Capstone Report

Convolution Neural Network Chess Board Recognition

Daniel Cuomo

11/10/2018

1. Definition

1.1. Project Overview

As computer vision techniques and camera technology have advanced in the previous years, the application of image recognition to completive games has grown. Chess, a game in which players constantly study to improve based on positions, is an area which processing of real time image data can help to provide new avenues for players to improve. Recognition of chess piece via image and game tracking have been explored but a good portion rely on a known start position.[1][2][3]

In this project a chess position image recognition application was created, using a convolution neural network classifier, to output a digital representation of the chess board. For this application a dataset of 1893 chess piece images were collected using the Staunton Tournament Chess pieces with WE Games Tournament Roll Up Chess Board. These were chosen as they are a commonly used set in US Chess clubs.

1.2. Problem Statement

The objective of this project is to perform the translation from an image representation of a chess board to a digital representation. This project will take an input chess board, provide the appropriate image pre-processing, divide the board into individual squares and using a CNN provide a best estimate for the piece in each square and then output a digital representation of the board. With this key work done it could then be translated to tracking live games on boards and extracting key moments to be fed into engines for positional analysis.

1.3. Metrics

Two metrics will be used for model evaluation:

1. Log Loss: -log P(Ytrue| Ypred) = -(Ytrue log(Ypred) + (1 - Ytrue) log(1 - Ypred)) [4]
2. Accuracy = True Positive + True Negative / Total[5]

Log Loss, also referred to as categorical cross entropy, is important to this problem as it is a multi-class problem. Log loss defines the loss function to heavily penalize high prediction probabilities for wrong classes and reward high prediction probabilities for the correct class. This will help the algorithm to attempt to guess the correct piece as opposed to the most prevalent piece in the dataset. Accuracy

[1] (Meyer, n.d.)
[2] (Danner & Kafafy, n.d.)
[3] (Xie, Tang, & Hoff, n.d.)
[4] (Sklearn Metrics Log Loss, 2018)
[5] (Keras metrics, n.d.)

while less important is still valuable as if the model is not comparable overall to human performance it is not desirable, it is also an easy to understand metric to display during training to track progress.

2. Definition

2.1. Data Exploration

The dataset was manually collected through use of a top mounted Google Pixel 2 camera. The process consisted of randomly placing a set of chess pieces on the board, taking an image, then splitting the image into individual squares. (The preprocessing and splitting will be discussed in more detail in the algorithms section) The individual piece images were converted to black and white as it was not necessary for color in this application. The total dataset consisted of 1893 chess piece images of 135x135 pixels. In Figure 1 is an example board image pre-split and in Figure 2 are a few example individual piece images.

Figure 1 Example Board Image

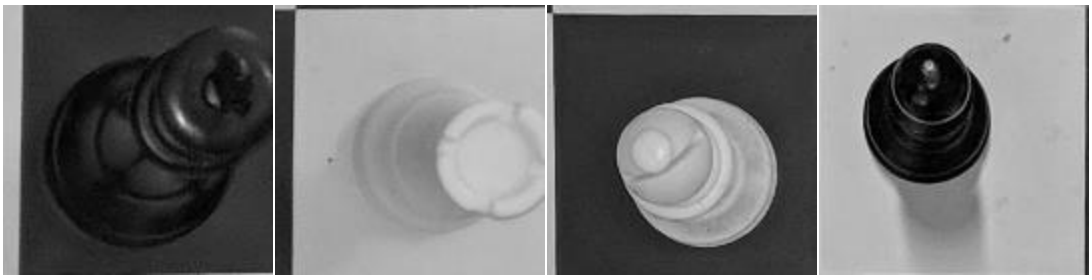


Figure 2 Example Piece Data Images

The same data set was grouped in two different ways to support the neural network architecture which will be discussed later. These two groupings were piece type and color of piece. In both scenarios empty was just considered square regardless of color, identifying the color of square is not important as the color of squares on a chessboard is fixed by position.

There was some significant class imbalance when it came to empty squares due to the nature of the data collection processed, because of this it was reduced to only 300 images in total as it outnumbered the rest of the dataset. Identifying an empty square should not be challenging for a neural network so eliminating a significant portion of this data was not viewed as detrimental.

The data was than split into groups of similar size for training, test and validation. After the split the data was reviewed, and mislabeled data was removed from the set. Figure 4 shows this breakdown by piece and color of the final dataset. There was still some minor class imbalance when comparing king and queens to other piece types, but given the total was ~9% for each was ~16% for the other classes no additional data was collected, and the goal was to use image augmentation during training to overcome the gap.

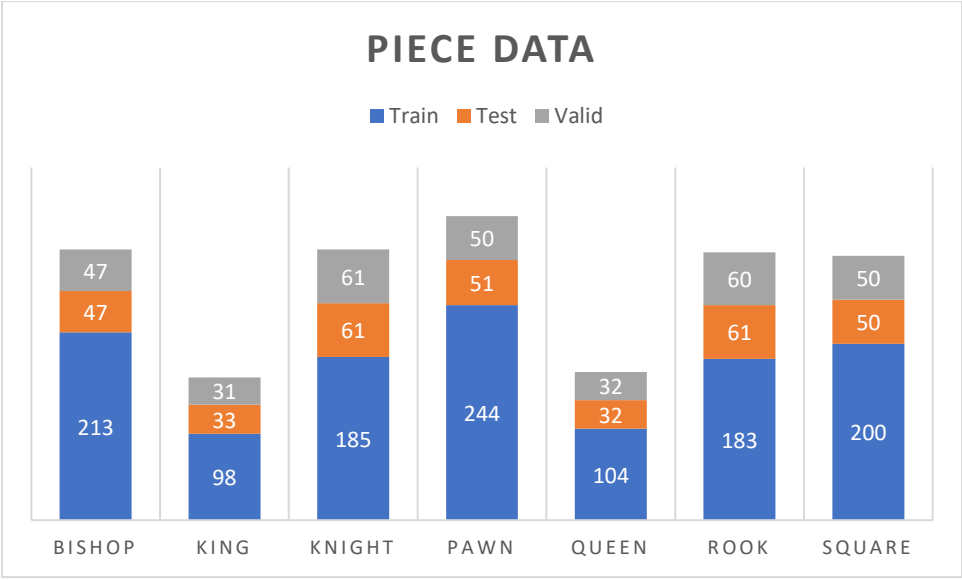


Figure 3 Piece data qty in dataset

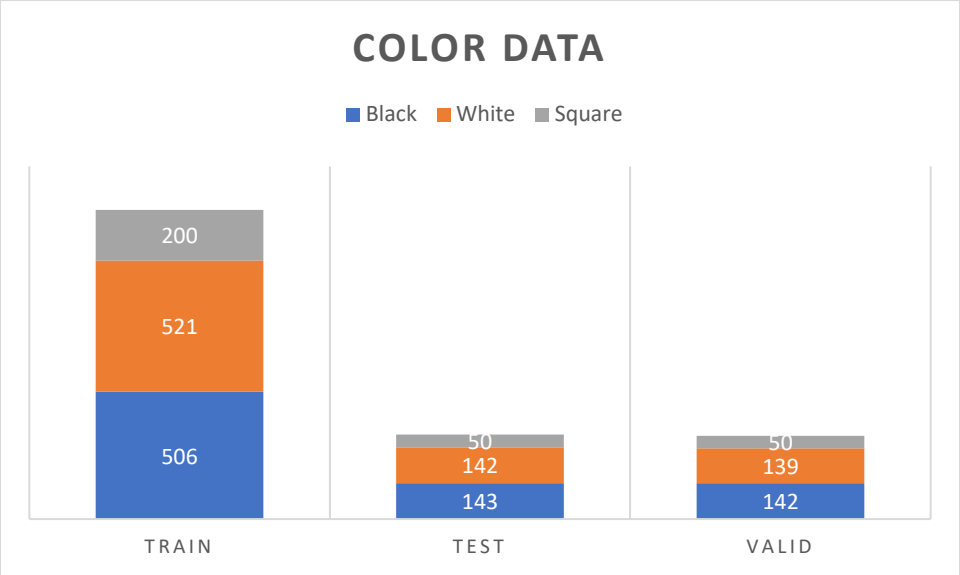


Figure 4 Color data qty in dataset

Two important things were noted when reviewing training data that could pose potential problems for the model. One issue was when pieces were positioned close to the edge of the square the top of the pieces were cut off, shown in Figure 5. This potentially posed an interesting problem for king's and queens and the cross and crown respectively are their unique identifiers. Another issue came for when a neighboring piece was on the edge of the square. In some cases, two pieces are shown in a picture, shown in Figure 6. This was clearly an issue with the data collection methods, however based on the camera position these were considered valid cases the model would need to be able to account for, and while few occurred, they were not removed from the data set. The goal would be to identify the piece in the majority square in the image.

Figure 5 Edge Feature Loss

Figure 6 Double Piece

2.2. Algorithms & Techniques

The classification algorithm consisted of two different convolutional neural networks, one responsible for piece identification and the other for color identification. The classifier would then take the output of the two to determine the piece. The decision to use two neural networks as opposed to one was an attempt to maximize training data to identify key features. This meant one CNN could focus on deciphering the difference between a king and queen regardless of color, and another would focus on color.

A simple CNN architecture using Keras[6] as shown in Figure 7 was considered to start with. This consisted of three sets of Convolution, Pooling and Batch Normalizations. After that a single Global Average polling layer which was connected to the output layer with outputs either corresponding to 7 for the piece and 3 for the color networks. During the development the following aspects of the neural network architecture were tuned:

- Total Number of Layers: Increased layer adds more neurons for learning
- Layer Types and Parameters
    - 2D Convolution Layers – Specific focus to filter number of filters and filter size to extract different key features.
    - Batch Normalization – To improve performance and reduce overfitting
    - Dropout Layers – To attempt to minimize overfitting by focusing on a specific area of the neural network

Another area in which tuning was focused was during the training process for the neural network. Key hyperparameters were adjusted during the process to minimize overfitting and come to the optimal solution:

- Model Learning Rate
- Total Number of Epochs
- Batch Size

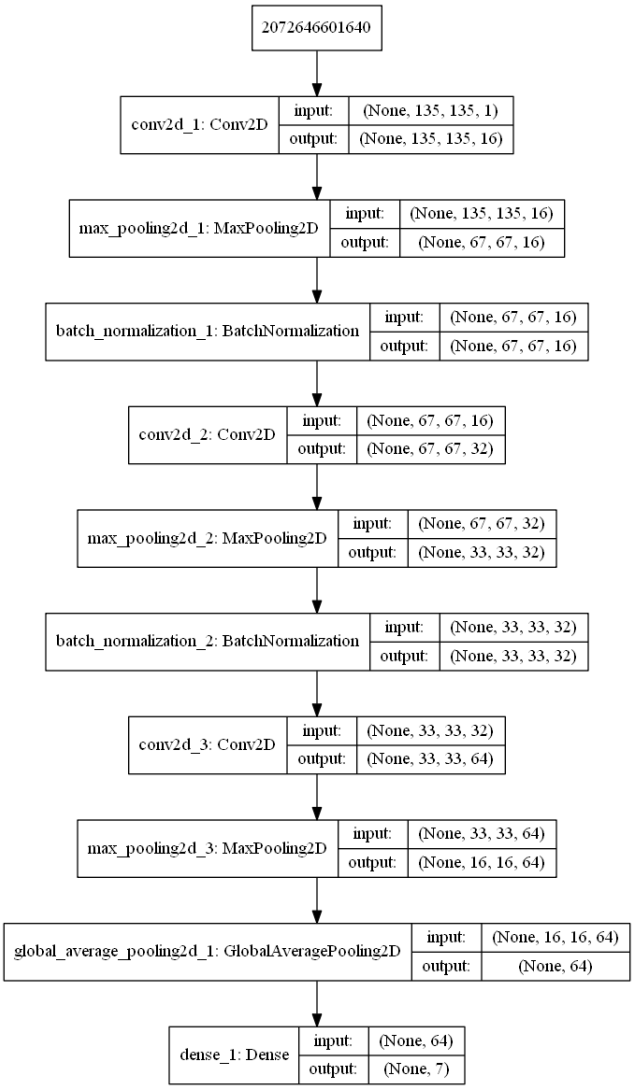


Figure 7 Baseline CNN Architecture

[6] (Keras Documentation, 2018)

2.3. Benchmark Model

The benchmark model had the CNN architecture shown in Figure 7 and was trained against the dataset for 100 epochs, with the RMSProp optimizer, batch size of 32, a learning rate of 0.01 and no image augmentation. Below are the results for the color and piece baseline model:

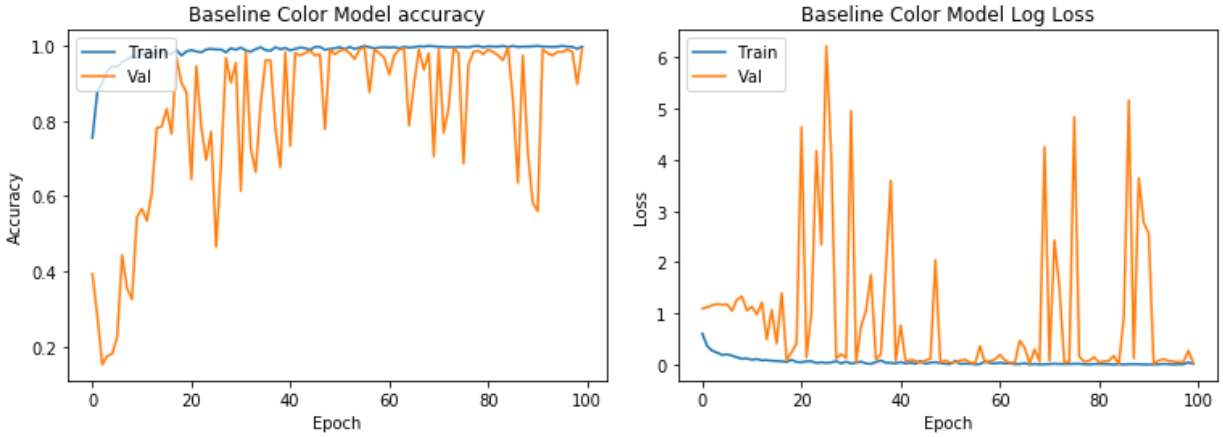


Figure 8 Baseline Color Accuracy and Log Loss

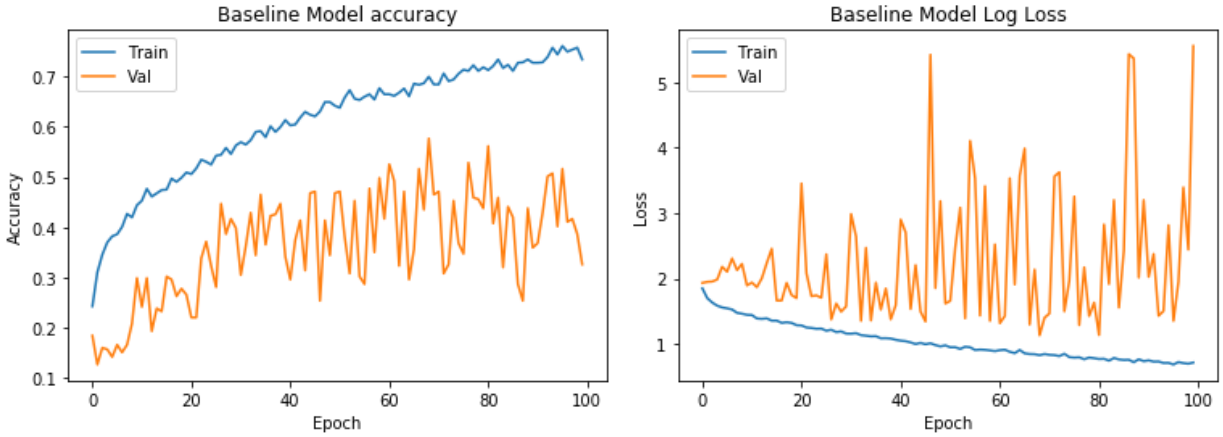


Figure 9 Baseline Piece Model Accuracy and Log Loss

Table 1 Baseline Model Test Results

	Accuracy Test Result	Log Loss Test Result
Color Model	100%	0.005
Piece Model	51.3%	1.23

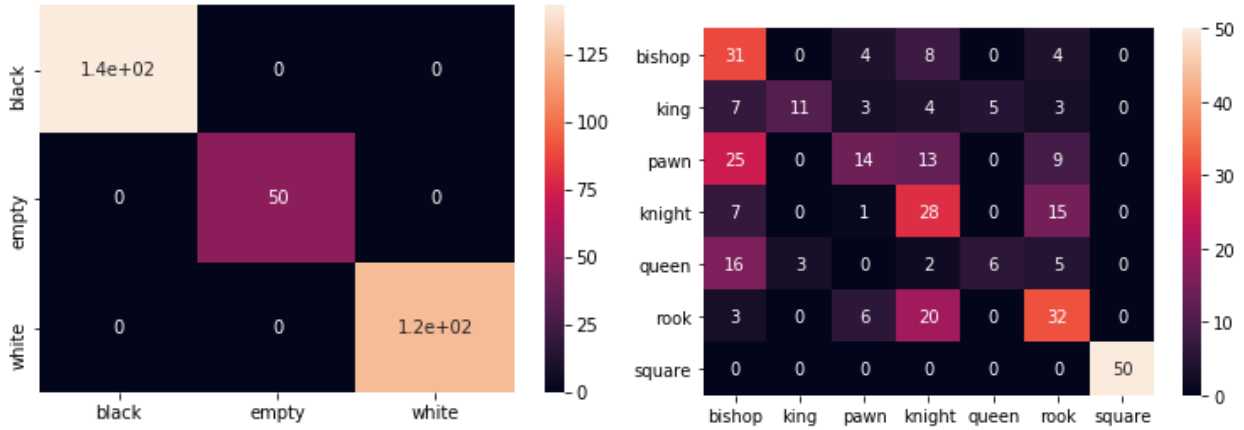


Figure 10 Baseline Model Confusion Matrices

One thing that is clear is in both cases the learning rate is too high, there is evidence of overfitting and no stability in the validation set metrics. The color model was able to find an optimal solution, suggesting maybe minor tweaking of the learning rate will be enough, but the piece model did not perform well. Both log loss and accuracy indicate poor results and when examining the confusion matrix, the only class that had perfect results was the empty square. Moving forward in this report we will focus on the piece model and any architecture decisions made will carry to the color model.

3. Methodology

Figure 11 shows a quick overview of the three main classes used throughout the project, this section will cover the detailed implementation for each section.

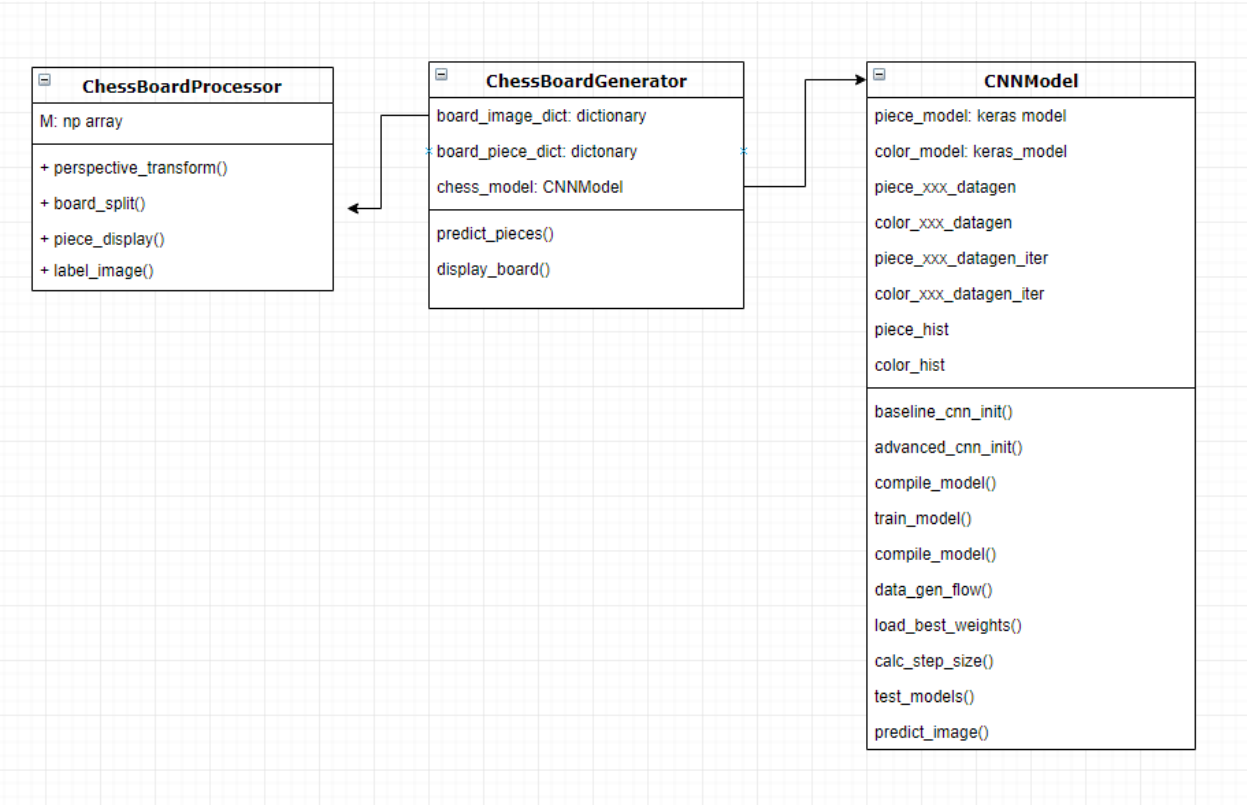


Figure 11 Software Architecture

3.1. Data Preprocessing

The image preprocessing was handled by the ChessBoardProcessor object. This preprocessing can be discussed in two separate but similar activities. The first, using an empty board to find the perspective transform for all boards. The second, applying the previously found perspective transform and slicing populated boards for pieces. For this application to work it required every time the camera or board was adjusted for the first step to be repeated, this was to ensure the chessboard grid could be identified. Identifying the chessboard grid while pieces were present proved to be challenging and was considered out of scope for this project.

The perspective transform calculation leverages the OpenCV[7] (Computer Vision) library and consists of the following steps:

1. Importing grayscale board image
2. Use OpenCV findChessBoardCorners to get chess board inner corners
3. Perform linear regression on inner corner diagonals to find outer corners
4. Calculate a geometric transform to take the outer corners and translate to a 1080x1080 image. Transformation array is stored to class variable M.

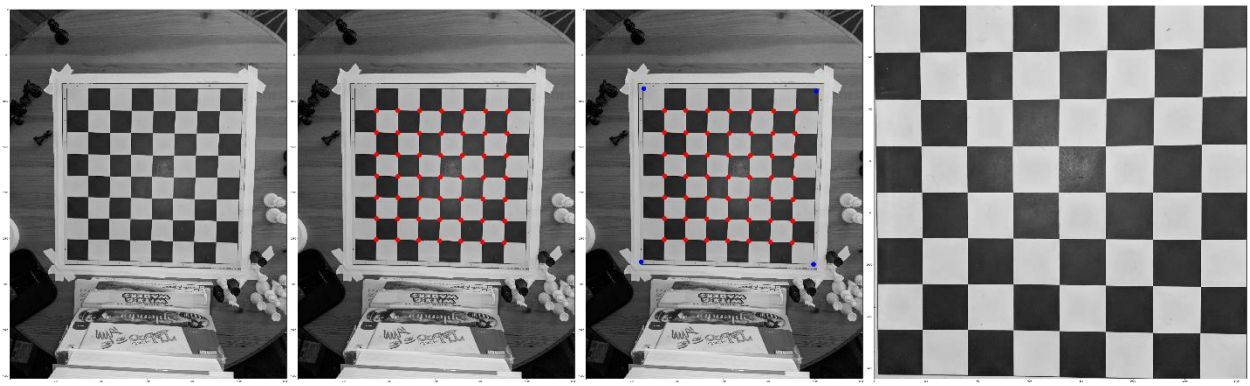


Figure 12 Visualization of Perspective Transform

Once the perspective transform is calculated off the baseline image it is possible to apply this to other images taken with the same camera position. Then once transformed they can be sliced. The steps are as follow:

1. Import board image with same camera position.
2. Apply perspective transform using cv2.warpPerspective
3. Slice image into 135x135 squares and store square images in dictionary.

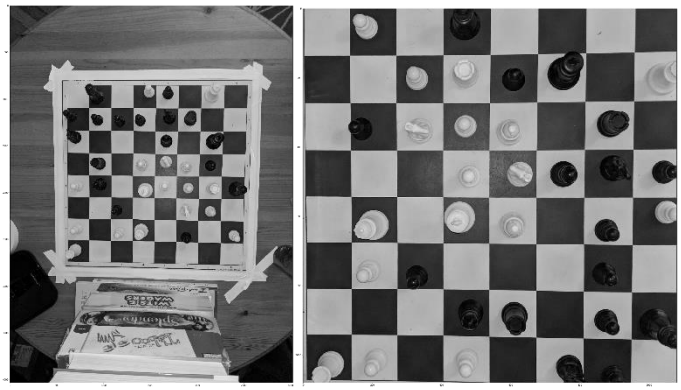


Figure 13 Board with pieces transformation

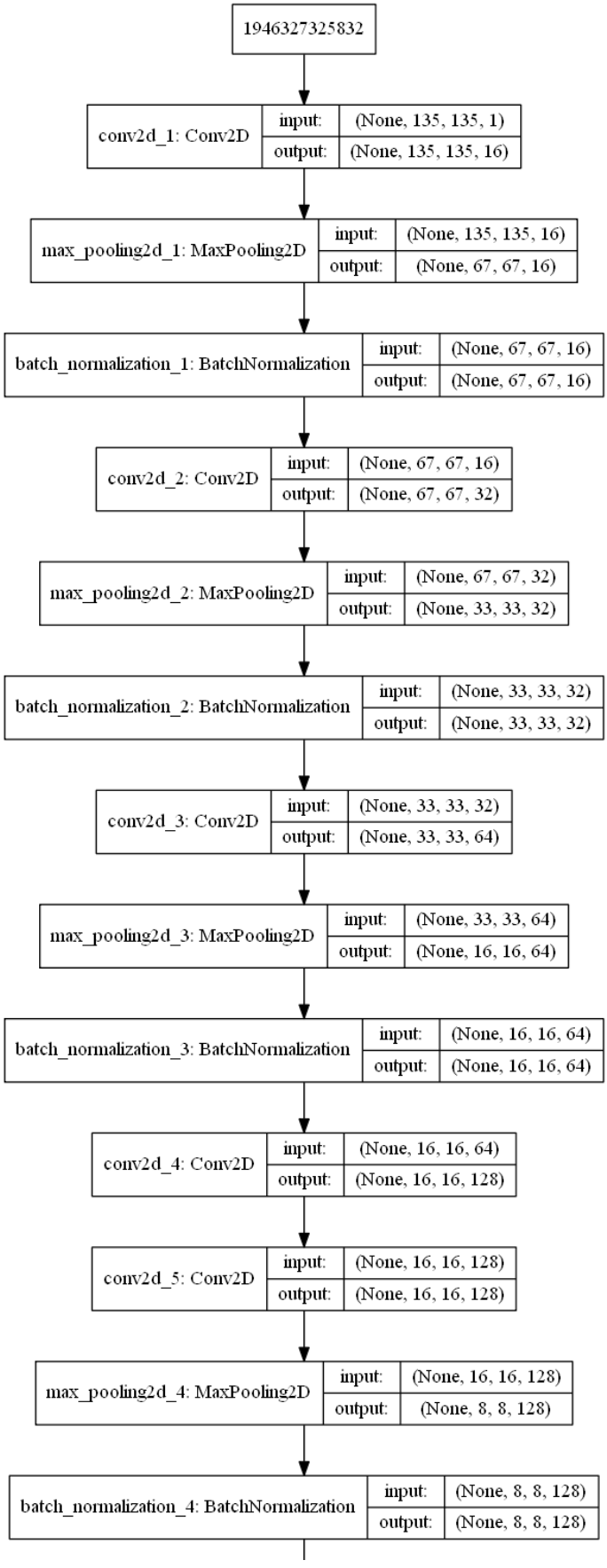
[7] (Open CV, 2018)

3.2. CNN Model Implementation

The CNNModel object contains the details with regards to the models, training, testing and prediction methods. These CNNModel was built off the keras library which takes advantage of TensorFlow as the backend for the deep learning algorithms. The CNNModel training and testing consisted of the following steps:

1. Initialization of the CNN model architecture.
2. Load training, testing and validation data into keras ImageDataGenerator, specific scaling and augmentation parameters for training data.
    a. Note: Keras ImageDataGenerator was used to allow augmentation to be easily added if needed and batch fitting.
3. Specify model hypermeters such as optimizer, learning rate, epochs, batch size etc.
4. Specify model loss functions and metrics. (Categorical Loss, Accuracy etc.)
5. Train network and store metrics to memory while saving best weights during process.
6. If the test results are satisfactory the process is complete, if not repeat tweaking model parameters and augmentation parameters.

The final chosen architecture is shown in Figure 14.



3.3. Chess Board Predictor

The final prediction of chess board pieces as carried out across two classes. The actual prediction function is defined in the CNNModel,

but the ChessBoardGenerator would execute the prediction using a passed in CNNModel objection. However, for this to function it would be wrapped in the main program flow as follows:

1. ChessBoardProcessor object created with baseline board image path passed in on initialization.
2. Test board loaded and split using ChessBoard Processor.
3. CNNModel object created
4. CNNModel object compiled
5. CNNModel either trained or pre-trained weights loaded depending on program flow.
6. ChessBoardGenerator objected created with split test board and CNNModel passed in on initialization.
7. ChessBoardGenerator predict pieces function call, which will predict color and piece type for each square using CNNModel.
8. ChessBoardGenerator output display run to display predicted board. Example output shown in Figure 17.

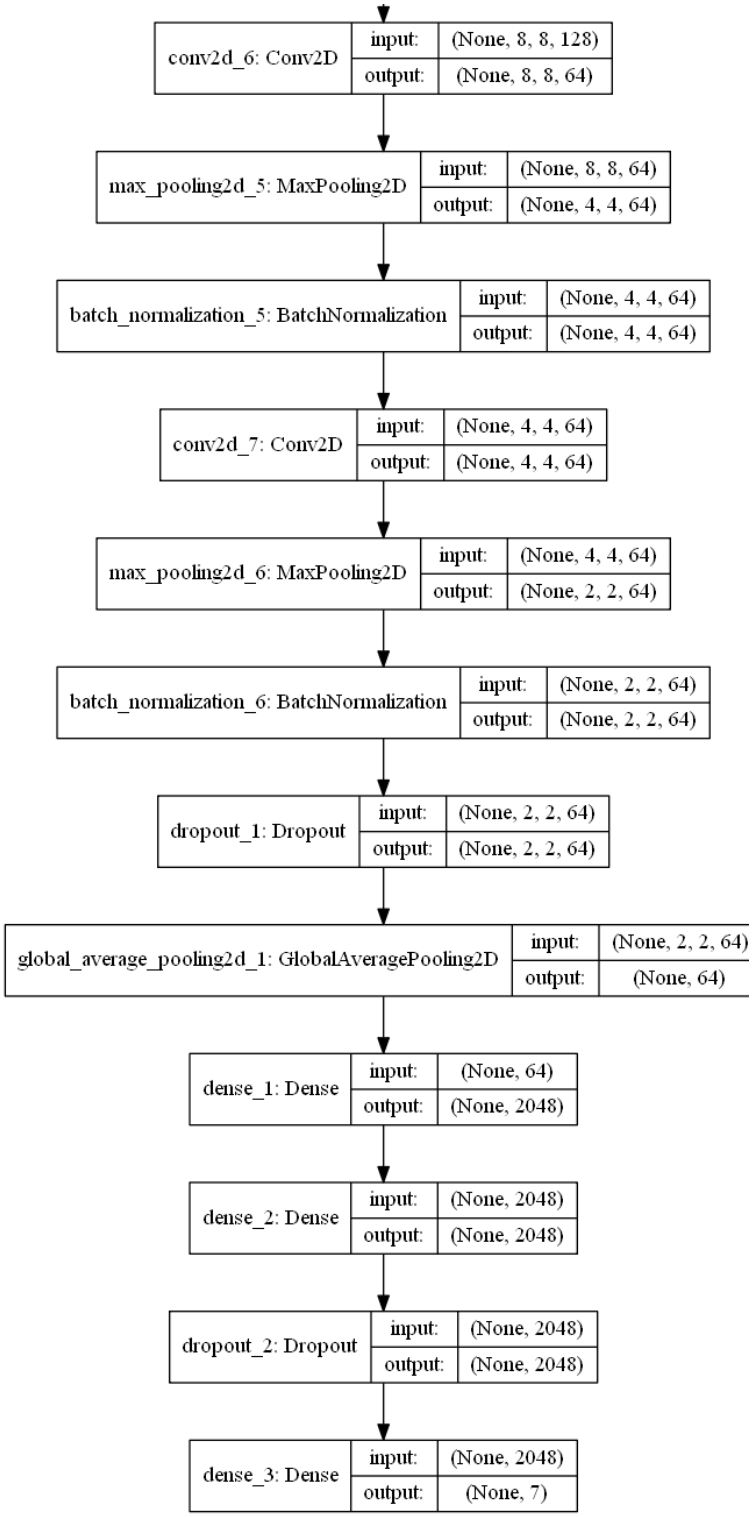


Figure 14 Final CNN Architecture

3.4. Refinement

Throughout the training process three important aspects were identified and tweaked that had large impacts on performance:

1. Learning Rate: It was noted across a variety of optimizers that learning rate had a huge impact on finding a reliable solution. As seen in the baseline model overfitting was prevalent with default learning rates, and they were reduced to prevent this.
2. Modifying and Adding CNN Layers: Additional convolution layers were added in attempt to pick out more features. Another technique that seemed to have good success was adjusting the filter size in earlier layers. By increasing the convolutional filter size, the idea was to identify larger features that could help distinguish the chess pieces.

3. Additional Dense Layers: The initial baseline took the output from the global average pooling layer and connected it directly to the output neurons. Tweaking the amount of additional dense layers and size provided some slight performance increase due to increased nodes.
4. Image Augmentation: Augmentation of the training data provided the largest boost in performance. Accuracy greater than 70% and log loss of less than 1 were not achieved without augmentation. It is possible that some of the complication in the final model could be removed when augmentation was added. Augmentation was important to help overcome the small amount of data and consider the many orientations a piece could have in a single square. This was only necessary for the piece model.

4. Model Evaluation

The final model as shown in Figure 14 consisted of the following:

- Three groupings of 2D Convolution, Max Pooling 2D and Batch Normalization layers of increase filters and decreasing kernel size from 5 to 2 with relu activation functions at the beginning,
- Back to back 2D Convolution layers of 128 filters of kernel size 2 with relu activation layers, followed by max pooling and Batch Normalization,
- Another two groupings of the same early pattern with the same number of filters and kernel size.
- A Global Average Pooling layer to minimize the number of parameters at the output of the convolution section of the neural network to reduce overfitting
- Two dense layers.
- A dropout layer to reduce overfitting
- Final dense layer with softmax activation to give a probability an image is in a certain class. (Note: 7 output layers for the piece model and 3 output layers for the color model)

The final model weights were obtained in a training run with the following hyperparameters:

- Optimizer: RMSProp
- Learning Rate: 0.00001
- Epochs: 1500
- Batch Size: 32

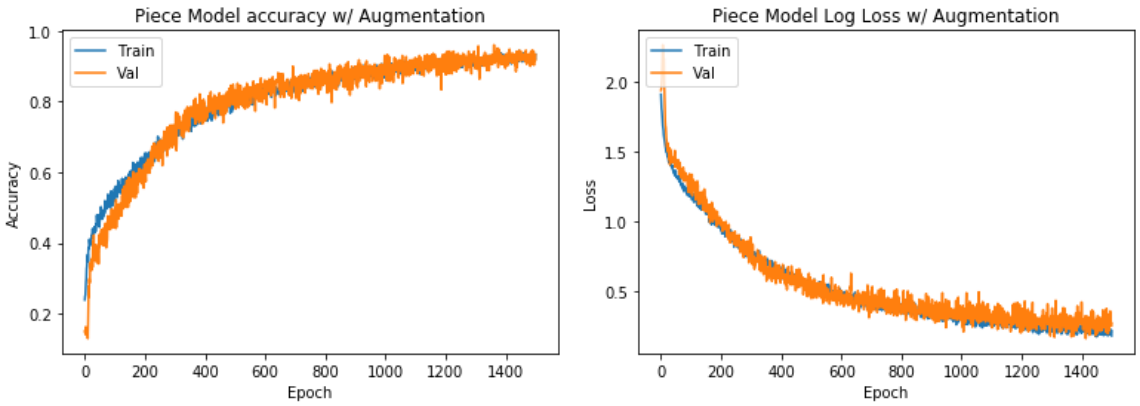


Figure 15 Final Loss and Accuracy vs Epochs during training

Table 2 Final Model Results

	Accuracy Test Result	Log Loss Test Result
Piece Model w/ Augmentation	91.0%	0.27

The results from the training far outperformed the benchmark. The model was able to raise the piece identification accuracy to 91% and reduce log loss to 0.27. Significant improvements from the 51.3% and 1.23 from the baseline. Upon review of the confusion matrix it appeared the only major issue that remained were king and queen identification. This was identified as potential issue earlier due to the dataset having potential loss of features. All other piece types had accuracy over 88%.

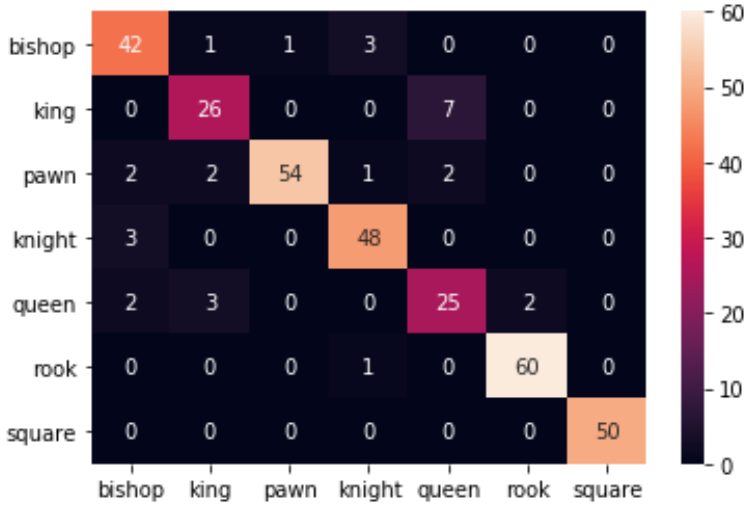


Figure 16 Final Confusion Matrix

5. Conclusion

5.1. Free Form Visualization

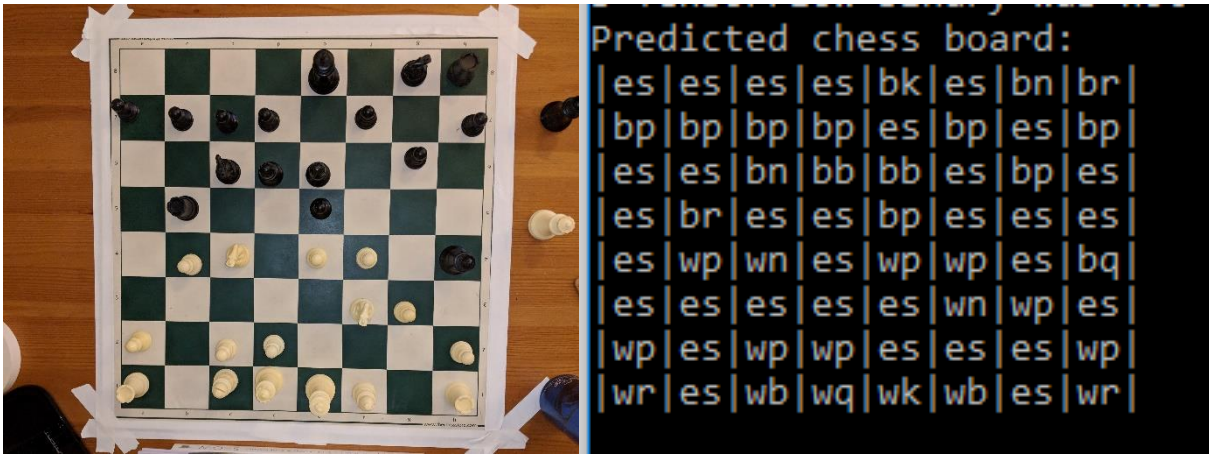


Figure 17 Example Output

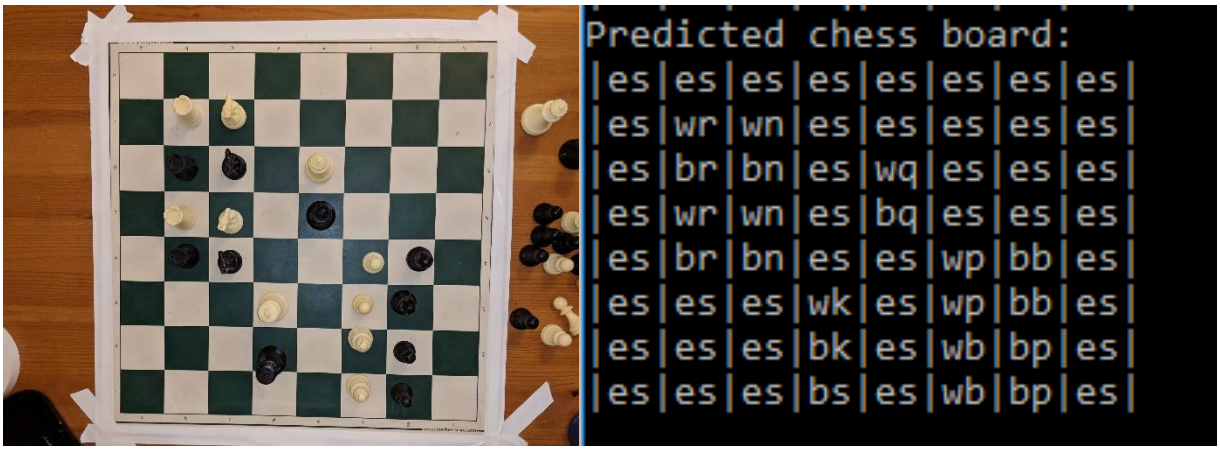


Figure 18 Example Output #2

5.2. Reflection

The solution to this problem consisted of the following steps:

1. Gathering and collection of chess piece data
2. Preprocessing of chess board data using OpenCV
3. Developing and training a benchmark model for classifier using keras
4. Refining the model and model training processes
5. Developing the end application that incorporated the model

The data gathering, and preprocessing were the most challenging and time-consuming portions of the project. The initial camera mounting scheme I had planned for provided poor quality piece data due to

the placement of the overhead camera. Once the data was obtained a portion of this problem still required classical computer vision techniques. I had no experience with computer vision previously so wanted to leverage OpenCV for this work. Properly understanding how the chess board corners algorithms and perspective transform would work took a lot of trial and error. My end implementation while providing satisfying results, was poor at precise corner location. However, I found this work also to be most interesting as well, largely due to tackling a new problem outside of an area I had experience.

5.3. Improvement

The result of the project was a classifier with >90% accuracy able to digitize a position of chess board. The output format however in its current state is not usable past visualization and a few of the parts of the process could use improvement outlined below:

1. Improved camera mount, the initial design failed and required a makeshift modification. A better camera and mounting scheme could help improve the quality of piece data and potential improve on the chopping.
2. A board built into a table vs a cloth board would help provide better corner identification as it would not have to consider the bending and bumps present.
3. As with most machine learning problems larger training, testing and validation sets could help, while the set was decent size all the pictures were taken in two runs and did not cover all potential orientations.
4. Improved output formatting, supporting standard formats used by chess engines.

6. References

Computer Vision Wikipedia. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Computer_vision#Recognition

Danner, C., & Kafafy, M. (n.d.). *Visual Chess Recogniition*. Retrieved from https://web.stanford.edu/class/ee368/Project_Spring_1415/Reports/Danner_Kafafy.pdf

Keras Documentation. (2018). Retrieved from Keras: https://keras.io/

Keras metrics. (n.d.). Retrieved from Keras Documentation: https://keras.io/metrics/

Meyer, J. (n.d.). *Raspberry Turk*. Retrieved from Raspberry Turk: http://www.raspberryturk.com/

Open CV. (2018). Retrieved from Open CV: https://opencv.org/

Sklearn Metrics Log Loss. (2018). Retrieved from Scikit Learn: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html

Xie, Y., Tang, G., & Hoff, W. (n.d.). *Chess Piece Recognition Using Oriented Chamfer Matching with a Comparison*. Retrieved from https://inside.mines.edu/~youyexie/paper/WACV2018.pdf