Chess-Ray Vision

Multi-Object Detection and Classification for Resolving Chessboard Photographs

Samuel Ryan (samryan)
samryan@seas.upenn.edu

Mukund Venkateswaran (mukundv)
mukundv@seas.upenn.edu

Michael Deng (michdeng)
michdeng@wharton.upenn.edu

Kurt Convey (kconvey)
kconvey@seas.upenn.edu

Team Member	Contributions
Samuel Ryan	Building dataset, data preprocessing, models, report
Mukund Venkateswaran	Building dataset, data preprocessing, models, report
Michael Deng	Building dataset, models, report
Kurt Convey	Data preprocessing, report

Abstract

We aim to solve the problem of detecting and classifying the types, colors, and positions of multiple chess pieces from photographs of chessboards. We frame this as a multi-class, multi-label task. Our results can be interpreted in terms of three distinct outcomes. First, we take 500 aerial photographs to create a new dataset of chess game images labeled in accordance with Forsyth-Edwards notation. Next, we devise an image preprocessing procedure that standardizes photos taken from different angles into consistent, cropped squares; this greatly reduces the complexity of the learning problem. Finally, and most importantly, we design several models, the best of which is a convolutional neural network that correctly classifies game states with over 99% accuracy.

CIS 520: Machine Learning
May 2019
Project Mentor: Arnab Sarker
Code: https://github.com/samryan18/chess-ray-vision/
Dataset: https://github.com/samryan18/chess-dataset/

Introduction

A game of chess can often take up to an hour or more to complete. When played with a physical board, it can be especially inconvenient when gameplay is interrupted. That is, to pause and resume a game, one would have to remember the game state (e.g. by taking a photo), find another time to meet up in-person, and manually recreate the board at a later time. Most likely, two players would be forced to come to terms with a dissatisfying, premature end to their game. To solve this problem, we build a multi-object detection and classification model for resolving chessboard photographs. By translating photos of physical chessboards into a virtual representation of the game state, one can save the game state and continue playing at a later time (e.g. by sending the game state to an online game engine).

There are additional use cases for this model. For instance, thousands of chess tournaments worldwide still employ people to manually record game states on paper score sheets. Companies also produce electronic chessboards. These physical boards have electrical components that track game states, and can cost between $50 and several hundred dollars. In both of these examples, anyone with a smartphone (i.e. camera) can instead use a model like ours to automate the real-time tracking of game states. Our model can also be used as a component in real-time game advice software or fully-autonomous chess-playing robots.

Related Work

Indeed, previous commercial projects and academic works suggest that there is demand for products based on our model. [1] For instance, a French company sells a paid "Webcam Chess" product that utilizes an unspecified algorithm. [2]

A canonical computer vision problem precursing ours is that of identifying the numbers in images of NxN Sudoku boards. [3]. By preprocessing an image of a Soduku board into N^2 separate images (i.e. one image per square), one can reduce the problem into that of classifying N^2 MNIST handwritten digits before recombining the results. Our approach follows a similar methodology.

Problem Formulation

We formulate the problem in terms of three discrete sub-problems: dataset creation, preprocessing procedure development, and machine learning model design. First, we must create an original dataset by taking aerial photographs of a standard tournament-style silicone chessboard in a variety of potential settings (e.g. different pieces on the chessboard, lighting, and aerial angle). Crucially, when taking hundreds of images, an efficient and precise process for taking and labeling photos must be developed. With regard to labeling, it is a standard practice to encode game states with Forsyth-Edwards notation (FEN), a virtual representation designed for the purpose of holding information necessary to recreate chess game states. This notation, however, is not well-adapted for labeling in the context of machine learning; we must find a way to transform FEN-notated (i.e. string) labels into a more suitable virtual representation (i.e. three-dimensional one-hot encoded tensors) for modeling.

Next, we have to develop a method for preprocessing photographs before model training. Finally, we must design effective machine learning algorithms to detect and classify chess piece positions, colors and types (e.g. pawn, knight, rook, etc.) and output these classifications in the form of the virtual representation which we devised. A natural choice of model for this computer vision problem is the convolutional neural network (CNN). However, we fit other models such as logistic regressions and fully-connected neural networks to contextualize the performance of our CNNs. To implement our algorithms, we use Python, with the support of OpenCV and PyTorch libraries. Our code and dataset can be found in the Github repositories linked on the cover page.

Accuracy and Loss Metrics

All misclassifications (i.e. missing, incorrect or extra pieces) carry equal weight. Our motivation for this is that if our model were to be incorporated into a real product, we would want the least amount of user swaps after errors. That is, loss should be correlated directly with the number of swaps necessary to rectify a misclassified board. Thus, our key metric is accuracy. We utilize a cross entropy-based loss function.

Noting that there are 64 spaces on a chessboard, 13 possible "pieces" in each space (6 possible white pieces, 6 possible black pieces and "no piece") and $B = 10$ images per batch, our chosen loss and accuracy metrics for a given batch of chessboard classifications are as follows:

$$batch\_accuracy = \frac{1}{64B}\sum_{i=1}^{B}\sum_{j=1}^{64}\mathbb{1}(\hat{y}_{ij} = y_{ij})$$

$$batch\_loss = -\sum_{i=1}^{B}\sum_{j=1}^{64}\sum_{k=1}^{13}\phi_{ijk}log(\hat{\phi}_{ijk})$$

In these equations, B represents the batch size, y_{ij} is the true label for "piece" (i.e. space) i in board j, ϕ_{ij} is the one-hot encoded vector of true probabilities for each piece in space i of board j, $\hat{\phi}_{ij}$ is the vector of predicted probabilities for each piece in space i of board j, and $\hat{y}_{ij}$ is the argmax prediction from $\hat{\phi}_{ij}$. Variable k iterates through the possible pieces in each space. Since multiplying the loss function by a constant does not affect the optimization procedure, we do not normalize it.

Dataset

Since a pre-existing dataset of labeled chess game photographs was unavailable, we create an original dataset by taking 500 aerial photos of various chessboard configurations, lighting conditions (i.e. with and without flash), aerial angles and backgrounds (e.g. floor surfaces, extraneous objects). We label them in accordance with Forsyth-Edwards notation. We streamline this process by playing out games from a database of famous chess games (that has already labeled each move of every game with FEN), taking a picture after we replicate each move.

(a) Flash off, gray wooden floorboard (b) Flash on, light wooden floorboard

Figure 1: Sample images from our final photo dataset. The noise created by the different backgrounds and angles of the photos made preprocessing imperative.

Preliminary Dataset for Proof of Concept

Using a simpler, preliminary dataset allows us to build a proof of concept that abstracts away the photo preprocessing steps of our project. That is, it enables us to not only build a data and modeling pipeline, but also test our hypothesis that we can build a CNN architecture that can classify these types of images with high accuracy—all before even creating our dataset of actual photographs or applying models to it. Hence, we fit models on a dataset of FEN-labeled digital chessboard images before we train models on our original photo dataset:

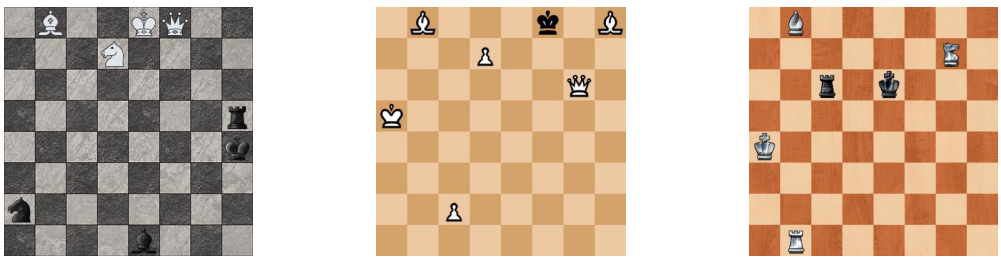


Figure 2: Sample images from our simpler, preliminary dataset.

All 100,000 images in the preliminary dataset have the same dimensions (400×400 pixels). However, the dataset consists of 28 different styles of chessboards mixed with 32 unique styles of chess pieces. The variation of these parameters in the dataset allows us to train models with greater capacity for generalization to different chessboards and sets of chess pieces. Simultaneously, since each image has identical dimensions and can be preprocessed in fewer and less-complicated steps, classifying these virtual images is an easier learning problem. Specifically, preprocessing in this case resembles that of the aforementioned Sudoku-board classification problem. We cut the images length-wise and width-wise into 64 separate tile images, each of which is treated as an individual classification problem. More information about our methods for classification on the simple dataset can be found in the Github repository linked in the cover page. Ultimately, we use a similar preprocessing approach on our preprocessed real dataset. Indeed, we find that, on this preliminary dataset, we can train a CNN classifier that identifies pieces with 100% test accuracy.

Label Preprocessing

Both of our datasets are labeled with FEN, which uses simple strings with backslashes to denote separations between rows. For example, the FEN label for the starting game position is as follows:

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR

We translate this notation into a representation better-suited for machine learning: three-dimensional one-hot encoded tensors. To do this, we first write the FEN labels (i.e. strings) to matrix conversion, which transforms these strings into 8×8 matrices with elements of integer-value between 1 and 13 to denote each piece (or empty space) on the board. Then, we one-hot encode each element of the matrix, stretching out a third dimension. This results in labels with $8 \times 8 \times 13$ dimensions.

Image Preprocessing

Even when photographs are varied or imperfect, our model must be robust. Hence, we preprocess photos into consistent images before the model attempts to classify them. Overall, our preprocessing steps transform our photos into perfect squares resembling the digital images from the preliminary dataset.

Unsurprisingly, finding corners of rectangles is a relatively common computer vision problem. Python's most popular computer vision library, OpenCV, has a built-in findChessBoardCorners function. [4] However, this function was unfortunately not robust enough for our purposes; it was unable to find the corners on most of the photographs in our dataset. Hence, we implement our own corner-finding procedure using a combination of existing and new implementations.

First, we use OpenCV to find corners on the boards. Then, we calculate a homography to warp the images into perfect squares. Given that preprocessing is not the focus of this project, we build our preprocessing on top of an an existing open-source project. [5] which was built with the OpenCV-Python package. [6] See Appendix 0.1 for our fully-detailed procedure.

This procedure successfully transforms all 500 photos in the dataset into perfect square boards:

(a) Original Image (b) Find outline of board (c) Transform and crop

Figure 3: General Steps of Preprocessing

In the final preprocessing step, images are cut into 64 squares. In the CNN model, these squares are which are fed in as individual features, each with three channels for RGB color inputs.

Models

(Naive Model) As previously mentioned, our criteria is raw accuracy of classifications. That is, the number of correctly classified spaces divided by the size of the board (i.e. 64). A naive model may choose to simply classify every piece or space on the board with the most common "piece" each time. Intuitively, since chessboards are relatively sparse (especially in the middle and end of games), a blank space is the most common "piece" on the board. In our dataset, guessing blank every time gives a 92.3% accuracy. We use this as a baseline by which to assess the performance of both other "baseline" models and our CNNs.

(Logistic Regression) Our first baseline model is a logistic regression with L2 regularization on its weights. This model takes all three RGB channels from every pixel in the image as inputs. The model learns weights for each of these 120,000 inputs, which are summed to create a $64 \times 13 \times 1$ tensor. This tensor is then fed through a softmax function over the second dimension to obtain probabilities for each piece in each space.

(Fully Connected Neural Network) Our next baseline model is a fully connected neural network. Like the logistic regression, this model takes all three channels from every pixel in the image as inputs. It feeds them through three linear layers with 512, 256 and 13 hidden nodes to output 13×1 vectors which are, again, fed through a softmax function. space.

(Convolutional Neural Network) We initially hypothesized that CNNs would be the best modeling approach. CNN models achieve translational invariance (i.e. object recognition unaffected by position) with significantly fewer weights. The shared CNNs weights provide three distinct benefits:

1. Forward and backward passes through the models are significantly less computationally expensive. The time to train a CNN for 25 epochs is less than a tenth of that of logistic regression. See Table 1 for the train times of each model.
2. The models converge in fewer batches because there are less directions to traverse in the parameter search space. This is demonstrated in the learning curves in Figure 4.
3. Most importantly, reduced complexity gives the models much stronger generalization performances and prevents overfitting on training data. Ultimately, this is corroborated by the high validation accuracy of our CNN models.

We train several different CNN architectures and compared their validation accuracy metrics. The architecture of our final CNN model is detailed later in this report.

Training and Evaluation

We build and train our models with PyTorch and run them on Google Cloud GPUs using Google Colab. The machine we use has a GPU and CPU with 11.5 and 5 gigabytes of available memory, respectively.

For the purpose of evaluation, we split our dataset of 500 photos into three sets: a training set of 420 photos (26,880 piece classifications), a validation set of 30 photos (1,920 piece classifications) and a test set of 50 photos (3,200 piece classifications). We use the validation set to choose between models and the test set to report the final generalization accuracy of our final model.

Due to memory constraints, we are limited to a batch size of 10 (which leads to a high degree of stochasticity in the accuracy of individual batches). During each epoch, the batches are reshuffled and fed sequentially through the models—followed by gradient calculations and optimizer updates. We use the Adam optimization algorithm for gradient descent, a commonly-used alternative to standard stochastic gradient descent with stronger convergence properties.

Algorithm 1: Training Loop (Implemented with PyTorch)

Input : model, batchLoader, optimizer, calculateLoss, calculateAccuracy
Output: model
for *epoch* $\leftarrow$ 1 **to** 25 **do**
  batchLoader $\leftarrow$ shuffle(batchLoader)
  for $i, (batch, labels) \in enumerate(batchLoader)$ **do**
    prediction $\leftarrow$ model(batch)
    loss $\leftarrow$ calculateLoss(prediction, labels)
    accuracy $\leftarrow$ calculateAccuracy(prediction, labels)
    gradients $\leftarrow$ gradients(loss, model)
    model $\leftarrow$ optimizer.step(gradients, model)
    saveToLogFile(accuracy, loss, i, epoch)
  end
end

Results

Model	Epochs	Train Acc	Validation Acc	Test Acc	Train Time
Naive Model (guess blank board)	n/a	92.3%	92.3%	n/a	n/a
Logistic Regression (L2 regularization)	25	76.70%	76.77%	n/a	34m 31s
3 Layer Fully Connected	25	95.11%	92.91%	n/a	21m 40s
CNN (64-64-64)	25	99.16%	99.58%	n/a	5m 50s
CNN (64-64-64) (Dropout)	25	98.87%	99.21%	n/a	5m 55s
CNN (64-64-64) (BatchNorm)	25	99.33%	99.63%	n/a	6m 10s
CNN (64-64-64) (BatchNorm, Dropout)	25	99.55%	99.58%	n/a	6m 18s
CNN (64-64-16) (BatchNorm)	25	99.46%	99.84%	n/a	2m 47s
CNN (32-32-16)	25	98.11%	97.08%	n/a	2m 25s
CNN (32-32-16) (BatchNorm)*	25	99.56%	99.84%	99.28%	2m 27s

Table 1: Comparison of various model results. All CNNs have three convolutional layers using 3×3 filters followed by three fully conencted layers. The parenthetical numbers to the right of each model name represent the number of filters per layer. The bolded CNN is our final model which we test on the test set.

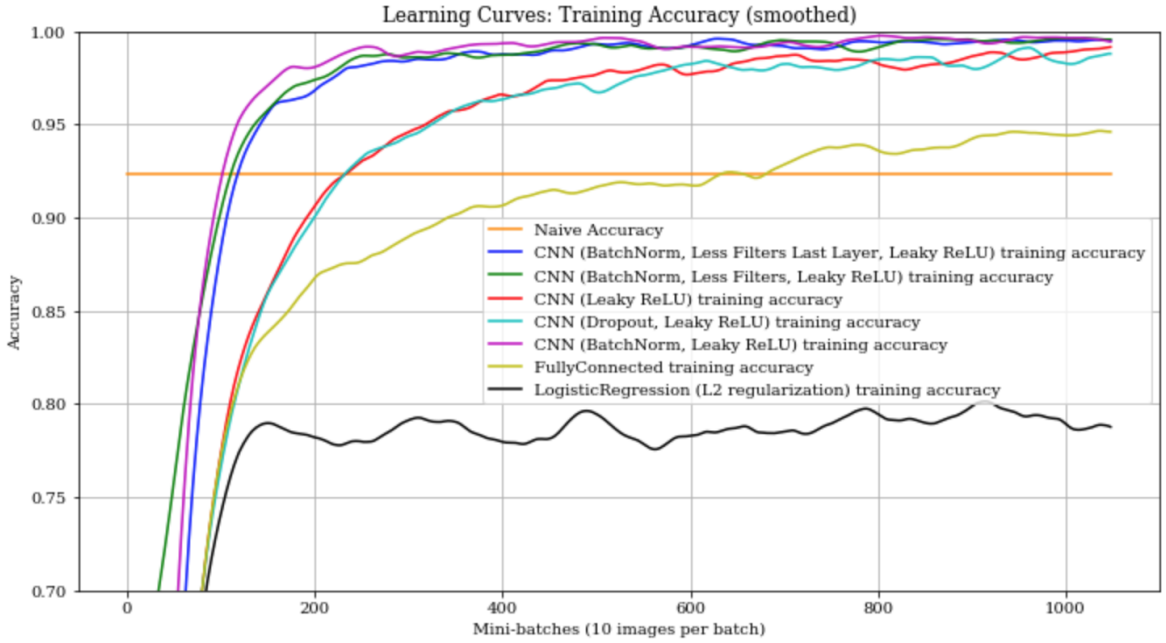


Figure 4: Learning curves for each model. For the purpose of visualization, these curves are smoothed. [7] Note that BatchNorm appears to help models converge more quickly.

Final Model

The highest-performing model on both the training and validation sets is the CNN with fewer filters per layer. Furthermore, this model trains in drastically less time than the other models. Hence, we choose to run this model on our test data. Given the scope of the project, it did not make sense to test many other architectures. Though, we do believe that it is possible that an even simpler model—possibly even a purely convolutional model with max-pooling layers to reduce dimensionality to the final 1 by 13 vectors– is likely achieve similar results.

Each 25×25 image of a board square is passed through three convolutional layers (with 32, 32 and 16 filters, respectively), without the use of max-pooling. Then, the tensors are flattened and passed through three linear layers, applying batch normalization and a leaky ReLu activation function after each layer, which is a fairly standard practice for designing CNN architectures. See Appendix 0.2 for an architecture diagram of our final model.

Although there is not an academic consensus on why batch-normalization works, it seems to have considerably improved the convergence speeds of our models. We use standard batch-norm implementation, normalizing the outputs of every layer in our network after applying nonlinearities. [8]

The accuracy statistics shown above are calculated across the entire training set with all squares counting equally. The model classifies 360/420 (85.71%) of train photos with 100% accuracy, 25/30 (83.33%) of validation photos with 100% accuracy and 35/50 (70.00%) of test photos with 100% accuracy. See Appendix 0.4 for specific examples of successful and unsuccessful model runs from the test set.

Conclusion and Discussion

Our results indicate that it is indeed possible to accurately classify pieces in photos of chessboards. A notable takeaway from this project is that preprocessing is a key step in making this a tractable problem. Taking raw images and attempting to classify them would have been significantly more challenging. With a transformation procedure that is applied to every image, we turn this multi-object classification problem into one that a relatively simple CNN can perform almost perfectly on.

Our model generalizes very well to our validation and test sets. However, all of the images in our overall dataset are selected from the same distribution of images taken in only four different settings on a single camera. In order to build a more robust product that can classify any chessboard taken on any camera, we would need a significantly larger dataset (which would be a rather tedious endeavor).

Acknowledgments

We would like to thank our project mentor, Arnab Sarker, for his helpful suggestions throughout the project.

References

[1] "Demand." [Online]. Available: https://www.chess.com/forum/view/chess-equipment/web-cam-based-chessboard-position-digital-recognition

[2] "Example of paid product." [Online]. Available: http://webcamchess.fr/

[3] J. H. Baptiste Wicht, "Camera-based sudoku recognition with deep belief network," *HES-SO, University of Applied Science*.

[4] "Opencv find chessboard corners." [Online]. Available: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#bool%20findChessboardCorners(InputArray%20image,%20Size%20patternSize,%20OutputArray%20corners,%20int%20flags)

[5] "Open source chess board detector algorithm." [Online]. Available: https://github.com/Elucidation/ChessboardDetect/blob/master/FindChessboards.py

[6] A. R. et al., "Open source computer vision (opencv-python) tutorials," 2017. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html

[7] "Tensorboard learning curve smoothing." [Online]. Available: https://github.com/tensorflow/tensorboard/blob/f801ebf1f9fbfe2baee1ddd65714d0bccc640fb1/tensorboard/plugins/scalar/vz_line_chart/vz-line-chart.ts#L55

[8] C. S. Sergey Ioffe, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *Google Inc.*, 2015.

[9] G. D. Illeperuma, "Using image processing techniques to automate chess game recording." [Online]. Available: https://www.researchgate.net/profile/Gayan_Illeperuma/publication/234491319_Using_Image_Processing_Techniques_to_Automate_Chess_Game_Recording/links/09e4150fe3be5bef54000000/Using-Image-Processing-Techniques-to-Automate-Chess-Game-Recording.pdf

[10] "Sobol filter." [Online]. Available: https://en.wikipedia.org/wiki/Sobel_operator

[11] "Canny edge detector." [Online]. Available: https://en.wikipedia.org/wiki/Canny_edge_detector#Gaussian_filter

[12] "Ramer douglas peuker algorithm." [Online]. Available: https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm

[13] "Finding countours with cv2." [Online]. Available: https://docs.opencv.org/3.1.0/d4/d73/tutorial_py_contours_begin.html

Appendices

0.1 Detailed Preprocessing Steps

1. Convert image to binary bitmap
2. Blur the image
3. Sobol Filter [10]
4. Canny Edge Detectors [11]
5. Find the contours [13]
6. Prune the contours—Ramer–Douglas–Peucker Algorithm [12]
7. Find line intersections
8. Sanity checks:
    - Board should be convex hull
    - Correct number of points found on board
    - Check angles between lines (should be close to 90 degrees)
9. Warp image with four-point transform to square shape

0.2 Final Model Architecture and Learning Curve

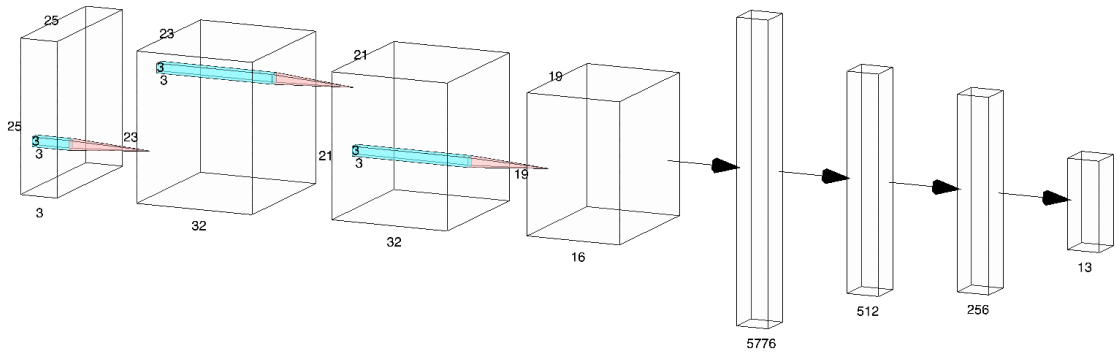


Figure 5: Architecture diagram of final model. Each one of the 64 tiles from the preprocessed board is run through this network.

The learning curve for our final model is as follows:

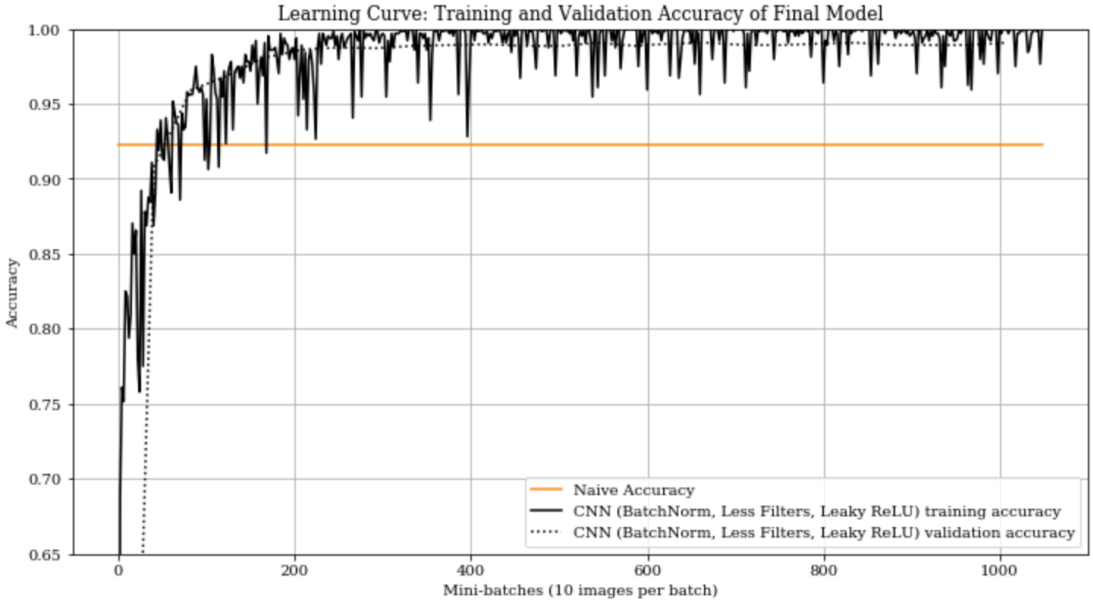


Figure 6: Learning curve for CNN with batch normalization. Note that the plotted validation accuracy is calculated periodically on a validation set, but is not used at all in training. The high stochasticity in batch accuracy is due to the small batch size (10) and small subset of images in which the model performs poorly.

0.3 Classifications During Training

The following figures illustrate how our final model improves at classifying examples from the validation set during the course of training:

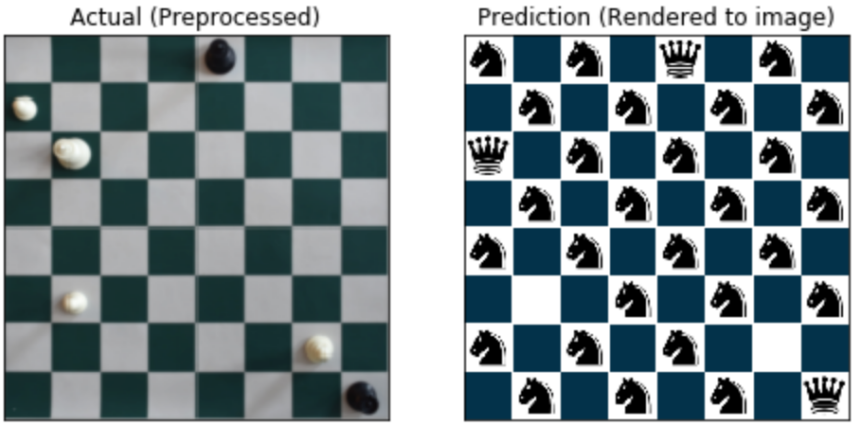


Figure 7: 48% classification accuracy on a random validation image after batch 10

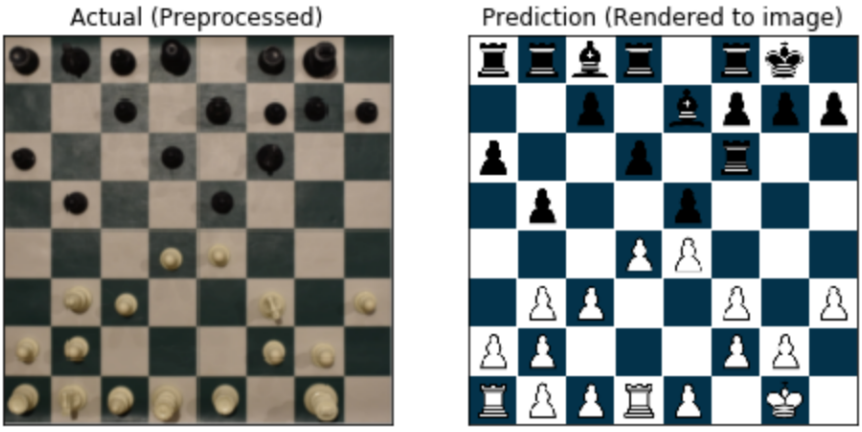


Figure 8: 86% classification accuracy on a random validation image after batch 42

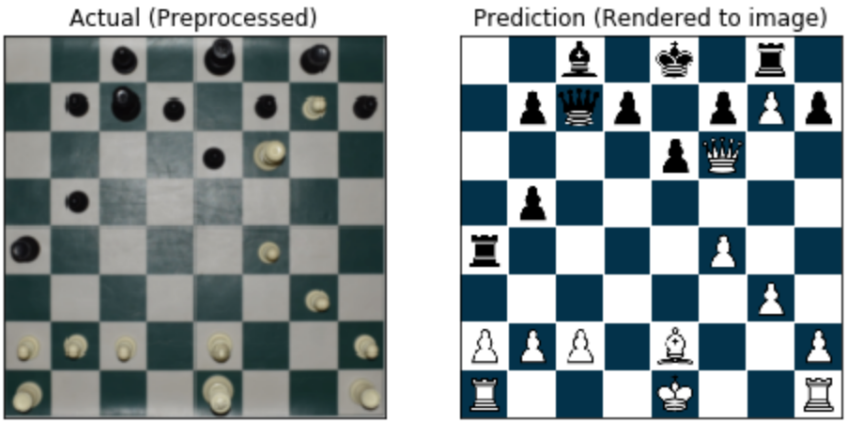
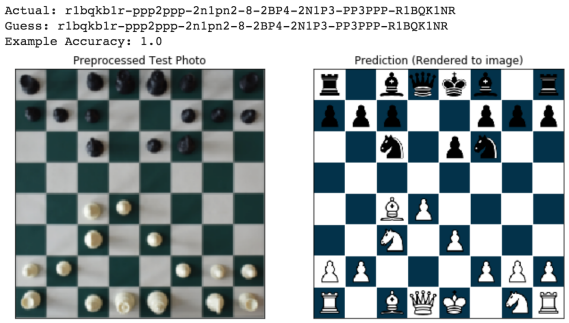


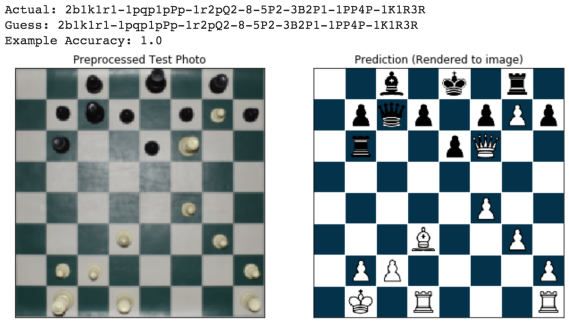
Figure 9: 100% classification accuracy on a random validation image after batch 254

0.4 Examples of Model Classifications on Test Images

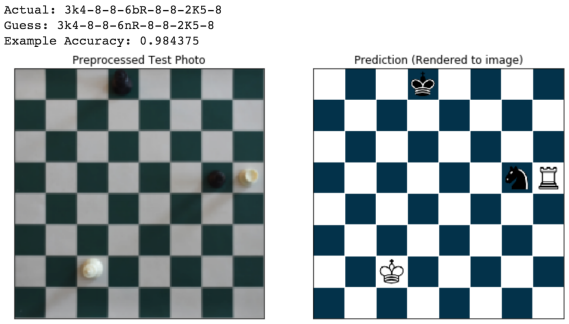
Final model classifications on eight selected images from our test set. For each example, the left image is the preprocessed image fed into the CNN and the right image is a rendering of the classification from the model (notated with FEN).



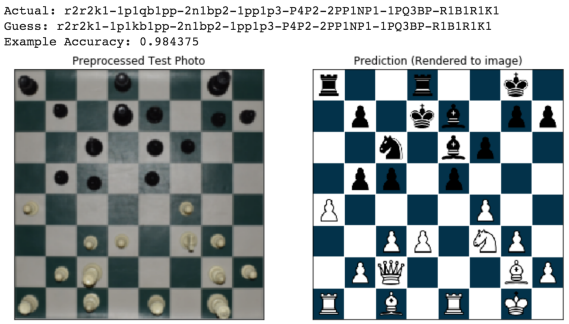
(a) Example test image 1:
flash off, perfect classification



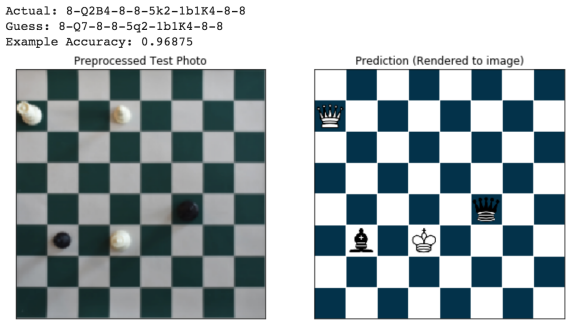
(b) Example test image 2:
flash on, perfect classification



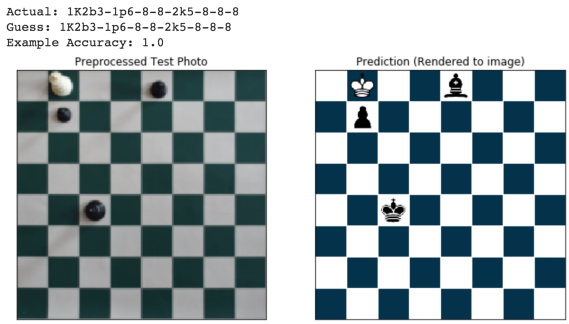
(a) Example test image 3:
flash off, 1 misclassified space



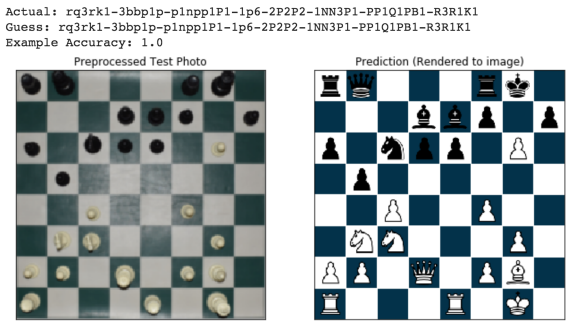
(b) Example test image 4:
flash on, 1 misclassified space



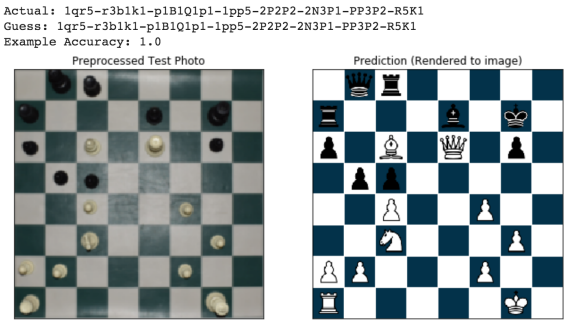
(a) Example test image 5:
flash off, 2 misclassified spaces



(b) Example test image 6:
flash off, perfect classification



(a) Example test image 7:
flash on, perfect classification



(b) Example test image 8:
flash on, perfect classification