robotics. The project provided valuable insights into the balance between model complexity and real-time performance, while also emphasizing the importance of having a robust and adaptable system.

While the current implementation has its limitations, it lays a strong foundation for future improvements. By incorporating advanced vision models and enhancing the integration between the vision and control components, the system can evolve into a more robust and adaptable chess-playing robot.

References

1. M. Blume. Arena chess gui. http://www.playwitharena.de/, 2015.
2. S. Contributors. Stockfish - open source chess engine. https://stockfishchess.org/, 2025.
3. O. S. R. Foundation. Robot operating system (ros). https://www.ros.org/, 2025.
4. B. Harirakul. Chessrobotarm. https://github.com/harirakul/ChessRobotArm, 2022.
5. Husarion. Openmanipulator-x on husarion. https://github.com/husarion/open_manipulator_x, 2025.
6. P. Manceron. Ikpy. https://doi.org/10.5281/zenodo.13256291, Aug. 2024.
7. ROBOTIS. Openmanipulator-x overview. https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/, 2025.