Using several pre-trained Convolutional Neural Networks for Object Dectection and Classification

Final Project Report of CS 460

ChenHaoyang, 1220022991;

September 2024

Shool of Computer Science and Engineering, Macau University of Science and Technology

---

1 Abstract

In this report, We focus on some issues concerning Chessboard detection, mainly using some convolutional neural network, a.k.a. CNN, Based on CNN, We're trying to convert some images of chessboard positions to a specific chessboard notation, Forsyth-Edwards Notation, a.k.a. FEN. After searching for the previous approaches to dealing with this kind of problem, we find that earlier researchers have always used techniques based on feature extraction, we propose a pipeline three-stage model: Form board edges detection to pieces Recognition and detection. Finally, we classify every square of the chessboard, identifying whether every square is occupied or not. While solving this detection and conversion problem, we explore different CNN architectures, including ResNet34 and DenseNet. And there is a better result compared with the previous work. On a blender dataset including more than 4500 images of different angles and light conditions of game images, our result is only with an average mistake count of less than 2 mistakes per chessboard. We also explore the minimized data size we need to optimize our model nearly to the best we can achieve and whether the model performs well in other chessboard datasets.

2 Introduction

2.1 Problem Statement

In this section, we start with the basic concepts and the core modules of the system, then state our problem formally.

A.Basic Concepts

Forsyth–Edwards Notation(FEN): It is defined as a standard notation for describing a particular board position of a chess game. And we use it as the output of our system.

Square: It is defined as a small section of the chessboard. As a part of our task, we need to separate the whole chessboard into 64 squares to generate the occupancy detection dataset.

FEN-Conversion system(FENC): Given an image dataset of the chessboard, the system could use different

vision models to finish the task of square detection, Occupancy Determination, and piece classification. And finally, output the FEN of the specific chessboard, which enables this analysis via chess engines.

B.Problem Definition

We next formally define the FEN Conversion problems.

Problem 1 (FEN Conversion from chessboard image): We formulate our problem by taking an image of a physical chessboard position and using and finetuning various CNN architectures (DenseNet, EfficientNet, ResNet, etc.) to output a corresponding FEN string.

2.2 System Overview

In this section, we mainly explore how to build the FENC system for our specific work. Our system can be modeled as a multi-layered pipeline. The FENC system consists of three main modules, which are the board detector, occupancy detector, and piece classifier. The raw dataset is input into the system, and the system outputs many labeled pieces with different classes.

Then, we use a mapping algorithm between the pieces' location and their classification to generate the corresponding FEN. We evaluate our system on a 3D engine-generated dataset of nearly 5,000 chess positions.

During the evaluation, We experiment with different open-source models for each subtask, including OpenCV for square detection and DenseNet, EfficientNet, and ResNet for occupancy and piece classification.

We also explore further evaluation considerations, such as convergence speed and transfer-learning capabilities of each model to different datasets.

The source code and dataset for this study are publicly available at https://github.com/HaoyangChen23/CS460_FinalProject.

3 Related Work

Some related the generation of FEN of a chessboard have been shown in [6][7][12], They compared the standardized form of online chessboard images with noisier physical positions. Our exploration of chess vision is limited to the range of physical chess positions. FEN-conversion from chess board images is not a new problem; it was solved by Bennett et al.[2] in 2012 using mathematical algorithms to feature engineer a model to extract the edges of squares. Based on the development of neural networks, modern approaches tried to train integrated models to solve this problem. [4][9].

Our FENC system divides the whole task into several subtasks. We train and finetune each stage, and then we integrate all submodules into one framework of this FENC system. For each stage of our work, there exist different approaches in the previous literature.

For board detection, Czyzewski et al. [4] use straight line and lattice-point detection to segment the chess board into its 64 squares. However, their approaches suffered from the limitation of low speed of generation under the different conditions of light and angle. Abeles [1]and Chen et al. [3] attempted to improve using a blur-reduction algorithm or X-corner detection.

For the occupancy detection problem, we define it as a Biclassification issue, using some pre-trained object detection models such as DenseNet, which has been shown to perform well on ImageNet [5][14].

In processing the dataset generated by the occupancy detector, we use the same approach as the occupancy detector used to solve the chess piece classification problem. Some previous models successfully solved this problem, such as Quintana et. al. [11]and Alex et. al.[13]. We offer our approaches by using some different deep learning architectures in our intermediate steps of constructing our model, optimizing the performance of accuracy and ability of transfer learning.

4 Methods

4.1 Board Detection

We choose an object detection model trained for implementation in Masouris et al. [9] as our baseline for board detection and will compare our results against the results shown in the paper. The paper explores a ResNeXt architecture for extracting the features of a chessboard, and then they fed it into a detection transformer (DETR). For what we do, we first run the Scharr operator for edge detection, which is an improved version of the Sobel operator. The algorithm is as follows:

1. Convert the image to grayscale and compute the derivatives in x and y directions using the Scharr operator, which uses the following 3×3 kernels:

   For x direction:

   $$f_x = \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

   For y direction:

   $$f_y = \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix}$$

2. Calculate the magnitude of the gradient as shown:

   $$|\nabla f(x,y)| = \sqrt{f_x^2 + f_y^2} \tag{1}$$

3. Convert the gradient magnitude to uint8 format using absolute value scaling to ensure proper normalization of edge strengths.

4. Apply adaptive thresholding with a Gaussian window to obtain binary edges. This approach automatically adjusts the threshold based on the local image region, making it more robust to lighting variations across the chessboard.

To further separate the board's edges from any edges detected from images, we perform LSD(Line Segment Detector) for its ability to pick out the lines we need from the result of edge detection. We explain the process as follows:

1. First, compute the image gradient orientation at each pixel (x,y):

   $$\theta(x,y) = \arctan\left(\frac{f_y}{f_x}\right) \tag{2}$$

where f_x and f_y are the image derivatives in x and y directions.

2. Then we scale the image by a factor of 0.8, and sort all the pixels by their y their gradient magnitude in descending order.This ensures that pixels with stronger gradients are processed first.

3. Starting from pixels with highest gradient magnitude, we group aligned pixels into line-support regions.A pixel is added to a region if its gradient orientation differs from the region's main angle by less than 22.5°.And the region grows until no more aligned pixels can be added.

4. For each line-support region, we compute the center of mass and principal inertia angle and fit a rectangle with width determined by the region's size. The rectangle's length and width define the line segment.

5. Last, we do the Line segment validation, which means we compute the the Number of False Alarms (NFA) by using follow equations:

$$\text{NFA} = N_{\text{tests}} \cdot P_{\text{false}} \tag{3}$$

Where:

- N_{tests}: The number of tests, representing the number of hypotheses (e.g., possible line segments).
- P_{false}: The probability of a false positive, computed as:

$$P_{\text{false}} = \binom{N_{\text{aligned}}}{k} \cdot p^k \cdot (1-p)^{N_{\text{aligned}}-k} \tag{4}$$

  Here:

  - N_{aligned}: Number of aligned pixels.
  - k: Number of pixels that align with the detected line segment.
  - p: Probability of a pixel aligning with the line segment by chance.

  The detection is statistically significant if:

$$\text{NFA} < 1 \tag{5}$$

After detection, we keep only segments within 30° of horizontal or vertical. This specifically targets the chessboard's grid structure

However, even though we use LSD to detect these lines, it is hard to classify what we need due to noise during processing. To continue detecting the right lines, we divide the lines into vertical and horizontal and then perform agglomerative clustering of the lines produced by the Hough transform, which are presented as polar coordinates to make grouping under different angle conditions with the help of SciKit-Learn library functions [10]. Then, we deal with each cluster of lines and take the median value of them as the representation of each cluster.

For further filtering the lines that remain, we generate all the intersection points and use them to run the RANSAC algorithm to sample points. Then, we derive a transformation matrix to give us a view of the board from a "birds-eye" perspective. It's an iterative process, we randomly sample a set of points and use a transformation matrix to project these points into a 2D view. At each iteration, we keep track of and update the best transformation matrix achieved across the trials. Transformation helps us detect any missing points and edges from our collection and gives us an easy method to segment each of the 64 squares to generate the pre-processed dataset for occupancy detection and piece classification.

4.2 Occupy Detection

For occupy detection, We sample the area around the square and pass it into an occupancy detector that helps us determine whether it's occupied or not. To do so, we have implemented four approaches. The baseline CNN architecture is CNN100 with three convolutional layers, three pooling layers, and three fully connected layers, with the final fully connected layer being the classifier head with two classes representing either empty or occupied. The second model is a ResNet34 model[8] pre-trained on ImageNet and finetuned on our occupancy data, with the classification head adapted to predict one of two classes. The third architecture uses DenseNet pre-trained on ImageNet; we finetuned the model's classification head in the same way as the ResNet model.

4.3 Piece Classification

If the square is determined as occupied by the occupancy detector, we pass these data to piece classifier to determine the color and type of piece. For constructing our piece classifiers, we choose the same three model architectures used in the occupancy detector on the square. But what we need to adjust in our model is the classification heads of the ImageNet-pre-trained models we need 12 classes to finish this subtask. Because there are twelve different (color, piece) tuples for chess game. By mapping all the pieces to an identifier, square, color, and type, we easily create a FEN representation of the board from this information set.

5 Experiments

5.1 Preprocessed Dataset

In our experiment, we chose a dataset containing 4,888 images (1200 x 800) of the chess game states hosted on the Open Science Framework site.[15] The title of the dataset we selected is "Rendered Chess Game State Images." The reason why we chose this dataset is the board positions were created under various orientation and lighting conditions, which gave us a chance to make our FENC system more robust to improve performance in different data environments.

For training our two modules of the system, we need to preprocess the raw dataset to generate a Scenario-specific dataset. Using our board detector, we generate the occupancy detection dataset by cropping a small section around each square and tagging it with whether it is occupied or not by referencing the ground truth FEN notation from the original dataset.

Similarly, we generate the piece classification dataset by cropping each occupied square and tagging it with the correct piece label of (color, piece) by referencing the ground truth FEN notation.

5.2 Occupancy Test

Our occupancy classifier was trained on the dataset preprocessed by our board detector for 20 epochs, a learning rate of 0.0005 with ReduceLROnPlateau scheduling (factor=0.5, patience=3), and a batch size of 64. These parameters were determined through a systematic grid search on the validation set. We employ binary cross entropy loss with label smoothing (smoothing=0.1) and use the AdamW optimizer with a weight decay of 0.0001. We created three occupancy models, each with a different CNN architecture described in our methods section (CNN100, ResNet34, and DenseNet). Training time for these three occupancy classifiers took 16 minutes, 1.2 hours, and 50 minutes, respectively on a single NVIDIA A100 Tensor Core GPU.

5.3 Segment Classify Test

For the more complex piece classification test, we enhance our training strategy while maintaining dataset consistency with the occupancy detector. But we trained our piece classifier with modified hyperparameters: 30 epochs, a learning rate of 0.001 with cosine annealing warm restarts ($T_0 = 5$, $T_{\text{mult}} = 2$), and a batch size of 128. We use standard cross entropy loss with label smoothing (smoothing=0.1) and use the AdamW optimizer with a weight decay of 0.01. Our three piece classifiers are also CNN100, ResNet34-finetuned, and DenseNet-finetuned. Training time for these three piece classifiers took 35 minutes, 1.5 hours, and 1 hour, respectively on a single NVIDIA A100 Tensor Core GPU.

5.4 Optimize Data Size

We also found that the occupancy and piece classifiers demonstrated strong validation set performance only a few epochs into training. So we do an experiment to test training jobs of our occupancy detector and piece classifier (same hyperparameters) for the ResNet34 and CNN100 but with stratified subsets of the training data of increasing sizes (1,000, 3,000, 5,000, and 10,000 examples). We write down all the records of training at each checkpoint to evaluate the performance if only a small fraction of the data is used. It also proves to us that the ImageNet data may help the pre-trained model converges faster on our dataset.

6 Resluts and Evaluation

We give three evaluation metrics for each subtask of our three-stage process.

For the board detector, we used pixel-wise Euclidean distance between ground-truth labels for the corners of the board and our detector's predicted locations.

For the occupancy detector and piece classifier, we use the accuracy and weighted F1 of their predictions to evaluate them as independent model components. Due to the imbalances of different class sizes, especially in the piece classification subtask dataset, which has way more pawns than queens, we chose to measure a weighted f1.

When evaluating our FENC system, we mainly focus on the average number of mistakes made per board. For further analysis, we also record the number average number of occupancy mistakes per board and the average number of piece classification mistakes per board.

6.1 Board Detection

Table 1 shows the comparison between the performance of our board detector model and the ResNeXt baseline model from Masouris et. al. [9].Our results, such as the average pixel error from ground truth and percentage of boards with no mistakes, generated by iterative transformation metrics are way better than the baseline solution we selected.

6.2 Occupancy Detection

Every model we chose demonstrates strong performance on the occupancy subtask. The strongest performance is posted by ResNet34, but it has the longest inference time, where the ResNet34 model takes nearly twice as long as

Table 1: Board detector evaluation of our approach vs ResNeXt baseline

Metric	Our Approach	ResNeXt Baseline
Mean incorrect squares per board	**0.12**	1.19
Boards with no mistakes (%)	**99.79%**	65.20%
Boards with ≤ 1 mistake (%)	**96.57%**	39.76%
Per-square error rate (%)	**0.17%**	1.86%
Average pixel error from ground truth	**1.27**	22.3

the CNN100 model. So it's trivial to choose which models we use, due to the relatively easy subtask of determining the situation of occupancy.

Table 2: Performance comparison of different model variants on occupancy subtask dataset test set

Model Type	Accuracy	Weighted F1
DENSENET	99.67%	99.67
CNN100	97.63%	97.31
RESNET34	99.96%	99.94

6.3 Piece classification

Similar to Table 2, Table 3 demonstrates that all model architectures exhibit strong performance on the pieces test set. However, there is a notable improvement between the pretrained CNN architectures (ResNet34 and DenseNet) compared to the CNN100, with a difference of about F1.

Table 3: Performance comparison of different model variants on pieces subtask dataset test set

Model Type	Accuracy	Weighted F1
DENSENET	99.89%	99.84
CNN100	97.32%	97.13
RESNET34	99.93%	99.95

6.4 System Results

The several parts of FENC system are seen as a fully-connected model, we examined the output of this system on the test set of full chessboard positions.

Table 4: Average number of mistakes per chessboard for different models on Blender test set

Model	Total Mistakes	Occupancy Mistakes	Piece Mistakes
DENSENET	1.73	0.15	1.58
CNN100	1.63	0.39	1.24
RESNET34	1.61	0.16	1.45

Observing the results of the previous sections, we find that the end-to-end model results are close between the three variants. The average number of mistakes is around 1.7 for each model, but the distribution of mistakes is uneven. In particular, the models all suffer from more piece identification errors than occupancy detection mistakes. This suggests that piece classification is harder for the models to perform, which aligns with our intuition, since certain chess pieces often tend to be mistaken for one another (king and queen, for instance). It is worth noting that

the ImageNet- pre-trained ResNet34 and DenseNet models performed better on occupancy detection, but worse than the CNN100 in piece detection.

7 Conclusion

Our objective is to automate the generation of a chessboard position from an image by dividing the task into three phases: board detection, square occupancy detection, and piece classification. We employed classical methods for board detection and trained various deep learning models for occupancy detection and piece classification. For board occupancy, ResNet34 demonstrated the highest prediction accuracy, although the margin over DenseNet and a CNN baseline was not significant. For piece classification, ResNet and DenseNet outperformed the simpler CNN. The results of the whole system showed strong classification performance, with all models averaging fewer than two total mistakes per board. However, more errors were shown in piece classification, with each model averaging fewer than 0.5 occupancy mistakes per board. Further analysis revealed that the ResNet model did not require training on the full dataset to achieve optimal performance, with performance tapering off when overtrained. With more time, we aim to explore transfer learning on larger and more diverse chessboard datasets to study the effects of catastrophic forgetting. Additionally, with greater computational resources and data, we plan to experiment with transformer architectures like CLIP and ViT to compare their performance against CNN-based models.

References

[1] Abeles, P. (2021). Pyramidal blur aware x-corner chessboard detector.

[2] Bennett, S. and Lasenby, J. (2013). Chess - quick and robust detection of chess-board features. *CoRR*, abs/1301.5491.

[3] Chen, B., Xiong, C., Li, Q., and Wan, Z. (2023). Rcdn – robust x-corner detection algorithm based on advanced cnn model.

[4] Czyzewski, M. A., Laskowski, A., and Wasik, S. (2020). Chessboard and chess piece recognition with the support of neural networks.

[5] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255.

[6] Ding, J. (2016). Chessvision : Chess board and piece recognition.

[7] Feng, X., Luo, Y., Wang, Z., Tang, H., Yang, M., Shao, K., Mguni, D., Du, Y., and Wang, J. (2023). Chessgpt: Bridging policy learning and language modeling.

[8] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition.

[9] Masouris, A. and van Gemert, J. (2024). End-to-end chess recognition. In *Proceedings of the 19th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*. SCITEPRESS - Science and Technology Publications.

[10] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830.

[11] Quintana, D. M., del Barrio García, A. A., and Matías, M. P. (2020). Livechess2fen: a framework for classifying chess pieces based on cnns.

[12] Saha, S., Saha, S., and Garain, U. (2024). Valued – vision and logical understanding evaluation dataset.

[13] Shan, A. and Ju, B. (2025). Chessboard understanding with convolutional learning for object recognition and detection. Stanford University.

[14] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015). Rethinking the inception architecture for computer vision.

[15] Wölfflein, G. and Arandjelović, O. (2021). Dataset of rendered chess game state images.