Wildlife Detection using Deep Learning

Jorge Viñuela Pérez

Table of Contents

List of Figures

List of Tables

1. Introduction

Computer vision is a rapidly evolving field within artificial intelligence that focuses on enabling machines to interpret and understand visual information from the world. Its applications span a wide range of domains, including autonomous driving, medical imaging, industrial automation, surveillance, and wildlife conservation. Object detection, the task of identifying and localizing objects within an image, is one of the most critical challenges in computer vision, as it combines both classification and spatial reasoning. Advances in deep learning, particularly with convolutional neural networks (CNNs), have significantly improved the accuracy and speed of object detection systems [1].

The aim of the project is to train and evaluate deep learning models capable of detecting four different animal species (buffalo, elephant, rhino, and zebra) using a labelled dataset containing bounding box annotations.

The assignment involves multiple stages, including dataset analysis, training models with different levels of complexity, applying data augmentation techniques, and critically evaluating model performance. This report documents the process of preparing the data, selecting the model architectures, training and evaluating multiple models under different conditions, and reflecting on the challenges and insights gained throughout the project.

2. Dataset and Preprocessing

For this project, I decided to use a dataset downloaded from Kaggle [2], containing images of four different wild animals: buffalo, elephant, rhinoceros, and zebra. The dataset is designed for object detection tasks, with each image accompanied by its corresponding bounding box annotation, specifying the location of the animal within the frame. The dataset is organized as follows:

- **Training set:** 1052 images
- **Validation set:** 227 images
- **Test set:** 225 images

The images vary in size, background, and lighting conditions, providing a moderately challenging environment for model training. The presence of bounding boxes is important because it allows not just classification but also shows where the animal is located within the image, as shown in the following Figure.

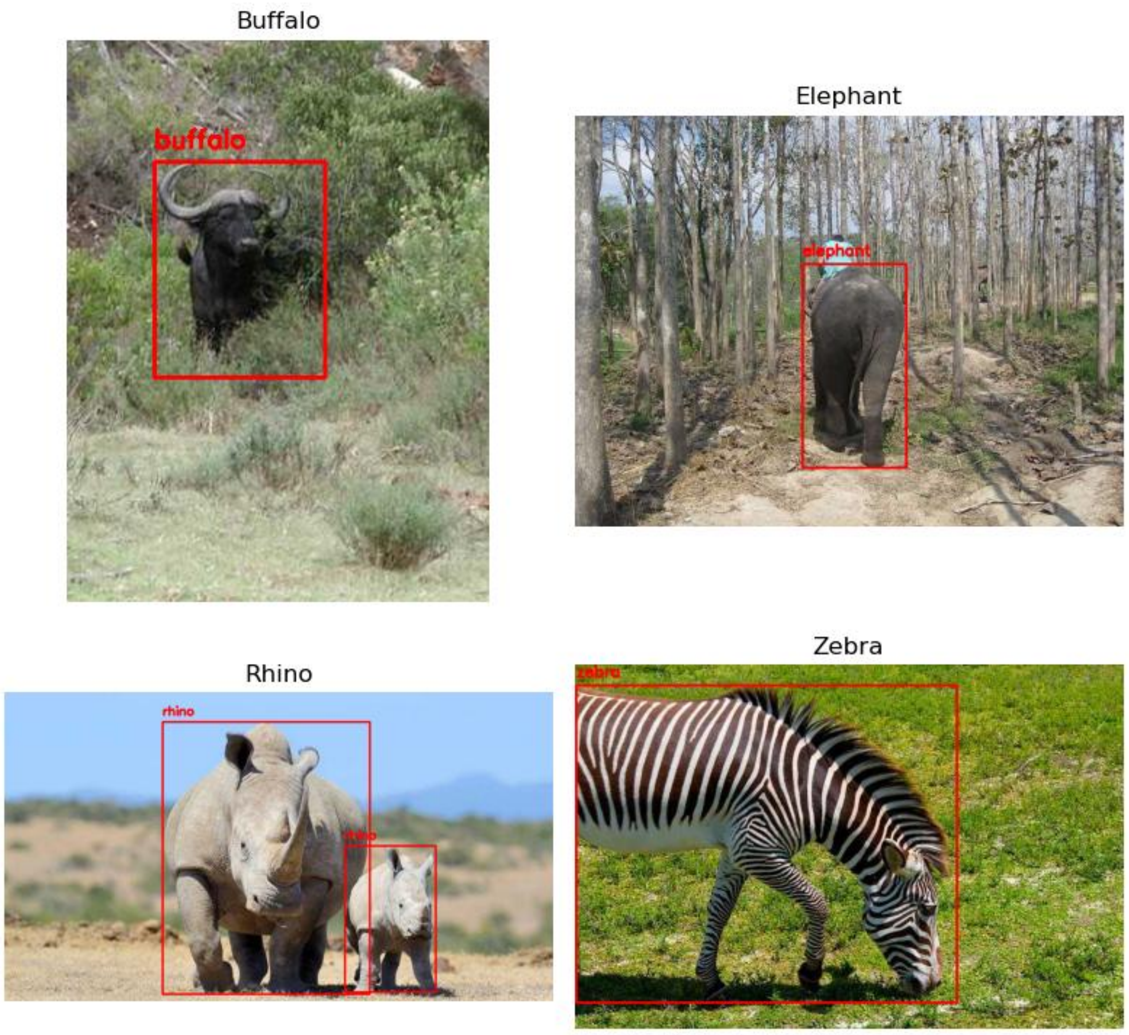


Figure 1. Sample Images from the Dataset with Bounding Boxes

Having observed the images of the dataset, the first step was to inspect and understand its structure. A key aspect of this process was evaluating the class distribution, as class imbalance can significantly influence a model's performance. For this purpose, a small script was written to parse the filenames, as each class was represented by a different number at the beginning of them, which revealed that the dataset was relatively well balanced across all categories.

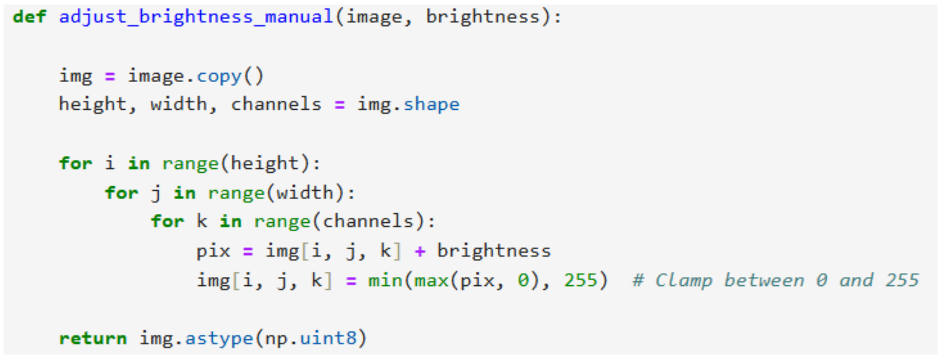
Table 1. Number of Images per Class

Classes	Number of Images
Buffalo	265
Elephant	268
Rhino	265
Zebra	254

This uniform distribution allowed me to proceed without the need to address an unbalanced dataset. Although the dataset was well-balanced across all four classes, data augmentation techniques were applied to enhance its diversity and help improve model generalization. However, it's important to note that this augmentation was only applied to the final model in my training pipeline (first models were trained on the original dataset without any additional transformations). Two different techniques were used: brightness adjustment (brighter and darker) and horizontal image flipping

2.1. Brightness Adjustment

This technique involves modifying the brightness levels of the images, simulating different lighting conditions in which animals might appear. By exposing the model to brighter and darker versions of the same scene, it learns to detect animals more reliably under varied lighting.



```python
def adjust_brightness_manual(image, brightness):

    img = image.copy()
    height, width, channels = img.shape

    for i in range(height):
        for j in range(width):
            for k in range(channels):
                pix = img[i, j, k] + brightness
                img[i, j, k] = min(max(pix, 0), 255)  # Clamp between 0 and 255

    return img.astype(np.uint8)
```

Figure 2. Brightness Adjustment Function

The brightness adjustment technique was implemented using a custom function, that created a copy of the original image and then iterated through every pixel and colour channel, adding a fixed brightness value to each pixel. The result was then clamped between 0 and 255 to ensure pixel values remained within valid RGB bounds. This manual approach allowed full control over the brightness level applied and helped simulate varied lighting conditions in the training data.

However, while implementing this technique I encountered a notable challenge related to performance. The custom function I used processes each image pixel by pixel, which made it considerably slow, especially when working with a dataset of over a thousand images. I explored alternative built-in OpenCV functions in an attempt to speed up the process. However, these faster methods often produced unintended results, such as colour distortions rather than a clean brightness change. After multiple attempts to fix the issue, I ultimately decided to prioritize reliability and correctness over speed, and proceeded with the slower, manual function that consistently produced the expected visual outcome.

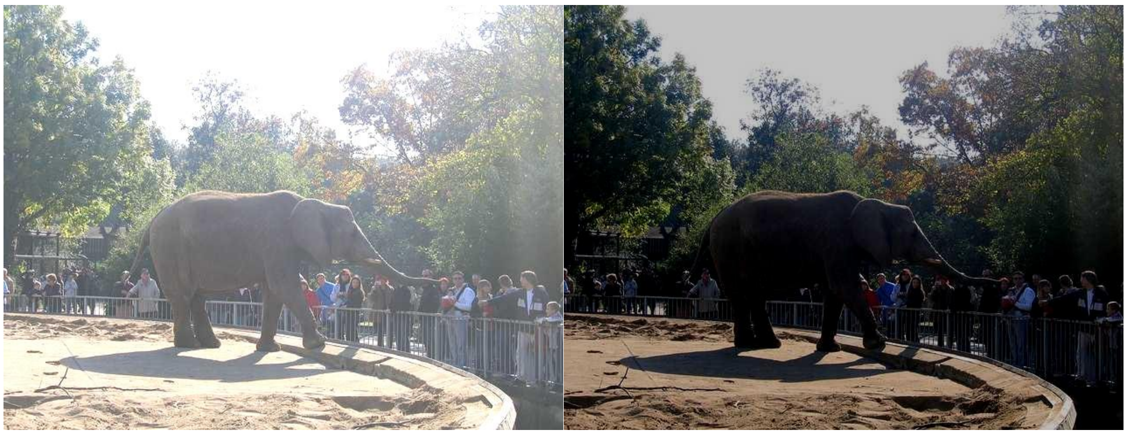


Figure 3. Example of Brightness Adjustment Data Augmentation

2.2. Horizontal Image Flipping

Horizontal flipping was applied to enhance the model's ability to detect animals regardless of their orientation. By mirroring the original images along the vertical axis, the model was exposed to both left-facing and right-facing representations of each animal, which helped reduce any directional bias and improved robustness during inference, especially in cases where an animal's position in the image varied from what the model had seen during training. The horizontal flip was performed using OpenCV's cv2.flip() function, where passing the value 1 flips the image over the y-axis. This effectively mirrors the image from left to right. The resulting flipped image was then added to the augmented dataset, providing a reversed version of the original while preserving the animal's visual features.

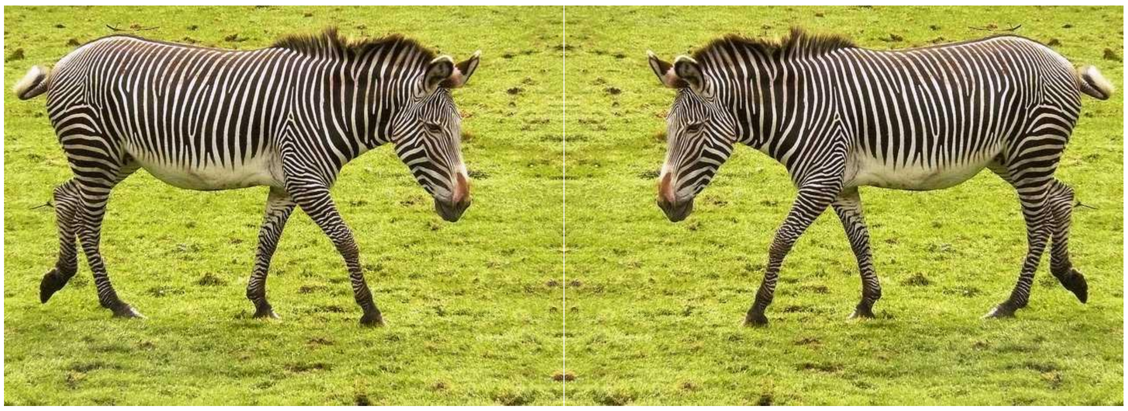


Figure 4. Example of Horizontal Flipping Data Augmentation

Although OpenCV supports both horizontal and vertical flipping, I chose to apply only horizontal flipping during augmentation, based on the nature of the dataset and the real-world behaviour of the animals being detected. Horizontal flipping is a realistic transformation, as animals can naturally appear facing either left or right in the wild. However, vertical flipping is not a common or natural occurrence in wildlife imagery; animals do not appear upside down in typical camera traps or field photography. Including vertically flipped images could potentially introduce unnatural patterns that may confuse the model, so I opted to exclude it to maintain the integrity and realism of the training data.

3. Model Architecture and Justification

Given the option to use pretrained models, I chose to work with the YOLO (You Only Look Once) family of architectures as it is specifically designed for real-time object detection, making it an ideal choice for this task, where the goal is not only to classify animals but also to accurately localize them within an image. Compared to other pretrained models such as ResNet, VGG, or EfficientNet, which are primarily designed for classification tasks [3], YOLO offers a more efficient and specialized solution for real-time object detection. Unlike classification-based pipelines, YOLO treats detection as a regression problem and processes the entire image in a single forward pass, resulting in significant speed and simplicity [4]. In addition, the dataset came with bounding box annotations already provided in YOLO format ("<class_id> <x_center> <y_center> <width> <height>"), which made it an even more natural choice, as it eliminated the need for any additional label conversion or preprocessing for object detection.

Since I was limited by the processing power of my computer, only small model architectures could be used. The options selected were YOLOv8n (nano) and YOLOv8s (small), which are fundamentally the same, following the structure of the YOLOv8 family.

Table 2. Model Summary

From	n	Params	Module	Arguments
-1	1	928	Conv	[3, 32, 3, 2]
-1	1	18560	Conv	[32, 64, 3, 2]
-1	1	29056	C2f	[64, 64, 1, True]
-1	1	73984	Conv	[64, 128, 3, 2]
-1	2	197632	C2f	[128, 128, 2, True]
-1	1	295424	Conv	[128, 256, 3, 2]
-1	2	788480	C2f	[256, 256, 2, True]
-1	1	1180672	Conv	[256, 512, 3, 2]
-1	1	1838080	C2f	[512, 512, 1, True]
-1	1	656896	SPPF	[512, 512, 5]
-1	1	0	Upsample	[None, 2, 'nearest']
[-1, 6]	1	0	Concat	[1]
-1	1	591360	C2f	[768, 256, 1]
-1	1	0	Upsample	[None, 2, 'nearest']
[-1, 4]	1	0	Concat	[1]
-1	1	148224	C2f	[384, 128, 1]
-1	1	147712	Conv	[128, 128, 3, 2]
[-1, 12]	1	0	Concat	[1]
-1	1	493056	C2f	[384, 256, 1]
-1	1	590336	Conv	[256, 256, 3, 2]
[-1, 9]	1	0	Concat	[1]
-1	1	1969152	C2f	[768, 512, 1]
[15, 18, 21]	1	2117596	Detect	[4, [128, 256, 512]]

The network is composed of 129 layers, organized into three main parts: the backbone, the neck, and the detection head. According to Ultralytics, YOLOv8 introduces several architectural improvements over its predecessors [5]:

- **Backbone**: YOLOv8 utilizes a powerful convolutional backbone for efficient feature extraction from input images. It is built with Cross Stage Partial (CSP) and C2f blocks to enhance gradient flow and reduce computational cost without compromising accuracy.

- **Neck**: The neck is responsible for feature aggregation across multiple scales. It incorporates upsampling, concatenation, and additional convolutional layers to combine spatially rich information from various backbone levels.

- **Head**: YOLOv8 uses an anchor-free detection head, which allows the model to predict object centres and box dimensions directly, simplifying the detection pipeline and improving both speed and accuracy.

While YOLOv8n and YOLOv8s share this architecture, the key difference lies in the width and depth scaling, resulting in a substantial difference in the number of parameters: approximately 3 million in YOLOv8n versus 11 million in YOLOv8s. This makes the nano version more lightweight and faster, while the small version offers improved accuracy and richer feature representation at the cost of greater computational demand. This architecture was chosen due to its balance of speed, modularity, and high detection performance, especially when paired with datasets that include bounding box annotations in YOLO format.

4. Model Training and Evaluation

After selecting the YOLOv8 architecture, I proceeded to train and evaluate multiple model variants to explore the trade-offs between performance and computational efficiency. I trained a total of three models:

- YOLOv8n (Nano)
- YOLOv8s (Small)
- YOLOv8s with data augmentation

5.1. Yolov8n (Nano)

The primary goal at this stage was to validate the training pipeline and obtain a performance baseline using limited computational resources. I set the training to run for 10 epochs, which provided a quick and effective way to evaluate the model's learning behaviour without excessive training time. This was particularly important given the constraints of training on a standard laptop.

The selected image size was of 640×640, which is the default and recommended resolution for YOLOv8 models, as it offers a good balance between detection accuracy and training speed. The batch size was set to 32, which was the largest stable value that my system could handle without running into memory issues. This allowed for reasonably efficient GPU/CPU utilization while maintaining training stability.

5.1.1. Evaluation and Results

Table 3. Precision, Recall, and F1-Score for YOLOv8n Model on Validation Set

	Precision	**Recall**	**F1-Score**
Buffalo	0.969	0.984	0.977
Elephant	0.95	0.99	0.97
Rhino	1.00	0.932	0.965
Zebra	0.992	0.992	0.992
Mean Value	0.9778	0.9745	0.976

The model achieved strong performance considering its lightweight design. Precision, recall, and F1-scores across all four animal classes were consistently high, with a mean precision of 0.978 and a mean F1-score of 0.976. The confusion matrix below shows that the model was highly reliable, particularly in distinguishing elephants and zebras, which had near-perfect classification rates.

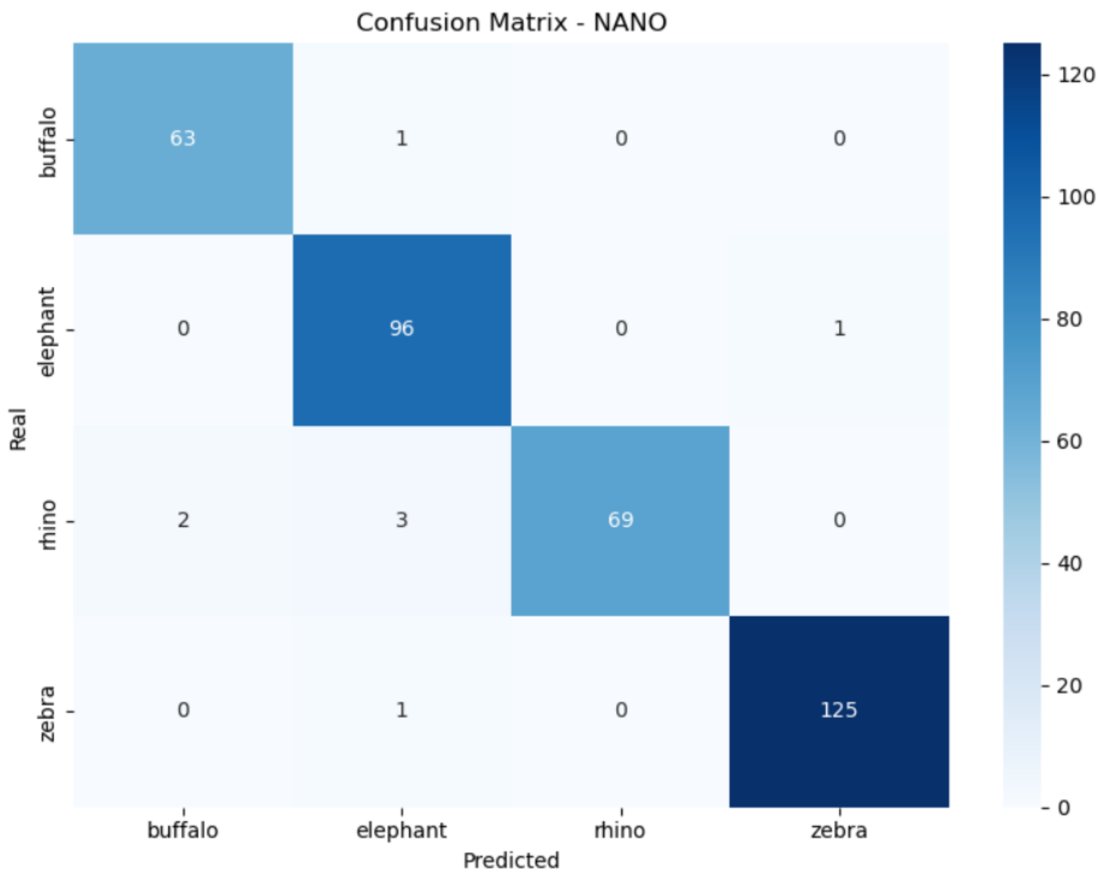


Figure 5. Confusion Matrix for YOLOv8n Model on Validation Set

However, some minor misclassifications were observed, notably with a few buffalo and rhino images being confused. These errors highlight the model's limited feature extraction capacity, a consequence of the relatively small number of parameters (about 3 million) compared to larger YOLO variants. While this nano model provided a very good baseline, its performance suggests that a more complex model, with deeper and wider layers, could extract richer features and further reduce misclassification.

5.2. Yolov8s (Small)

Following the encouraging results obtained with the nano model, I decided to train the YOLOv8s model to explore whether increasing the model's capacity would lead to better detection performance. The training hyperparameters remained mostly consistent with those used for the nano model to ensure comparability, the only difference was related to the batch size, which had to be reduced to 16 to avoid running into memory issues on my laptop, due to the larger number of parameters.

5.2.1. Evaluation and Results

Table 4. Precision, Recall, and F1-Score for YOLOv8s Model on Validation Set

	Precision	**Recall**	**F1-Score**
Buffalo	0.985	1.00	0.992
Elephant	0.96	0.99	0.975
Rhino	1.00	0.933	0.966
Zebra	0.985	0.992	0.988
Mean Value	0.9825	0.9788	0.9803

As can be extracted from the table, precision and recall metrics exceeded 0.96 for all classes, with rhino and zebra reaching near-perfect precision (1.00 and 0.985, respectively). The mean F1-score reached over 0.980, showing an improvement over the nano model's performance. Finally, Buffalo detection achieved perfect recall (1.00) and very high precision (0.985), demonstrating the model's ability to consistently and accurately recognize this class.

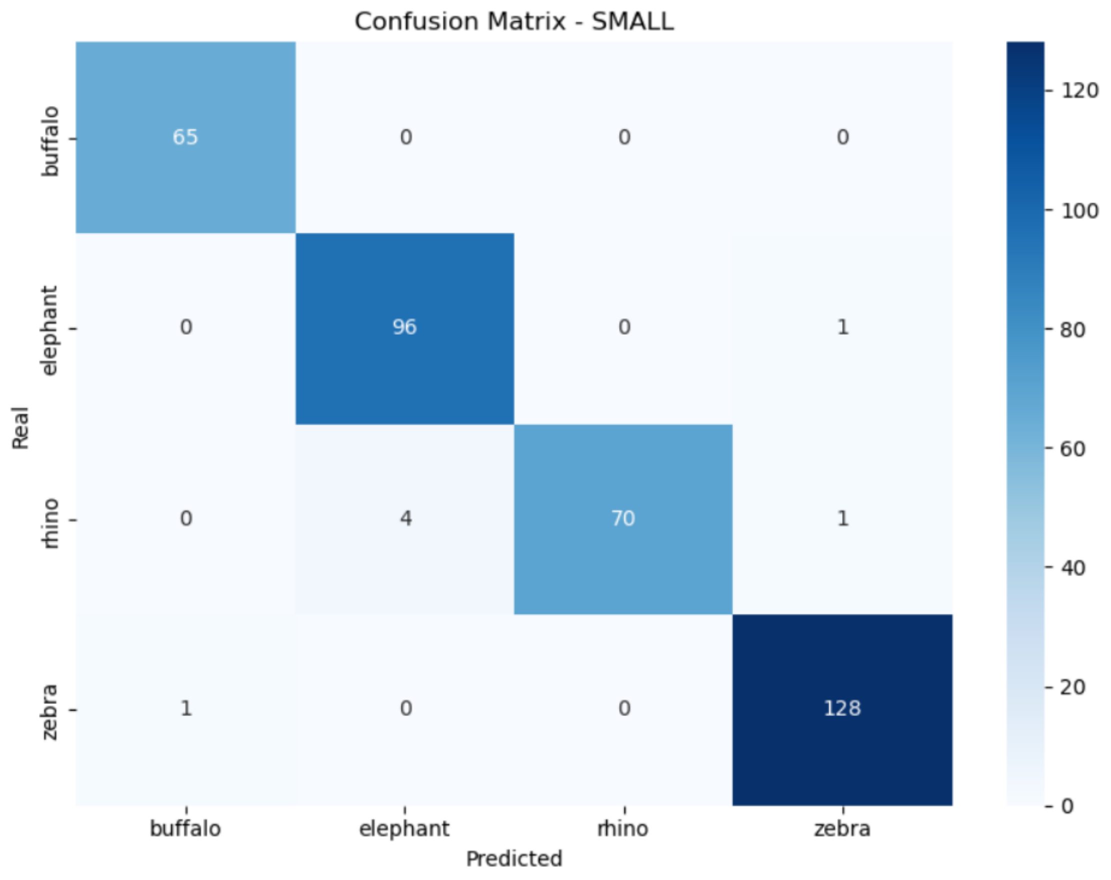


Figure 6. Confusion Matrix for YOLOv8s Model on Validation Set

Training the small model led to a noticeable improvement in detection performance compared to the earlier nano one. The confusion matrix above reveals that the model achieved perfect classification for buffalo images and maintained exceptionally high performance across all other classes. Misclassifications were extremely rare and mostly occurred between rhino and elephant images, which can be expected due to their similar shapes and sizes in certain poses.

Overall, the YOLOv8s model exhibited a stronger generalization ability, fewer misclassifications, and greater overall stability. These results highlight the advantage of using a more complex model with higher feature extraction capacity, especially when computational resources permit it.

5.3. Yolov8s with Augmented Data

Although the previous YOLOv8s model already showed excellent performance, part of the coursework requirement was to train a model using data augmentation techniques. I therefore retrained the small model, this time incorporating horizontal flipping and brightness adjustments during the training process. All hyperparameters (10 epochs, batch size 16, and image size 640) remained the same to ensure a fair comparison.

5.3.1. Evaluation and Results

Table 5. Precision, Recall, and F1-Score for YOLOv8s Model with Data Augmentation on Validation Set

	Precision	Recall	F1-Score
Buffalo	0.984	0.984	0.984
Elephant	0.96	0.99	0.975
Rhino	1.00	0.959	0.979
Zebra	0.992	0.992	0.992
Mean Value	0.984	0.9813	0.9825

The evaluation results demonstrate that the model maintained an extremely high level of performance. The confusion matrix shows very few misclassifications across all classes, with all the classes achieving precision and recall values above 0.96. In particular, the mean precision (0.984) and mean F1-score (0.9825) slightly improved compared to the non-augmented model.

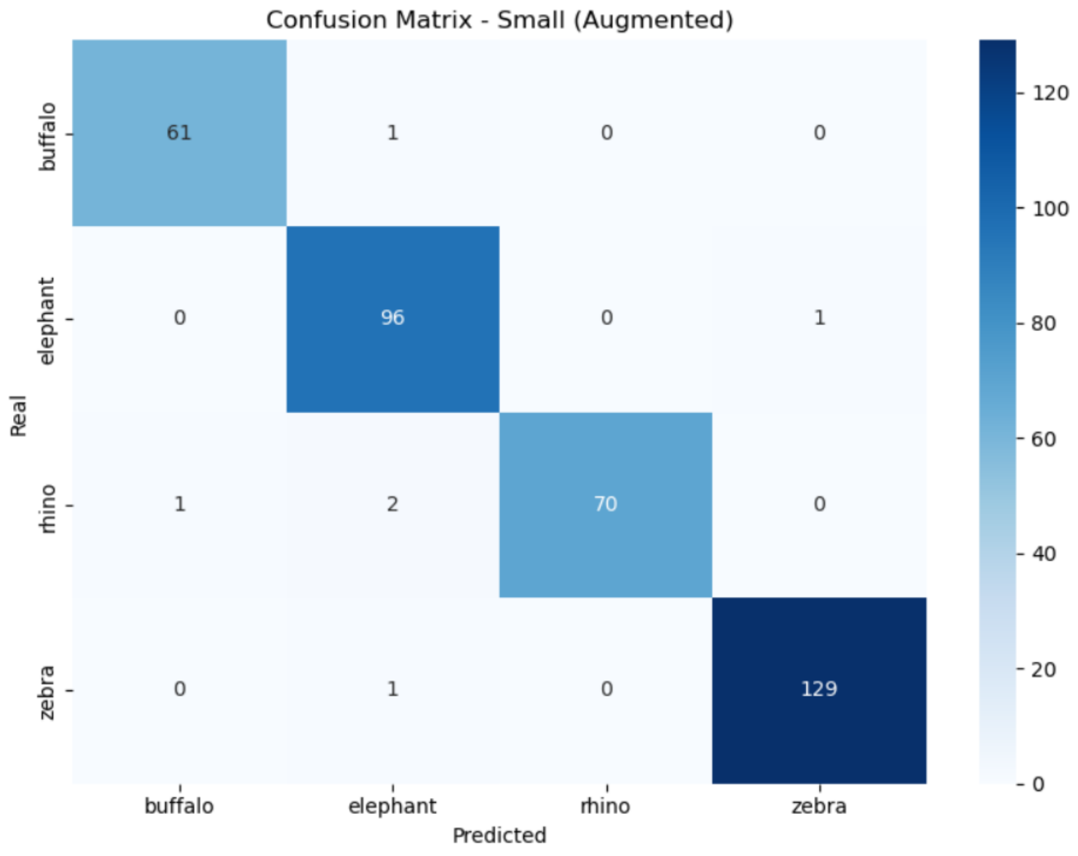


Figure 7. Confusion Matrix for YOLOv8s Model with Data Augmentation on Validation Set

Interestingly, while augmentation introduced small fluctuations in individual class scores (e.g., a very slight decrease in buffalo recall), the overall robustness and consistency of the model improved. This suggests that the augmentations helped the model generalize better by making it more resilient to variations in image conditions, one of the main goals of data augmentation.

5. Conclusion

This project provided a complete experience in applying deep learning techniques for object detection, covering dataset analysis, preprocessing, data augmentation, model training, evaluation, and critical reflection. Starting with the lightweight YOLOv8n model allowed me to quickly validate the setup and establish a strong performance baseline, while training the more powerful YOLOv8s model demonstrated the benefits of using architectures with greater feature extraction capacity.

The introduction of data augmentation techniques, specifically horizontal flipping and brightness adjustment further enhanced the model’s robustness. Although the performance improvements were relatively small due to the already high baseline, the use of augmentation proved valuable in increasing the model's generalization ability and resilience to variations in image conditions.

Through this coursework, I developed a deeper understanding of managing computational limitations, selecting suitable model sizes, applying effective augmentation strategies, and critically interpreting evaluation metrics. I also realized the importance of systematic experimentation and the trade-offs between model complexity and resource availability. Overall, this assignment significantly strengthened my practical skills and theoretical knowledge in building and evaluating modern object detection models.

References

[1] X. Wu, D. Sahoo, and S. C. H. Hoi, "Recent advances in deep learning for object detection," *arXiv preprint arXiv:1908.03673*, 2019.

[2] B. Ferreira, "African Wildlife: A Data Set of 4 African Animals in the YOLO Labeling Format," *Kaggle*, 2020. [Online]. Available: https://www.kaggle.com/datasets/biancaferreira/african-wildlife

[3] F. Fisranda, F. Austin, and M. Sadrawi, "Comparison Analysis of YOLO, ResNet, and EfficientNet for Contagious Skin Diseases," in *Proc. Int. Conf. on Recent Innovations in Systems Engineering (ICORIS)*, Oct. 2023. [Online]. Available: https://doi.org/10.1109/ICORIS60118.2023.10352306.

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 779–788.

[5] Ultralytics, "YOLOv8 Models," *Ultralytics Documentation*, [Online]. Available: https://docs.ultralytics.com/models/yolov8/. [Accessed: Apr. 25, 2025].