2D Chess Piece Classification Report

Rajtilak Indrajit

December 16, 2018

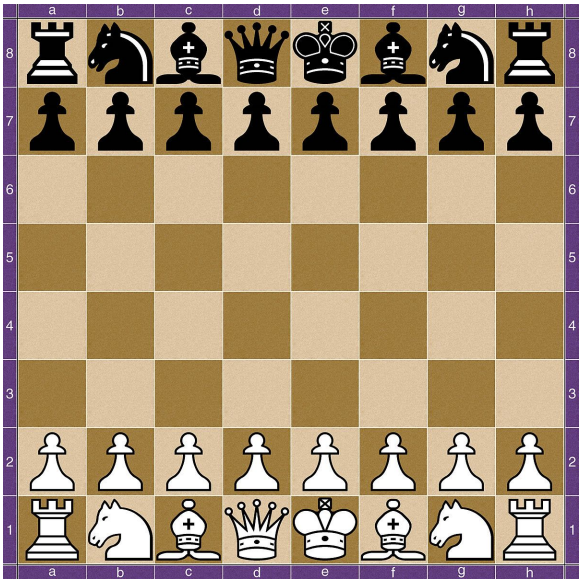
I. Definition

Project Overview

The focus of this capstone is to take the first step to enable real-time engine analysis and move suggestion of online chess games through a Chess Bot. To explain the first steps better, we need a quick intro to the game of chess.

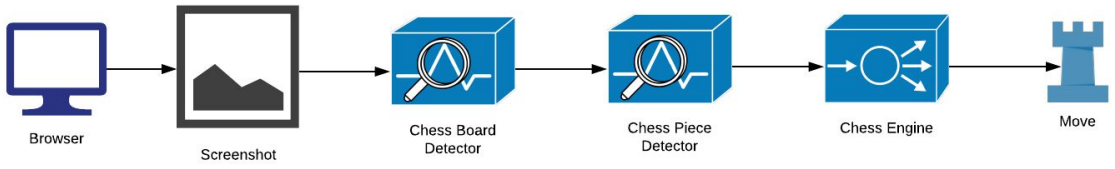
Project Domain: Game Setup and Dynamics

Chess is a two-player strategy game based on an ancient Indian board game called Chaturanga. Chess is played on a square board of eight rows (numbered from 1 to 8) and eight columns (lettered from a to h). The colors of the 64 squares alternate and are referred to as light and dark squares. The pieces are divided into white and black sets, and the players are referred to as White and Black respectively. Each player begins the game with 16 pieces of the specified color, which consist of one king, one queen, two rooks, two bishops, two knights, and eight pawns. The player with the white pieces always moves first. After the first move, players alternately move one piece per turn. The objective is to checkmate the opponent's king by placing it under an inescapable threat of capture. To this end, a player's pieces are used to attack and capture the opponent's pieces, while supporting each other.



Problem Statement

The overall objective is to enable real-time engine analysis and move suggestions for online 2D chess games. Enabling this at high level involves identifying the chess board, classifying the chess pieces, interpreting the chess position, and feeding it to an engine for analysis. The project pipeline can be visualized as follows.



This project will focus on the central component of this problem: 2D Chess Piece Detection using Machine Learning. In other words, our goal is to develop a robust classifier for chess pieces. The classifier will then be used to convert online chess games to the standard FEN notation, which in turn can be used for engine analysis and move suggestion.

Classifier Inputs & Outputs

The classifier expects square images of individual chess pieces as input, and provides predicted class of the chess piece as output. The image will be classified into one of the seven corresponding classes - King, Queen, Rook, Bishop, Knight, Pawn, and Blank.

Assumptions about the Input

- The image contains zero or one chess piece, not more
- The image is grayscale
- The image is square (1:1 aspect ratio)
- The chess piece design is 2D

The classifier is robust to variation in size, color, pattern, and even translation of chess pieces.

Not in Scope of the Project

The following activities are not within the scope of this project

- Classification of color of the chess piece - black or white
- Support for 3D chess piece design
- Splitting of the chess board into constituent pieces
- Detecting the chess board in a full screen screenshot

Key Considerations

- The ML model we train for this problem needs to be able to generalize well to the variability in 2D chess piece designs for online chess
- The model needs to have exceptionally high accuracy. Since the goal of the model is to detect all 64 chess squares accurately to describe the FEN notation, our classification accuracy needs to be really high.
- We have no trade-off benefits between precision and recall, the model has to index high on both precision & recall.
- In the worst case, even a model with accuracy as high as 0.9 will end up making wrong FEN predictions 53% of the time or more. This is because the accuracy of FEN notation is exponential in the accuracy of the model.
- The model should be reasonably fast at predictions since our application goal is to perform near real-time analysis.

Proposed Solution

Our goal is to train a model that is able to correctly classify individual 2D chess pieces with very high accuracy, and generalize well enough to make up for variability in 2D chess piece designs. Given these constraints, a deep-learning approach is a good option.

We will train a Deep Learning model using Convolutional Neural Networks, to learn the underlying concepts of 2D chess piece designs. A multi-class classification approach is optimal since we would want the model to be able to make relative trade-offs between the different classes, and since our dataset is fairly balanced with equal number of samples for each class.

Metrics

The domain of our problem requires a model that performs well on all dimensions. We need to be able to correctly classify all 64 tiles of a chess board. Getting even one of the tiles wrong would result in a wrong board position, and therefore fail to be useful. This implies that we have no room to trade off on precision and recall. Accuracy is a good enough metric to use here, since the dataset is fairly balanced overall. However, a stricter metric that would be resilient to any imbalance in dataset sizes would be the F1 score.

Hence, the metric of choice is the multi-class F1 score. For choice of averaging, I considered micro, macro, and weighted. When it comes to predicting a chess board, the model is surely more likely to see blanks and pawns, and much less likely to see the queen and the king. However, since our dataset is fairly balanced it came down to whether the model should have a bias towards any of the classes. Another factor is the problem context, a model that has a bias towards the more frequent classes is no better when it comes to predicting chess pieces, since it is an all-or-nothing result. Hence, we chose to use ‘micro’ averaging for our F1 score.

The metric discussed above is at a piece level. Eventually, we would like to introduce a new metric: Board Accuracy. Board accuracy would be a binary metric, true when we predict the board accurately, and False otherwise, regardless of whether we mislabelled one or ten squares.

Board Accuracy = # true board positions / total board positions

II. Analysis

Data Exploration

Data Acquisition

Internet searches revealed that there is not an existing chess image dataset that can be used for this classification task. Google Image search did not prove to be very useful either. There are some additional criteria for our dataset, we require a dataset that is non-uniform and representative of the variability of chess pieces in the real world. Hence, I decided to build a dataset of chess piece images from real-world images of chess pieces.

Data Characteristics

The data was created manually by screenshotting and cropping various chess piece and board designs from lichess.org, chess.com, chess24.com. The dataset consists of 1k images based of the 2D chess boards on online chess websites.

A total of 35-40 designs of chess boards was used. The dataset has images that vary in background, texture, shape, color, and design. This variability is a close approximation to online chess boards that the model is meant to cope with - online and on-screen 2D chess boards on popular chess sites.

The dataset will be used for both training, validation and testing of the model. Variability in zoom, shear, and rotation is introduced during training and testing using keras. The dataset is organized into 6 subdirectories based of the different classes of the chess pieces. K for king, Q for queen, N for knign, B for bishop, R for rook, P for pawn, and B for blank. Blank is treated as a separate class since it overall fits nicely into the multi-class classification paradigm. It is equally possible to exclude the blank class and use a rule-based system to classify it beforehand.

The dataset is well balanced, with equal number of images for most classes. However, the queen and king classes have fewer images - this is simply a function of the fact that there is only one set of king & queen per board, whereas two sets of all other pieces. The number of pawns has purposefully been limited so as to not introduce imbalance in the dataset. Here are a few of the images from the dataset.

Dataset Statistics

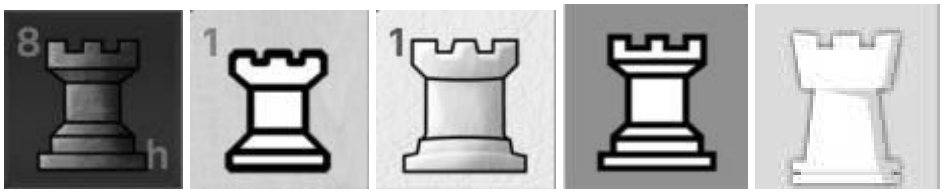
Class	Count of Train	Count of Train	Count of Train	Count of Train

	Images	Image Distinct Designs	Images	Image Distinct Designs
ROOK	176	44	16	4
BISHOP	176	44	16	4
KING	114	32	12	6
QUEEN	114	32	12	6
PAWN	176	44	16	4
BLANK	97	38	16	8

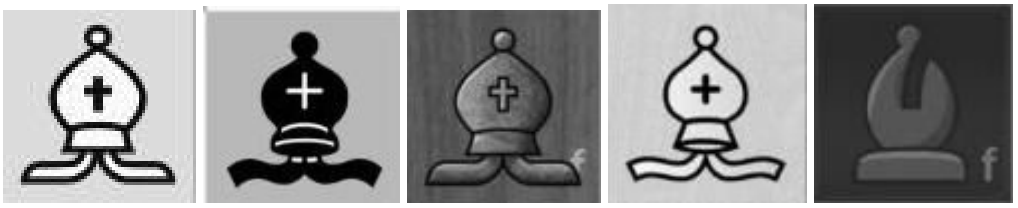
Data Visualization

Sample Images

ROOK

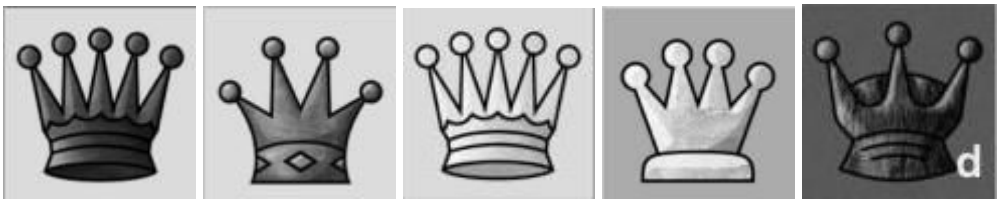


BISHOP



KING

QUEEN



PAWN

BLANK

Algorithms and Techniques

In order for our model to generalize to 2D chess piece designs, we are making a big assumptions about there existing, and our model learning the commonality between different designs. This has likeness to the concepts of eigenfaces and eigenvectors used in classical machine learning and image detection.

Since the solution hinges on the eigenvector approach, one classic benchmark model would be a Support Vector Machine. Since we are dealing with image data, we would be using a non-linear kernel to be able to learn and separate the dataset, and a good choice here would be the Radial Basis Function or RBF kernel. The model is trained to perform multi-class classification on the image data across the seven classes. To facilitate generalization and identification of the underlying eigen-pieces (similar to eigenfaces), we would add a data pre-processing and dimensionality reduction step. I am following the standard image classification technique in supervised learning - Principal Component Analysis. We are likening this classification task to that of face detection. In the same vein as eigenfaces, we are generating eigen-pieces that represent the different chess pieces using Principal Component Analysis. Also, we will use GridSearchCV for tuning the hyperparameters of our Support Vector Machine. The model would closely follows the example shown here. The data preprocessing would involve converting all the images to grayscale, standardizing shape to be 1:1 aspect ratio, and the size to be 64x64 pixels, and normalizing the pixel image data to the scale of 0-1. The image data will then be flattened to a single row of 4096 feature vectors with values in the range of 0-1.

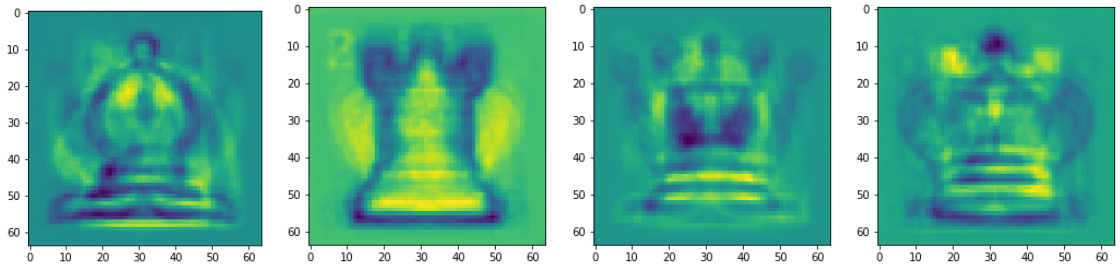
However, as previously mentioned, the final model would involve Convolutional Neural Networks in a Deep Learning setting. The motivation for using CNNs is clearly their superior ability to detect patterns in image data. CNNs are able to convolve on image data to detect patterns while retaining the spatial separation of pixel data. This allows CNNs to perform significantly better at detecting underlying patterns in images. Added to this, our model architecture will benefit from layering: we would have multiple layers of CNNs that feed forward, thereby giving us the ability to detect more complex patterns in our image data. Each successive convolution filter will detect increasingly complex features, from lines, to edges, to basic shapes, to eventually the underlying eigen-piece representation for each class. The model architecture will closely follow the architecture of VGG16. The image data will be converted to grayscale, and the size and shape will be set to 64x64 pixels in a 1:1 aspect ratio, and normalized such that the values are in the range of 0-1. This is then fed into the top layer of the neural network as input to the CNNs.=

Benchmark

While there is existing work in this domain, there isn't an existing benchmark model that could be used to directly compare the model we are setting out to build. TensorFlow Chessbot by Sameer Asari uses CNNs to perform this task on 2D chess boards and achieves fairly robust performance. Chess Piece Recognition by Anoshan Indreswaran also takes a similar approach. However, the models solve different problems using different datasets and cannot be used to directly compare performance on our dataset.

As discussed before, we will treat this problem similar to the problem of image classification using SVMs and PCA. The model would closely follows the example shown here. The first step is to separate the data into training and test sets, which is done as described previously. Image preprocessing steps include converting all the images to grayscale, resizing them in a 1:1 aspect ratio to be 64x64 pixels. The image data is flattened to a single row of 4096 feature vectors. We then perform principal component analysis transformation on the dataset. The choice of number of components is somewhat arbitrary, after a couple of attempts I realized that the accuracy improvement diminishes beyond 25 components. The model is a SVM with an RBF kernel. We perform GridSearchCV to identify the best hyperparameter values for C and gamma. The best performing model had a C value of 1000.0 and gamma of 0.01.
Ultimately, the benchmark model we trained using this dataset achieves an F1 score of 0.82.

A few of the eigen-pieces generated by the benchmark model are shown here: they appear to resemble a (in order) bishop, rook, queen, and king.



III. Methodology

Data Preprocessing

As mentioned before, the data preprocessing steps are as follows:

Splitting Squares

An early pre-processing step outside the model pipeline is to split the full square board into the different squares of the chess board. This is done fairly naively, with the expectation that the board image is perfectly square and we are able to simply cut it into 64 smaller squares. The code for this step can be found here. This step allows us to generate the data for training and testing.

Size & Aspect Ratio

We expect each image to be a perfect square with 1:1 aspect ratio with width and height of 64x64 pixels. We use openCV to resize the images, with the default INTER_LINEAR interpolation.

Grayscale

The images are converted to grayscale such that the array dimensions are reduced from (64, 64, 3) to (64, 64, 1)

Rescale

All the image pixel data is rescale such that the values lie between 0 and 1. This is said to have performance improvements during the training and backpropagation steps of our neural network.

Train, Test & Validation Data

As mentioned before, the testing data was hand-picked to include variance in designs that are previously unseen by our model. The test and validation data are randomly split using *keras.ImageDataGenerator.* We use a validation split of 0.25, along with introduction of zoom and horizontal flip.

Implementation

Within the realm of chess piece classification, there were a number of approaches to consider and choices to make. Here is a brief discussion of some of those choices.

Predicting Individual Pieces vs Entire Positions

The first problem to solve was how to set up the ML model. The two clear choices were to either predict at a piece-level or a board-level. A model that were to predict entire board positions seemed impossible at first glance, since the number of possible positions reaches 10^{120} as noted by the Shannon Number. Even a more conservative estimate puts this at around 10^{40}. Hence I chose not to pursue this any further.

2D vs 3D chess pieces

Another point of consideration was whether or not to support 3D chess pieces, and if yes, whether to include them as part of the same class or to separate the 2D king (2DK) from the 3D

king (3DK). The challenge here turned out to be segmenting the chess board into squares such that the 3D chess pieces were contained in each square. After much trial and error, I decided to abandon 3D piece support and focus on 2D. If I were to attempt 3D again, based on the trial and error, I would separate the 3D classes from the 2D classes for some pieces (King, Queen), but perhaps keep it the same for others (Knight, Rook, Bishop).

Detecting Piece Color

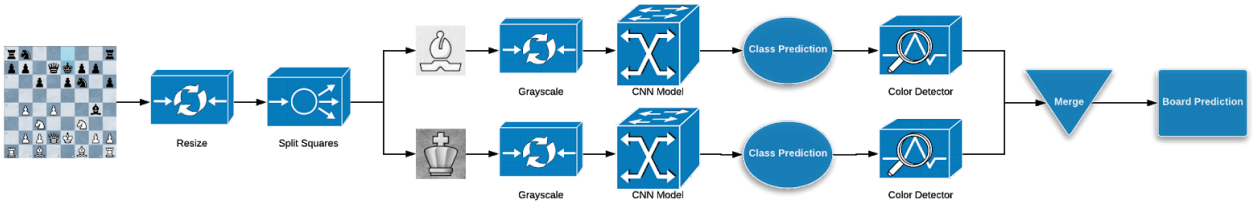
Should the model detect just the piece, or also attempt to detect the color of the piece? The general feeling here was that the color is very subjective, and it is difficult to predict in isolation. It would be a much simpler problem to predict color based on the global board level, rather than the local information at a square level. Hence, this problem was pushed further up-stream. My idea is to solve this problem using a naive pixel averaging approach to being with, and build on top over time.

Google Search vs 2D Boards

Since I had to generate the dataset myself, I had the choice of Image search or manually screenshotting and splicing images from popular online chess websites. The former approach proved to be too difficult, as there was simply too much noise in image search. Hence I decided to take the manual route, as already mentioned above in the dataset description.

Pipeline

Ultimately, the design I landed on is the most simplistic, but also the most effective. The process steps include creating the dataset by sourcing and manually classifying the training and validation set into the corresponding image classes. The images will then be converted to grayscale and scaled to desensitize our detector to color and size variations. These images are then used to train a CNN that uses 5 convolution layers and nearly 100k trainable parameters. The resulting model will then be run on our test set to measure classification accuracy.

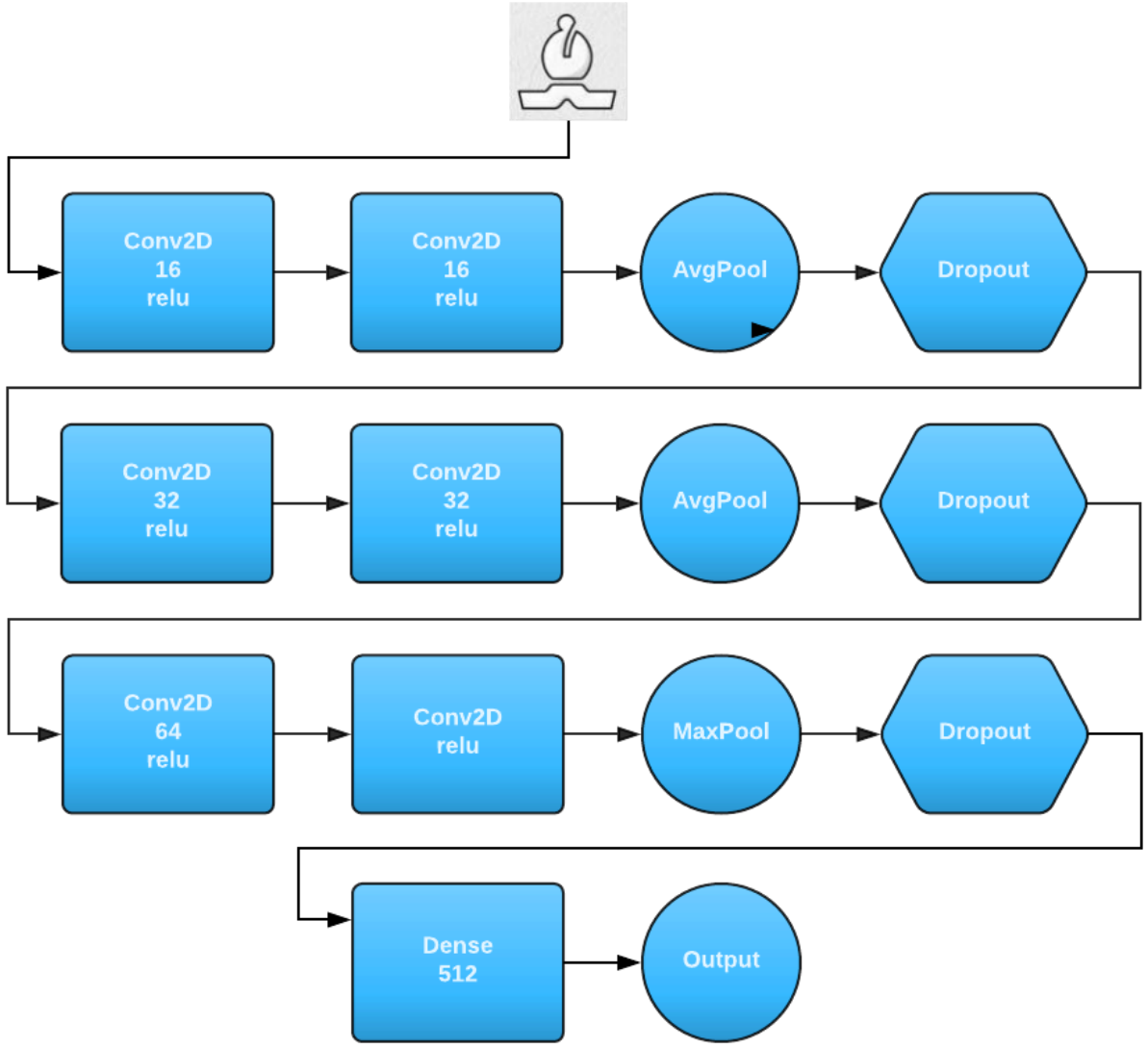


Refinement

Initial Solution

The initial solution was quite different from the ultimate model that we landed on. The initial solution started with the architecture diagram seen below. Fundamentally, there were a few learnings and iterations that lead to improvements to the architecture. What was evident from the

initial training cycles is the the model had high bias this was evident from the high training error on the dataset. The following aspects were tweaked, in order to improve the model performance.



Pooling Layers

Replacing the AveragePooling2D with MaxPooling2D reduced training error significantly. I imagine this is because of the nature of image data that we are dealing with: the average pooling layers would basically do more hard than good at distinguishing the different layers, detecting edges, and so on. MaxPooling2D on the other hand does a much better job of the same.

Activation Function

I started with *relu* activations on all the convolution filters. I then found out about *elu* and that it was faster at converging and also solved for the *dying ReLU* problem. Indeed elu performed slightly better than ReLU.

Filters & Layers

The first model had three layers with two filters per layer. Since the model had high bias, I decided to increase both the number of layers, and the number of filters per layer. I used the VGG16 model architecture as a blueprint for making these choices. The resulting model has six layers with three filters per layer. I tried experimenting with changing filter sizes, but it did not have material impact on the accuracy of F1 score of the model.

Dropout

The first iteration had a dropout in each layer of the model, and the dropout rate was set to 0.25. To reduce the bias, I decided to drop the dropout layers and reduce the rate to 0.1. The final model has a dropout at the final layer, before the Dense layers with a rate of 0.1.

Loss Function

In order to support the multi-class setting, we are restricted to few choices of loss function. I decided to use the most classic *categorical_crossentropy* loss function.

Optimizer

I tried a few different optimizers, starting with adam, RMSprop, adagrad, and adadelta. Of these, *adadelta* was the best performing.

IV. Results

Model Evaluation and Validation

The final model was a result of multiple iterations of network architectures, choice of layers, filters, filter sizes, dropouts, dense layers, and so on. The guiding principle for making these choices were the performance on the training and validation datasets, and influenced by the architecture of the VGG16 model architecture.

Model Architecture Parameter Choices

The parameters chosen for the final model are:

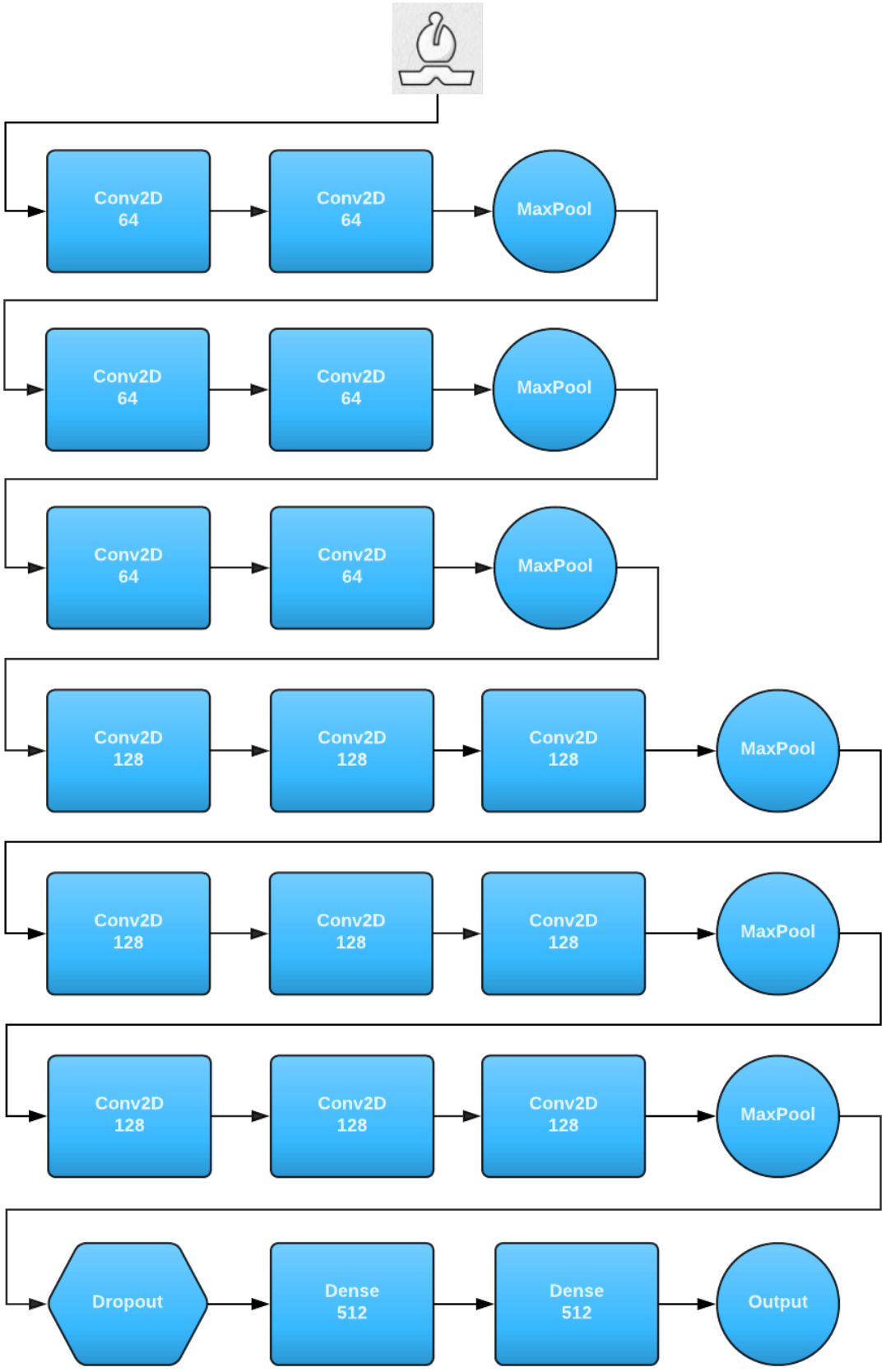
- Number of layers: six
- Filters per-layer: two for first three layers, three for the next three layers
- Number of filters: 64 in first three layers, 128 in the next three layers
- Activation: elu for all the layers; final layer is softmax to allow for multi-class outputs
- Pooling: MaxPooling2D at each layer of the model
- Filter size: Standard filter size of 3x3
- Dropout: One dropout layer prior to dense layers with rate of 0.15
- Padding: same padding to ensure all of the data is convolved
- Optimizer: adadelta
- Loss: categorical_crossentropy
- Epochs: 50
- Batch Size: 25
- EarlyStopping: Validation loss with patience of 10

Model Performance on Training & Validation sets

The model was trained on 773 training images and 256 validation images. As mentioned before, the validation and training sets were shuffled randomly using ImageDataGenerator. The model was trained for 50 epochs in batches of 25 images, with early stopping criteria based on validation loss.

The training data logs show that the model started poorly (as expected) and improved dramatically over the first ten epochs, with loss dropping to ~0.2 and validation loss of ~0.1. The next twenty epochs show clear trade-offs between validation and test set loss. The lowest loss of ~0.01 was reached in epoch 26, however the validation loss of this model was higher. Over the next four epochs, validation loss does not improve any further. The best model is the one reached in epoch 25, where validation loss was a low 0.00073.

Final Model Architecture



Justification

The final model we developed performs much better than our benchmark model. The best visualization of the difference in performance is to look at the F1 scores and the confusion matrices of both models.

Benchmark Model

F1 Score

0.82

Classification Report

	precision	recall	f1-score	support
BISHOP	0.82	0.88	0.85	16
BLANK	0.94	1	0.97	16
KING	0.77	0.83	0.8	12
KNIGHT	0.76	1	0.86	16
PAWN	0.71	0.62	0.67	16
QUEEN	0.8	0.67	0.73	12
ROOK	1	0.75	0.86	16
micro avg	0.83	0.83	0.83	104
macro avg	0.83	0.82	0.82	104
weighted avg	0.83	0.83	0.82	104

Confusion Matrix

	BISHOP	BLANK	KING	KNIGHT	PAWN	QUEEN	ROOK
BISHOP	14	0	0	0	2	0	0
BLANK	0	16	0	0	0	0	0
KING	0	0	10	1	0	1	0
KNIGHT	0	0	0	16	0	0	0
PAWN	0	1	1	4	10	0	0
QUEEN	2	0	0	0	2	8	0
ROOK	1	0	2	0	0	1	12

Final Model

F1 Score

0.98

Classification Report

	precision	recall	f1-score	support
BISHOP	1	1	1	16
BLANK	1	1	1	16
KING	1	1	1	12
KNIGHT	1	1	1	16
PAWN	1	1	1	16
QUEEN	1	0.83	0.91	12
ROOK	0.89	1	0.94	16
micro avg	0.98	0.98	0.98	104
macro avg	0.98	0.98	0.98	104
weighted avg	0.98	0.98	0.98	104

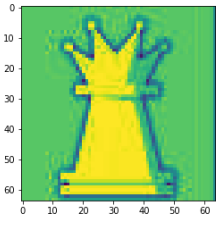
Confusion Matrix

	BISHOP	BLANK	KING	KNIGHT	PAWN	QUEEN	ROOK
BISHOP	16	0	0	0	0	0	0
BLANK	0	16	0	0	0	0	0
KING	0	0	12	0	0	0	0
KNIGHT	0	0	0	16	0	0	0
PAWN	0	0	0	0	16	0	0
QUEEN	0	0	0	0	0	10	2
ROOK	0	0	0	0	0	0	16

Comparison

The benchmark model performs reasonably well on the dataset - reaching an F1 score of 0.82. The classification report shows that the model performs really well on classifying blank cells, but poorly on all other classes. The confusion matrix also reveals that there is a lot of confusion between pawn, queen, and bishop. Overall, this model does poorly on the dataset.

The final model is a significant improvement over the benchmark, we are see that the model a high F1 score of 0.98. Precision and Recall are perfect for all classes. The model only fails on one particular queen design that it has not seen before, where it mistakes it for a rook, even this mistake is understandable given the image to the right.



V. Conclusion

Reflection

We started this project with the goal of enabling real-time engine analysis and move prediction for online 2D chess games. At the core of the process, was a chess-piece classifier that would classify the pieces with high accuracy. The scope of the project was to develop a model that would perform really well at the classification problem, with the ability to generalize to differences in color, shape, and design of 2D chess pieces.

The model we have trained during the project achieves most of these goals - the model trained on roughly 30 distinct designs and was able to generalize and predict ~4 previously unseen designs of chess pieces, making only one notable exception (the misclassification of queen as a rook). The most difficult aspect of the project was data collection - given the strict data requirements of the problem statement, we had to create a dataset by hand. Clearly, the model will only get better with more data to learn from. However, there is more work to be done even within the realm of classification.

Improvement

More Data

The first and most important is more data. The model we have trained has only seen ~700 images in total. If we could generate a dataset of 10,000 images or more, the model would be much more robust and be capable of generalizing much better.

3D Chess Piece Support

The model is restrictive in that it only supports 2D chess pieces. Extending the model to support 3D boards would be an ambitious but useful improvement area. It could however be a completely separate model, or perhaps the same model would be capable of discriminating 2D vs 3D pieces.

Color Detection

The model we have trained so far does not detect color of the chess pieces. This feels like a tricky problem to solve using ML given the variability in chess piece colors generally, and a simpler problem to solve using a rule based system, especially since it requires comparison across different pieces on the board. However, the model could be extended to detect color also.

Variability in Image

Presently, we expect the chess pieces to be square and be reasonably centered in the image. We could start supporting non-standard aspect ratios and non-centered images in future. We could also support translations other than horizontal flips and zooms, such as skews, shears, rotations, and translations.