VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

STATISTICAL MACHINE LEARNING

FINAL PROJECT

OBJECT DETECTION WITH YOLO

Students: 21127004 - Tran Nguyen An Phong
21127135 - Diep Huu Phuc
21127307 - Tong Gia Huy
Class: 21CNTThuc
Instructors: Le Long Quoc
Ngo Minh Nhut

Ho Chi Minh City, August 2024

Abstract

Computer vision has long been one of the main focus in the field of computer science. This branch aims to utilize images or videos, and the computational power of machineries to perform tasks like classification, object detection, and recognition. Since the invention of convolutional neural network (CNN), computer vision models have been enhanced tremendously. Different from normal feed-forward networks, CNN utilizes kernels to extract regional features from the image, thus making calculations more accurate. Architectures like You Only Look Once (YOLO) leverages this advantage of preserving dimensionality to perform robust and high precision object detection. In this project, we aim to establish an overall understanding of this series of models, as well as use it to construct our own detection website. We will discuss the technicalities and notable features of YOLO, as well as give an overview of our website and details on our own model.

TABLE OF CONTENTS

1 INTRODUCTION

You Only Look Once (YOLO) is a well-known series of object detection models. Throughout the years, more revised verions have been developed and published, which each later models having more modification thus boosting their accuracy and performance.

In this project, we aim to establish an understanding in YOLO's architecture. We also used our own dataset to retrain the model, a web interface for object detection is also developed. In this report, we present our overview on YOLO's features and our website's development. Details on our training process, including data, evaluation metrics, and result are also discussed.

2 OVERVIEW ON OBJECT DETECTION MODELS

2.1 Architecture

Object detection models have a number of components that help extract features, perform detection, and some additional supporting modules. In this section, we give an overview of the architecture of such models.

Backbone. As the name suggests, this part of the model is very important to the whole architecture as it is tasked with extracting the features from input images. Instead of encoding the features into a vector like in convolutional network for classification, the backbone network save the features into a stacked feature maps. This approach is crucial for preserving spacial information of the pixels, which allows for accurate detection of object throughout the entire image. Some widely used networks for this task include: CSPDarknet53 (YOLOv4, Bochkovskiy et al. 2020), and EfficientNet (YOLOv8).

Neck. This section is an optional part of the model and lies between the feature extraction stage (backbone) and detection stage (head). The purpose of this part is to construct multiple scales of the feature maps obtained from backbone. By expanding the dimension of the feature maps, the head can detect objects at different sizes. Some widely used methods for this part are: Spatial Pyramid Pooling and Path Aggregation Network (YOLOv4),

Head. One last main component of the object detection is responsible for performing the actual detection on the feature maps. For architectures that make use of the neck stage for enriching the scales, detection is performed at each scale independently from each other. Small scale maps are used for detection of large objects, and larger scale maps are used for smaller objects. It is in this stage that the bounding box for each object are defined, a classification probability is also calculated.

2.2 Additional features

In addition to the base architecture, some techniques for boosting the model's performance can also be incorporated during various stages. Bochkovskiy et al. 2020 call these adjustments *Bag of Freebies* and *Bag of Specials*. In this section, we give a simple explanation and example for each of those terms.

- Bag of Freebies: a set of techniques or methods that can alter the training strategy to improve the prediction accuracy, namely: Data Augmentation - used to alter the input images which can help create diversity in training data, Semantic Distribution Bias, and Objective Function of Bounding Box Regression.
- Bag of Specials: a set of modules and post-processing methods that can further improve the model's accuracy exchanging for the model's inference time. These methods can range from enhancing features to introducing advanced mechanisms.

3 REQUIREMENT 1: Web interface for pre-built object detection model

In this section, we present the design of our web interface, along with the integration of pre-trained YOLO model to perform object detection. We used Ultralytics' version 10 medium model for the detection task. The model is loaded onto global variables at the website's booting stage. If the pre-trained `.pt` file is not found on the local machine, it will be automatically downloaded. We present screenshots of the interface in Figure 1 and 2.

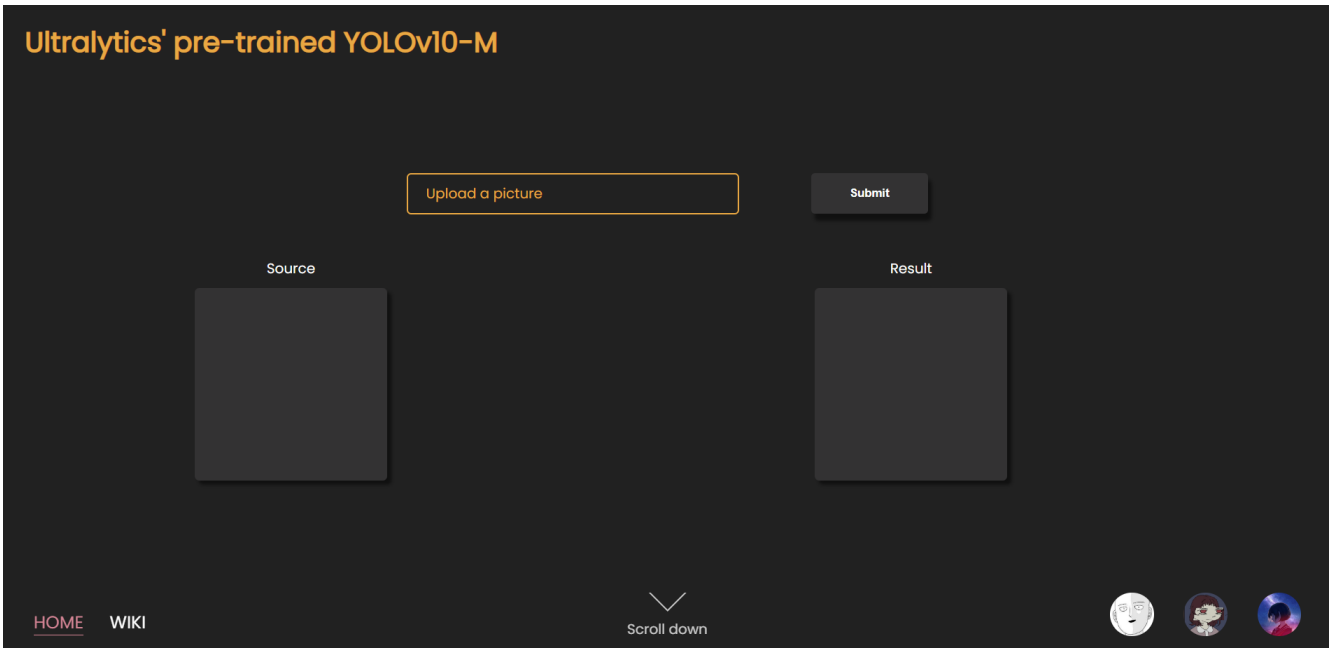

Figure 1. Default user interface of object detection task using Ultralytics's pre-trained model.

4 REQUIREMENT 2: Custom model and application building

4.1 Data Preparation

In this part we will discuss about the data source and its distribution information. We will not go specifically on the coding aspect of it because it is messy and also unnecessary in this report.

4.1.1 Data source

Although there are many image detection datasets available online (mainly on Kaggle platform), most of them do not meet our requirement which is the ability to filter by class (in our case animal) and have label written in YOLO format. And because our main goal to train the YOLOv8 model so we must come up with a solution to acquire the dataset as freely as we will, which will be done by the help of Google Open Images Dataset V7 and `fiftyone` library.

YOLOv4: Which is implement by us by Pytorch and perform all its duties with detection box format as `[id, x1, y1, x2, y2]` (which is different from both Google Open Images Dataset V7 and YOLO format). Fortunate for us that there is dataset on Kaggle with this format, also because we don't have high demanded on this model, so it is convenient to leverage resource that are already available. We will use a subset of the Animals Detection Images Dataset to work with this model.

YOLOv8: As mentioned above, our database will be Google Open Images Dataset V7 and the required data will be crawl and transform into YOLO format with the help of `fiftyone` library.

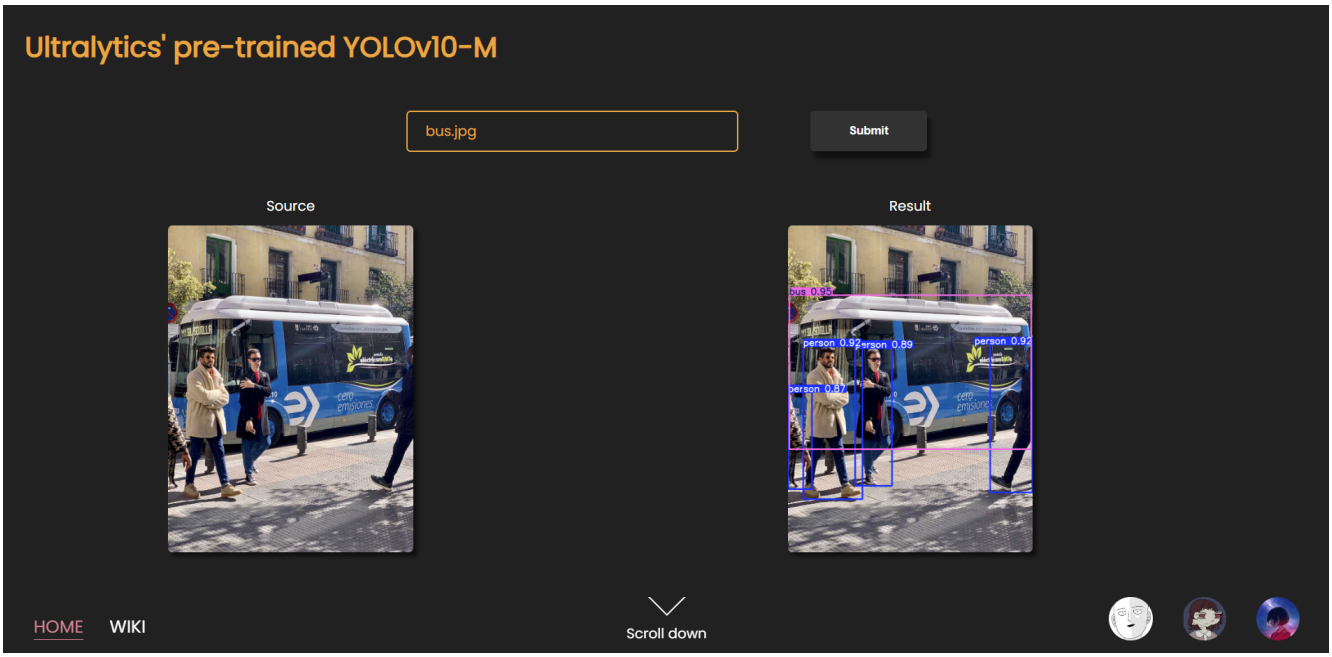

Figure 2. Interface of Ultralytics' model after being prompted with an input image.

By doing this, we can crawl data freely with the classes we need and as many as we wan (on the limit that the database can provide). There is also one more advantage of crawling our own data, which will be explain in the next part.

```
def download_animal(animal):
    classes = []
    classes.append(animal)
    dataset = fiftyone.zoo.load_zoo_dataset(
        "open-images-v7",
        split="test",
        label_types=["detections"],
        classes=classes,
        max_samples=400,
        dataset_name="Animal_test_" + animal
    )
    dataset.export(
        dataset_type=fiftyone.types.YOLOv5Dataset,
        classes=classes,
        split="test"
    )
✓ 0.0s
```

Figure 3. A part of code using `fiftyone` to download and convert to YOLO format.

4.1.2 Data distribution

YOLOv4: We pick a subset of 9 classes from the original dataset, which have highly uneven distribution (Figure 4), which is one of the reasons why we have to crawl our own dataset for YOLOv8. There are a total of 1858 images for training and testing.

YOLOv8: We choose to pick 22 class from the database and divide it into three complete division of train/test/validation, which each have 7094/3691/1692 images respectively. So, the ratio for train/test/validation is nearly 57/30/13.

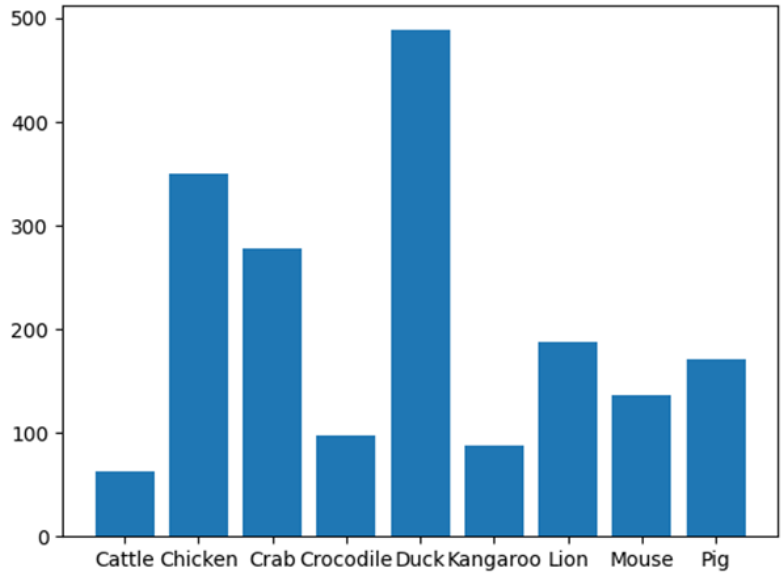

Figure 4. Distribution of the data used to train our YOLO4 model.

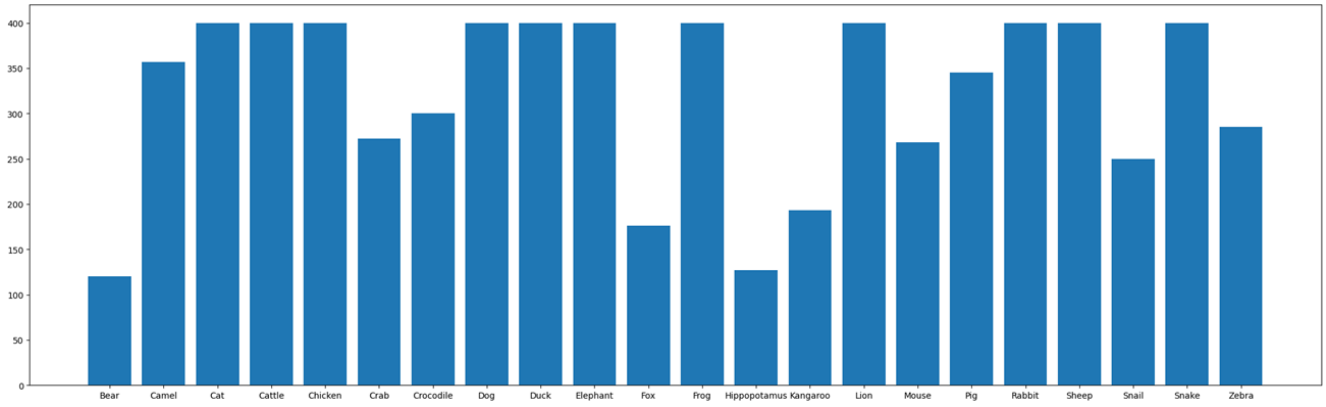

Figure 5. Distribution of the data used to train our YOLO4 model.

Figure 5 is a distribution plot for the training set, which is more evenly distributed thanks to the help of crawling tool. Although there remain some classes that have a small number of images because that is the providing limit of Google Open Image Dataset, it still has a reasonable quantity which is enough to avoid the bias of providing too many or too few data for a some class.

4.2 Training

4.2.1 How to train a YOLO model

Because the YOLO architecture comprises of a Backbone network (which is used to extract feature from image) and a Detection head (which use to predict bounding boxes, objectness scores, and class probabilities) so in summary, we first need to train the backbone network so the information it extracts is more useful for the detection layer. Then we apply for regression training using loss function for the Detection head so it can at the same time predict bounding box and performing classification better.

A training process go through these step:

- Forward Pass: The input image is passed through the model. The model predicts bounding boxes, objectness scores, and class probabilities for each grid cell.

- Loss Calculation:
  - Bounding Box Loss: Measures the error between the predicted bounding boxes and the ground truth bounding boxes using mean squared error.
  - Objectness Loss: Evaluates how well the model predicts the presence of an object in a bounding box using binary cross-entropy loss.
- Class Loss: Measures the error between predicted and true class probabilities using cross-entropy loss.
- Backpropagation: The total loss is backpropagated through the network to update the weights using an optimization algorithm like Stochastic Gradient Descent (SGD) or Adam.
- Iteration: This process is repeated for multiple epochs until the model converges to a set of weights that minimizes the loss function.

When talking about training a YOLO model, we often skip the Backbone training part as the Backbone network is highly universal and can be trained once for many different Detection head. And by knowing that, it is recommend transferring the Backbone network weight from notably well-trained model such as Darknet53 instead of training it's from scratch. Now the training process mainly focus on training the Detector.

YOLOv4: Because we only use this model for further understanding of how a YOLO model training process work, so we only trained this model for 100 epochs. The training code is created based on the spirit of the training process we mention above so it is not yet polished and many flaws may still remain. Which was alright considering the research and self-taught purpose of this model. The code used for this stage is referenced from Tianxiaomo 2021.

4.2.2 Train YOLOv8 model by Ultralytics and Kaggle

The reason for this being separated from part a is because we going to use the help of Ultralytic framework for most of our work in this project and because the training process is heavy, so it is going to be hold in Kaggle. A training process by using Ultralytics have following step:

- Prepare dataset in YOLO format (which is discussed in Section 4.1).
- Prepare a configuration file to guide Ultralytics for the new classes and train/val dataset you want to use.
- Host these dataset and configuration file to Kaggle since we use Kaggle to perform the training process.
- Tell Ultralytic to carry the weight for Backbone network from its base YOLOv8 model and using our new config to train for animal detection problem. While training, we need to specify the number of epochs and the input image size of the training dataset, which in our case are 200 epochs and 640.
- When the training process is done on Kaggle, the notebook return output contains the trained model and several insight report of the training process.

4.3 Result and Evaluation

For the sake of simplicity, we are going to evaluate only the YOLOv8 model trained by Ultralytics, also because this is the most potential model to have high evaluation score. Since we are working with an object detection model, the most suitable metric for grading should be mAP. Mean Average Precision (mAP) score is a popular evaluation metric used in object detection tasks to measure the accuracy of a model in detecting objects. It is particularly useful because it considers both the precision and recall of the model's predictions. Following is a breakdown of what the mAP score is and how it is calculated:

- The precision-recall curve is a plot that shows the trade-off between precision and recall for different threshold values. As you increase the threshold for considering a detection as positive, precision generally increases, but recall decreases, and vice versa.
- AP is the area under the precision-recall curve for a particular class. It is calculated by taking the average of precision values across all recall levels from 0 to 1. In some cases, interpolation is used to ensure that precision is monotonically decreasing. For each object class, the AP is calculated separately.
- mAP is the mean of the Average Precision (AP) values across all object classes. It provides a single score that summarizes the model's overall performance across all classes.
- Ultralytics definition of mAP point is a little bit different, they define class' mAP as a mean between all "easy detection" of one class. They also have a model's mAP, which is mean of all class mAP value. Ultralytic provide mAP score calcuted under many IoU (intersection over union) threshold and we will chose mAP-50 as the common metric in this report.

4.3.1 Evaluation of the training process

Ultralytics provide mAP-50 score evaluated per epoch using the validation set. Since we are training the model on 200 epochs it is redundant to show mAP score for every epoch in this report. Instead, we report mAP-50 score every 20 epochs in Table 1 (mAP-50 score for every epoch can still be accessed through the notebook output or the "`result_per_epoch.csv`" file in "`modelEvaluating`" folder).

Table 1. Reported mAP-50 score of the model while training every 20 epoch.

Epoch num	mAP-50
20	0.50381
40	0.65829
60	0.71419
80	0.736
100	0.74504
120	0.75555
140	0.76199
160	0.76561
180	0.76819
200	0.76527

As can be observed from Figure 6 and 7, all the metric of our model (including mAP-50) saw significant changes while training less than 100 epochs. After 150 epochs, while there are still

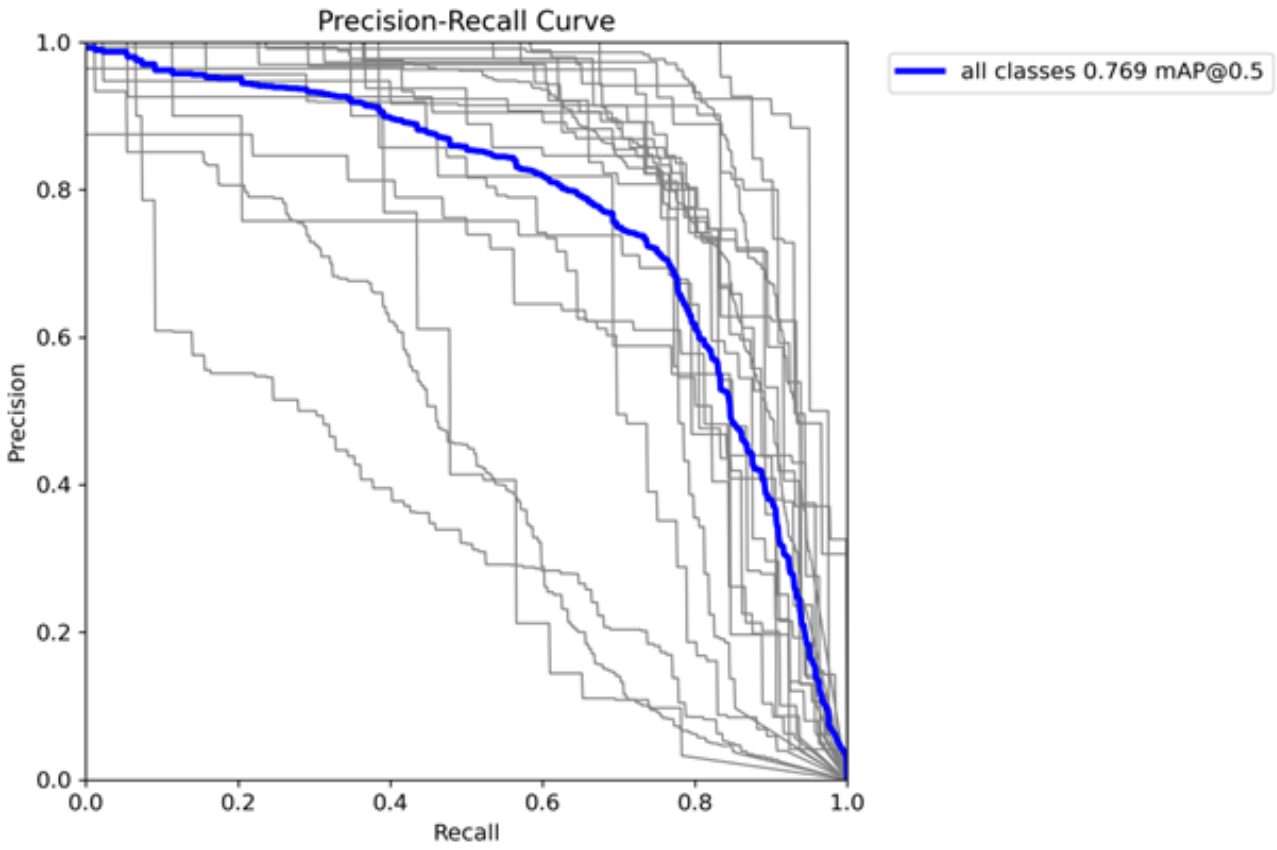

Figure 6. Precision-recall curve.

notable changes, we can almost say that the value hit its plateau. Maybe with the current dataset and configuration, we just need to train our model for 150 epochs.

In summary, our trained model achieves the mAP-50 score of 0.769, which is a fairly good point (a model with mAP-50 score above 0.5 can be seen as usable). Almost all class have their own mAP-50 score above 0.5, except for some special case like Bear, Cattle or Sheep. The explanation for the oddly low score of Cattle and Sheep may need a more deep-down look into, maybe those classes have low numbers of train images or there are too many variation of animal classified as Cattle.

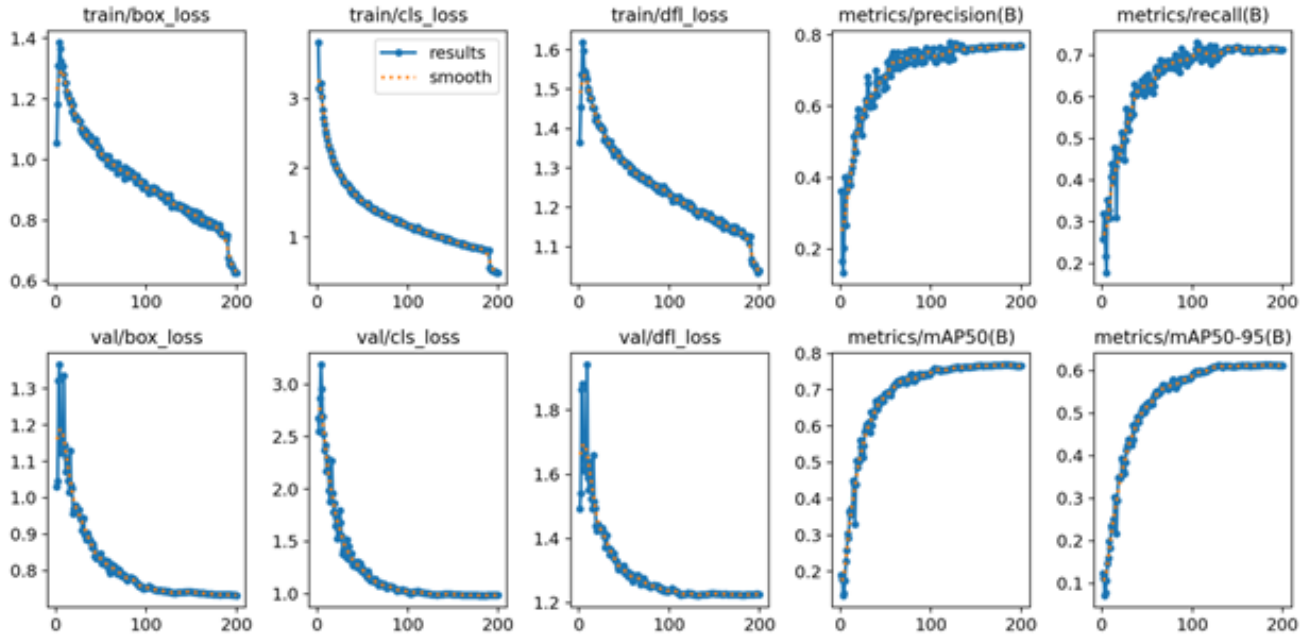

Figure 7. Metric report of the model while training.

```
Model summary (fused): 168 layers, 3,009,938 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 106/106 [00:15<00:00,  6.93it/s]
                   all       1691       2260      0.769      0.711      0.769      0.614
                  Bear         21         23       0.43      0.478      0.515      0.462
                 Camel         25         30      0.735      0.833      0.856      0.747
                   Cat        345        389      0.962       0.78      0.908      0.756
                Cattle        134        259       0.62      0.405      0.445      0.328
               Chicken         49         72      0.828      0.738      0.809      0.651
                  Crab         66         72      0.916      0.875      0.931      0.697
             Crocodile         39         53       0.92      0.655       0.82      0.595
                   Dog        400        480       0.92      0.624      0.845      0.705
                  Duck         68        149      0.943      0.661      0.787      0.573
              Elephant         33         56      0.782      0.786       0.84      0.681
                   Fox         41         44      0.529      0.795      0.718      0.671
                  Frog         39         48      0.861      0.833      0.883      0.721
          Hippopotamus         10         13      0.535      0.846      0.746      0.664
              Kangaroo         21         32      0.591      0.719      0.666       0.52
                  Lion         57         65      0.827      0.769      0.819      0.684
                 Mouse         45         55       0.89      0.734      0.833      0.513
                   Pig         51         76      0.757      0.592      0.669      0.539
                Rabbit         64         71      0.855      0.748      0.856       0.72
                 Sheep         62        122       0.37      0.402      0.361      0.261
                 Snail         34         40      0.917        0.9      0.941      0.744
                 Snake         61         66      0.903      0.707      0.897      0.654
                 Zebra         26         45      0.829      0.753      0.772      0.632
Speed: 0.2ms preprocess, 1.9ms inference, 0.0ms loss, 1.0ms postprocess per image
Results saved to runs/detect/train
```

Figure 8. Metric report of the model after running on the validation set.

4.3.2 Evaluation of the testing process

Apart from the score evaluated while training, we also evaluate on the completely separated test set. The evaluation on test set shows us the performance of our trained model on unseen data.

Ultralytics also provide a method to valuation after training by calling `val()` function and provide the configuration file the same as training process and also the trained model, except this time the train and validation configuration will be our test dataset.

```python
from ultralytics import YOLO

# Load a model
model = YOLO("/kaggle/input/yolov8-model-after-animal-training/last.pt")

# Customize validation settings
validation_results = model.val(data="/kaggle/input/yolov8-config-for-test/YOLOv8_for_Test.yaml", device="0")
```

Figure 9. Calling validation method with the trained model and test dataset.

```
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 231/231 [00:27<00:00,  8.31it/s]
                   all       3690       5188      0.733      0.685      0.715      0.565
                  Bear         59         70      0.482      0.386       0.37      0.311
                 Camel         56         91      0.611      0.681      0.603      0.489
                   Cat        400        438      0.861      0.785      0.866      0.722
                Cattle        388        765      0.648       0.41      0.435       0.32
               Chicken        176        240      0.848      0.767      0.826       0.72
                  Crab        157        195      0.874      0.759      0.787      0.603
             Crocodile         96        108      0.889      0.816      0.869      0.642
                   Dog        400        489      0.887      0.624      0.827      0.687
                  Duck        199        382      0.879      0.652       0.78      0.601
              Elephant        137        234      0.606      0.704      0.673      0.546
                   Fox        103        110      0.732      0.873      0.835      0.742
                  Frog        111        131      0.809      0.702      0.824      0.607
          Hippopotamus         39         58      0.548      0.741      0.593      0.449
              Kangaroo         65         87      0.571      0.701      0.679      0.545
                  Lion        138        156      0.813      0.754      0.819      0.709
                 Mouse        210        245      0.791      0.664      0.747      0.467
                   Pig        135        212      0.688      0.655      0.704      0.607
                Rabbit        215        267      0.831      0.683      0.769      0.636
                 Sheep        179        396      0.453      0.475      0.378      0.282
                 Snail        129        139      0.721      0.755       0.77      0.603
                 Snake        234        243        0.9      0.848      0.905      0.659
                 Zebra         64        132      0.672      0.644      0.668      0.483
Speed: 0.2ms preprocess, 2.2ms inference, 0.0ms loss, 1.1ms postprocess per image
Results saved to runs/detect/val
```

Figure 10. Valuation metric of our trained model on test dataset.

It is delightful to know that our trained model can achieve the mAP-50 score of 0.715 and having classes mAP-50 score similar to when valuate validation set, considering the fact that the test dataset size is more than two times the validation size. Our trained model is capable if not saying excel in detection animal on large unseen dataset.

4.4 Application

Here we describe a practical usage of the trained model by using it in an animal detection website. In terms of technicalities, the model is loaded and used the same way as described in Section 3, with the exception of automatically downloading the trained weights.

We summarize the features of the site as follows:

- Perform wildlife animal detection on input image.
- List facts and related links on detected animals on the Wiki tab.
- Order the facts based on confidence of each detection or based on the name of animals.

To access the wildlife detection model, simply scroll down from the initial Ultralytics' homepage, or click on the *Scroll down* button on the bottom of the screen. To see the facts/links related to the animals, click on the *Wiki* button. Screenshots of the webpage are presented in Figure 11, 12, 13, and 14.

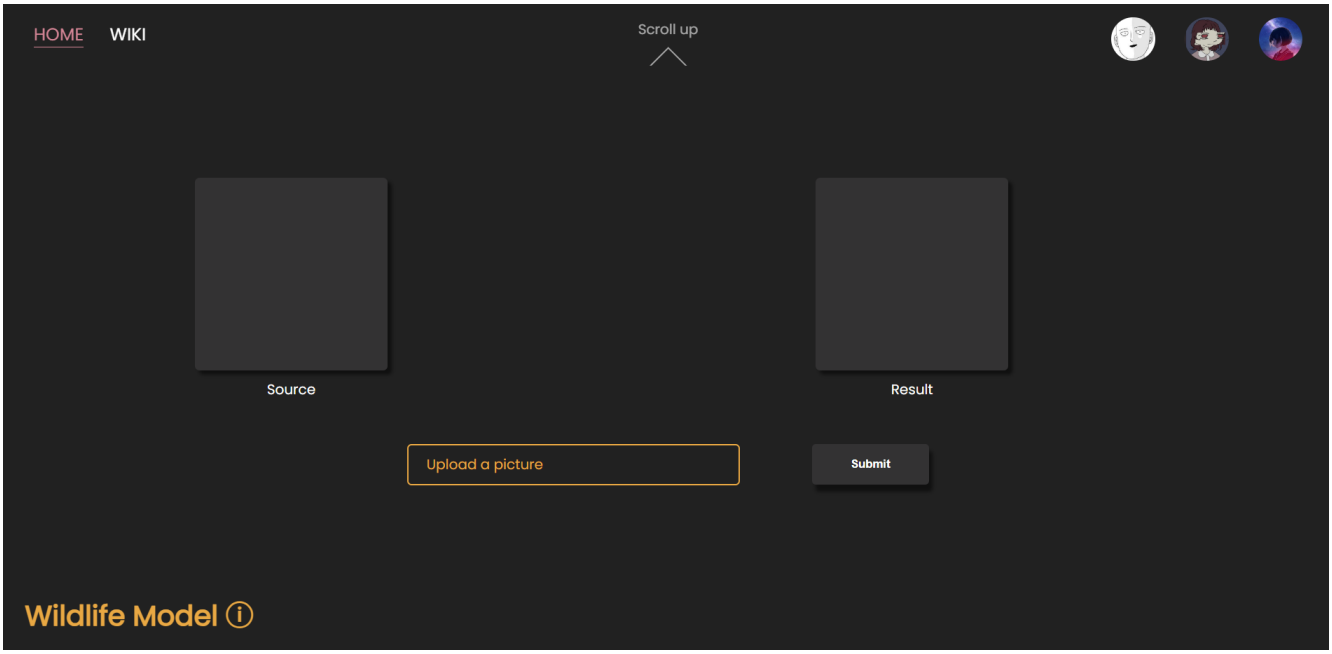

Figure 11. Interface of wildlife detection model's homepage.

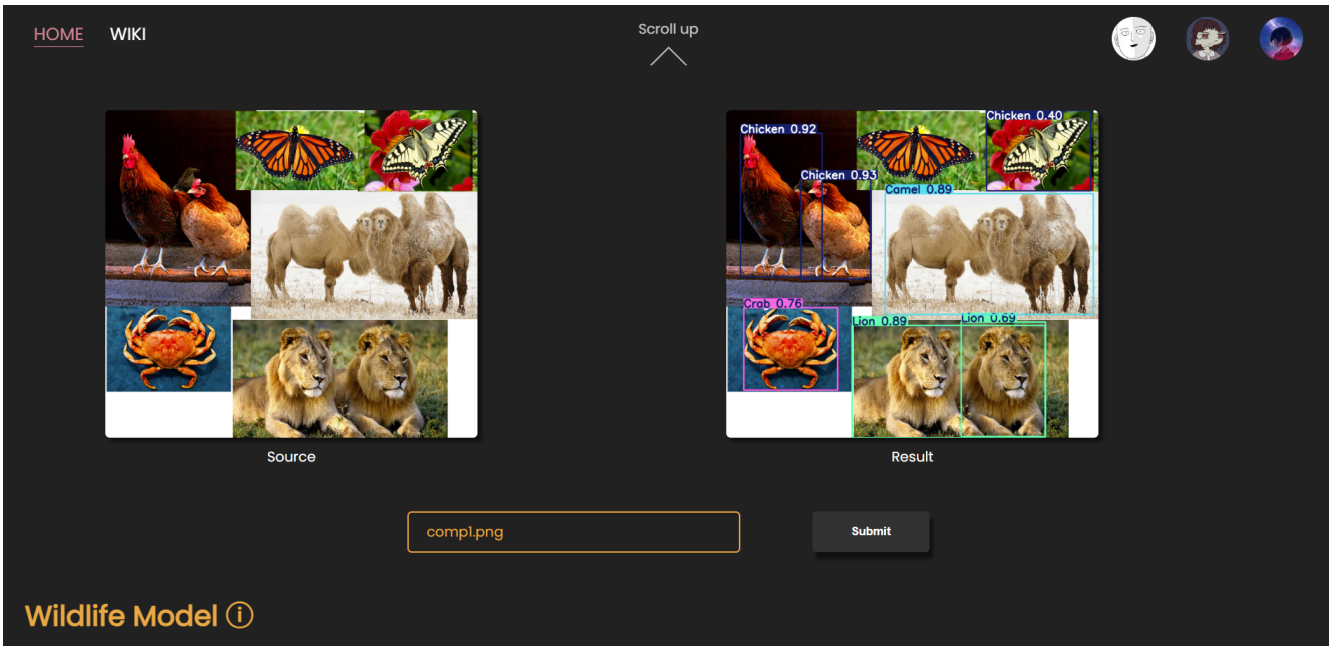

Figure 12. Interface of wildlife detection model after being prompted with an input image.

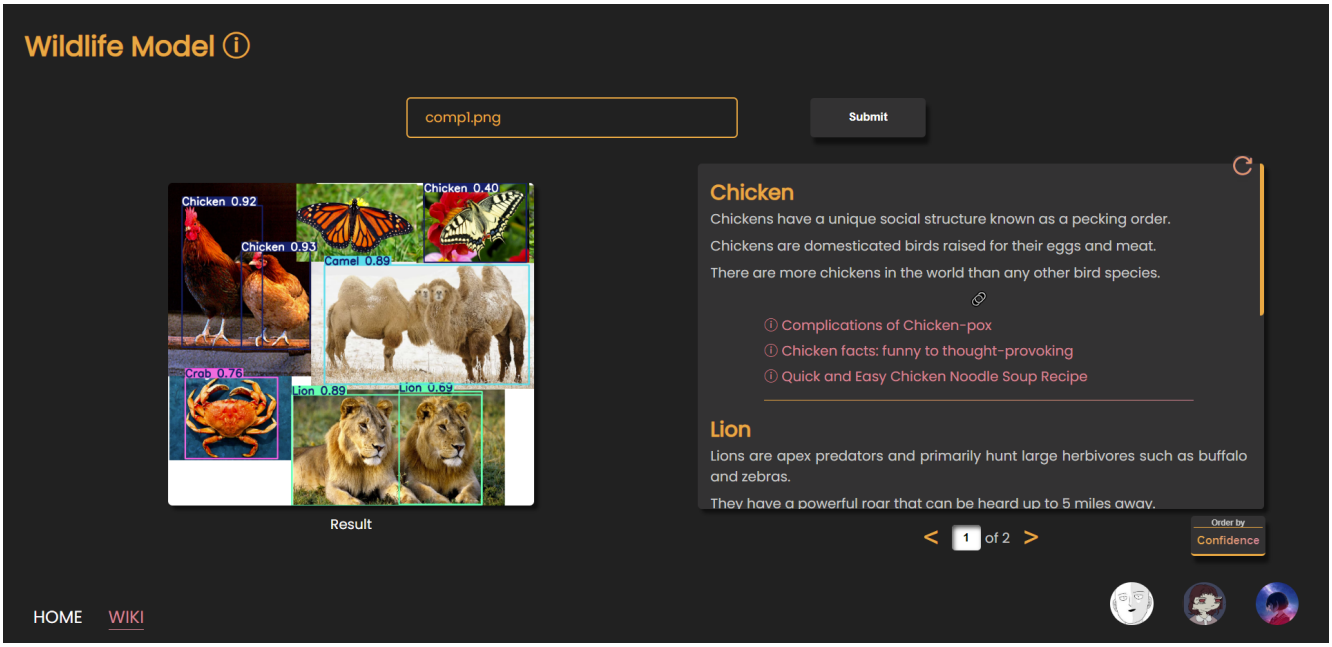

Figure 13. Interface of Wiki page. The fact list is being ordered by confidence of each detection.

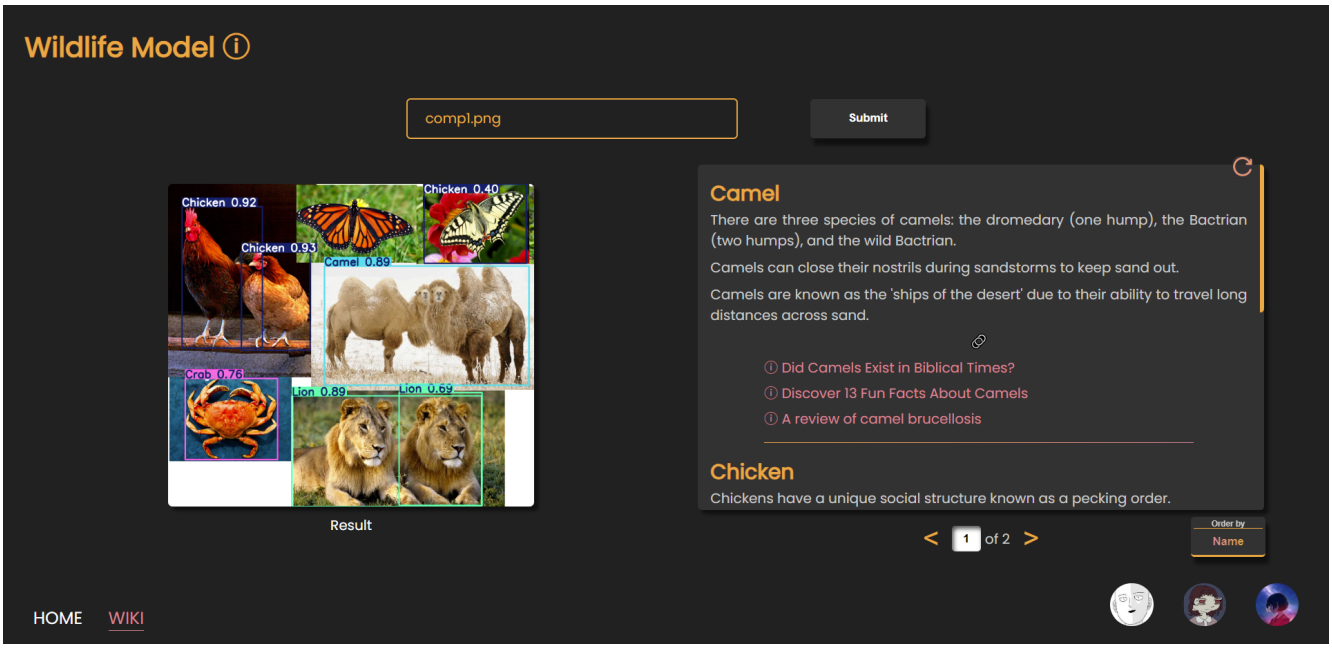

Figure 14. Same as Figure 13, but being ordered by alphabetical order of animal names.

ACKNOWLEDGMENT

The authors would like to thank the lecturers of the *Statistical Machine Learning* course at the Ho Chi Minh University of Science (Le Long Quoc, and Ngo Minh Nhut) for their instruction in theory and lab session as well as support during the project duration.

The structure and style of this report is based on the Monthly Notices of the Royal Astronomical Society (MNRAS) template and instructions for authors (https://academic.oup.com/mnras/). All data and codes used in this project are for educational and research purposes.

REFERENCES

Bochkovskiy A., Wang C.-Y., Liao H.-Y. M., 2020, YOLOv4: Optimal Speed and Accuracy of Object Detection (arXiv:2004.10934)

Tianxiaomo 2021, pytorch-YOLOv4, https://github.com/Tianxiaomo/pytorch-YOLOv4