Practical implementation of YOLOv8 model for detection and tracking of wildlife animals

# Abstract

# This study investigates the viability of YOLOv8 for wildlife tracking and monitoring, highlighting its adaptability and accuracy. Employing YOLOv8 on a savannah wildlife dataset reveals its proficiency in swiftly recognizing various animal species in Nat Geo Wild videos.

# 

# Introduction

In the domain of comuter vision, object detection is one of the most significant fields. Findings in this area find applications in rapidly advancing fields like autonomous vehicles, medical imaging, face recognition, and all kinds of monitoring, from human to wildlife.

Among the algorithms for object detection, You Only Look Once (YOLO) stands out as one of the best Open Source models for its real-time efficiency and accuracy.

Here I will be focusing on implementing YOLO for the purpose of detecting savannah animals, a task that aligns well with the algorithm’s capabilities. The YOLO algorithm’s unique grid-based approach, which predicts object locations and class probabilities simultaneously, grants it an advantage in scenarios where speed is essential.

This implementation wil cover data preprocessing, model training and its application to Nat Geo Wild Footage.

# Implementation

## Dataset

For this savannah animal detection implementation, I employed a curated animal detection dataset gathered from Kaggle. This dataset comprises a diverse collection of images showcasing various savannah animal species in their natural habitats. Each image is annotated with bounding box coordinates and corresponding class labels, facilitating the training of our YOLO model.

The Labels are already ready for the model, made in the format of

<class-index> <x> <y> <width> <height>

Where x and y are the coordinates of the bounding box followed by the width and height of the bounding box. The only thing that was needed was to change the name of the class with the class index as the dataset contains the names in the labels.

Because this is a homemade project the dataset has to be limited to fewer animals than the original as the bottleneck of 16Gb ram kept crashing the program when running even 20 animals so this animal detection will be working on 11 of the 21 animals provided.

The animals to detect will be "Cheetah","Crocodile","Elephant","Giraffe","Hippopotamus","Leopard","Lion","Monkey","Rhinoceros","Tiger","Zebra".

## EDA

Before proceeding with the model training process, I conduct a very basic Exploratory Data Analysis (EDA) on the animal detection dataset. EDA serves as a preliminary step to gain insights into the dataset’s composition and characteristics.

A graph on a black background

Description automatically generated

This plot represents the amount of images we have of each class that we will be training and testing on. The blue bar represents the training data and the orange represents the test data. As we can observe the monkey is somewhat overrepresented in the data as compared to the rest of the animals. This might create a bias in the model further on.

## Preprocessing

Since the data is mostly in the correct format there is not much preprocessing necessary. There are no serious outliers, too much noise or anything like that.

The entire preprocessing necessary is resizing the image and then adequately changing the label data to fit with the resized image. Additionally, a new folder system is created for more practical access to the training and testing data.

We separate the animal classes we will be detecting into a separate training folder where they aren’t separated into subfolders by their class and resize them to 640 by 640 to fit with the requirements of the YOLO model. Then we create new labels where the class name is changed for the class index that we decided on for the corresponding class and then normalize the bounding box coordinates and height and width to correspond to the new size of the images.

The last necessary step before starting the training is to create a YAML file. This is a required file for YOLO training where the necessary data for the paths and classes is kept. The YOLO model then reads this file as a replacement for what would normally be kept in the parameters of the model.

A screenshot of a computer

Description automatically generated

This is what a simple YAML file looks like for the purposes of this model training.

## Image Processing

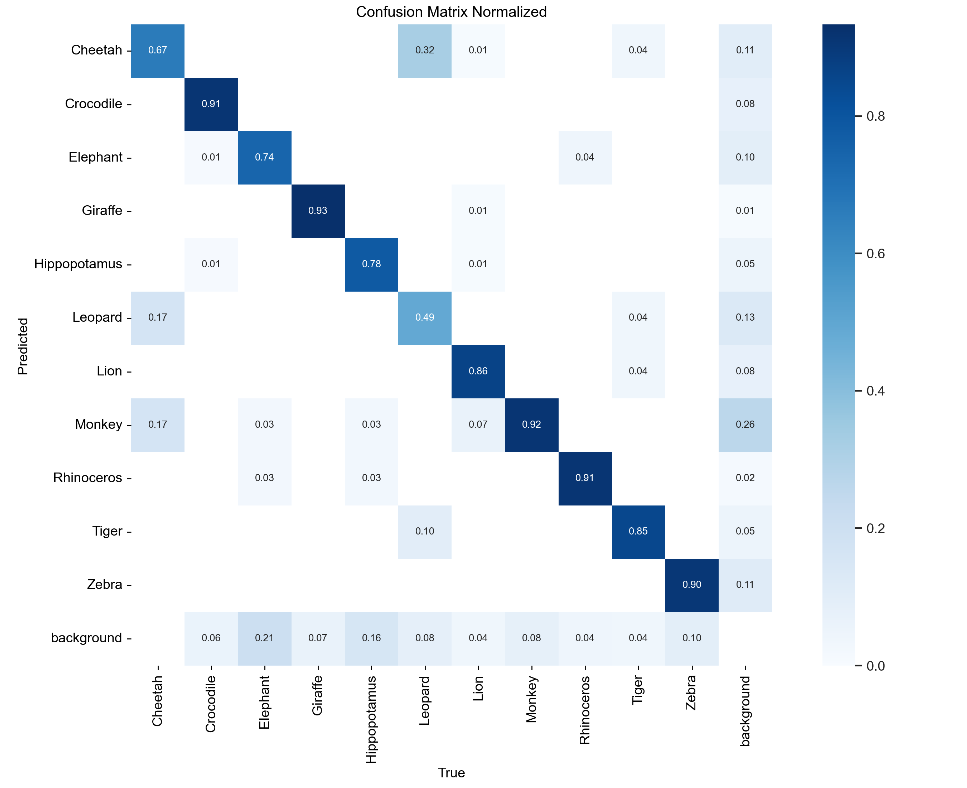
After some research on the model, it turns out that the YOLOv8 (and any version after v3) can process images of any size as long as both sides are a multiple of 32. This is because the maximum stride of the backbone is 32 and it’s a fully convolutional network. Still when processing a batch of images they have to be resized into the same size(this is because tensors of different shapes cannot be concatenated), and regardless of whether it is a batch it has to be a square image. The YOLO v8 implementation automatically solves issues with different sized images by taking the largest side of an image and resizing it to a multiple of 32 then resizing the other side to keep the aspect ratio and using padding to get it to a multiple of 32 as well so that the aspect ratio is preserved. This means we can input the images into the model as they are.

An additional change I made was changing the colors that the images save as. It appears as though the way cv2 works has changed from the sources I used to learn how to use it so all the images were saved to BGR instead of RGB. I fixed this issue and now the correct colors are used to train the set.

## Model Training

For the Model Training a pre-trained model is used. YOLOv8 has a pretrained model which is optimal for training your own datasets on. We overwrite the old classes and train it on our own data (the animal dataset) The model is loaded and we run it for 50 epochs on the data. This will most likely result in an overfitting on the training data however the YOLO architecture automatically saves the weights for the best model which is updated after each epoch, so we can overtrain and then just take the one that gave the best results afterwards.

The YOLO Library has several useful mechanisms aside from just saving the best weights that make it easy to work with. It has built in tracking of important features of the model such as the F1 curve, P curve, PR and R curves as they progress through the epochs. It also automatically generates a confusion matrix on how well the model predicts each class at the end of the training process.



This is the confusion matrix of the validation dataset from the last trained model. We can see that it seems to be well trained on some classes while its lacking in its ability to detect others. The monkey and giraffe for example seem to be easy to detect while a leopard gets mistaken for a cheetah or a tiger about as often as it is predicted correctly. This is normal as both the cheetah and the leopard are somewhat underrepresented in the dataset as well as having much more similar features unlike most other animals.

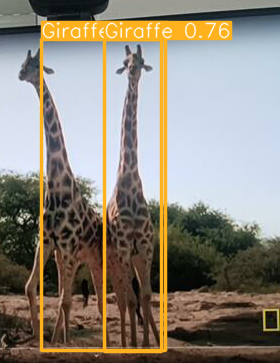
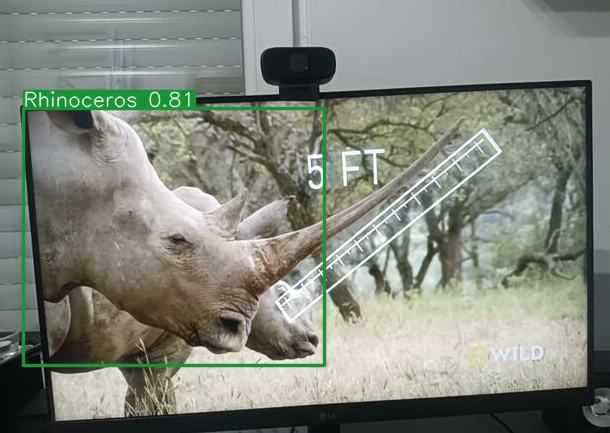
A graph of different colored lines

Description automatically generated

On the image above we have the F1-Confidence curve. This curve represents the values of the F1 score depending on the confidence threshold. YOLO automatically calculates the optimal confidence value for us as having 0.8 F1 score on average when set to a confidence threshold of 0.59.

## Results

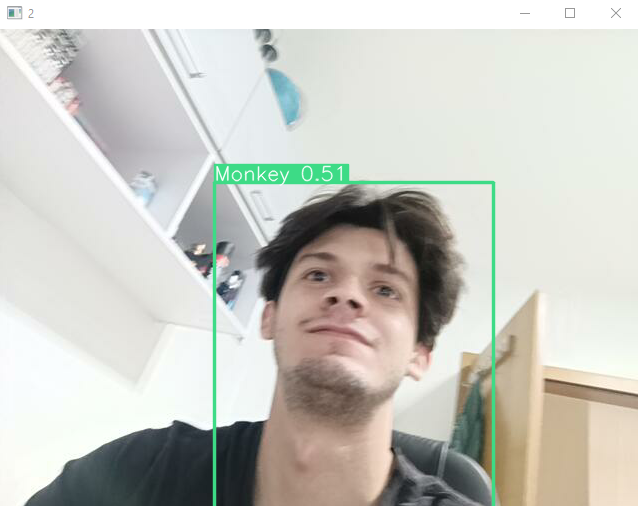
As the goal of this project is to conduct animal detection on a live feed we will have to connect a camera with the program and use the feed as prediction instead of images as we did during validation.



These are some examples of how the model predicts.

It creates a bounding box around the animal that is detected writing which class it belongs to and a confidence score of how likely that class is to be correct.

The model however does often make mistakes. Here is a screenshot of it detecting me as a monkey.



This is likely to be because of the overrepresentation of the monkey in the dataset along with having slightly similar features to a person. It is difficult to make it detect a person as a monkey but if you shake the camera you can do it.

## Model training 2 and the results

I have decided to make this a separate segment to showcase the difference in the model before and after implementing the changes in resizing. I’ve also changed the train val test split so that 20% of the training set is used for validation and then the test part is used for testing (which also has to be done using the val() function as that one returns graphs of the results. I just use a separate yaml file to differentiate which one I want to use.

The results are greatly improved with this being the new validation F1-confidence curve

A graph of a graph of a number of numbers

Description automatically generated with medium confidence

The testing F1 curve shows us that the model definitely isn’t as good as it seems on the validation set as for example it can hardly distinguish a cheetah.

A graph of different colored lines

Description automatically generated

Here is the curve with the test set. We can comfortably say the model is greatly improved with the changes that were implemented because its F1 score is nearly the same as the one the previous model got for just the validation set which as we established is never as good as when tested.

Here is the confusion matrix for the test set showing us what kind of mistakes the model most usually makes. It shows that the leopard and cheetah are the most difficult to distinguish from other classes such as lion monkey and even the background itself.

A screenshot of a computer

Description automatically generated

Thanks to the great implementation of the val() function we get a lot more data to work with when testing.

A screenshot of a computer screen

Description automatically generated

These are the testing results.

Box Precision and recall show the proportion of true positive predictions for the class out of all the predicted bounding boxes(precision) and the proportion of true positive predictions out of all the instances of the class in the dataset(recall).

As with the previous results we can see the cheetah being very unrecognisable by the model with by far the lowest scores out of any animal.

mAP50 is the Mean Average Precision with a threshold of 0.5. it is a common metric used for evaluation of object detection models. It measures the precision-recall trade off across different confidence thresholds for the bounding boxes.

mAP50-95 is the same thing just with a different threshold range and provides a more comprehensive evaluation of the model.

Generally a mAP50 score of 0.7 or higher is considered good, meaning the model can reliably detect objects while a score of above 0.9 is considered excellent.

The mAP50-90 scores above 0.5 are good while 0.7 or higher are excellent indicating high accuracy in detecting objects.

This model is considerably good having a 0.8 mAP50 score and a 0.68 mAP50-90 score.

Here is a sample of the validation batch. The first image is the original labels while the second will be the model’s predictions

A screenshot of a computer

Description automatically generated

A collage of different animals

Description automatically generated

## Conclusion

In conclusion, our exploration of YOLOv8's applicability to wildlife tracking and monitoring yields promising results. YOLOv8's integration of cutting-edge techniques underscores its adaptability and accuracy across various scenarios, positioning it as a robust solution for real-time, multi-object detection, especially in the context of wildlife. This practical implementation using a customized dataset demonstrates YOLOv8's adeptness in swiftly and accurately identifying distinct animal species within Nat Geo Wild footage.

Sources:

<https://www.kaggle.com/datasets/antoreepjana/animals-detection-images-dataset>

<https://github.com/ultralytics/ultralytics> (YOLOv8 model)