Chess Board Vision Final Report

Ao Chen, Lowell Dever, Tianqing Feng, John Ner

Section 1. Team Report:

Project Aim:

The goal of this project was to output a portable game notation (PGN) file from a game of chess recorded over a video camera. PGN files display the subsequent order of chess moves using algebraic chess notation. Most importantly, this project gives chess players an affordable option to record and review their chess matches. Boards with comparable functionality such as the DGT Electronic Chessboard cost hundreds to thousands of dollars, while the implementation of this computer vision-based setup could potentially cost under $50 depending on the type of camera and stand purchased.

Expected Outcome:

The input is a live video feed from a webcam, and the output is a PGN file. This PGN file is generated during the match and can be viewed after the game's completion. Any change to the piece location will be detected, but only valid moves according to the python chess library are recorded and saved to the PGN file.

Evaluation Metrics:

The project was broken down into four distinct parts: board detection, piece classification, move event detection, and PGN generation. For the board detection, it needed to find all corners of the board, including the outside edges. If done correctly, it should find an 11x11 set of points. The outer points are then trimmed so only the corners of the 8x8 squares are saved. Desired piece classification consists of precise identification of all 6 piece types, including both black and white color distinction. The resulting COCOmAP and confusion matrices are appropriate evaluation tools to judge the effectiveness of piece classification. Move event detection is evaluated by comparing the input move from the user to the output move displayed on the digital board. This digital board is on-screen during gameplay. The corresponding move on the digital board needs to match the input move from the user. The PGN generation file needs to accurately reflect every valid move made during the game, expressing the moves via algebraic chess notation. Evaluation of the file is done by reviewing the list of actual moves made during the game to the moves generated in the PGN file.

Results & Summary:

As seen by the images below, the corners of the board were successfully identified and labeled. After the corners are correctly detected, we can then use perspective transform to easily create the 8x8 grid of a chessboard.
The resulting COCOmAP for piece classification is 0.989. A confusion matrix was made for test images in the supplement material.

		predicted											
		WK	WQ	WB	WN	WR	WP	BK	BQ	BB	BN	BR	BP
actual	WK	8	0	1	0	0	0	0	0	0	0	0	0
	WQ	0	12	0	0	0	0	0	0	0	0	0	0
	WB	0	0	21	0	0	0	0	0	0	0	0	0
	WN	0	0	0	22	0	0	0	0	0	0	0	0
	WR	0	0	0	0	20	0	0	0	0	0	0	0
	WP	0	0	0	0	0	59	0	0	0	0	0	0
	BK	0	0	0	0	0	0	9	0	0	0	0	0
	BQ	0	0	0	0	0	0	0	12	0	0	0	0
	BB	0	0	0	0	0	0	0	0	21	0	0	0
	BN	0	0	0	0	0	0	0	0	0	24	0	0
	BR	0	0	0	0	0	0	0	0	0	0	20	0
	BP	0	0	0	0	0	0	0	0	0	0	0	59

The accuracy is 99.653%

Board Detection Results:

Detected Corners:

Original vs Final Implementation:

For board detection, we originally planned to do a perspective transform before detecting the corners, but in the final implementation, a perspective transform was used to make it easier to find the 8x8 grid.

There was no deviation between the original and final piece classification methods. The method involved training a convolutional neural network (CNN) capable of identifying all 6 types of white and black pieces. The training and validation datasets consisted of images taken from the actual chess board setup used for project testing.

For move detection, the original method for implementation was to use the screen coordinates of the detected pieces to determine their relative location on the board. By comparing two matrices, a moving piece would be detected in a new location in the final matrix. If this move was valid, the move would be confirmed and pushed into the program stack. The final implementation of this process was a bit more complex, as the board was not a perfect square due to the camera angle. Several transforms were required to normalize the board into a perfect square, which then enabled the group to perform the above implementation.

For PGN, it was as planned.

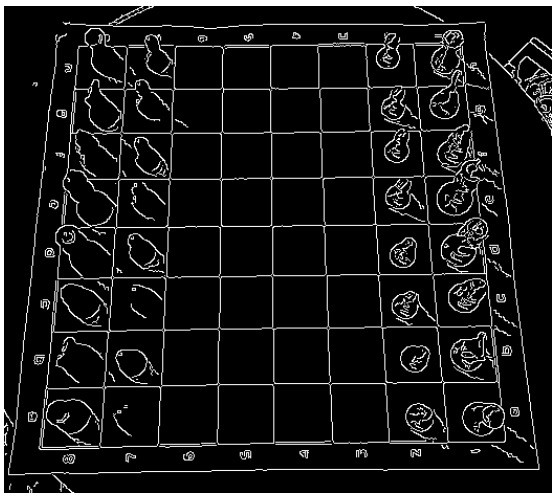
Section 2. Individual Report

My main task for this project is to work on Board Detection and help Ao train the model for Piece Detection and label the training images, but I also helped in integrating different parts of the project. I also helped generate the detection accuracy results by adding the number of pieces it detected and calculating the average confidence per image.

Board Detection:

The board detection process starts by cropping the image so that the bottom part of the chessboard is about the same width as the image. After this is done, the image is stored in a file to be used later.

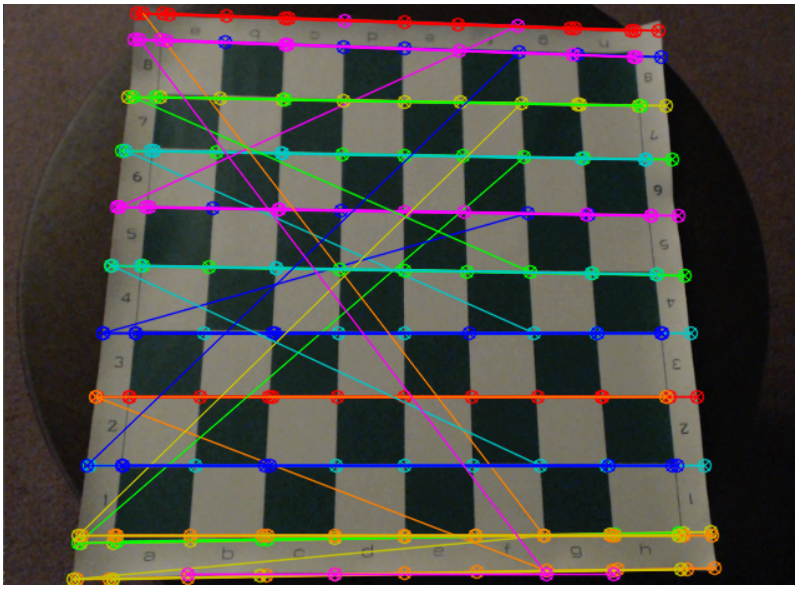
The function detect_corners() is used to load the image and convert it to grayscale for the Canny Edge detection. I also applied increased the contrast and applied some blur to improve the results.



Canny Edge Detection

After the edges are detected, it is sent to the hough_line() function to get the vertical and horizontal lines. Once the lines are generated, I called another function to find the intersection between each line and return all of the points where they intersect. However, this results in multiple corners where there should be one. This is because the hough_line generated more lines than was expected. To fix this, I added a function called remove_duplicates() which returns one point within a cluster. Preferably the middle point.

Once clustered, the points need to be augmented so that the order of points is based on their position from top to left and then down. Without removing the duplicates and augmenting the points, the result will look like this:



Board without points augmentation, cluster removal, and outer points still included

Once the coordinates are rearranged, another function called remove_outside_points is used to remove the outer points that we will not need for the chessboard grid. Once it returns the inner points of the board, I used cv2.drawChessBoardConers to output the 9x9 inner points. Once called, it looks like this:

The corner points and the corner image is returned to the calibrate_board function for further processing.

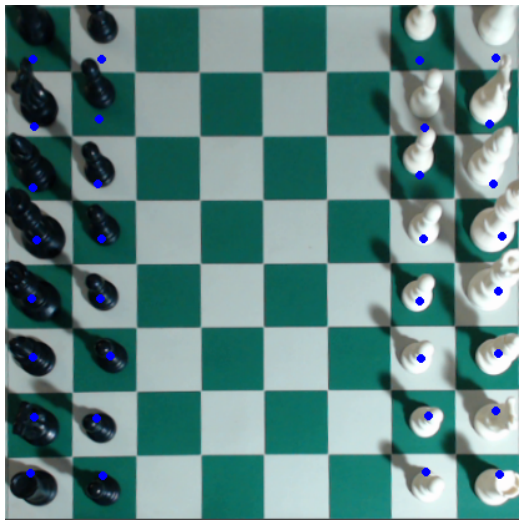
The corner points generated will need to be stored in a 9x9 array so we can easily add pieces to the grid later. A function called save_corner_points is used for this. Once it finishes adding the points to the grid, it returns the new array.

Piece Detection:

I also helped Ao and the other members to take pictures of the board and label them using Roboflow and LabelImg. We were able to label at least 200 pictures which we split into Validation, Testing, and Training. The training images are then augmented further by using mosaic since this yielded the best results compared to other augmentations. After applying this augmentation, Roboflow allowed us to generate 300 training images which we used to train our best model. We tried using the public dataset provided by Roboflow to test yolov4 but it resulted in only 50% accuracy and most of the pieces while it can detect it, it does not classify it correctly. The best model we achieved is by using Scaled-YOLOV4-Tiny Multi-Scaled with a 0.989 cocomAP using the mosaic and random crop augmentation after 326 epochs.

Board and Piece Detection Integration:

As mentioned in section 3, when calling detect.py, it returns an array of boundary boxes and classified pieces. We now need to figure out how to add each piece to the chessboard. First, we generated a transform matrix so that when used in the board image, it will stretch the top corners of the board and bottom corners into a squared image.



Warped Image after Transform Matrix

This transformation is useful since we know exactly the size of each square, and we can figure out a general location for the chess pieces without checking every square. This will save us time since we do not need to loop 64 times just to add a piece to a square. We used the middle of the bottom corners of the boundary box to determine where a piece is located relative to the board. Ao developed an algorithm that I helped implement in the function classify_squares:

```
def classify_squares(size, midpoint):
    classify_arr = []
    for i in midpoint:
        # print("Midpoint", i[0])

        x = i[0]
        y = i[1]

        square_x = x // size
        square_y = y // size

        classify_arr.append([int(square_x), int(square_y)])

    return classify_arr
```

Note: In this function, the size is the width of each square.

After returning this classified array, we then used another function to add the classification array to the 2D Matrix:

```
def classify_2d(classify_arr, predict_arr):
    category_reference = {0: 'wk', 1: 'wq', 2: 'wb', 3: 'wn', 4: 'wr', 5: 'wp', 6: 'bk', 7: 'bq', 8: 'bb', 9: 'bn',
                          10: 'br', 11: 'bp', 12: 'em'}

    predicted_list = np.empty(shape=(8, 8), dtype=object)
    predicted_list.fill('em')

    try:

        for classify, predict in zip(classify_arr, predict_arr):
            predicted_list[classify[0]][classify[1]] = category_reference[predict]

        flipped_arr = np.fliplr(predicted_list)

        return flipped_arr

    except IndexError:
        pass
```

The array generated is then passed to the move-detection part, which Lowell worked on.

Section 3. Codes/Programs:

Github link: https://github.com/jkner/Chess-Vision

cv_chess.py is the main python code that calibrates the board, loads the model and takes the images from the webcam, and sends it to the model for an output.

cv_chess_functions.py stores most of the functions that we use for cv_chess.py to make the code more organized.

detect.py runs the detection model and takes in an image of a chessboard and outputs the boundary boxes and classifies each piece.

After running python3 cv_chess.py, the YOLOv4 model is loaded first since it takes at least 10 seconds to load. After loading the model, it performs the board calibration. The user will press c until the board is properly calibrated. Once the board detects 81 corners and the user presses s, the coordinates of the corners are stored in an array for later use. The detection starts after the array is stored.

The PGN is created and data is written to it until the program is over and the file is closed.

The detection process starts by sending the model to detect.py which will load the image stored in the directory for detection. After the detection process is over, it returns the boundary boxes of each piece and then classifies each piece along with a confidence score. If the confidence is less than 60%, we do not output it since our nms-score-threshold is set to 0.6. Our iou threshold is set to 0.3.

After the detection process returns the boundary box and classifies the array, we need to determine where the piece is located on the chessboard. To do this, we took the middle two bottom corners of each piece and shifted them slightly to be closer to the middle of the piece. This will serve as the piece location when we add it to the grid array.

We then retrieve the perspective transform matrix on the corners of the board to transform it to a 412x412 image. This way the board can be easily divided into an 8x8 grid. This returns a matrix that we can use to transform the other data to easily localize each piece.

This transform matrix is used to transform the middle points we generated before and the corners of the chessboard we generated from the board calibration step.

Once we have the transformed middle points and corner points, we can now add each of the detected pieces to an 8x8 grid. After the pieces are stored in an 8x8 array, we can now start the move detection process.

To detect the move, we compared the current board state to a previous valid board state. If nothing changed, then a move has not been found. If the two boards are not equal, then we'll perform a check to see if the move is a valid one from the list of legal moves generated by the python chess library.

If the move is valid, then we can add it to the board state and then push the move into an array for PGN generation later.

This step is repeated until the python chess library detects a checkmate or stalemate or if the user presses q which will end the game. The result is updated to the pgn file and the array is converted to a valid chess notation using the python chess library. Once converted, we can now write the notation to the PGN file and close it.

References:

Board Detection & Piece Localization

Chess piece detection - digitalcommons.calpoly.edu. (n.d.). Retrieved May 14, 2022, from https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1617&context=eesp

Board Detection:

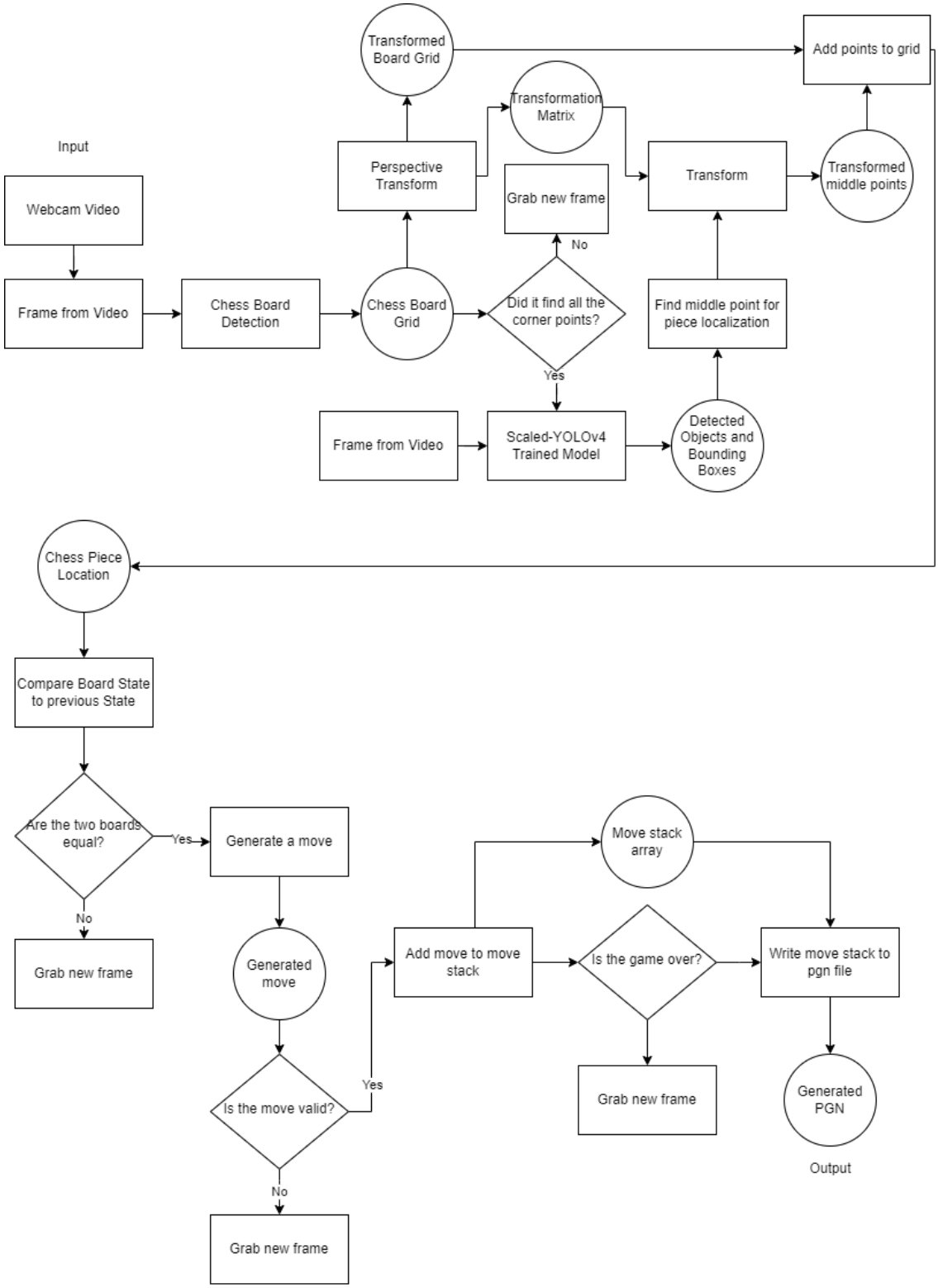
Underwood, A. (2020, October 22). *Board Game Image Recognition Using Neural Networks*. Medium. Retrieved May 13, 2022, from https://towardsdatascience.com/board-game-image-recognition-using-neural-networks-116fc876dafa

Training and Validating Model:

wangermeng2021. (n.d.). *Wangermeng2021/scaled-yolov4-tensorflow2: A tensorflow2.x implementation of scaled-yolov4 as described in scaled-yolov4: Scaling cross stage partial network*. GitHub. Retrieved May 13, 2022, from https://github.com/wangermeng2021/Scaled-YOLOv4-tensorflow2

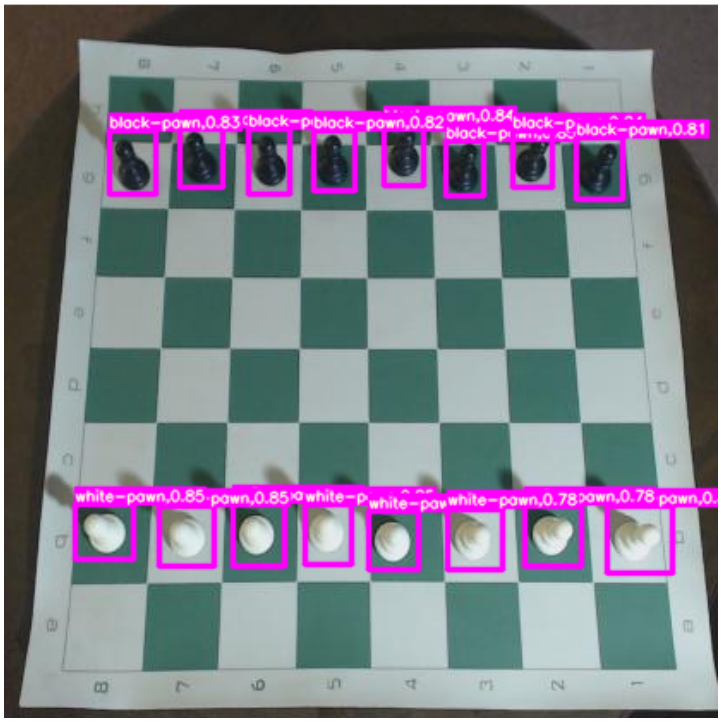
Section 4. Supplementary Material

Program Flowchart



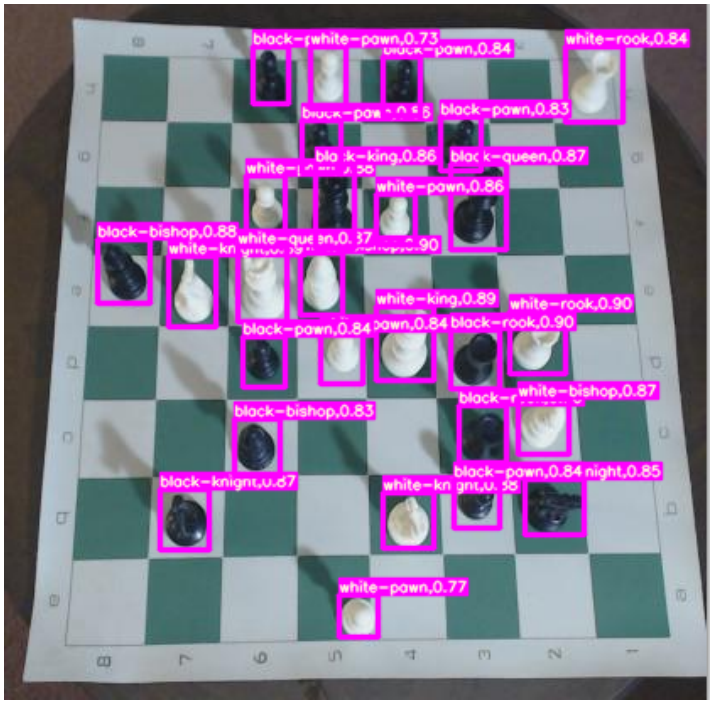
Real-time demo: https://www.youtube.com/watch?v=Nw5VhdQbd-M

Detection Test Result Images:

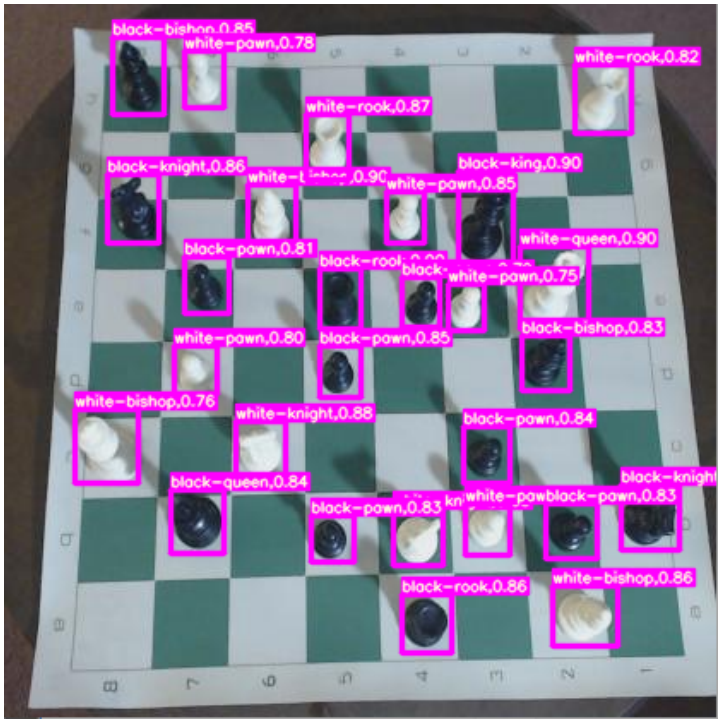


Average Confidence: 0.8306597545742989

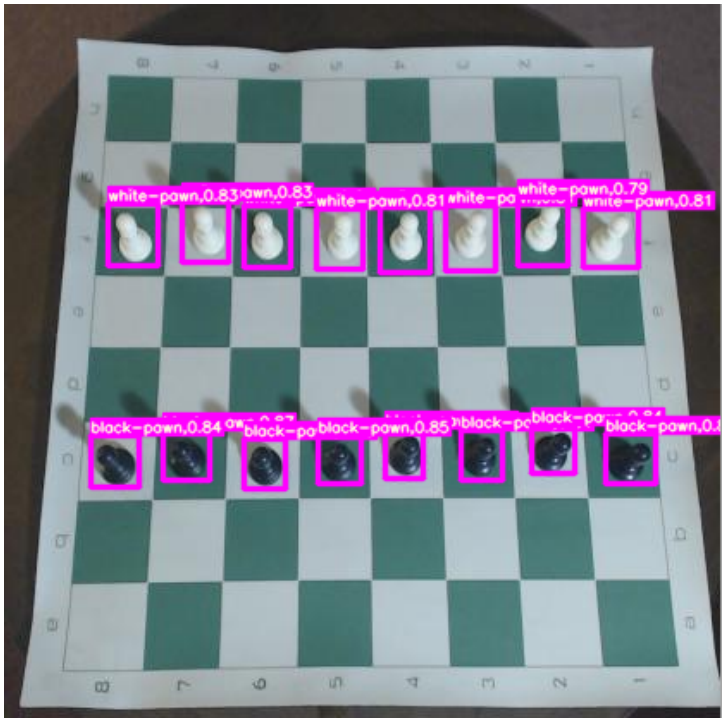
Average Confidence: 0.8438122272491455



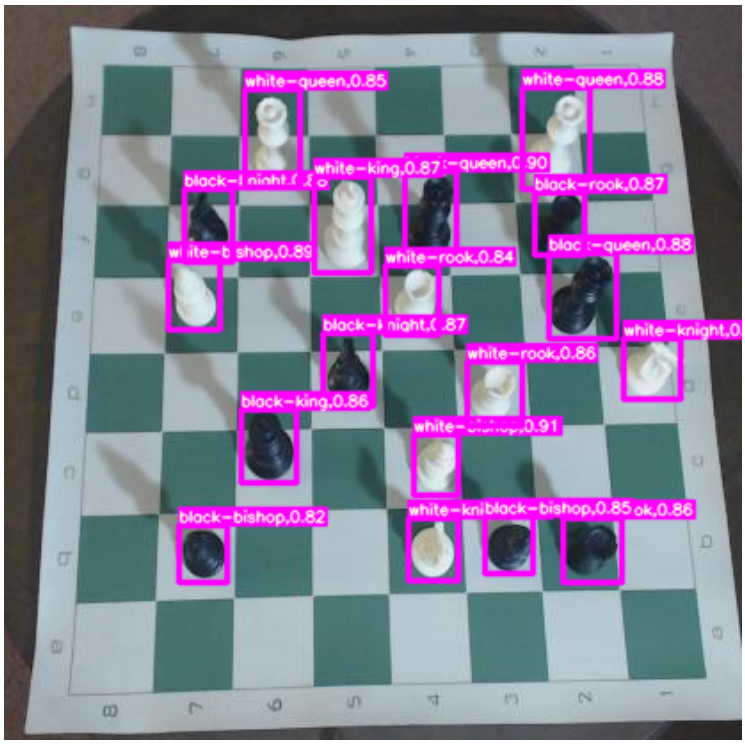
Average Confidence: 0.8553084090903953



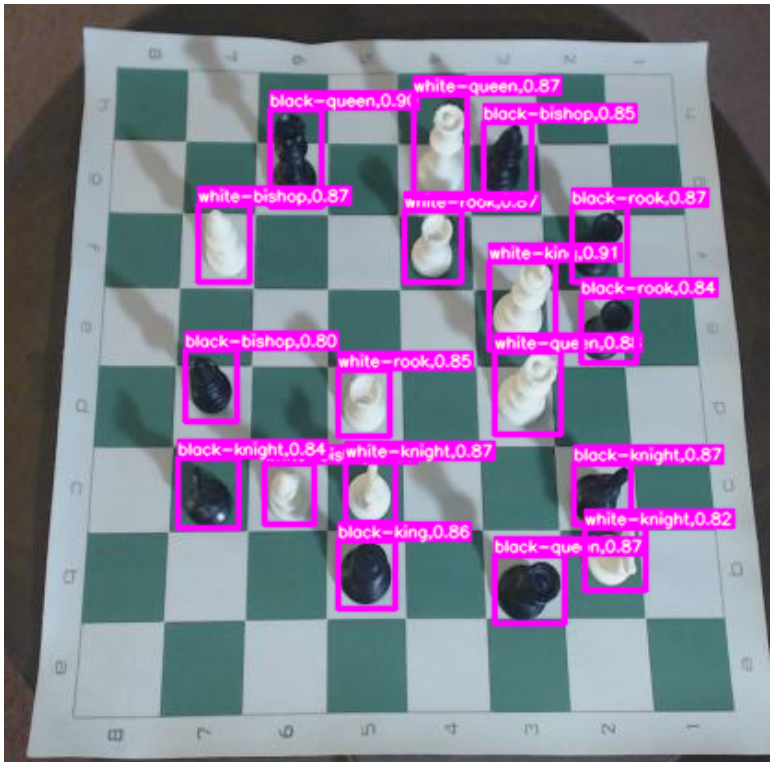
Average Confidence:0.8419597104743675



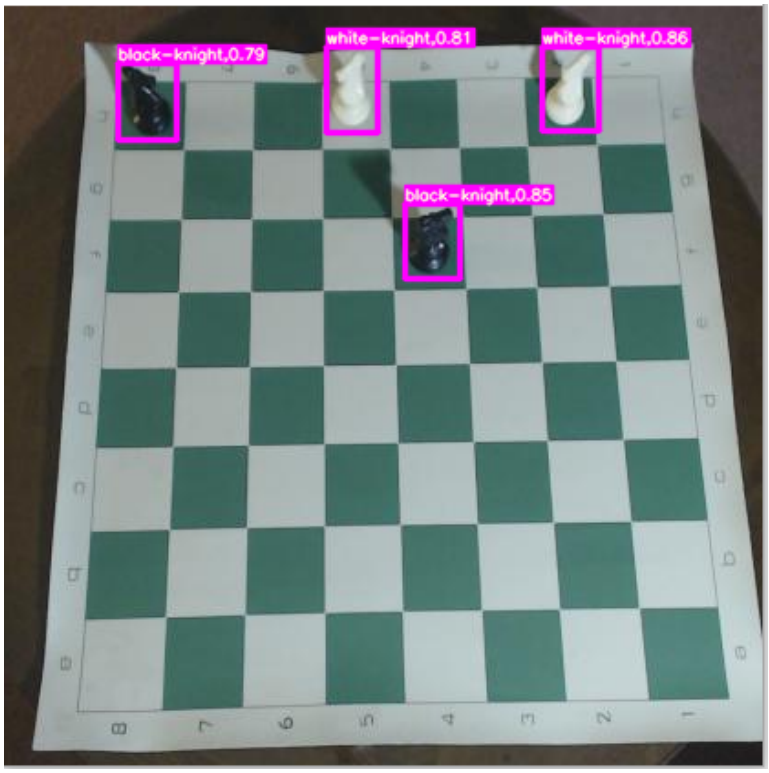
Average Confidence:0.8305948749184608



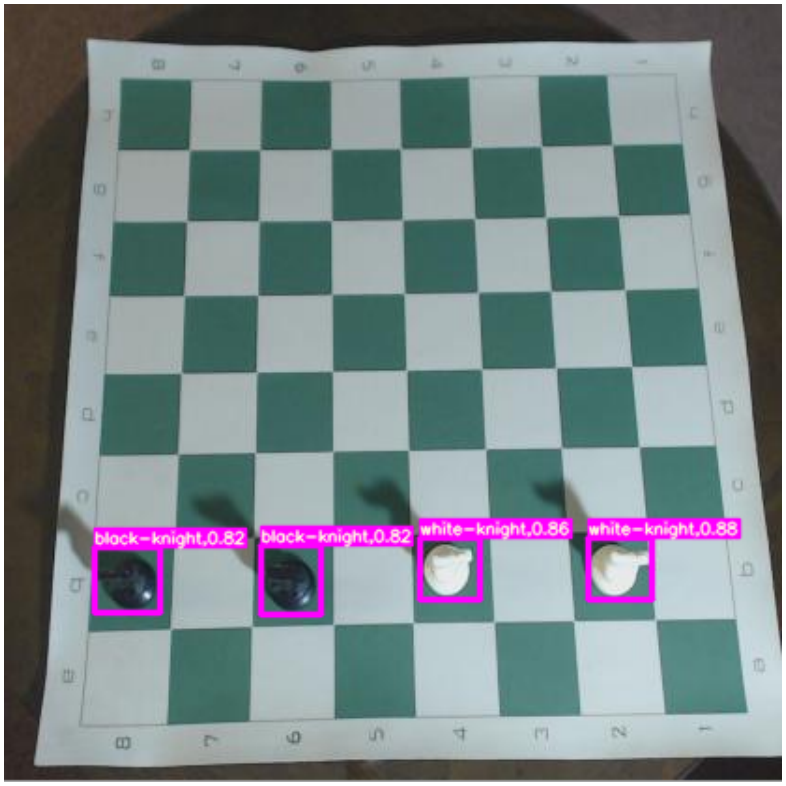
Average Confidence: 0.8663979040251838



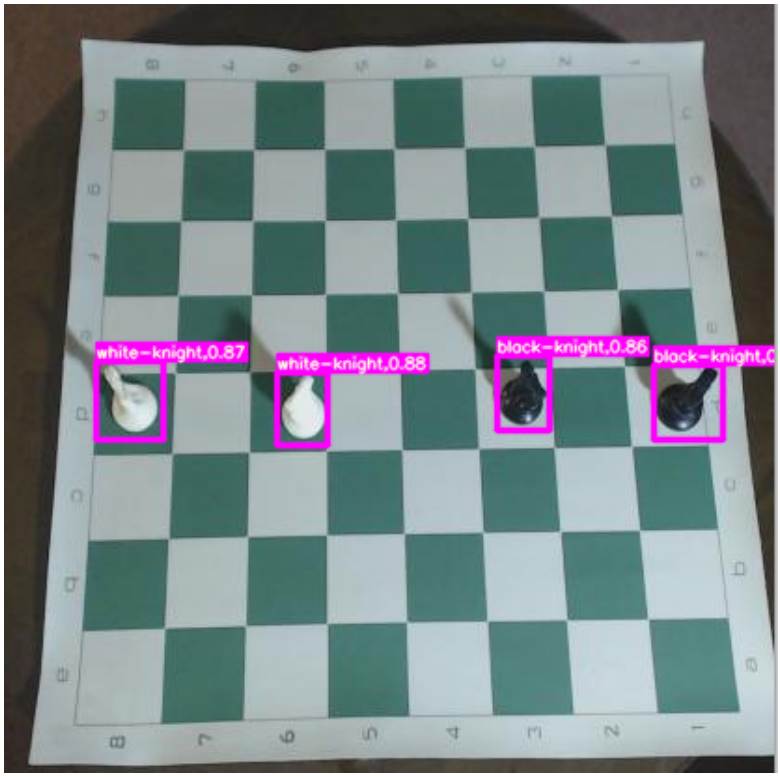
Average Confidence: 0.8644250366422865



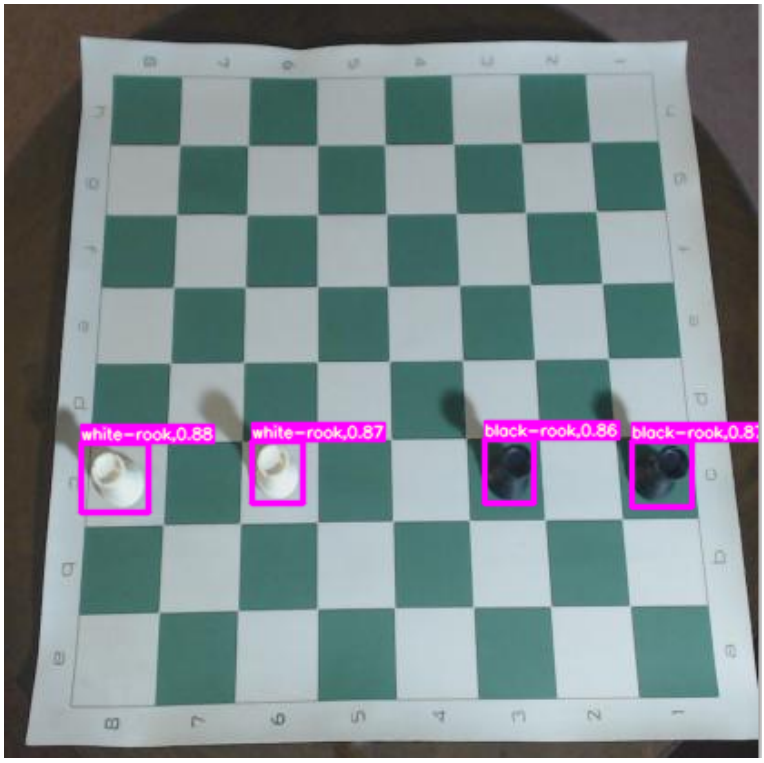
Average Confidence: 0.8310530483722687



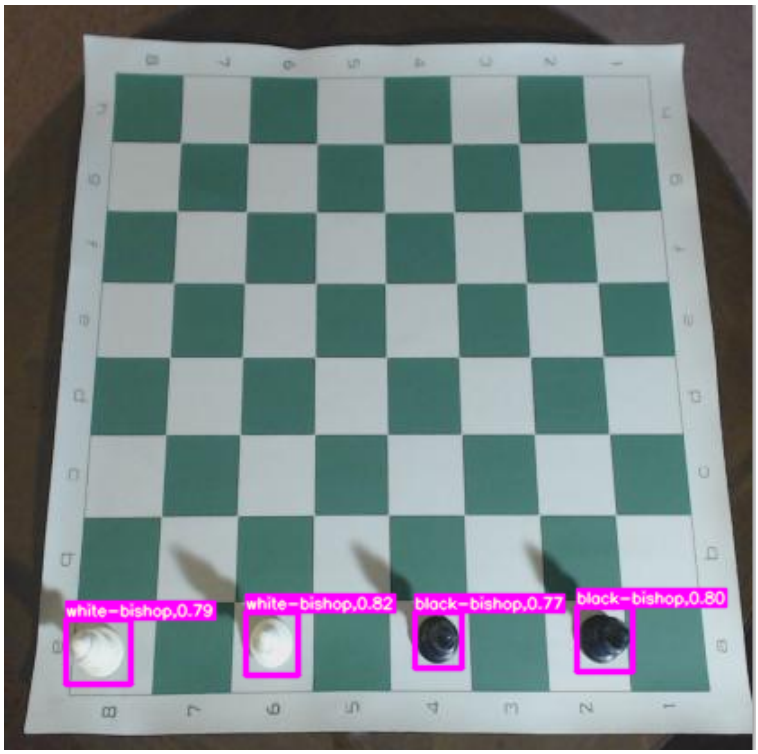
Average Confidence: 0.8426888883113861



0.8702127188444138

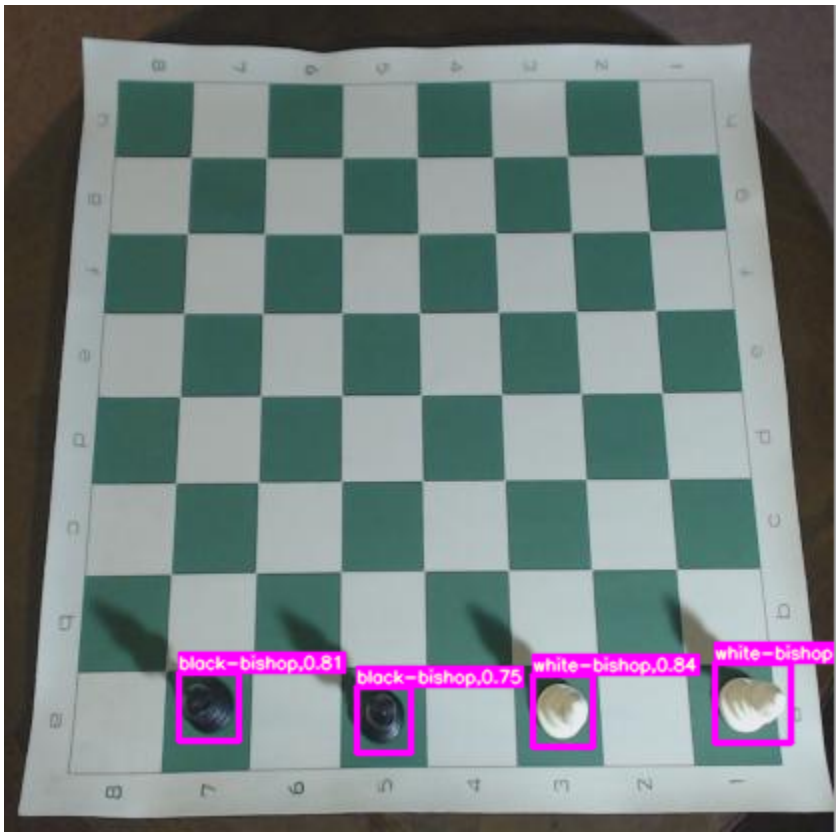


Average Confidence: 0.8715315908193588

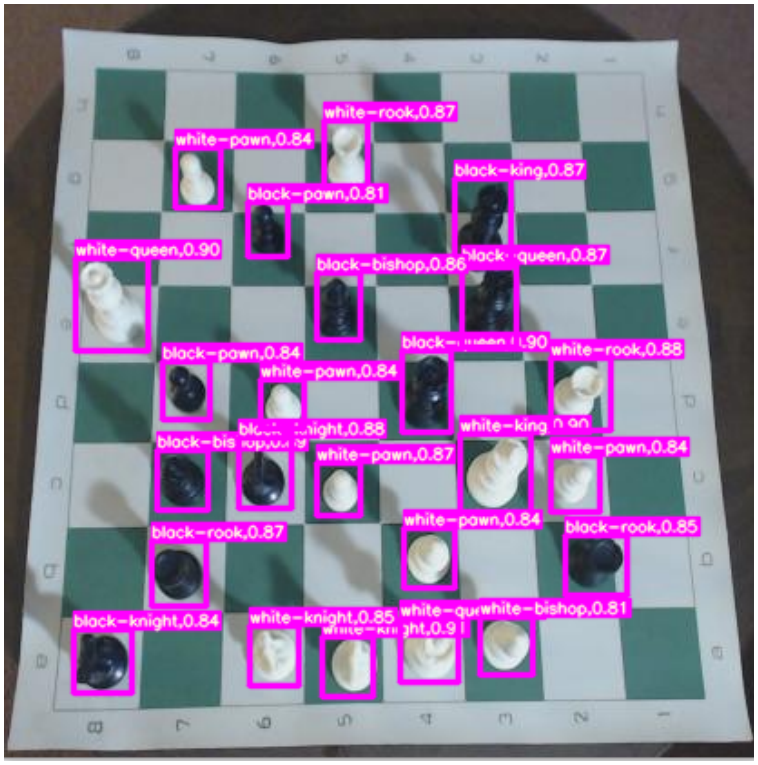


Average Confidence: 0.7916754931211472

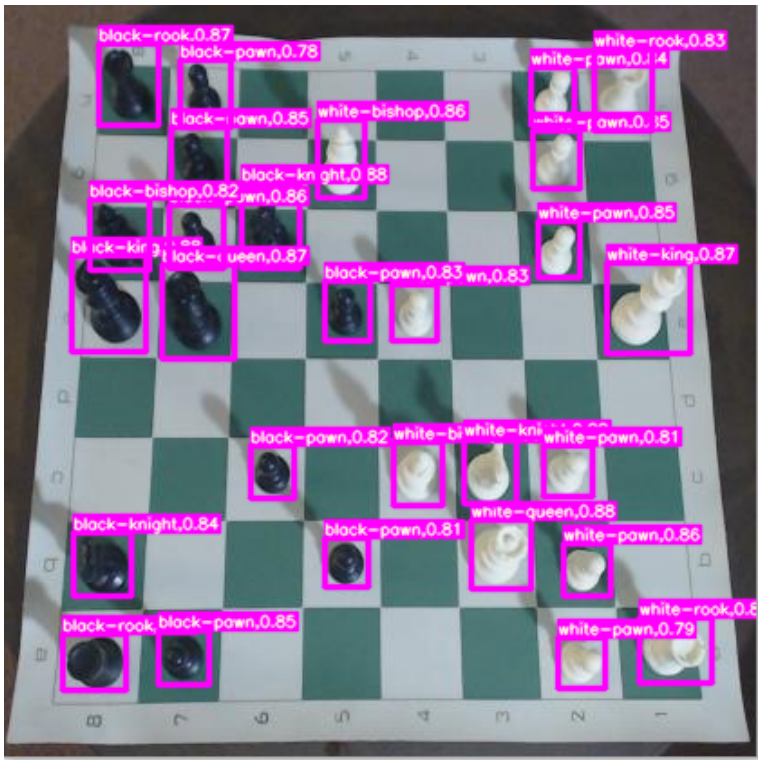
Average Confidence: 0.8522761724889278



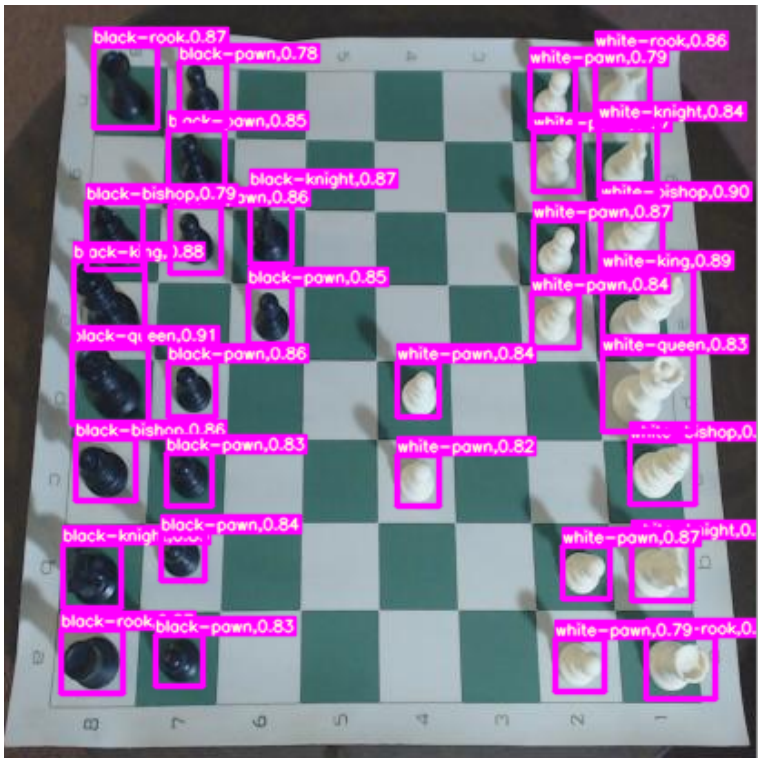
Average Confidence: 0.8103634715080261



Average Confidence: 0.8632304867108663



Average Confidence: 0.8473674931696483



Average Confidence: 0.8498046863824129

Total Average confidence for all test images: 0.8449036451001526

		predicted											
		WK	WQ	WB	WN	WR	WP	BK	BQ	BB	BN	BR	BP
actual	WK	8	0	1	0	0	0	0	0	0	0	0	0
	WQ	0	12	0	0	0	0	0	0	0	0	0	0
	WB	0	0	21	0	0	0	0	0	0	0	0	0
	WN	0	0	0	22	0	0	0	0	0	0	0	0
	WR	0	0	0	0	20	0	0	0	0	0	0	0
	WP	0	0	0	0	0	59	0	0	0	0	0	0
	BK	0	0	0	0	0	0	9	0	0	0	0	0
	BQ	0	0	0	0	0	0	0	12	0	0	0	0
	BB	0	0	0	0	0	0	0	0	21	0	0	0
	BN	0	0	0	0	0	0	0	0	0	24	0	0
	BR	0	0	0	0	0	0	0	0	0	0	20	0
	BP	0	0	0	0	0	0	0	0	0	0	0	59

Accuracy = 99.653%