`

Project Report - “Get Chess Board“

Watching chess on YouTube:
-But why not <random_chess_move>??
The video moves on without hearing you…

In a parallel universe:
The video still moves on without hearing you… but you open that position on a chess engine.

1. Motivation and Goals

Videos on YouTube provide a lot of content for enjoying chess games played and commented on by extraordinary players. The videos also happen to be fast and provide a lot of exposure to chess but they sometimes lack in commentary regarding alternative moves - mostly rightfully so. In such cases, the ability to have an analysis board of the current position open up in a browser could help the viewer study the game more effectively and better answer “what-if” questions that they might have.

With this project, I aim to decrease the friction to study a chess game played on a video platform. This could:

1. Shorten the feedback-loop on learning by making the analysis board (an infallible guide for almost all purposes) available quicker.
2. Introduce a programmatic tool for capturing chess games from videos.
3. Make watching chess videos more fun!

The project will take a screenshot of a chess game on a video platform and return a string that represents the piece placement.

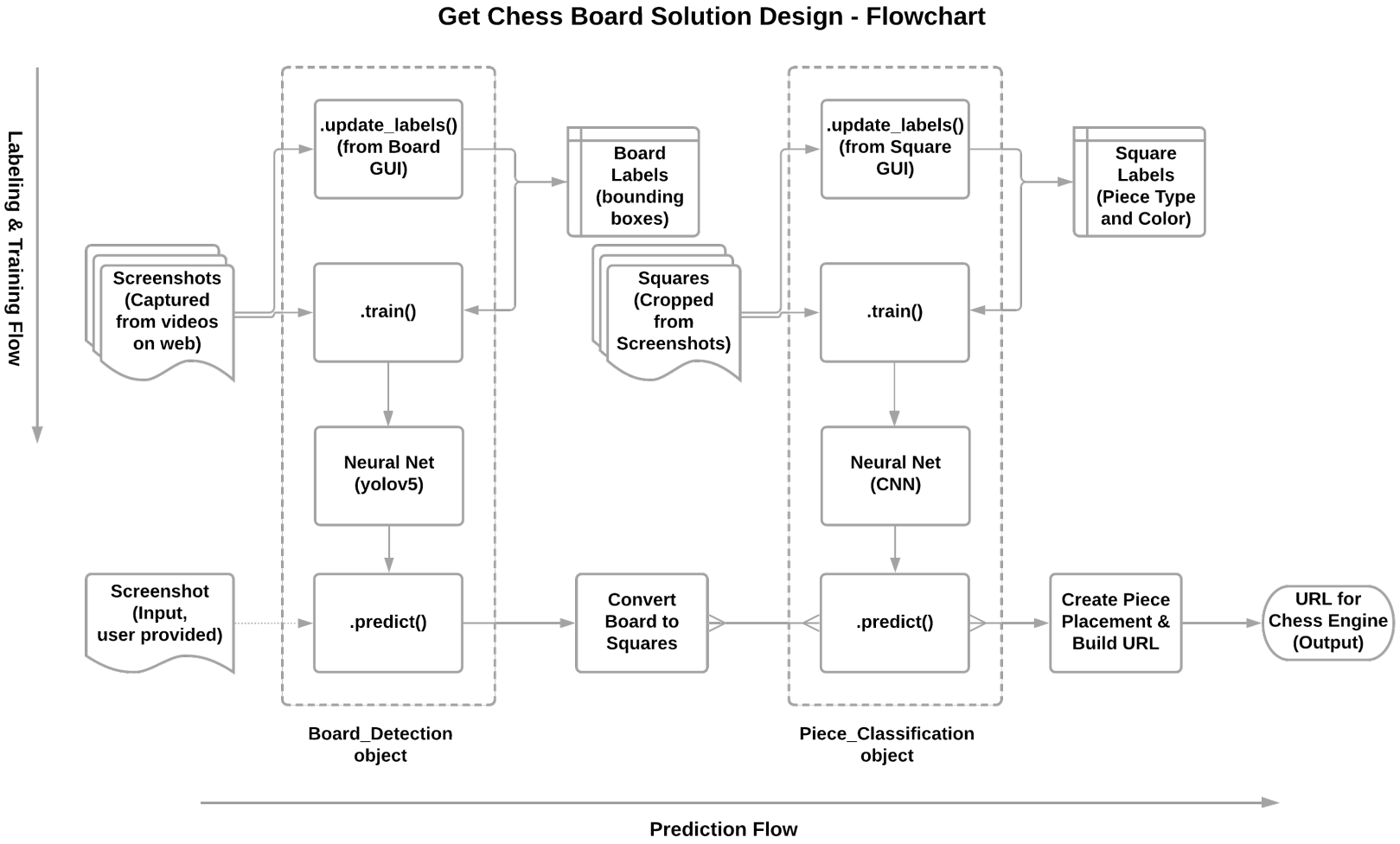
Rest of the report will go over a summary of the methodology, steps taken in the methodology and conclude with a discussion of next steps.

2. Methodology Summary

For this problem, my solution design involved breaking the problem down into manageable and relatively independent sub-problems that could be tackled sequentially. This way, rather than having one problem that could be intractable to debug (screenshot -> placement), I would have smaller problems some of which were relatively more tractable “black-boxes” (screenshot -> board; squares->pieces) and some of which were relatively straightforward manipulations (board -> squares, pieces -> placement).

The methodology follows the flowchart[1] below:

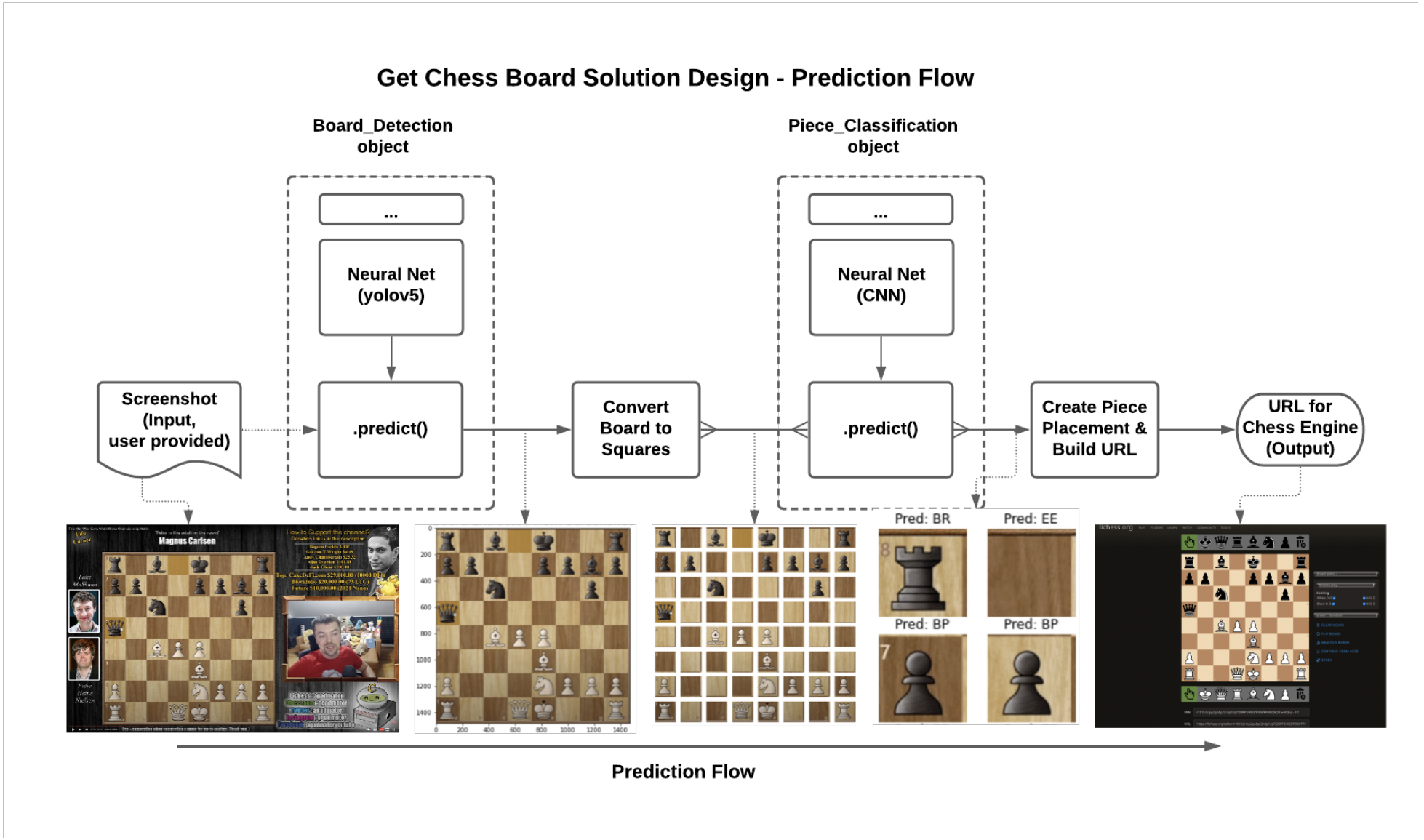
[1] Flowcharts by lucidchart.com



In the above design, Board_Detection and Piece_Classification are two core objects perform detection and classification functions for boards and squares, respectively. Each object has .update_labels(), .train() and .predic() functions that are used to respectively update and correct data labels, train neural networks and make predictions.

On the other hand, "Convert Board to Squares" and "Create Piece Placement" boxes are implemented via a set of utility functions all together called gcb_utils. The module gcb_utils also contains the data labeling GUIs that are precursors to the training stage.

Below is a pictorial representation of the journey of a screenshot on the Prediction Flow axis. A walkthrough can be found in the iPython Notebook: Get Chess Board Part 1.



In the next section, I will go briefly go over the main building blocks of the solution. For each, I will focus on guiding ideas, some design choices, outputs and comment briefly on possible improvements. The report will conclude with some possible next steps for the project in order to better achieve the goals outlined in the first section. The Appendix that follows will outline the directory structure to help navigate the github repo.

3. Methodology Steps

3.0 Data Acquisition & Labeling GUI across Boards and Squares

For this project, after a brief exploration, I decided to gather my own data. Reasons were twofold: First of all, for whatever data was available, it looked like I would have to do the labeling myself. Secondly, if I needed more data (as this is a deep-learning project at its core), I would have to come up with a way to do it myself. Given those, gathering my own data seemed to make the best sense.

As for the GUI development, I decided to take that on myself as well. First reason was that the available labeling programs didn't seem to run on my Apple Silicon machine, yet. Second, I had never done any GUI programming before so I thought it might be a UX capability that I could develop on my own (and tailor the ouput for my own project at hand.) My choice has been tkinter as it comes built-in with Python, it works almost universally on all operating systems and seemed simplest to learn.

3.0.a Data Acquisition & Labeling GUI for Chessboard Detection

The design aims to lessen the cognitive load of the labeler. A natural extension is to keep the labeler away from dealing with the data source (a csv file in this case) as well as the manual handling of file names and bounding box pixels therein.

To this end, the user should be able to:

- Label a chessboard object (set a bounding box) within the GUI with mouse drag
- Label multiple objects
- Delete any labels that are not desired
- Label screenshots in succession
- Move to the next image (or exit) without worrying about whether the work was saved or not. (abstract from data source)

The implementation can be found in gcb_utils/gcb_utils.py. The preferred way of running is via the Object_Detection object. Please see Get Chess Board Part 2a for a more detailed walkthrough.

Various Labeling Screens:

In the first screen, please note that bounding box pixels are all NaN before labeling. Labeling is a click-drag-unclick sequence. The GUI also handles auromatic scaling of the source image (any height/width) to fit into the tkinter window.

In the second screen to the right, please note that bounding box pixels are all NaN before labeling. Labeling is a click-drag-unclick sequence. Note the label “Chessboard” on the upper left hand corner of the bounding box.

Finally, the screen at the bottom, demonstrates that the GUI can handle the insertion and deletion of multiple labels. (deletion not shown)

Notes on Screenshots and Preparing Labels for Training:

Labeling Speed with the GUI:

-With the GUI, I was able to label around 80 screenshots pretty quickly – in approximately 2-3 hours.

-At the time, the labeling process gave me confidence that if I needed more data, I could easily get more.

Information on Screenshots and Screenshot Size Statistics:

-Each screenshot is 2880x1800 (width x height in pixels) in png format captured on a macOS system. The average size is 3.8M (min/max/std: 1.6/7.2/1.3Mb)

Information on Labeling Data Source and Use in yolov5:

-Label data is held in a csv file where each row indicates a chessboard label associated with a file name, image size information and that particular chessboard's bounding box information

-yolov5 requires a yaml file that indicated directories for training and validation images which also asks the user indicate the number and name of classes.

-In addition, yolov5 also requires a specific format for image labels where the bounding boxes are represented as normalized image centers and image height/width pairs. In addition to handling randomization of training and validation images and their labels, the data acquisition code also handles this conversion.

3.0.b Data Acquisition & Labeling GUI for Piece Identification (Squares)

Similar to the chessboard detection GUI, this design also aims to lessen the cognitive load of the labeler. A natural extension is also to abstract away the data source from the labeler and let the interface handle file names and square properties.

In the GUI, main data fields to be filled are:

- Piece Color: EBW for (E)mpty squares and (B)lack, or (W)hite pieces.
- Piece Type: EPRNBQK for (E)mpty squares and pieces of type P(awn), R(ook), k(N)ight, (B)ishop, (Q)ueen or (K)ing
- Human Check: YN for (Y)es or (N)o. This is to indicate that a human has checked the labeling. For a future feature where the Piece_Identification object will do the preliminary labeling and human will complete labeling without many clicks.

To this end, the user should be able to:

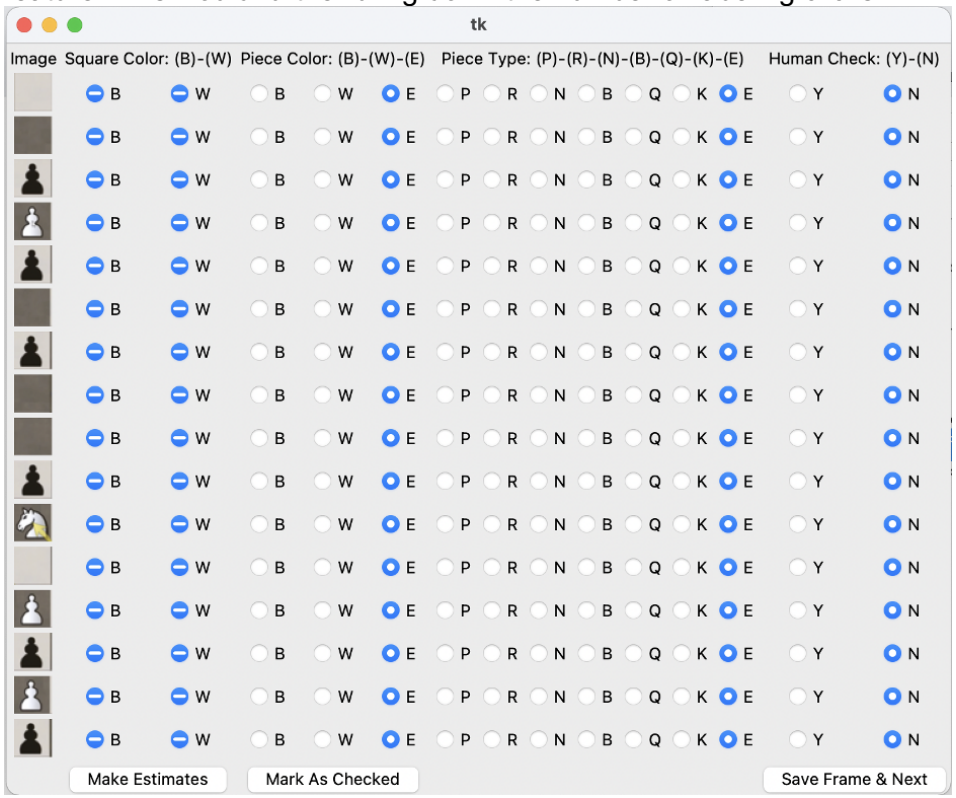
- Label a square object within the GUI with mouse click – radio buttons are used for their toggle property.
- Use minimum clicks
    - The default state of any square is set to E(mpty) for both Piece Color and Piece Type as one can at most have half the squares filled in a chess game.
    - A button is added to confirm "Human Check" wholesale for the screen

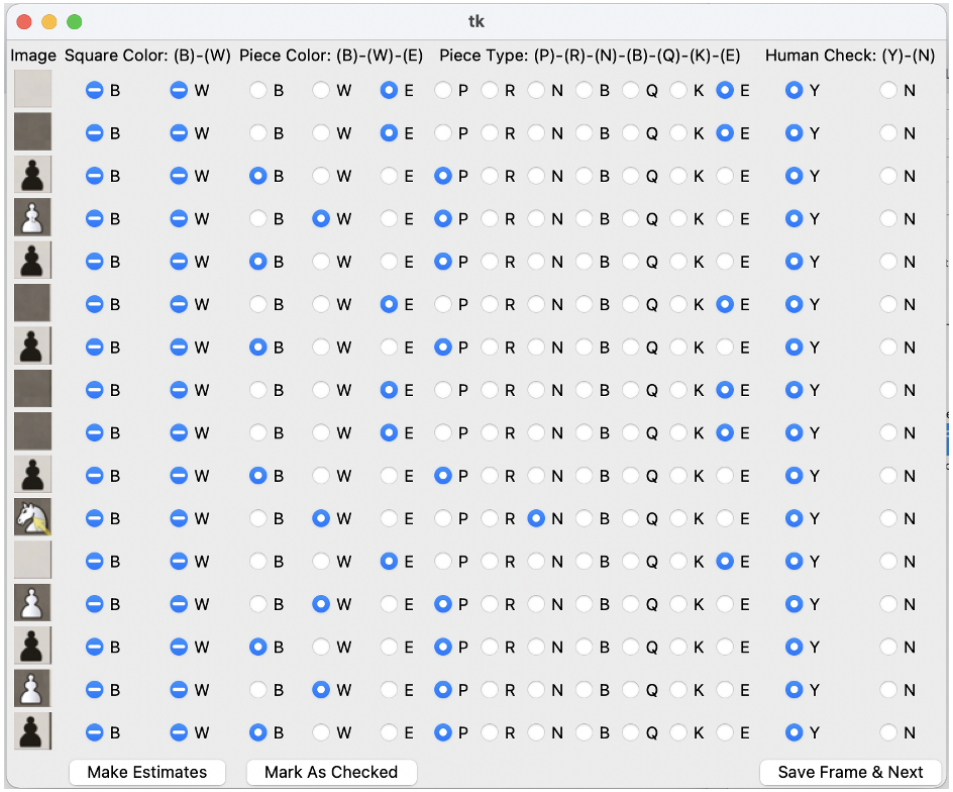
- Label sets of squares in succession
- Move to the next image (or exit) without worrying about whether the work was saved or not.

The implementation can be found in gcb_utils/gcb_utils.py. The preferred way of running is via the Piece_Classification object. Please see Get Chess Board Part 2b for a more detailed walkthrough.

Square Labeling in Pictures:

Please note that the defaults for Piece Color and Type are E(mpty) and Human Check is N(o). The button “Mark as Checked” marks all Human Check: N’s as Y’s. On the average this minimizes the number of clicks. Another improvement would be to activate the “Make Estimates” feature. This would further bring down the number of labeling clicks.





Notes on Square Labeling Performance and Use:

Labeling Speed with the GUI:

-With the GUI, I was able to label around 5600 screenshots in perhaps around 6 hours or so. When creating the squares, I made use of functions from gcb_utils module. It is essentially the “Convert Board to Squares” step applied successively.

A Look at the Square Distribution:

-A breakdown of labeled squares is given below:

```
Count of Pieces by Color and Type – includes E(mpty) squares
```

fname								
PcType-PRNBQKE	**B**	**E**	**K**	**N**	**P**	**Q**	**R**	**All**
PcColor-BWE								
B	115.0	-	87.0	110.0	524.0	64.0	143.0	1043
E	-	3540.0	-	-	-	-	-	3540
W	118.0	-	87.0	114.0	531.0	66.0	143.0	1059
All	233	3540	174	224	1055	130	286	5642

- From the above, we can see that, as expected Empty squares make more than half (62.7%) of the sample creating a bias. In aggregate, next are Pawns (18.7%) followed by Rooks (5.1%), Bishops (4.1%), kNights (4%), Kings (3.1%) and Queens (2.3%).

-However, the "in aggregate" statement above is comes with a potentially big caveat for classification. The labels indicate unbalanced classes as they are and when coupled with the split into black and white pieces, the unbalance will become more pronounced. One silver lining is that the weights of the black and white pieces seems balanced.

3.0.c Retrospective and Possible Improvements

In retrospect, I am pretty happy with how the GUI and its interaction with the data source files have turned out. In addition to executing the “Make Estimates” function in Square labeling, I would also like to filter and display the squares based on Piece Type and Color. I believe that would help checking immensely.

3.1 Board Detection

Board detection is implemented via the Board_Detection class. The class contains a labeler, trainer and a detector for finding chessboards and finding bounding boxes in screenshots. Labeler is, in turn, implemented gcb_utils while trainer and detector are interfaces into PyTorch implementation of yolov5 (https://github.com/ultralytics/yolov5, see https://pjreddie.com/media/files/papers/yolo.pdf for Yolo.) Main methods are update_labels, train and predict.

The core element of Board_Detection is PyTorch implementation of yolov5. I have decided to use this algorithm as it seemed to be fast and relatively small. It also helped that the board detection task was relatively undemanding and generalizable: i.e. the chess board is always a rectangle (closer to a square) and it is always a chequered figure with 8 rows and columns. The choice of PyTorch came from the availability of yolo on PyTorch and my curiosity to also work with the PyTorch environment to what extent that I could.

For a Board_Detection object walkthrough, please refer to the iPython Notebook Get Chess Board Part 3a.

3.1.a Labeling

Please refer to section 3.0.a above for a detailed explanation and the iPython Notebook for a walkthrough.

3.1.b Training

Training Inputs and Parameters:

-The dataset contained 64 screenshots for training and 18 for validation. The screenshots at this stage are 1/4th the size (halved width and height at 1440x900) and converted into jpg's for space considerations.

-The initial pretrained weights are those of the smallest pretrained model provided by yolov5: 'yolov5s.pt' – initially trained on COCO dataset.

-The long side of the image is set to 1440 as per the new size above

-Epochs was set to 1500 out of abundance of caution as the cost of early termination was high.

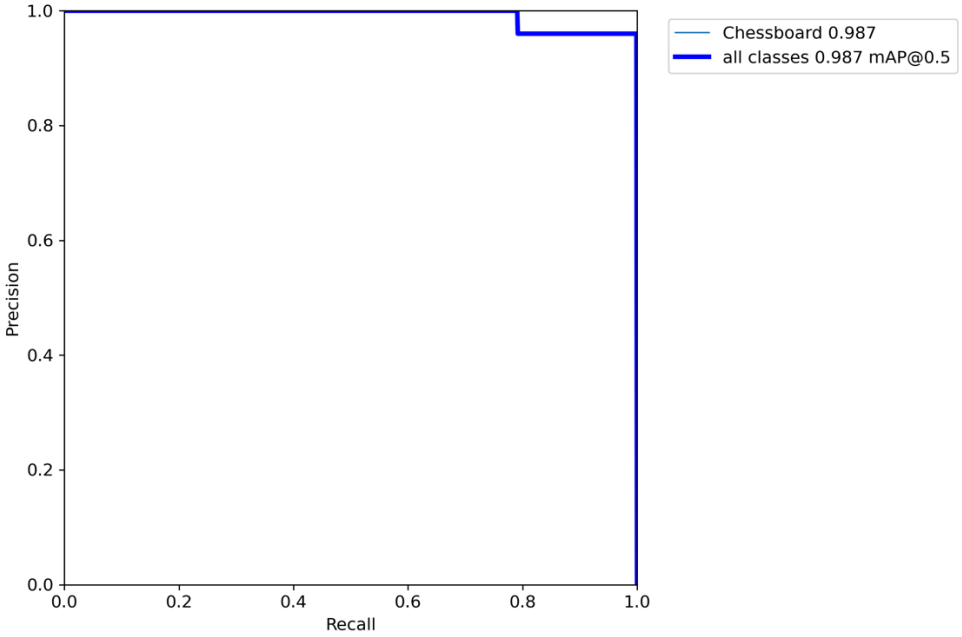
-Other parameters and defaults are referenced in the notebook referenced above.

Results:

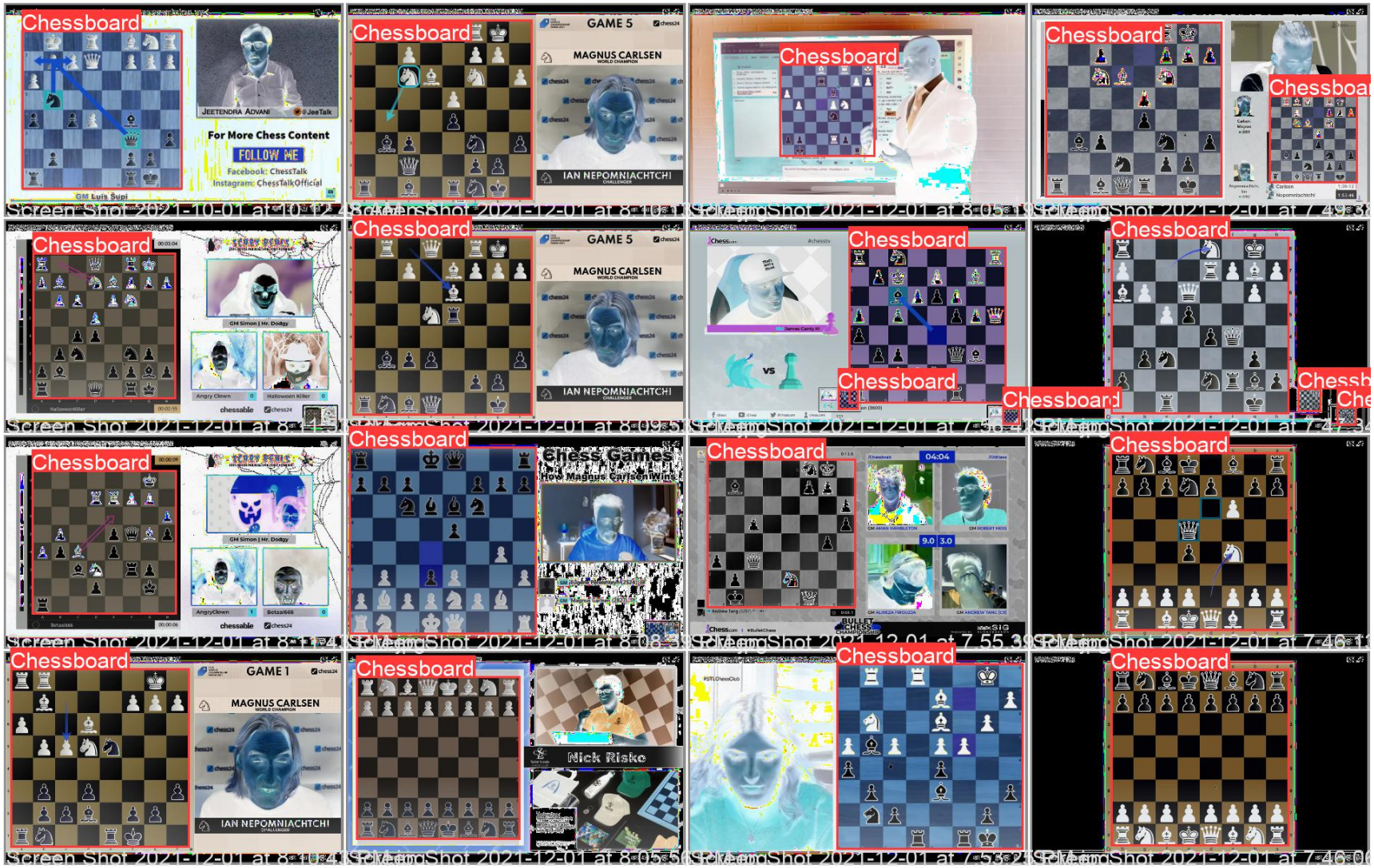
-Each epoch took approximately 4 minues on an Apple MacBook Air (2020) M1 with 16GB of memory. The total runtime for the model was 39 hours where the final epoch was #598 and the best epoch was #498. This early termination (<1,500 epochs) was due to the default 'patience' parameter being hit – where the model did not improve for 100 consecutive epochs. A screenshot of these epochs are below:

```
Results of the experiment (actual best set, took over 39 hours, best epoch 498, end at 599.)
     Epoch   gpu_mem       box       obj       cls    labels  img_size
  498/1499        0G    0.0154  0.006404         0        49      1440: 100%|██████████| 4/4 [04:00<00:00, 60.11s/it]
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100%|██████████| 1/1 [00:05<00:00,  5.33s/it]
                 all         18         24       0.96          1      0.987      0.953
     Epoch   gpu_mem       box       obj       cls    labels  img_size
  598/1499        0G   0.01687  0.004755         0        52      1440: 100%|██████████| 4/4 [03:32<00:00, 53.17s/it]
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100%|██████████| 1/1 [00:05<00:00,  5.32s/it]
                 all         18         24       0.96          1       0.96      0.911
Stopping training early as no improvement observed in last 100 epochs. Best results observed at epoch 498, best model saved as best.pt.
To update EarlyStopping(patience=100) pass a new patience value, i.e. `python train.py --patience 300` or use `--patience 0` to disable EarlyStopping.
599 epochs completed in 39.422 hours.
```

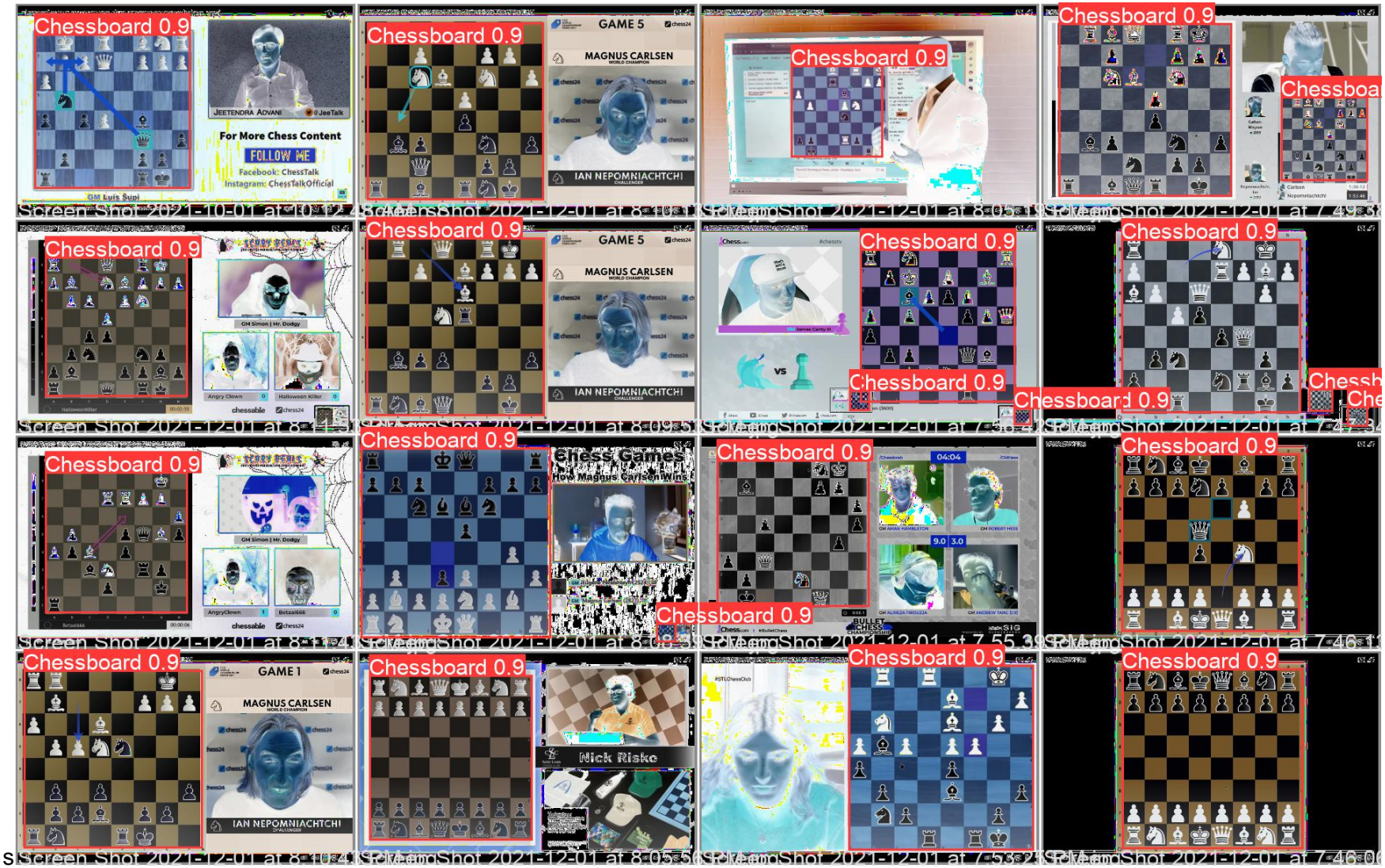
-Over the 18 images used for validation, mean Average Precision (mAP) at an Intersection Over Union (IOU) ratio of 0.5 was 0.987. Over a range of IOU's from 0.5 to 0.95 mAP stood at 0.953. Also, please note that precision is 0.96 and recall is >0.995 in the validation set. The precision-recall graph from yolov5 is presented below.



-Although, there is a chance that model could be overfit due to these high figures, unfortunately I did not have the chance to experiment with the model given the high time cost of training. The results from validation images however, show that the fit is still functional out of the sample for the purpose of detecting chessboards. Images are below: Actual labels are below immediately followed by predictions.



Validation Set – Labels



Validation Set – Predictions

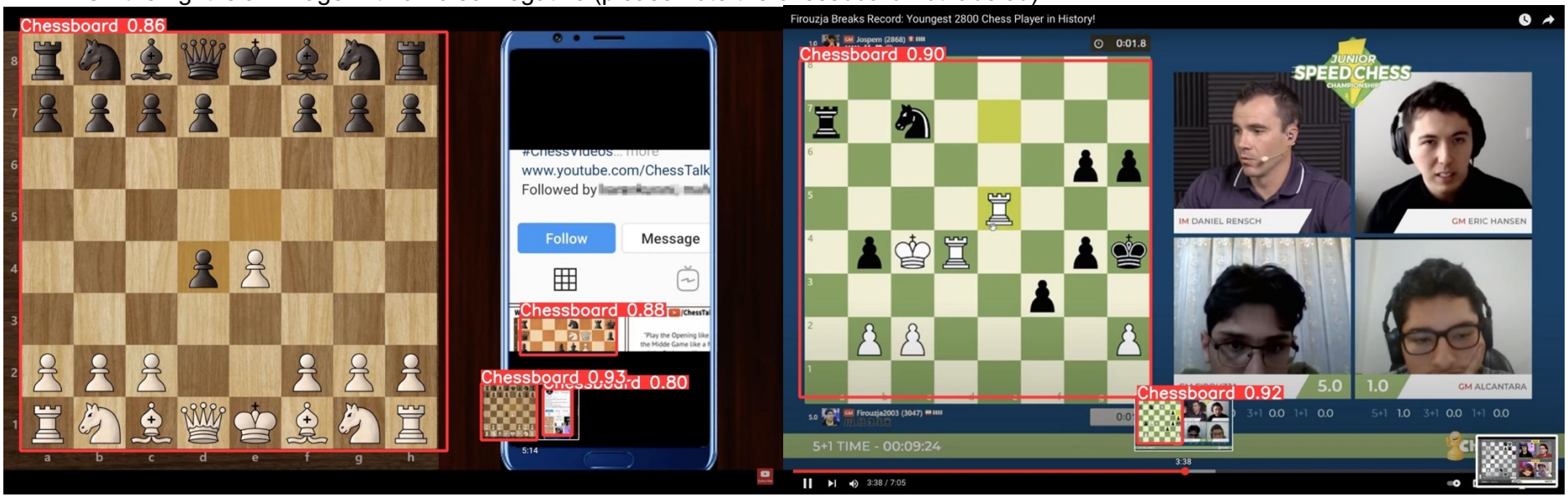
3.1.c Prediction

Let's take a look at the board detection output by yolov5 on images not previously encountered by the neural net. In general, yolov5 does a good job in classification and localization however we still encounter some False Positives and False Negatives:

Correct Classifications:

Incorrect Classifications:

On the left is an image with a False Positive (please note the partial chess board and the smaller rectangle with lower confidence values)
On the right is an image with a False Negative (please note the chessboard not labeled)



3.1.d Retrospective and Possible Improvements

In retrospect, I am happy with how the output of the board detection even though I could only run it once. Furthermore, the recall of the model is good. While the bounding box estimation could use some more work, that may not be the issue to focus on at the deep learning stage. Given some confidence in bounding parameters, I can think of a methodology to push/pull borders to improve the board capture IOU.

I am also happy having chosen to use PyTorch, it was a good exposure to a second deep learning framework.

3.2 Convert Board to Squares

In this step, the output from the Board_Detection prediction will be split into 64 squares and input into the prediction function of Piece_Classification object. The process can be implemented using functions in gcb_utils module. For view of the implementation, please the appropriate section in Get Chess Board Part 1.

One suggestion to improve the square selection could make use of the chequered structure of the chessboard. For each side, the border line (1 pixel wide) could be correlated with similar one-pixed wide lines that are nearby.. For a small region around this border, a change in correlation could indicate a chessboard border.

3.3 Piece Classification

Piece_Classification is implemented via the Piece Classification class. The class contains a labeler, trainer and a detector for classifying squares. Labeler is, in turn, implemented in gcb_utils while trainer and detector are implemented in piece_classification.py.

The core model is a simply connected CNN in Tensorflow with two Convolution&MaxPooling layers followed by two dense layers, the last one being softmax. It is similar to an MNIST classification exercise.

The object's main goal is to classify a square: Is it empty (EE)? If not, what is the piece that it contains (WP, WR, WN, WB, WQ, WK, BP, BR, BN, BB, BQ, BK)? This gives us a classification problem over 13 classes.

The solution itself however has to function/generalize extremely well across several dimensions:

-difference in piece shapes across different chess sets
-imbalance in multiclass data in favor of empty,
-different types of pieces looking alike (e.g. kings and queens),
-chess square colors showing great variation
-improperly captured squares coming in from the board detection step and

The solution also has to get this right for each of the 64 squares. Naively, say, for an (arbitrary) 97% chance of perfect placement prediction, accuracy should be at (0.97)^(1/64) = 0.9995. This calculation alone is somewhat daunting.

This version of the CNN solution seems to generalize well across all of the problems above[2].

3.3.a Labeling

Please refer to section 3.0.b above for a detailed explanation and the referenced iPython Notebook for a walkthrough.

3.3.b Training

Training Inputs and Parameters:

-The dataset contained 4231 squares for training and 1128 for validation. When input into training square cropped from the boards were shrunk down to 80x80 squares

-The model didn't use any initial pretrained weights and trained from scratch.

-The model included a callback function, where for each epoch, the model was saved as "the best" if the choice of metric on the validation metric was so far the best. If validation data wasn't available, the metric of the training data was used instead.

-The final choice of validation metric was "Cohen's Kappa" which has some adjustment for class imbalance. Another metric that I used was "objective function loss".

[2] One issue that I can think of is the color of chess pieces being relative. Imagine two sets: light blue vs blue and blue vs dark blue where blue was labeled white for one set and black in another. Although even this can be solved with this methodology, a better approach could be hierarchical classification: Classify empty vs full. Run a 2-means clustering algorithm across pieces based on the average color of the square center. Classify the 'lightter one as white'. Then run shape classification on the pieces.

-The model ran for 40 epochs, each of which took approximately 5 seconds on an Apple MacBook Air (2020) M1 16 GB machine running macOS.

The best epoch of the best model was as follows:

```
Epoch 00031: val_cohen_kappa improved from 0.97573 to 0.97573, saving model to data/model/piece-train/exp/cnn_pieces.h5
133/133 - 5s - loss: 3.3121e-06 - accuracy: 1.0000 - cohen_kappa: 1.0000 - val_loss: 0.2934 - val_accuracy: 0.9858 - va
l_cohen_kappa: 0.9757 - 5s/epoch - 40ms/step
```

-Please note that the Cohen's Kappa stand at 1 for the training set while the Cohen's Kappa on the validation set is 0.9757. It is still possible that the model is overfit. It looks like I could use a callback function that compared metrics across both training and validation sets rather than looking at only one of them.

Training Analysis:
Below are confusion matrix of the training set and a sample of correct classifications demonstrating the variety in the data.

Confusion Matrix for the Training Set – _pred stands for model prediction

	EE	WP	WR	WN	WB	WQ	WK	BP	BR	BN	BB	BQ	BK
EE_pred	2658	0	0	0	0	0	0	0	0	0	0	0	0
WP_pred	0	389	0	0	0	0	0	0	0	0	0	0	0
WR_pred	0	0	108	0	0	0	0	0	0	0	0	0	0
WN_pred	0	0	0	83	0	0	0	0	0	0	0	0	0
WB_pred	0	0	0	0	91	0	0	0	0	0	0	0	0
WQ_pred	0	0	0	0	0	50	0	0	0	0	0	0	0
WK_pred	0	0	0	0	0	0	70	0	0	0	0	0	0
BP_pred	0	0	0	0	0	0	0	392	0	0	0	0	0
BR_pred	0	0	0	0	0	0	0	0	106	0	0	0	0
BN_pred	0	0	0	0	0	0	0	0	0	80	0	0	0
BB_pred	0	0	0	0	0	0	0	0	0	0	91	0	0
BQ_pred	0	0	0	0	0	0	0	0	0	0	0	50	0
BK_pred	0	0	0	0	0	0	0	0	0	0	0	0	63

400 Randomly Chosen Correct Classifications [Training Set - 4231 Images]

For correct classifications, most of the data seems to come from “clean cut” squares and seeping from the sides do not seem to make a difference. The model also seems to be robust to the color of the chessboard square.

For the final model, I trained another model that used all of the available squares without leaving any for validation. The idea was that benefit of more data could outweigh costs of the overfit. At this point, I am not sure if this was actually the correct trade-off decision as there are many models that have a Cohen’s Kappa of 1 and choosing the first one that was achieved might have been inferior.

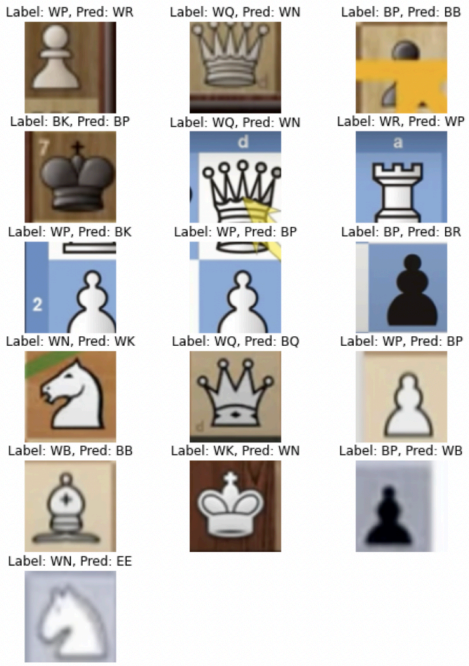
3.3.b Prediction

Prediction will take using the above model recreating the validation set results in the training section. Confusion matrix and the errors are as follows:

Confusion Matrix for the Validation Set – _pred stands for model prediction

	EE	WP	WR	WN	WB	WQ	WK	BP	BR	BN	BB	BQ	BK
EE_pred	707	0	0	1	0	0	0	0	0	0	0	0	0
WP_pred	0	109	1	0	0	0	0	0	0	0	0	0	0
WR_pred	0	1	28	0	0	0	0	0	0	0	0	0	0
WN_pred	0	0	0	24	0	2	1	0	0	0	0	0	0
WB_pred	0	0	0	0	19	0	0	1	0	0	0	0	0
WQ_pred	0	0	0	0	0	10	0	0	0	0	0	0	0
WK_pred	0	0	0	1	0	0	13	0	0	0	0	0	0
BP_pred	0	2	0	0	0	0	0	103	0	0	0	0	1
BR_pred	0	0	0	0	0	0	0	1	27	0	0	0	0
BN_pred	0	0	0	0	0	0	0	0	0	22	0	0	0
BB_pred	0	0	0	0	1	0	0	1	0	0	20	0	0
BQ_pred	0	0	0	0	0	1	0	0	0	0	0	11	0
BK_pred	0	1	0	0	0	0	0	0	0	0	0	0	19

All Incorrect Classifications [Validation Set - 1128 Images]



-Recall error rates seem to be for white pieces. In particular, they tend to be highest for white Queens and lkjljl kNights.
-Imprecise cuts on the blue board seems to impact the result, perhaps for white pieces.

3.3.d Retrospective and Possible Improvements

In retrospect, I am again happy with the performance of the classifier although there seems to be some indication of overfit. At this stage, any improvement to the validation set would probably come from a better extraction of squares. More square data, augmentation and better board prediction modeling could "remedy" this. However another low-cost approach could be the methodology discussed in Section 3.2. One other approach to the better square extraction problem is to intentionally train for it: If one chose to train on a larger square that included some portion of all bounding chess squares (there'd be overlap), then we could side-step the square extraction problem somewhat without introducing too much model/mental complexity

Another approach for a better Cohen's Kappa could be via using a smaller kernel size or a higher number of filters in the first convolution layer. I didn't have the opportunity to explore these changes to the architecture although, I have coded in the ability to do that for kernel size.

Finally, I could probably devise a better callback function to save the model considered a balance of validation training set metric difference (overfit) and sum (high average metric value).

3.4 Create Piece Placement & Build URL

3.4.a Create Piece Placement and FEN

In this step, we are going to use the prediction data to create a piece placement string and embed that in an FEN - Forsyth-Edwards Notation. (https://en.wikipedia.org/wiki/Forsyth–Edwards_Notation; https://www.chessclub.com/help/PGN-spec - see 16.1: FEN)

An FEN is a 'space-separated' single ASCII text line that defines the state of a chess game. It is made of six text blocks that represent: piece placement (a8->h1, grouped by rows); active color; castling availability; en passant target square; half-move clock; full move number.

As any FEN component other than the piece placement will require some analysis of history, it makes sense to move forward only with the FEN for now.

For piece placement, the analysis will also assume white is always playing from the bottom of the screen. This is a limiting assumption but not a terrible one at this stage. A suggestion to improve this aspect may lie in a few directions: looking at the placement of black&white pieces and running a binary classification based on a history of games; attempting to pick up a move history and deduce direction, or getting (the/another?) neural network to recognize the letters and numbers on/around the chess board. Also, for a left to right game, one can always make sure to rotate the board such that a black square is on the bottom-left and reduce to problem to one we could already solve.

Following shows piece placement transferred from piece classifier predictions first into text and later into an FEN piece placement string.

```
r.b.k..r
pp..ppbp
..n...p.
q.......
..BPP...
....B...
P...NPPP
R..QK..R
```

FEN Piece Placement String:

r1b1k2r/pp2ppbp/2n3p1/q7/2BPP3/4B3/P3NPPP/R2QK2R

Full FEN string making use of the six FEN component liberally (once on the website, it should be fairly easy to fix any errors)

r1b1k2r/pp2ppbp/2n3p1/q7/2BPP3/4B3/P3NPPP/R2QK2R w KQkq - 0 1

3.4.b Build URL

This is the final step for this project. Using Lichess API with some imitation, I've modified the FEN string into a form accepted by lichess. In the future this section can be abstracted and a variety of transformations can be written for various online chess engine APIs.

The Lichess URL is as follows:

https://lichess.org/editor?fen=r1b1k2r%2Fpp2ppbp%2F2n3p1%2Fq7%2F2BPP3%2F4B3%2FP3NPPP%2FR2QK2R+w+KQkq+-+0+1

4. Conclusion and Next Steps

I believe that in its current form the project has achieved what it has set out to do: Take a screenshot of a chess game played on a video platform an d return string that represents the piece placement.

At this point, some next steps that will move the project closer to its original objective of: “decreasing the friction to study a chess game played on a video platform” could come in form of either add-ons or model improvements.

In terms of add-ons, following come to mind:
-Write a new class that wraps the end-to-end process and outputs the relevant string/link for the desired chess engine.
-Make model available on Docker for widespread use independent of platform.
-Deploy on the cloud (Heroku?) for even wider adoption.

In terms of model improvements, following are top of mind (some mentioned in text above)
-Create experiments to judge the relative impact of board detection and piece classification for prioritizing model improvement
-Test to see if piece classification can be improved by a hierarchical classification
-Generate more training and validation data to compare data’s impact on models’ outcomes
-Test to see if a model could be created to estimate the direction of play