Deep Learning Approach for Chessboard Piece Recognition

Adriano Machado (up202105352), Félix Martins (up202108837),
Francisco da Ana (up202108762), and João Torre Pereira (up202108848)
Faculty of Engineering, University of Porto, Portugal
(Dated: June 18, 2025)

I. Introduction

This report presents a deep learning computer vision approach for detecting and localizing chess pieces on a chessboard using PyTorch and Ultralytics. Our project focused on three main objectives: detecting the number of pieces, finding bounding boxes for each piece, and building a digital twin of the board. We followed the original train/-val/test splits of the dataset, using the ChessRed2k dataset for bounding-box detection. Models were compared on the validation set, with the best ones evaluated on the test set for consistency.

II. Task 2: Number of Chess Pieces

The goal was to predict the number of chess pieces on the board. We focused on regression, with **Mean Absolute Error (*MAE*)** as the primary metric, and explored various methods to optimize performance.

Our initial setup used a ResNet50 base model, MSE loss, AdamW optimizer (learning rate 1×10^{-4}, weight decay 0.01), and a simple learning rate scheduler (halves learning rate each 5 epochs). We used a sigmoid activation, $out = 30 \cdot \sigma(x) + 2$, to ensure outputs are within $[2, 32]$. Images were resized to 256 and cropped to 224, and our models were fine-tuned (all layers) using PyTorch pre-trained weights for 30 epochs.

A. Methodology and Improvements

We improved the model iteratively in a "greedy-like" fashion, applying successful changes to subsequent experiments.

In this task, we explored multiple methods in attempts to improve the performance of our models, such as:

- Experimenting with different **model architectures**
- Introducing different **data augmentations**
- Different **activation functions**
- Different **loss functions** (although all are regression based)
- Specific **model hyperparameters** (learning rate, weight decay, etc.)

1. *Experimenting with manual augmentations*

Initially, models with basic transformations achieved a validation MAE of 0.73. Introducing **manual augmentations** (rotation, random perspective distortions, Gaussian blur, color jitter) improved performance to 0.62 MAE. We avoided aggressive operations like random cropping to prevent obscuring pieces. In Section II A 3, we will see that there are better augmentation options.

2. *Different model architectures*

We compared several architectures, chosen based on the PyTorch vision models documentation and their performance in the ImageNet dataset, leveraging pre-trained weights and fine-tuning.

- ResNet50
- ResNeXt-101 64x4d
- EfficientNetV2 Small
- Swin Transformer V2 Small

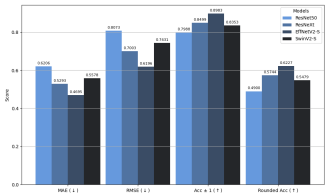

FIG. 1: Comparison of the performance of different model architectures. *EfficientNetV2-S*(green) has the best performance

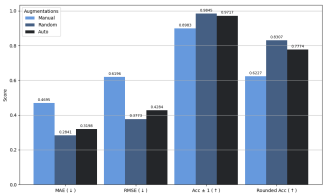

FIG. 2: Comparison of the performance of EfficientNet with different data augmentation techniques. *RandAugment*(orange) yields the best performance

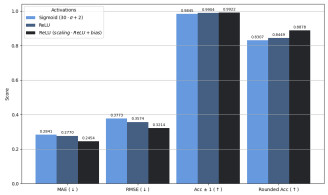

FIG. 3: Comparison of the performance of different activation functions for the prediction. *ReLU*(black) with learnable parameters has the best performance.

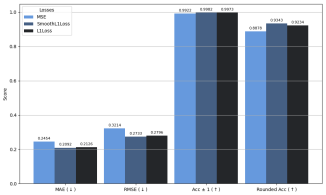

FIG. 4: Comparison of the performance of different loss functions. *SmoothL1Loss* and *L1Loss* (darker colours) have the best performances

Since we always used pre-trained weights, we adjusted the transformations to each model's pre-training input size, keeping other augmentations consistent.

EfficientNetV2-S demonstrated the best performance (Figure 1), becoming our primary model for subsequent experiments.

3. *Automatic augmentations*

Instead of performing manual data augmentations, we can use automatic augmentations: RandAugment and AutoAugment. RandAugment randomly applies a set of augmentations, while AutoAugment uses a specific policy to select them. RandAugment showed better performance with our EfficientNetV2-S setup, as shown in Figure 2.

4. *Final activation functions*

We experimented with other activation functions, replacing the original sigmoid activation. First, we used a plain ReLU activation, which improved performance. We then added scaling and bias parameters: $out = scaling \cdot ReLU(x) + bias$, which further increased performance (Figure 3).

5. *Different loss functions*

We experimented with **L1Loss** and **SmoothL1Loss** (both based on MAE) in addition to the original **MSE**. Both L1Loss and SmoothL1Loss yielded better performance than MSE (Figure 4).

6. *Hyperparameter tuning*

Using the *optuna* package, we tuned the learning rate, weight decay, loss function, and scheduler over 50 trials, each trained for 15 epochs. Out of 50 trials, 17 were pruned. The optimal configuration involved **L1Loss**, a learning rate of 5.26×10^{-4}, weight decay of 0.0326, and

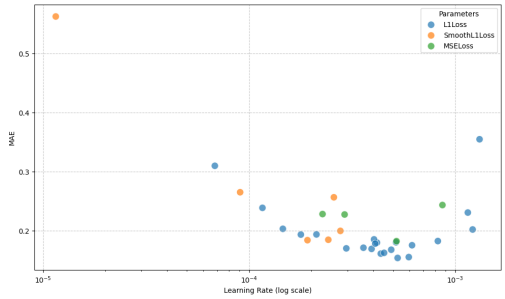

FIG. 5: MAE (y) vs. Learning Rate (x) by Loss

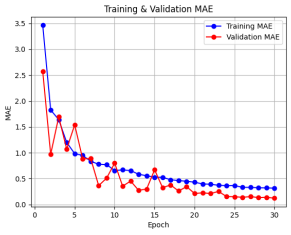

FIG. 6: *MAE* training curve for a tuned model (30 epochs).

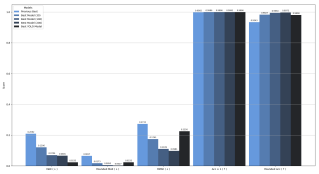

FIG. 7: Comparison of tuned models with different epoch counts with previous best model (Section II A 5) and with best YOLO model (Section III A)

a cosine annealing learning rate scheduler, with a performance of 0.15366 validation MAE. Some results from the tuning process are shown in Figure 5.

B. Best Models Comparison and Results

We trained the tuned configuration for 30, 100 and 200 epochs. After experimenting with rounding prediction values, we noticed this increased the performance. So, during training of the 100 and 200 epochs models, we instead decided to save the best model with respect to the rounded predictions MAE. During training, the best plain validation MAE achieved for the 100 epochs model was 0.0603 and, for the 200 epochs model, it was 0.0376. However, by optimizing for rounded prediction values, we saved different models, with 0.0704 MAE and 0.0050 rounded values MAE for 100 epochs, and 0.0665 MAE and 0.0027 rounded values MAE for 200 epochs. The 200 epochs version had the best results, with **0.0027** rounded values MAE on the validation set and also consistent with the test set, with **0.0056** rounded values MAE. This slight validation test difference likely comes from selecting models based on the best rounded values MAE instead of plain MAE. We show the loss curve for the model trained for 30 epochs in Figure 6, with the others having similar curves. The comparison of our best models is in Figure 7.

Surprisingly, our best YOLO model (from Task 3) evaluated on the full validation dataset outperformed our dedicated model in finding the number of pieces, when not rounding predictions. It achieved 0.0233 validation MAE and 0.0202 test MAE, using a confidence threshold of 0.5. This is likely due to its larger size and robust training. Nevertheless, our best and primary model for Task 2 is the tuned model trained for 200 epochs, including rounding of its predictions.

For this primary model, we also show two additional visualizations obtained from the test set, in Figures 8 and 9, to better contextualize its performance.

III. Task 3: Bounding box detection and Digital Twin Extraction

The objective of this task is to develop a pipeline that receives an image of a chessboard and outputs two main results: bounding box detections for all chess pieces, and

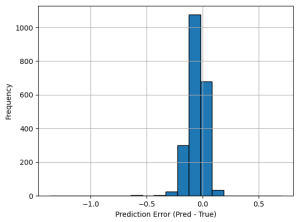

FIG. 8: Prediction Error Histogram

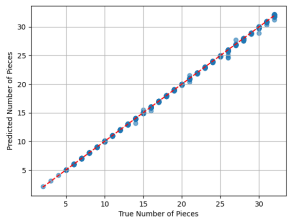

FIG. 9: Predicted (y) vs. True (x) Number of Pieces

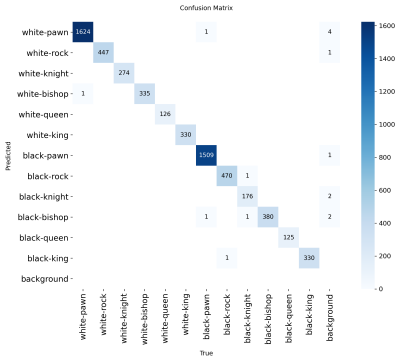

FIG. 10: Confusion Matrix for Chess Piece Recognition with Yolo-v8x

the digital twin of the board (piece types and their matrix locations).

A. Bounding Boxes

We started by training a model capable of detecting the bounding boxes of the pieces and their class (White Pawn, Black Queen, etc.), choosing **Ultralytics Yolo-v8n** as baseline. Labels were extracted from the provided annotations file by parsing it to obtain the desired dataset format.

The model was trained for 100 epochs with batches of 64 samples and images resized to 960 pixels (the remaining parameters follow the default setup). To measure the performance of the model in bounding box detection, we used **mAP (mean Average Precision)** with 2 IoU thresholds, obtaining mAP50 = 0.99 and mAP50-95 = 0.86. Performance in the individual piece classification sub-task was tracked by **Precision** and **Recall**.

Later, we trained more advanced models, like Yolo-v8x and Yolo-11s, in order to obtain better results in these subtasks. The best results for piece recognition can be seen in Figure 10.

B. Digital Twin extraction

Initially, an end-to-end digital twin extraction approach as described by Masouris et al. (2023)[3] led to significant overfitting, performing poorly beyond common chess positions. We then shifted to a multi-step pipeline leveraging our trained YOLO model's predictions.

The main idea was to use the YOLO bounding box predictions to correctly locate the pieces in the board matrix. We first experimented with the traditional techniques developed during Task 1 to extract the board's corners and the chessboard grid. By applying the computed transformation for board warping, we can obtain the estimated square for a piece in the chess grid by considering a key assumption: the central pixel of the bounding box's bottom half will always be mapped to the correct square. This is enough to extract the **Forsyth-Edwards Notation (FEN)** representation of the board and measure the

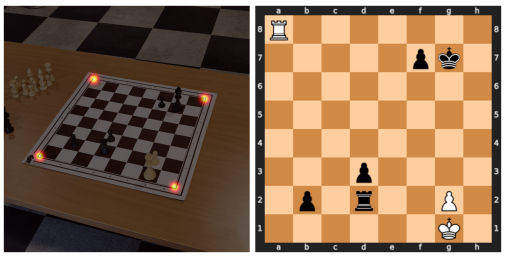

FIG. 11: Physical board with detected heatmaps (left) and its digital twin (right)

pipeline's performance by computing the **Edit Distance** between the expansion of the obtained FEN string and the ground truth value. This metric counts the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into the other, giving us a measure of misplaced, incorrect, or missing pieces.

The first results from the entire ChessRed2k dataset were not very impressive: the average edit distance of 1.59 indicates that there is considerable margin to improve the pipeline. By storing mismatched strings and analyzing them, we were able to identify issues in our pipeline, mostly caused by poor detection of the board's orientation.

C. Detecting Corners and Orientation with Deep Learning

We explored deep learning to replace the tradition board warping, which used less reliable techniques like canny edge detection, polygon approximation and feature matching.

a. Yolo for Pose Estimation

Initially, we trained **Yolo11s-pose** to detect both the bounding box of the chessboard and its four corner keypoints. We trained it on the ChessRed2k dataset for 100 epochs, with a batch size of 64 and an image size of 640. Although it produced very accurate bounding boxes (mAP50 = 1 and mAP50-90 = 0.99), the predicted keypoints rarely coincided with the true corners.

To measure corner predictions, we calculated the **Mean Euclidean Distance (MED)** between the normalized corner coordinates and our predictions. Here, we obtained 6.9% for the validation set and 3.6% for the test set. A percentage of 0% would indicate an optimal value, without any error.

b. Regression with a CNN

Given our initial lack of success, we used a simpler approach: pre-trained CNN backbones (**ResNet50**) to directly regress the four chessboard corners. We froze the backbone and trained some added layers, using MSE with Adam, with the outputs based on a sigmoid activation. After 150 epochs, predictions were still unreliable, with a MED of 14.91% in validation and 13.19% in test. Performance significantly improved by using EfficientNetV2-S with fine-tuning of all its layers and **manual data augmentations** to improve generalization, achieving a MED of 0.58% on the validation set and 0.45% on the test set.

c. Corner Heatmap

After encountering reliability issues with basic coordinate regression, we explored a more robust heatmap-based approach. This involved training a **U-Net** with a pre-trained **ResNet34 backbone** to predict probability distributions for each corner's location. The decoder, trained from scratch, consists of a series of up-sampling and convolutional layers that construct four final heatmaps, one for each of the board's corners.

This method achieved a significant accuracy improvement. The model consistently generated well-defined heat maps in which the peak activation aligned with the true corner position, with a **0.50%** MED on the validation set and **0.31%** on the test set. An example is shown in Figure 11.

Nevertheless, we noticed that the model occasionally confused the identity of the corners, for example, by swapping the heatmaps for the top-left and bottom-right corners. This indicates that while the model learns to find corner-like features, it struggles to consistently assign the correct label based on global orientation.

d. Segmentation mask and Orientation Prediction

As an alternative approach, we adopted a two-stage pipeline for board warping and rotation. First, a **U-Net** architecture was trained from scratch to produce a **binary segmentation mask** of the chessboard. Second, we used a pre-trained **ResNet18**-based classifier to predict the orientation of the masked board, with **four classes** representing 0°, 90°, 180° and 270°. This approach achieved a MED of 2.23% on the validation set and 0.49% on the test set.

IV. Conclusions

We successfully developed a deep learning pipeline for chessboard analysis. For Task 2 (Number of Chess Pieces), our tuned EfficientNetV2-S model achieved excellent results, with an MAE of **0.0027** on the validation set and **0.0056** on the test set, after using rounded predictions.

For Task 3, we achieved very good bounding box detection performance with YOLOv8x (mAP50 = 0.99). While extracting the full digital twin proved to be more complex, our iterative experimentation with various techniques led to significant improvements in accuracy, with the heatmap corner detection technique the most successful.

The main limitations of our project are in digital twin extraction, where the smaller dataset for annotated corners limited our model's ability to generalize across diverse conditions. We showed that using data augmentation techniques in the simpler corner extraction method increased performance. Future work could address these limitations by incorporating data augmentation into the more advanced methods, or by implementing an explicit orientation detection module to resolve labeling ambiguities.

---

[1] Craig Belshe. Chess piece detection. https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1617&context=eesp, 2021. Senior Project, Electrical Engineering Department.

[2] Andrew Chen and Kevin Wang. Robust computer vision chess analysis and interaction with a humanoid robot. *Computers*, 8:14, 02 2019.

[3] Athanasios Masouris. Chess recognition dataset (chessred). https://data.4tu.nl/datasets/99b5c721-280b-450b-b058-b2900b69a90f/2, Sep 2023.

[4] Daylen Yang. Building chess id: Featuring computer vision! deep learning! https://medium.com/@daylenyang/building-chess-id-99afa57326cd, Jan 2016.