Chessboard Understanding with Convolutional Learning for Object Recognition and Detection

Alex Shan
Stanford University
azshan@cs.stanford.edu

Brent Ju
Stanford University
brentju@cs.stanford.edu

1. Abstract

We present a convolutional neural network (CNN) based approach for converting images of physical chessboard positions to Forsyth-Edwards Notation (FEN). While historical methods have used traditional feature extraction or attempted to train a model end-to-end, we propose a three-part model: a board detector, an occupancy detector for each square, and a piece classifier. By training these components independently and tying them together, we evaluate their performance on a 3-D Blender generated dataset of nearly 5,000 game states. We also explore different CNN architectures, including ResNet and InceptionV3. We post impressive results on the Blender dataset, with an average mistake count of just 1.7 mistakes per board. We also explore additional questions regarding training size and transfer learning, demonstrating that our approaches do surprisingly well with small amounts of data and even generalize well to other chessboard datasets.

2. Introduction

In this work, we attempt to build a computer vision system that is capable of converting images of physical chess boards and pieces to Forsyth-Edwards Notation (FEN). FEN is a compact representation of a chess position used to communicate and analyze the state of a chess game, describing the location of each piece on the board. It is challenging for existing systems to analyze the state of an over-the-board chess game, motivating a FEN-conversion system which enables this analysis via chess engines. To accomplish this goal, we construct a model, internally using three different vision models, to detect squares, determine the occupancy of each square, and finally classify which piece belongs in each square. We experiment with different open-source methods for each subtask, including OpenCV for square detection and ResNet and InceptionV3 for occupancy and piece classification. After these three stages, we use a mapping algorithm between piece location and piece classification to systematically generate FEN for the board position. We train our model variants on a 3-D visual engine-generated dataset of nearly 5,000 chess positions. During evaluation, we compare our model performance against a baseline 3-layer CNN approach. We find that while all CNN architectures perform well, achieving an average error count of around 1 per image, examination of subtask performance demonstrates slight advantages of the InceptionV3 and ResNet models. We also discuss further evaluation considerations, such as convergence speed and transfer-learning capabilities of each model to different datasets.

2.1. Problem Statement

We formulate our problem as taking an image of a physical chess board position and using various CNN architectures (ResNet, InceptionV3, etc.) to output a corresponding FEN string.

"r1b1qrk1/p1p3pp/1bpp1n2/6B1/4P3/3N4/PPP1NPPP/R2Q1RK1"

3. Literature Review

A key distinction to clarify is that this work deals with FEN generation for over-the-board positions, not screenshots of online chess positions. These are distinct problems in chess vision which are tackled in different ways due to the standardized form of online chess positions compared to noisier physical positions as shown in [6, 8, 15]. Therefore, when using the term chessboard image, we refer to an image of a physical chess position. FEN-conversion from chess board images has been handled with approaches dating back to 2012, where Bennett et. al. [2] use mathematical algorithms to feature engineer a model to extract the edges of squares. Modern approaches center around using neural networks to train end-to-end models to recognize chess boards and chess pieces to generate FEN [4, 11]. Our three-stage approach of square detection, occupancy detection, and piece classification differs in that we use open-source

models and finetune each stage separately before integrating all three to complete our model. Within each subcategory, different approaches have been taken in existing literature. For square detection, Czyzewski et. al. [4] use straight line and lattice-point detection to segment the chess board into its 64 squares. However, these approaches can be limited by lighting and angle distortions and are generally slow [4]. Attempts have been made to improve in these aspects, such as a blur-reduction algorithm by Abeles [1] and X-corner detection by Chen et. al. [3]. We can formulate the occupancy detection problem as classifying the object inside of a cropped image of a chess square, enabling the use of object detection models such as Inception V3, which has been shown to perform well on ImageNet [16, 5]. Lastly, once occupation detection is complete, chess piece classification can be formulated as a general classification problem with a class for each (piece, color) tuple. Image classification has significant overlap with object detection models, leading us to use the same model architectures for each subtask. There have been examples of previous two and three-stage models having success in this problem, such as Quintana et. al. [13]. We still offer novel research as previous multi-stage models have not used new deep learning architectures in their intermediate steps, opting for the earlier feature extraction-based approaches. Another key difference in architecture is that some current approaches attempt to predict the most likely board state given the model's understanding of piece placement, using knowledge of what popular positions look like to inform prediction [4]. However, we attempt conversion without using prior database information to influence the model's decision-making, relying purely on the model's understanding of square locations and piece recognition.

4. Dataset

4.1. Blender Dataset

The dataset we are using is titled "Rendered Chess Game State Images," hosted on the Open Science Framework site.[17] The dataset contains 4,888 images (1200 x 800) of chess game states that occured in real games played by Grandmaster Magnus Carlsen, recreated and rendered using Blender software. A large factor as to why we selected this dataset is because the board positions were created under various orientation and lighting conditions, which will allow us to train a robust model that can handle images from various different environments. The data are also very well annotated with training labels: each image is associated with a JSON object containing data fields such as FEN notation, camera orientation, board coordinates, and piece information. Each piece is also labelled with its type, color, square location, and a tuple of data that represents a bounding box for the piece in the given image. For each subtask model such as the occupancy detector and piece classifier, we preprocess the raw dataset to generate a task-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 raw dataset. Similarly, we generate the piece classification dataset by cropping each occupied square and tagging it with the correct piece label of (color, piece), e.g. (white, pawn) by referencing the ground truth FEN notation. Our train/validation/test splits of the dataset were generated using a 85/5/15 split, respectively. After processing the original dataset, the occupancy subtask dataset had over 300,000 total examples (100 x 100 resolution images) and the piece classification subtask dataset had around 30,000 examples (100 x 200 resolution images). Some examples of full test images, as well as the processed occupancy and piece classification samples are shown below.

Figure 1. Sample image from test set on entire board.

Figure 2. Sample occupancy subtask example showing an occupied square.

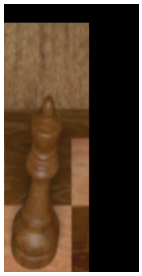

Figure 3. Sample piece classification subtask example showing a black queen.

4.2. Transfer Learning Dataset

We were also motivated to examine how well our models trained on the Blender dataset would generalize to unseen examples, so we use a handcrafted dataset from Wolflein et. al. [18], which contains photos of chess board positions that do not resemble the Blender dataset. We apply the same preprocessing strategy as in section 3.1 to generate our subtask finetuning data. Ultimately, we have an intentionally miniscule finetuning training set of 2 positions, with 30 test positions to evaluate on. Our processed subtask datasets have 128 occupancy examples and 64 piece examples for the training set.

Figure 4. Sample image from transfer learning dataset.

5. Methods

Our proposed model will be comprised of a three step pipeline. **As a baseline** for board detection, we will compare our results against a object detection model trained for implemented in Masouris et. al. [11]. The paper utilized a ResNeXt backbone for feature extraction of an image, which is then fed into a detection transformer (DETR). For our implementation, we run the Canny edge detector [14] on the image to extract all edges. The algorithm is as follows:

1. Suppress noise in the image and compute the derivatives of the grayscale image in x and y directions by applying a $3x3$ Gaussian filter.

2. Calculate the magnitude and direction of the gradient as shown:

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

$$\theta = \tan^{-1}\left(\frac{f_y}{f_x}\right)$$

3. Round all gradient directions to the nearest $45°$. For all pixels, if the gradient value at that location is not the largest of the three pixels in the direction of and opposite the direction of its own gradient, its value is set to 0. This is to ensure specificity of the edges.

4. All remaining pixels are subjected to thresholding of edge strength: we experimented with various threshold values to divide pixels into strong edges, weak edges, and no edges. Pixels below our low threshold are set to 0 to indicate no edge, pixels above the high threshold are set as strong edges, and pixels in between the boundaries are marked as weak edges.

5. Consider pixels deemed as strong edges from the previous step as true edges. To connect the edges and determine which weak edges to keep as true edges as well, we run breadth or depth first search starting from strong edge pixels and converting any weak pixels that are directly linked to strong pixels into strong pixels as well until all edges are detected.

To separate the board's edges from any edges detected from pieces, we perform Hough transform [7] for its ability to detect lines from the result of edge detection and its simplicity and ability to handle occluded ranks and files. To do so,

1. First, consider an edge point of known coordinates (x, y). There are many potential lines that can pass through this point at a variety of angles, which we will deem as a family expressed in the polar coordinate form:

$$-x\cos\theta + y\sin\theta = \rho$$

2. Now, consider the lines in (ρ, θ) space: what this means is that ρ and θ are now our variables, and x and y are constants. Any lines that intersect in (ρ, θ) space will thus give us the ρ and θ values in (x, y) space that correspond to a line in our image.

3. Discretize the (ρ, θ) space by quantizing the space into "accumulator" cells. For each (x, y) edge point, we "vote" on cells that satisfy the corresponding (ρ, θ) line equation by incrementing a counter in the cell.

4. Similar to the Canny edge detector, we establish a threshold for votes to be determined as a line, and pick the cells with more votes than the threshold to be our lines.

However, this produces far more lines than necessary due to noise during processing. To solve this issue, we divide the lines into vertical and horizontal and then perform agglomerative clustering of the lines produced by the Hough transform, which are nicely presented as polar coordinates to make grouping by angle differences simple with the help of SciKit-Learn library functions [12]. Once the clusters of lines are established, for each cluster, we take the line with the median ρ value as the singular representative of

that cluster. From there, we calculate as many intersection points as we can from the generated lines. Using these intersection points, we run RANSAC [9] to sample points and calculate a transformation matrix that allows us to view the board from a "birds-eye" perspective. To do so, for k iterations:

1. Randomly sample a set of points from the data.
2. Compute a transformation matrix that will project these points into an aerial 2D view.
3. Using this model, find the number of inlier points to the transformation matrix.

We chose to run our algorithm for a minimum of 200 iterations to gather a large sample of candidate transformations and a maximum of 10,000 iterations before deeming our search inconclusive. At each iteration, we keep track of and update the best transformation matrix acheived across the trials. This transformation simultaneously helps us detect any missing points and edges from our collection and gives us an easy method to segment each of the 64 squares if they are projected at an angle where each square is of roughly equal size in order to collect inputs for occupancy detection and piece classification.

Next, for each of the 64 squares, we will sample the area around the square and pass it into an occupancy detection model that determines if the square is either empty or occupied by a piece. To do so, we have implemented three approaches. The first is a **baseline CNN model** with 3 convolutional layers, 3 pooling layers, and 3 fully connected layers, with the final fully connected layer being the classifier head with 2 classes, representing either empty or occupied. The second model is a ResNet model [10] pretrained on ImageNet and finetuned on our occupancy data, with the classification head adapted to predict one of two classes. Specifically, we use the ResNet18 pretrained model variant. The third architecture uses InceptionV3 pretrained on ImageNet; we finetune the model's classification head in the same way as the ResNet model.

Finally, if the square is detected as occupied, we run a piece classifier to determine the color and type of piece that is on the square. For our classifiers, we use the same three model architectures used in the occupancy detector but we adapt the classification heads of the ImageNet-pretrained models to have 12 classes, one for each (color, piece) pair. We will then create a mapping for each piece to an identifier, square, color, and type. With these mappings and the locations of the squares of the chessboard, we have the information to manually create a FEN representation of the board.

In order to evaluate the model, we have chosen criteria at each step to check performance. For board detection, our dataset contains ground truth labels for where the 4 corners of the board are located, and we measure the distance between our prediction and the actual location in pixels. For occupancy detection and piece classification, we can use binary cross-entropy loss for training and weighted validation set F1 as a metric for how the model is performing. To evaluate the FEN representation of the board, we use the Python chess library to populate a board object based on which squares were classified as occupied with the identified piece at that square. We gauge the accuracy of our model on the number of squares correctly classified as occupied or empty, the number of pieces correctly identified, and the overall accuracy of the 64 squares of the board being filled in.

We use all of the same methods to train and evaluate the models on the transfer learning dataset.

6. Experiments

6.1. Occupancy Detection

We train our occupancy classifier on the Blender dataset-generated subtask data for 10 epochs, a learning rate of 0.001, and batch size 32; these were chosen using grid search based on validation set results. We train using binary cross entropy loss (occupied or empty) and use the Adam optimizer. We created three occupancy models, each with a different CNN architecture described in our methods section (3-layer Convnet, InceptionV3, and ResNet18). Training time for these three occupancy classifiers took 30 minutes, 2.5 hours, and 1.5 hours, respectively on a single NVIDIA RTX A5000.

6.2. Piece Classification

Similar to the occupancy detector, we train our piece classifier on the Blender dataset-generated subtask data with the same hyperparameters. These hyperparameters ended up having the best performance on the validation set out of the different hyperparameter arrangements. We use standard cross entropy loss and the Adam optimizer. Our three piece classifiers are also a 3-layer CNN, InceptionV3-finetuned, and ResNet18-finetuned on the dataset. It took 1 hour, 3 hours, and 2 hours, respectively on a single NVIDIA RTX A5000.

6.3. How much data do we need?

During experimentation, we found that the occupancy and piece classifiers demonstrated strong validation set performance only a few epochs into training. This motivated us to explore how many examples we needed to achieve strong performance. Therefore, we ran the same occupancy and piece classification training jobs (same hyperparameters) for the ResNet18 and 3-layer CNN but with stratified subsets of the training data of increasing sizes (1,000, 2,000, 10,000, and 20,000 examples). We examine the model per-

formance at each checkpoint to see if only a fraction of the data may yield comparable results to the fully trained models. We chose to experiment with one ImageNet-pretrained model (ResNet) and the non-pretrained model (3-layer CNN) to see if the ImageNet data may help the pretrained model converge faster on the Blender dataset.

6.4. Transfer Learning

We also examine how well our models generalize to different physical chessboard arrangements from outside the Blender dataset. We take the trained occupancy and piece classifiers for the 3-layer CNN and ResNet18 architectures and finetune them on the training set of the transfer learning dataset. We finetune using the same hyperparameters as the original training, except that we lower the learning rate to 0.0001 from 0.001 to avoid large updates overshooting the adjustments in parameters needed to transfer the learned features effectively to the new dataset. We also experiment with different numbers of epochs (35 and 50 epochs); we found that the original 10 epochs resulted in barely any change from the baseline (no finetuning), so we hypothesized that looping over the dataset more could contribute to improved results given how little data there was to learn from.

7. Results and Evaluation

Each subtask of our 3-stage process has its own evaluation metric. 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. We chose to measure weighted F1 because of the class size imbalances, especially in the piece classification subtask dataset, which has disproportionately more pawns than queens, for instance. When evaluating our entire system, we mainly focus on the average number of mistakes made per board. A mistake is defined as a missing, hallucinated, or misidentified piece. For granularity in analysis, we also record the number average number of occupancy mistakes per board and the average number of piece classification mistakes per board.

7.1. Board detection

Appendix 5 shows the performance of our board detection model compared to the ResNeXt baseline model from Masouris et. al. [11], which pales in comparison to our preliminary results across the various metrics that we computed such as the average pixel error from ground truth and percentage of boards with no mistakes.

Table 1. **Performance comparison of different model variants on occupancy subtask dataset test set**

Model Type	Accuracy	Weighted F1
RESNET18	99.88	99.93
3-layer CNN	99.47	99.47
InceptionV3	99.93	99.93

7.2. Occupancy detection

Unsurprisingly, every model architecture demonstrates strong performance on the occupancy subtask. While the InceptionV3 model posts the strongest performance (99.93 F1), the miniscule difference in performance trades off with inference time, where the InceptionV3 model takes nearly twice as long as the ResNet and 3-layer CNN models. We believe that the strong performance of all model variants reflects the relative ease of the task; intuitively, determining the occupancy of a square should be somewhat trivial.

7.3. Piece classification

Table 2. **Performance comparison of different model variants on pieces subtask dataset test set**

Model Type	Accuracy	Weighted F1
RESNET18	99.81%	99.81
3-layer CNN	96.24%	96.20
InceptionV3	99.92%	99.93

Similar to Table 1, Table 2 demonstrates that all model architectures exhibit strong performance on the pieces test set. However, there is a notable improvement between the pretrained CNN architectures (ResNet18 and InceptionV3) compared to the 3-layer CNN, with a difference of about 3 F1.

In order to better understand what features our model uses to predict the piece type, we generated saliency maps with a collection of image samples varying in piece type, color, and residing square color. The process of generating saliency maps involves computing the gradient of the output class score with respect to the input image. This gradient, when visualized, reveals the areas in the image that the model considers most important for its classification decision. By examining the saliency maps generated from both models, we were able to identify key differences in how each network processes and prioritizes different parts of the input images, as well as understanding certain pitfalls the model may face when classifying pieces. For instance, both the CNN and ResNet saliency maps have more defined structures when detecting queens and kings, but the maps are noisier and less interpretable for pieces like pawns.

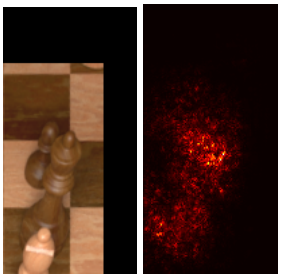

Figure 5. An occluded black queen on a black square (left), along with its ResNet saliency map (right.)

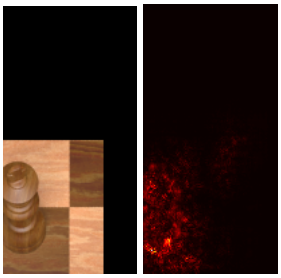

Figure 6. An isolated black king on a black square (left), along with its ResNet saliency map (right.)

Additionally, the model also has noisier saliency maps when there are other pieces in view, as evidenced by the example of this pawn's saliency map with lots of neighboring pieces.

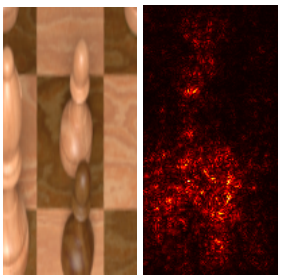

Figure 7. A heavily occluded white pawn on a black square (left), along with its ResNet saliency map (right.)

7.4. End to End Results

We now examine the results for the fully-connected model on the test set of full chessboard positions.

Table 3. **Average number of mistakes per chessboard for different models on Blender test set**

Model	Total mistakes	Occupancy Mistakes	Piece Mistakes
ResNet18	1.77	0.19	1.58
3-layer CNN	1.70	0.44	1.26
InceptionV3	1.66	0.19	1.47

In line with 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-pretrained ResNet18 and InceptionV3 models performed better on occupancy detection, but worse than the 3-layer CNN in piece detection.

7.5. Stratified Data

Table 4. **Average number of total chessboard mistakes for end-to-end evaluation on Blender test set**

Model Type/Num examples	1K	2K	10K	20K
ResNet18	3.85	2.30	1.64	1.53
3-layer CNN	10.28	8.77	5.51	3.63

Table 4 shows that in certain cases, a lower amount of training data does not necessarily lead to worse test set performance on the end-to-end model. For instance, the ResNet18 achieves an average error count of 1.53 mistakes per chessboard, which is actually lower than the ResNet18 model trained on the entire training set (3). This suggests that the ResNet18 model ends up overfitting to the training set after it trains on too many examples and therefore, training on a subset of the training data leads to a model more robust to generalization. Another result of this experiment is that after 20K examples, the ResNet18 model has already reached performance that is equal to its final state; the 3-layer CNN model is at 3.63 mistakes per board, while its final state has 1.7 mistakes per board. Therefore, we can conclude that the ResNet18 model converges faster than the 3-layer CNN. These results make sense considering that the ResNet18 model's pretrained feature extraction allow it to learn faster than the 3-layer CNN; this may also explain why the ResNet18 model which trained on the full dataset generalized worse to the test set than the 20K examples model.

7.6. Transfer Learning

For our results, see Table 6 (moved due to formatting issues).

Both the ResNet18 and 3-layer CNN improve after transfer learning compared to their trained baselines, but they ultimately fall short of the excellent results found in Table 3. In particular, it seems that the piece mistakes drive the majority of the errors during evaluation, suggesting that the models do not generalize well to the new piece designs. The ResNet model does slightly better than the 3-layer CNN in this aspect, which may be a result of its vast pretraining data exposing it to a broader set of images, allowing it to generalize to the new data better than the 3-layer CNN. We

believe that our results are still impressive given the scarcity of our training set and hypothesize that a greater number of examples would produce dramatically better results.

8. Conclusions and Future Directions

Our objective is to automate the generation of a board position given an image of a chess board by splitting the task into 3 phases of board detection, square occupancy detection, and piece classification. We use classical methods for board detection and trained a variety of deep models for occupancy detection and piece classification. For board occupancy, InceptionV3 displays strongest performance in prediction accuracy; however, not by a large margin compared to ResNet and a CNN baseline. For piece classification, ResNet and InceptionV3 display stronger performance than the simple 3 layer CNN. Finally, our end-to-end results show strong performance in classification, with all 3 models averaging less than 2 total mistakes per board. However, the distribution of mistakes is skewed more heavily towards piece classification errors, with each model averaging less than 0.5 occupancy mistakes per board. Stratification of the data into various training dataset sizes reveals that the ResNet model does not require training on the full dataset to achieve optimal performance and performance actually tapers off with further training. We conclude that the pretrained models we used converge quickly to the training set, and can overfit the data if trained for too many epochs. While transfer learning on the handcrafted dataset did show improvement over the pretrained baselines, the discrepancy between occupancy mistakes and piece classification was even wider, suggesting that the models had a difficult time generalizing to new piece designs, a problem that we acknowledge to be common due to different piece design choices across chess set makers. With more time, we would be interested in exploring transfer learning across different, larger datasets of chessboard images and observing the effects of catastrophic forgetting, for instance. Additionally, with more compute resources and data, we would experiment further with transformer architectures such as CLIP and ViT to compare their performance against our CNN based architectures.

A. Board Detection Results

Table 5. **Board detection evaluation of our approach vs ResNeXt baseline**

Metric	Our Approach	ResNeXt Baseline
Mean incorrect squares per board	**0.15**	1.19
Boards with no mistakes (%)	**93.86%**	39.76%
Boards with ≤ 1 mistake (%)	**99.71%**	65.20%
Per-square error rate (%)	**0.23%**	1.86%
Average pixel error from ground truth	**1.36**	22.3

Table 6. **Average number of end-to-end mistakes per chessboard for finetuned ResNet and 3-layer CNN model variants on transfer learning test set**

Model Type	Total Mistakes	Occupancy Mistakes	Piece Mistakes
ResNet18$_{base}$	12.22	1.37	10.85
ResNet18$_{35}$	6.19	0.07	6.11
ResNet18$_{50}$	6.19	0.07	6.44
3CNN$_{base}$	22.48	0.11	22.37
3CNN$_{35}$	10.37	0.15	10.22
3CNN$_{50}$	10.00	1.25	8.74

Note: the subscript *base* next to a model name means that the model was not finetuned on the transfer learning data; these are the baselines. The subscripts with numbers correspond to the number of training epochs for each model.

B. Contributions and Acknowledgements

Alex wrote the occupancy and classifier model training code and infrastructure for training and evaluating these subtask models. Alex also wrote dataprocessing code to load in downloaded datasets and ran the experimentation and training jobs. For the report, Alex helped write/edit the Abstract, Introduction, Literature Review, Dataset, and Experiments sections. Alex is part of Stanford's Artificial Intelligence Lab (SAIL) and used three GPUs from his lab cluster to run the experiments shown in our paper. Brent wrote the code for board detection, saliency map generation for piece classifiers, and the end-to-end pipeline for performing inference on a board image. Brent also wrote the code for evaluating board detection and FEN generation accuracy for the full model. For the report, Brent helped write the Dataset, Methods, Results, and Conclusions section.

References

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

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

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

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

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

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

[7] R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, jan 1972.

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

[9] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, jun 1981.

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

[11] A. Masouris and J. van Gemert. 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, 2024.

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

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

[14] W. Rong, Z. Li, W. Zhang, and L. Sun. An improved canny edge detection algorithm. In *2014 IEEE International Conference on Mechatronics and Automation*, pages 577–582, 2014.

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

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

[17] G. Wölflein and O. Arandjelovic. Dataset of rendered chess game state images, 2021.

[18] G. Wölflein and O. Arandjelović. Determining chess game state from an image. *Journal of Imaging*, 7(6):94, June 2021.