An Object Detection Ramble with YOLOv8n : Application to Chess Pieces

Edouard Chappon and Samira Ahrouch

Abstract

This study focuses on adapting the **YOLOv8n** model, initially pre-trained on the COCO dataset with 80 object classes, to detect **12 new classes** corresponding to **chess pieces**, identifying both their type and color. To achieve this, multiple object detection experiments were conducted on a dataset of chess pieces images, exploring various aspects of model adaptation.
Change it Two Jupyter notebooks are attached to this report: one for the training phase, including data preparation, fine-tuning, and hyperparameter optimization, and the second for the evaluation phase, involving testing on different test datasets and performing both qualitative and quantitative analyses. The results provide insights into the impact of the adaptation on the overall performance of the model.

1 Introduction

Computer vision has various applications such as facial recognition, autonomous vehicles, robotics, and even medical image analysis. It involves diverse tasks, including image classification, object detection, semantic segmentation, and automatic image generation. Advances in convolutional neural networks (CNNs) [3] have played a crucial role in enhancing the performance of image processing models, making these tasks more accurate and efficient.
Object detection is particularly a fundamental task in image analysis. It involves identifying and localizing object instances within an image and subsequently classifying them into different categories. Over time, different techniques have been developed, evolving from feature extraction methods to deep-learning models based on CNN architectures [1, 4]. Among these models, YOLO (You Only Look Once) [2] has emerged as one of the most popular and efficient architectures, known for its single-stage design that enables real-time detection. Furthermore, its latest light version, YOLOv8n, offers significant improvements in terms of accuracy and efficiency.
In this study, we focus on adapting YOLOv8n for the classification and detection of chess pieces. The objective is to finetune this model and evaluate its performance in several settings, compare it with other architectures.
This report is organized as follows: Section 2 describes the YOLOv8n architecture, Section 3 outlines the used evaluation metric, Section 4 presents the dataset and finally and Section 5 presents the experiments and discusses their results.

2 YOLOv8n Network Architecture

YOLOv8n is a lightweight version of the YOLOv8 object detection model, optimized for speed and efficiency while maintaining high accuracy. It belongs to the YOLO (You Only Look Once) family, renowned for real-time object detection. The network processes an RGB input image and produces bounding boxes, confidence scores, and class probabilities for detected objects. Its architecture consists of three primary components: the **backbone** for feature extraction, the **neck** for multi-scale feature aggregation, and the **head** for generating detections. Below, we describe each part of the network. The Figure 1 shows its full architecture.

2.1 Input

The model takes an RGB image of size 640x640 pixels.

2.2 Backbone

The backbone is responsible for extracting features from the input image through 8 blocks composed of convolutional layers, C2f blocks, and a final Spatial Pyramid Pooling - Fast (SPPF) layer. It progressively reduce the spacial domain and increasing the feature space to capture relevant informations. The backbone consists of the following layers:

- **Convolution layers** with batch normalisation and SILU activation function.
- **C2f blocks** which are convolutions layers with skipping connections to avoid gradient vanishing.

2.3 Neck

The neck aggregates features from different backbone layers (the $4^{th}, 6^{th}$ and 8^{th} feeds the 14^{th}, 11^{th} and 9^{th} layers) to enable detection at multiple scales.

It employs 13 blocks composed of upsampling (to detect objects at 3 different scales), concatenations

(to combine information from skip connections), C2f blocks to combine low-level and high-level features.

A special **SPPF block** process the output of the output of the last layer of the backbone, it consists of a convolutional block followed by three MaxPool2d layers. Every resulting feature map from the MaxPool2d layer is then concatenated at the end and fed to a convolutional block to capture multi-scale informations.

Two skips connections are also added between the 9^{th} and 21^{th} layers and the 12^{th} and 17^{th} to obtain representation to avoid gradient vanishing.

2.4 Head

The head generates the final predictions by processing features from multiple scales. It consists of a single detection layer, it takes feature maps from layers 15, 18, and 21. It outputs bounding boxes, confidence scores, and class probabilities for our 12 classes.

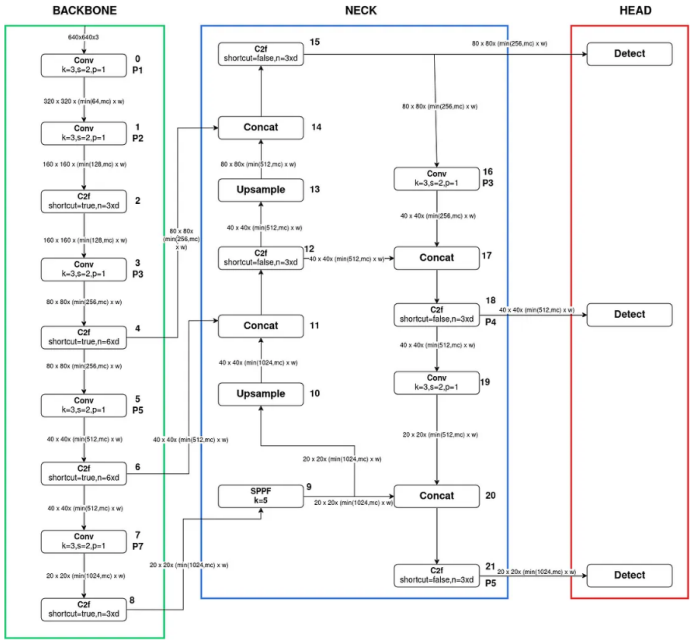

Figure 1: Yolov8 achitecture

2.5 Output

The final output of YOLOv8n consists of bounding boxes, confidence scores, and class probabilities for each detected object in the input image. The multi-scale features from the neck and head enable the model to detect objects of varying sizes efficiently, making it suitable for real-time applications such as object detection in constrained environments.

3 Evaluation metrics : the mAP50-95 (B)

The *mean Average Precision* (mAP) metric is widely used to evaluate the performance of object detection models. In this study, we use the **mAP50-95 (B)** metric, which involves computing the mean of the Average Precision (AP) across a range of Intersection over Union (IoU) thresholds from 50% to 95% (in steps of 5%). The suffix "(B)" indicates that this evaluation is performed on bounding boxes.
For each IoU threshold $t \in \{0.50, 0.55, 0.60, \ldots, 0.95\}$, the Average Precision is first computed for each class. The **mAP50-95 (B)** is then obtained by averaging the AP values over all these thresholds and across all classes. This process provides a comprehensive measure that reflects both the precision and robustness of the model's predictions.
The AP metric computation is based on the IoU concept, which is defined as:

$$\text{IoU} = \frac{A_{\text{intersection}}}{A_{\text{union}}}, \tag{1}$$

where $A_{\text{intersection}}$ denotes the intersection area between the predicted box and the ground truth box, and A_{union} represents the union area of the two boxes. A prediction is considered correct (true positive) if the IoU exceeds a given threshold, typically 0.50. By varying this threshold from 0.50 to 0.95, the **mAP50-95 (B)** metric provides a more thorough assessment of detection quality, accounting for cases where stricter overlap is required.

4 Dataset

The dataset used in this project has been extract from the train set of this kaggle dataset. It consists of 693 images of the same chess board and chess pieces in different configurations. The dataset has been treated with roboflow:

- Rename classes
- resize image in 640×640 pixels
- Separate between train and validation sets.
- data augmentation of the training set with several techniques of image modification: Grayscale, Hue, Saturation, Brightness and Exposure.
- Generate COCO and YOLO dataset formats.

We end up with 1,557 images of chess pieces, split into 1,452 images for training and 105 images for validation. The dataset features 12 distinct classes, representing the six types of chess pieces—pawn, knight, bishop, rook, queen, and king—in both black and white variants. Figure 2 shows a training sample from this dataset with pre-treatment.

Figure 2: Image from training set

Each instance of a chess piece in the images is annotated with a bounding box and its corresponding class label, enabling the model to learn both the localization and classification of the objects.
The distribution of the classes within the dataset (combining training and validation sets) is details in table 1. We can observe that it is imbalanced, with certain classes having a significantly higher number of instances compared to others. In total, there are $15,390$ annotated instances across all images. The pawns are the most prevalent. Conversely, the queens are the least represented. All the classes range from 590 to $3,592$ instances. Additionally, the number of images containing each class varies. The dataset is not consistent with the standard composition of a chessboard because around 50% of the images contains black king and white king, which should be 100%.

Class	Nb Instances	Nb Images	% Instance
black-bishop	726	438	4.72%
black-king	762	762	4.95%
black-knight	1,015	676	6.60%
black-pawn	3,592	778	23.34%
black-queen	457	457	2.97%
black-rook	1,074	714	6.98%
white-bishop	915	630	5.95%
white-king	789	789	5.13%
white-knight	1,025	694	6.66%
white-pawn	3,438	780	22.34%
white-queen	590	590	3.83%
white-rook	1,007	676	6.54%

Table 1: Class Distribution in the Chess Dataset

This dataset provides a diverse set of images with varying numbers and types of chess pieces, offering a suitable foundation for training and evaluating the object detection model.

Notice that we ran all our experiments only once. To obtain more reliable results we would have to compute the mean score over several runs. But Kaggle free plan is not infinite.

5 Experiments

In this section, we present the experiments conducted in this study. We used a pre-trained YOLOv8n from the library ultralytics.

We start by a first fine-tuning and a first observation to the performences in subsection 5.1, then we compare performences for several values of the freezing layers hyperparameter in subsection 5.2. In the subsection 5.3 we compare YOLO with other architectures. The last experiment 5.4 shows the effect of a pre-finetuning on 3 new pre-finetuned datasets and our finetuned chess dataset.

For each experiment, we outline the methodology, report the results, and discuss the corresponding conclusions.

Experiments has been leaded on Kaggle with T4 GPU.

5.1 Experiment 1: First Training

We fine-tuned a simple YOLOv8n model for 50 epochs. In this initial experiment, we froze the first four layers and let the default parameters.

We achieved a validation score of 79.7% mAP50-95. Figure 3 displays the mAP50-95 for each individual class, plotted against the percentage that class represents in the training set. One might expect a positive correlation between these two variables, but it does not hold in practice. For example, pawns are the most prevalent class, yet they exhibit a lower mAP compared to many other classes.

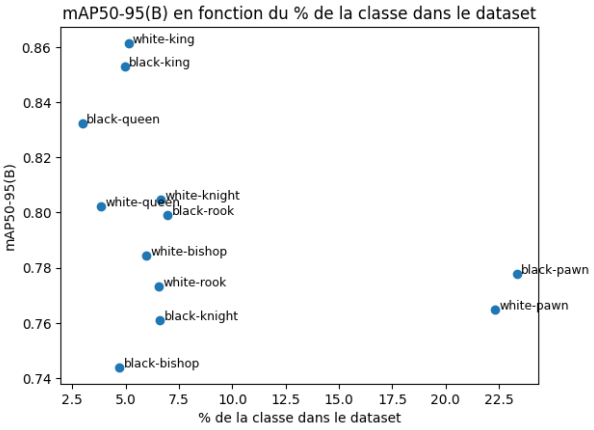

Figure 3: mAP50-95 by class vs. class percentage in the training dataset.

We tried to pair predictions with ground truth box by finding the prediction not used by an other GT that have the best IoU with the GT. In Figure 4 a sample with GT boxes (blue), well detected classes box (green) and wrong detected classes (red) are displayed

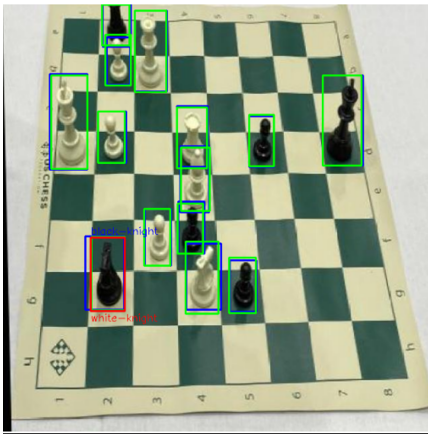

Figure 4: Classes box (green) and wrong detected classes (red) displayed for a validation image.

In total, we identified that only one ground truth (GT) bounding box was not matched with a true prediction, whereas 21 predicted bounding boxes remained unmatched. This discrepancy comes from the way we compute the matching which is not symetric (it also depends on the order of treatment of the GT).

The GT not matched is the first picture of the figure 5. We can see that the GT piece is a partly masked by an other one. The sample of prediction not matched, has detected a piece (with the wrong label), but an other prediction (with the good label) has a slightly better IoU with the GT and thus has been match with it. All of the 21 predictions not matched are in this case.

Figure 6 shows the class confusion matrix for all matched pairs of GT/predictions. We observe near-perfect classification. After analyzing the mistakes, we can see that the errors made by the model reproduce errors in the labels. Some pieces are labeled with the wrong color, some mix up piece types, and some pieces are not labeled at all. Figure 7 shows the types of errors and how they are reproduced by the model. The last type can explain the lack of prediction for the first image in Figure 5.

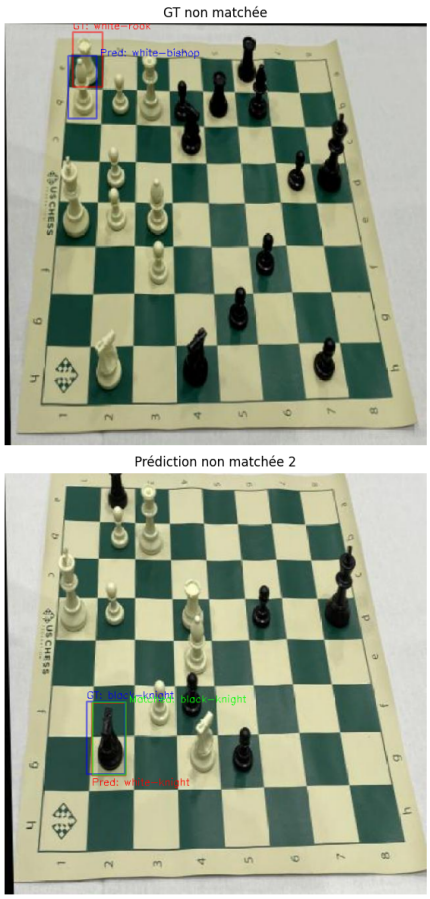

Figure 5: Image 1: GT (red) not matched and prediction with the biggest IoU (blue). Image 2: Pred not matched (red) and the GT with the biggest IoU (blue) and its matched prediction (green)

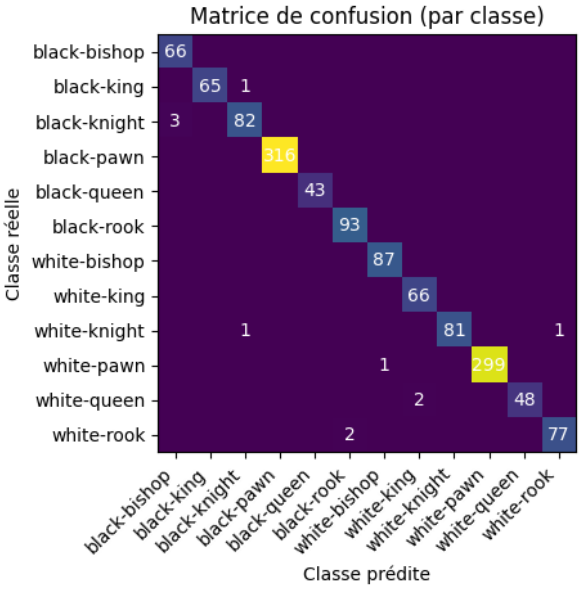

Figure 6: Confusion matrix for the YOLOv8n model. Rows are ground truths, columns are predictions.

5.2 Experiment 2: Performances Accordingly to the Number of Frozen Layers

5.2.1 Method

For this second experiment, we again employed the YOLOv8n model architecture on the same dataset but varied the number of frozen layers. We compared six configurations with 0, 4, 8, 12, 16 and 20 layers frozen. We used the default hyperparameters for YOLOv8n and trained each configuration for 20 epochs. The goal was to observe how freezing layers would affect performances. We used mAP50-95 metric.

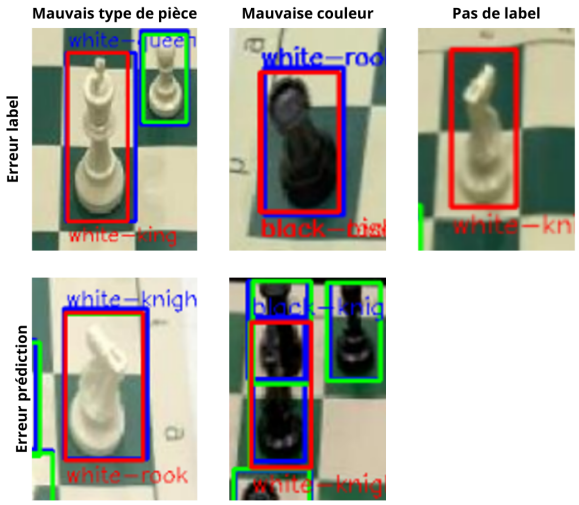

Figure 7: Errors within labels (first row) and within predictions (second row). The red box is the prediction which not match with the label in blue.

5.2.2 Results

Figure 8 shows the evolution of the validation mAP50-95 across epochs for all freezing setups. We displayed the validation score of the first epoch to the 20^{th}. We can notice that the rise of the metric in the firsts epochs is higher accordingly to the number of unfrozed layers. This is because more trainable parameters leads to more flexibility and faster convergence within the optimization process. Table 2 shows that all experiments reached between 60% and 70% of mAP50-95 except the 20 layers frozen which is way slower. The table shows that the number of trainable parameters tend to increase the time of compute, but less than expected. It is maybe due to the parallelization of calculations offered by GPU.

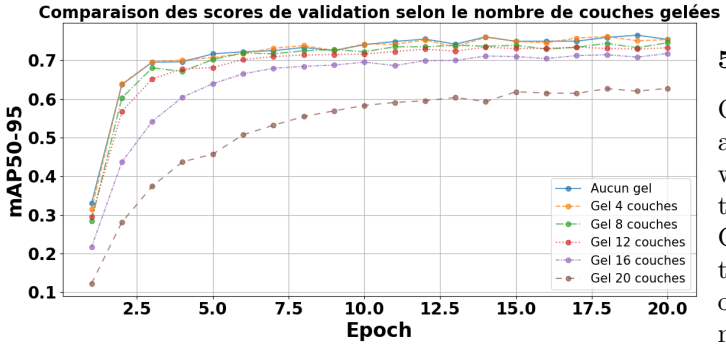

Figure 8: Comparison of validation mAP50-95 by epoch for different numbers of frozen layers.

The best score reached within 20 epochs is for 4 frozen layers. We chosed to keep this number for the rest of the experiments.

nb frozen	seconds	mAP50-95	Parameters
0	349	0.7841	3 013 188
4	314	0.7846	2 982 132
8	300	0.7657	2 365 428
12	282	0.7487	1 740 532
16	286	0.7345	1 555 060
20	286	0.6594	1 246 708

Table 2: Results of the experiment

5.2.3 Conclusion

In this experiment, we showed that moderate freezing of the first backbone's layers can yield to a gain of time without degradation of the performences. Moreover, excessive freezing of 20 layers is too restrictive and leads to poor results.

5.3 Experiment 3: YOLOv8n vs R-CNN vs DETR

5.3.1 Method

In this experiment, we compare the performances of three different object detection architectures: YOLOv8n, R-CNN, and DETR. Each model was trained on the same dataset for 20 epochs with 4 freezing layers for Yolov8 and DETR. We used default settings for other hyperparameters. The goal was to evaluate the validation mAP50-95 score during training.

DETR: is a one-stage model like YOLO; however, instead of selecting boxes with a CNN-based architecture, it uses a transformer to process the image's global features. We used the RT version from ultralytics library which leverages of some optimisations.

F-RCNN: is a two-stage model that first generates region proposals and then classifies them using a CNN, resulting in high accuracy but increased computational complexity and slower inference. We used the implementation of the library Detectron2.

5.3.2 Results

On Figure 9 we can observe that YOLOv8 (blue) and DETR (orange) have similar validation scores, while Yolo is the fastest by far like we can see on table 3, due to its little number of parameters. R-CNN (green) shows a slower convergence in terms of time and epochs. It reached a lower score than the others, even it has not converged. This approach is not recommanded .

Model	seconds	mAP50-95	Parameters
DETR	1 787	0.7698	31 809 712
F-RCNN	4 960	0.5152	41 477 536
YOLOv8n	288	0.7722	2 982 132

Table 3: Models comparison

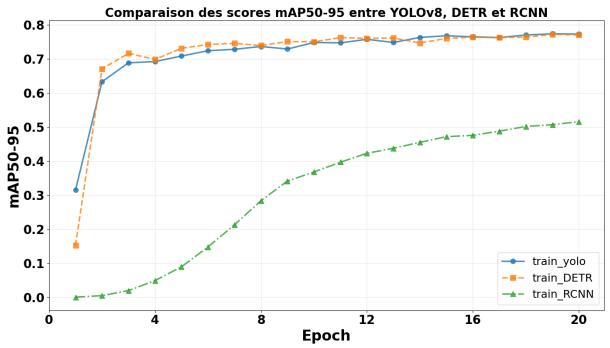

Figure 9: Comparison of mAP50-95 scores between YOLOv8, DETR, and R-CNN.

5.3.3 Conclusion

In this experiment, YOLOv8n outperforms both DETR and R-CNN in terms of convergence speed and final performance. DETR, while effective, requires more epochs to reach its full potential. R-CNN, despite being a strong model, has the slowest convergence and does not perform as well as the other two methods. Overall, YOLOv8n proves to be the most efficient architecture for object detection in this context, making it the preferred choice for real-time applications.

5.4 Experiment 4: Fine-tuning on an Intermediate Dataset before Fine-tuning on `chess_data`

5.4.1 Method

In this experiment, we used YOLOv8n and introduced an additional *pre-fine-tuning* step. We first fine-tuned the model for 200 epochs on a different dataset:

- `COCO`: 288 train images and 20 for validation with 80 classes. This is a small number of images for many classes, which affects its performance.
- `Vehicle`: 180 train images and 30 for validation with bikes and cars.
- `chess_new`: 177 train images and 15 for validation, consisting of chess-piece photos with the same classes but different chess boards, backgrounds, and pieces. This allows us to test how well the model generalizes from our `chess_data` dataset.

After that, we re-trained the same model for 20 epochs on the standard `chess_data`.

The goal was to assess how the mAP50-95 evolves during the pre-fine-tuning phase and then on the `chess_data` validation. In particular, we wanted to see if the model could generalize from one chess dataset to another.

Some adjustments were needed in the architecture and `yaml` files for the COCO and Vehicle datasets due to the new classes, as shown in Figure 13.

5.4.2 Results

Figure 10 illustrates the evolution of the mAP50-95 score during the pre-fine-tuning phase and the final fine-tuning on `chess_data`. Table 4 summarizes the results, showing the performance after each of the two phases.

The score after pre-fine-tuning on `chess_data` starts at 0, except for the `Chess_new` pretraining, while the mAP of the dataset used for pre-fine-tuning drops to 0 in one or two epochs.

The size of the head of the network (the output layer) is larger for COCO than the others (see Figure 13), which leads to slower learning during the second fine-tuning.

The green curves show that there is a transfer of knowledge from one chess dataset to the other. We can also see that the second fine-tuning leads to faster convergence for similar data, and the mAP doesn't drop to 0 for fine-tuning with `chess_new`.

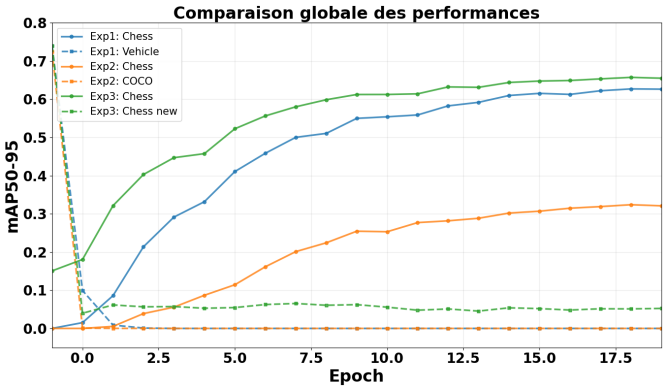

Figure 10: Evolution of mAP50-95 scores during the second fine-tuning on dataset used for pre-finetuning and `chess_data` used for the second one.

Dataset	mAP50-95 before	mAP50-95 after	Gain
– Chess	0.0000	0.6263	0.6263
- - Vehicle	0.7399	0.0000	-0.7399
– Chess	0.0000	0.3209	0.3209
- - COCO	0.7240	0.0000	-0.7240
– Chess	0.1506	0.6550	0.5044
- - Chess_new	0.7399	0.0523	-0.6875

Table 4: Results of pre-fine-tuning on various datasets followed by fine-tuning on `chess_data`.

Figure 11 shows a validation sample from `Chess_new` after fine-tuning. We can see that the model makes many mistakes. This is because the dataset is not

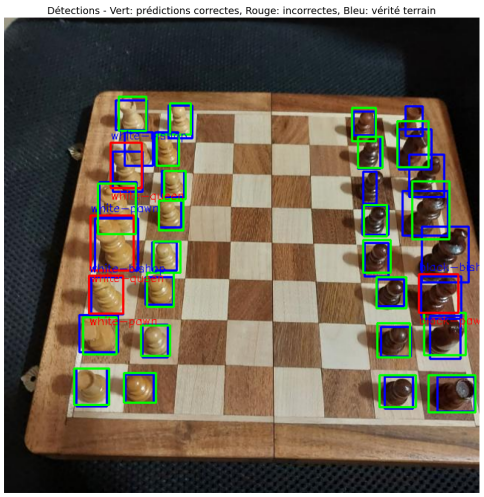

Figure 11: Prediction -good in green bad in red- after prefine-tuning on `chess_new` sample. GT boxes are in blue.

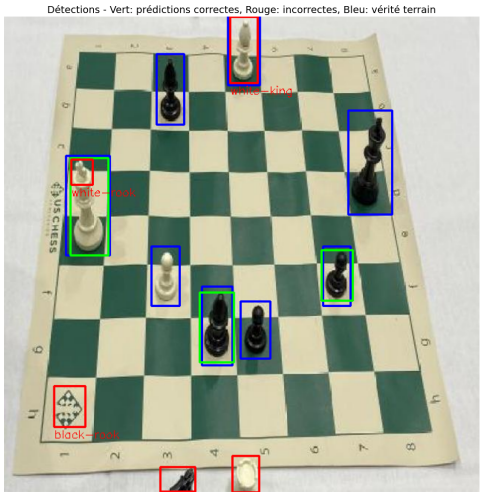

Figure 12: Prediction -good in green bad in red- after prefine-tuning on `chess_dataset` sample. GT boxes are in blue.

as large as the `chess_dataset`.

Figure 12 shows that the model fails to generalize to other types of data. It detects some part of piece as entire piece or a logo on the chessboard as piece and misses other pieces.

5.4.3 Conclusion

This fourth experiment highlights how an additional pre-fine-tuning phase on a different dataset can affect final performance on `chess_data`. In some cases (e.g., using `chess_new`), pre-training can improve the training and leads to knowledge transfert. However, when the dataset is too dissimilar (as in `Vehicle` or a small subset of `COCO`), it may actually hinder the final model performance on `chess_data`. Finally, our experiments with `chess_new` have shown that the YOLO architecture is highly sensitive to the underlying structure of the dataset.

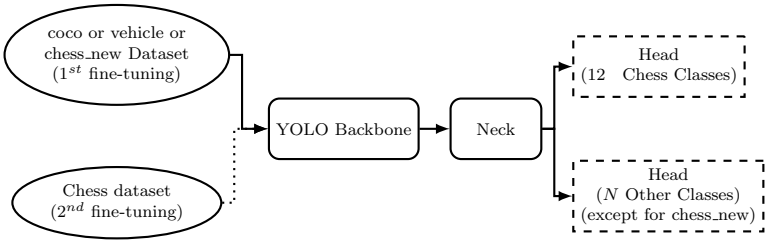

Figure 13: YOLO architecture for sequential fine-tuning: pre-training for 200 epochs on one dataset followed by fine-tuning for 20 epochs on chess images.

6 Conclusion

YOLOv8n proves highly adaptable to specialized object detection tasks with impressive efficiency. This lightweight architecture delivers competitive accuracy with significantly faster training and inference than alternatives.

Moderate layer freezing provides a balance between flexibility and efficiency during model adaptation. YOLOv8n's is more adaptable over complex models like DETR and R-CNN. It becomes particularly apparent in specialized tasks with limited training data.

Knowledge transfer experiments revealed limited benefits from datasets with similar structure.

These results offer practical guidance for adapting vision models to specialized domains.

References

[1] Ravpreet Kaur and Sarbjeet Singh. "A comprehensive review of object detection with deep learning". In: *Digital Signal Processing* 132 (2023), p. 103812. ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2022.103812. URL: https://www.sciencedirect.com/science/article/pii/S1051200422004298.

[2] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV]. URL: https://arxiv.org/abs/1506.02640.

[3] Xia Zhao et al. "A review of convolutional neural networks in computer vision". In: *Artificial Intelligence Review* 57.4 (Mar. 2024), p. 99. ISSN: 1573-7462. DOI: 10.1007/s10462-024-10721-6. URL: https://doi.org/10.1007/s10462-024-10721-6.

[4] Zhong-Qiu Zhao et al. “Object Detection With Deep Learning: A Review”. In: *IEEE Transactions on Neural Networks and Learning Systems* 30.11 (2019), pp. 3212–3232. DOI: 10.1109/TNNLS.2018.2876865.