CS230

ChessNet: Transcribing Chess Positions from Images

Adam Stanford-Moore
Department of Computer Science
Stanford University
sasm@stanford.edu

Hristo Stoyanov
Department of Computer Science
Stanford University
stoyanov@stanford.edu

Abstract

This work develops a model for turning an image of a chess board into a text transcription describing the locations of every piece, which can easily be fed into a chess engine (e.g. *lichess.org*). Our framework finds the location of the board in the input image, then crops all of the individual squares and feeds each into classifier, outputting the piece name (or "empty") for every square. Our classifier achieves an average F1 score of 0.94 across all classes and is able to transcribe a chess board with on average ≤ 1 mislabelled square.

1 Introduction

Chess game analysis is a critical component of improving chess playing. It is a common practice among amateur players to take photographs of the board positions during a live game in order to allow analysis with a chess engine afterwards.

Previously, chess players had to manually enter the locations of up to 32 pieces into a chess engine. We aim to build a neural network that can transcribe the position of a chess board from a photograph. The input to our algorithm is a color image of tournament style chess board with pieces. We then use image processing and a convolutional residual network to output a Forsyth-Edwards Notation (FEN) text representation of the board, which is the input to a chess engine.

More broadly our project is part of the growing field of object detection and classification. A future autonomous agent may have to play chess on a real board, and while the AI may have excellent chess playing abilities, it will also have to perceive the board and translate it into a format interpretable to its built-in chess engine. This work focuses on that translation task.

2 Related work

Previous work has taken a variety of approaches toward solving the same problem of transcribing an image of a chess board. Yang (7) (building on work by Buchner (1)) creates a dataset of images by cropping images of chess boards (more details in the *Methods* section). Yang then uses transfer learning to build a chess-piece classification model starting from the AlexNet model. We followed a similar approach to Yang although we create our own classifier.

Ding (3) sidesteps the cropping algorithm used by Yang and has a user manually pick the four board corners. Then Ding classifies pieces through use of a sliding window and using a model trained on features determined by a scale-invariant feature transform (SIFT) and histogram of oriented gradients (HOG). Piece color is determined via a ratio of pixel colors in the square. The author does not use neural networks.

CS230: Deep Learning, Spring 2019, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

Another approach by Czyzewski et al. (2) uses a similar line/corner detection system as Yang and then builds a convolutional neural network to assign probabilities for each possible class given a picture of a piece. Then, possible board configurations are evaluated by a chess engine and the most probable board position is selected. Using a chess engine improves accuracy over Ding and Yang, but takes extra time.

3 Dataset

The project started with a dataset collected and labelled by Yang (7) containing 10,545 images of pieces, pawns and empty squares. However, most of those images were empty squares and pawns leading us to collect and label an additional 5,400 pictures (refer to Table 1 and Figure 1). For every class from both data sets we shuffled and split into train, dev, and test sets in the ration 80/10/10.

	Yang		Our dataset		Train Set (19789)		Dev Set (1398)		Test set (1428)	
	white	black	white	black	white	black	white	black	white	black
Bishop	524	540	350	334	1255	1229	88	87	89	90
King	289	289	175	162	645	615	47	45	48	47
Knight	523	509	355	351	1266	1246	87	86	91	88
Pawn	1536	1521	233	210	1786	1717	176	173	179	175
Queen	377	375	304	306	1026	1031	68	68	71	70
Rook	321	280	366	376	1132	1123	68	65	71	68
Empty	3461		1878		5718		340		341	

Table 1: Overview of data used in training.

All of our images have resolution 224x224, which is the image resolution expected by the ResNet50 network. Pre-processing is done with the ResNet50 *preprocess_input* function that shifts the color channels to BGR and zero centers the color values with respect to the ImageNet dataset without scaling.

Our data collection effort was guided by results showing that fewer images within a class caused it to have a lower F_1 score, where both precision and recall would be worse relative to a class with more images. As a result, when building off Yang's data we collective more pictures of rooks, queens, kings, knights, and bishops to balance the classes. To quickly label images we trained a model on Yang's set then used it to crop and sort squares into a directory structure, which we then reviewed and corrected. We then further increased the data set by augmenting our collected data with 90^o rotations and horizontal flips.

(a) white bishop (b) black queen (c) white king (d) black rook (e) Bad Crop

Figure 1: Examples of images we collected. Image (e) is the result of a bad crop with unclear correct label; therefore it was omitted from the dataset.

4 Methods

Since our limited dataset did not allow an end-to-end approach, we divided our problem into two tasks: cropping board squares and square classification. The two parts are equally important, but we focused our energy on the second classification task. Our two-part approach follows the method developed by Daylen Yang (7) and Matt Buchner (1). Our cropping task builds directly on Yang's open-source implementation.

4.1 Cropping Board Squares

The algorithm for the first task takes in a raw color photograph of a chess board, finds all of the squares in the image and returns an ordered array of all of the cropped squares where each location in the array corresponds to a location on the chess board. The algorithm works through a process of 1) Canny Edge detection, 2) Hough line detection and corner detection, and 3) geometric transformation and cropping.

First, the OpenCV implementation of Canny edge detection is applied to the raw photograph. The algorithm smooths the photo with a Gaussian filter to remove noise then finds the intensity gradients of the image and applies non-max suppression to remove faint edges. Lastly it removes weak edges that are not connected to stronger edges. Next the edged image is fed through a Hough line detection algorithm (OpenCV) to detect straight edges. The intersection points of these lines are then found and clustered to determine the the four corners of the board. Then, using the four corners, the board itself is cropped and transformed into a square shape, removing any background. If this transformation is done accurately, then each chess square is equal in size and all 64 squares fully cover the transformed image. Lastly the transformed board is cropped into 64 equally sized smaller images containing one square each.

This process worked well with boards on a uniform background. However, if the edge of the table were visible in the image, then the corner detection and transformation could be thrown off and the cropped images might not contain just a single square (see last panel of Fig. 1). The presence of other objects in the picture next to the board significantly decreases the effectiveness of this method.

Figure 2: The steps converting an input image to the cropped squares. Canny Edge detection was used before Hough line detection but was omitted for brevity.

4.2 Square Classification

During this task each of the 64 cropped squares was provided as input to a classifier to determine which piece was on the square or if it was empty. We used transfer learning starting with a pre-trained ResNet50 model (4) and explored different modifications to the architecture.

During our first attempt, we removed the last layer of ResNet50 (corresponding to a fully-connected layer with 1000 outputs) and substituted it with three fully-connected layers: two of 512 nodes each and a final FC layer with 13 outputs (one per class). We used ReLU activation functions for the first two layers and a softmax activation function for the last layer since we defined the problem to expect that each picture would have only a single image of a piece.

This architecture would severely overfit our training data, which at this point was limited to only a few hundred images per class. Within a few epochs, we would get a training loss of virtually 0 and 99.9% accuracy with no further progress possible. In contrast the dev set performance would be limited to 60 - 80%, mostly fitting to the empty squares that dominated the dataset. In order to reduce such high variance we tried to add regularization by adding dropout to the fully-connected layers, but that had almost no effect. By decreasing the number of trainable layers, we could decrease the variance, but then the accuracy of the dev set decreased.

Further, we attempted to add a single fully-connected layer and attempt to re-train more of the layers of the convolutional network. Our final model replaces the ResNet50 final fully-connected layer of 1000 nodes with a single fully-connected layer of 13 nodes and uses a softmax activation function. The layer is initialized with a truncated normal distribution with mean 0 and a standard deviation of 0.05 (truncated at 2 std dev).

Our loss function is the sparse categorical cross-entropy, where we added a weight vector $w_c = 1.2$ for bishops and kings, the most difficult pieces to classify.

$$J = \frac{1}{m} \sum_{o=1}^{m} \sum_{c=1}^{K} y_{o,c} \log(p_{o,c})$$

Where $y_{o,c}$ has value 1 only for the true class label for that data set and $p_{o,c}$ is the probability predicted by the network for that class label.

5 Experimental Results & Discussion

Using our modified version of the ResNet50 we tuned the number of trainable layers, the batch size, and the number of epochs.

Layers: Our best result was achieved by re-training 7 convolutional layers. This achieved an average score of $F_1 = 0.94$ across classes on the test set. Training more layers produced seemingly better F1 scores (see Table 2), these performed worse on actual boards and were very sensitive to the edges of other pieces visible in the image.

Batch sizes: We found that the batch size does not significantly affect the performance of the algorithms. Small batch sizes, e.g. 16, would lead to an unstable loss function, as one class might be too prominently featured. We found that batch sizes of 128 or 256 worked fairly well. When trying to use 256 batch size and training 33 convolutional layers, we found that the hardware we were renting was insufficient and we would get out-of-memory errors. Over all, we attempted to use batch sizes of 32, 64, 128, 256 and 512. Given the hardware we were using (G3S.xlarge instance on AWS), batch sizes of 512 could not be proccessed.

Optimizer: We used Adam with the default parameters described in the original paper (5). Using the default parameters of learning rate of 0.001, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ provided a fast learning environment. Hence we did not try to adjust those.

Error analysis of best model:
We observe that training more layers corresponds to higher F_1 scores. However, these models often overfit which resulted in many mistakes when given a new board. Table 2 shows a comparison of the models we have.

board mistakes	$F1_{Dev}$	$F1_{test}$	Trained Layers	Batch Size	Epochs	M_{train}	$M_{dev/test}$
5.2	0.934	0.944	7	256	20	∼17323	∼1295
5.6	0.955	0.954	14	256	30	∼19789	∼1398
9.8	0.983	0.984	33	128	20	∼19789	∼1398

Table 2: Comparison of model parameters. Board mistakes are average over 5 boards the model has not seen before.

Figure 3 shows a run of the model on a picture that contains the initial position. This is done with the model that achieves $F_1 = 0.944$. However, the model achieving $F_1 = 0.98$ on our dev and test sets makes many mistakes, recognizing empty squares as black nights.

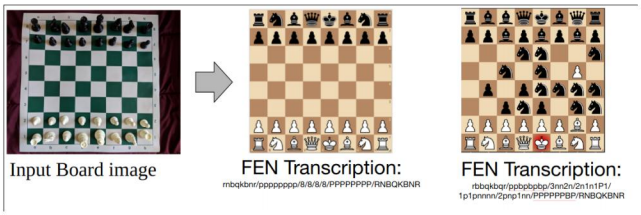

Figure 3: Perfect transcription of the initial board position produced by model with $F_1 = 0.94$ (left). Right transcription done with model $F_1 = 0.98$, indicating over-fitting to the dev set.

Figure 4 shows a confusion matrix for the best model and a few of the misclassified pictures. As noted by Ding (3), bishops and pawns seem to be the hardest to classify. Queens and kings are often classified accurately if the whole piece is visible since they have distinct features, but in images where the tops are cut off (for instance the king's cross), these pieces are often confused.

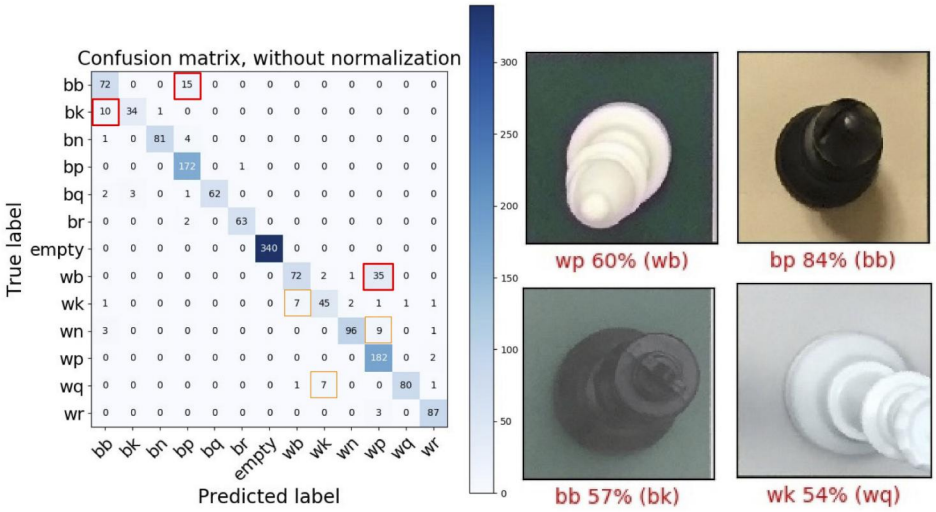

Figure 4: Model sometimes confuses bishops,pawns and kings.

One of the issues we encountered throughout this project was that the loss converged to almost zero very fast (usually within 1 epoch), likely due to our limited dataset. However, the training loss would continue to decrease after multiple epochs and then reach a steady state. This is observable in Figure 5.

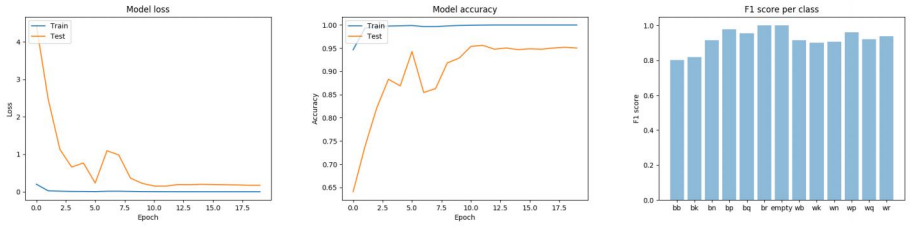

Figure 5: Loss, accuracy, and F1 scores for the best model (Average F1 = 0.94).

6 Conclusion/Future Work

We were able to partially replicate some of the results claimed by Yang (7). Further, we achieved an ability to recognize a complete board close to the results achieved by Czyzewski (**?**) using only computer vision models and without employing further analysis of the possible options.

In conclusion, our best model achieved an average F1 score of 0.94 on the test set and was able to accurately transcribe chess boards on average with ≤ 2 mislabelled squares per board. Such accurate transcription was contingent upon a good cropping completed in the first half of the model, since a badly cropped square was impossible to classify. Our model and data are available at `https://github.com/renegat96/chess-vision`.

Currently the size of public datasets of transcribed boards and labelled images of chess pieces is prohibitively small for more sophisticated algorithms. With more data we could extend our model to classify different styles of chess pieces on different styles of boards taken at different angles. With a large enough dataset of transcribed boards we could also develop an end-to-end neural network model similar to the YOLO model (6) that could identify and classify images at the same time.

7 Contributions

Both authors worked closely together on all tasks associated with the project.

8 Acknowledgements

We would like to thank our TA Sarah Najmark for constant feedback and support. Also thank you to Professors Ng and Katanforoosh for teaching us.

References

[1] Buchner, M. and profile, V. (2011). *Chessboard picture recognition project - part 1*. [online] Codebazaar.blogspot.com. Available at: http://codebazaar.blogspot.com/2011/08/chess-board-recognition-project-part-1.html [Accessed 7 Jun. 2019].

[2] Czyzewski, Maciej A. *An Extremely Efficient Chess-board Detection for Non-trivial Photos*. 2017. http://arxiv.org/abs/1708.03898

[3] Ding, Jialin. *ChessVision: Chess Board and Piece Recognition*. 2016. https://web.stanford.edu/class/cs231a/prev_projects_2016/CS_231A_Final_Report.pdf

[4] He, Kaiming et al. *Deep Residual Learning for Image Recognition* 2015. https://arxiv.org/abs/1512.03385

[5] Kingma, Diederik P. and Jimmy Ba. *Adam: A Method for Stochastic Optimization* 2015. https://arxiv.org/abs/1412.6980

[6] Redmon, Joseph and Santosh Kumar Divvala and Ross B. Girshick and Ali Farhadi *You Only Look Once: Unified, Real-Time Object Detection*.CoRR 2015. http://arxiv.org/abs/1506.02640,

[7] Yang, Daylen. *Building Chess ID*. 2016. https://medium.com/@daylenyang/building-chess-id-99afa57326cd

[8] Yang, Daylen. *Chess ID*. 2016. https://github.com/daylen/chess-id

[9] Chollet, François et al. *Keras*. 2015. https://keras.io

[10] Pedregosa, F. et al. *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research. 2011