Open Peer Review on Qeios

Tweeting AI: A Machine Learning Approach for Bird Species Detection and Classification

Ayonabh Chakraborty[1], Pushan Kumar Dutta[1]

1 Amity University Kolkata

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.

Abstract

The rapid decline in global bird populations has generated an urgent need for efficient and accurate monitoring of avian species. In this study, we propose a novel machine learning approach, Tweeting AI, for the automated recognition of bird species based on their images. To solve this problem, we have used Convolutional Neural Network (CNN). This model is the best regarded in image prediction. In this study, we have used CNN to classify the bird species in which we use a dataset for training and predicting. This system allows us to easily identify the species of any bird that we want to know, and segregate the endangered species to preserve, care and take every possible action for their survival.

** *A study, analysis, and usage of Machine Learning for the identification of Endangered Species*

Ayonabh Chakraborty

Amity School of Engineering and Technology Kolkata Amity University, Kolkata, India

Email: ayoncchakraborty7@gmail.com

PK Dutta

Amity School of Engineering and Technology Kolkata Amity University, Kolkata, India

Email: pkdutta@kol.amity.edu

Keywords: Convolution Neural Networks, bird species detection, avian conservation, environmental monitoring, machine learning, deep learning, model-training.

I. Introduction

Bird species recognition has become a significant issue in today's date. As birds are part of nature and rapidly adapt to ecological changes, for the common man and for the people involved in this field of study it has become essential to identify birds that we encounter in our day-to-day life. Birds play a crucial role as an element of nature. They help in plant pollination; they eat up a large percentage of insects that come into the way of farming. This way they help in controlling the pests that hamper agricultural production. These are a few of the cost benefits that can be helpful for humans if we protect the birds. The aim of our model is to discover more birds and to identify those that are endangered. The advent of machine learning techniques has opened up new possibilities for detecting bird species more efficiently than ever before. Specifically, by using convolutional neural networks (CNNs), which are capable of identifying patterns within large data sets leading towards improved outcomes over time. This innovative approach that leverages AI-powered technology could significantly improve our understanding of avian ecology while enabling researchers to collect data on birds' behaviors more accurately thereby enhancing overall productivity levels benefiting both individuals/teams involved and the organization as a whole, ultimately resulting in better outcomes achieved faster than expected otherwise. Objectives and Contributions:

The main objectives as proposed are as follows:

1. To develop a machine learning-based approach for detecting bird species based on their vocalizations captured via audio recordings coupled with image analysis capabilities provided by CNN models.
2. To evaluate the effectiveness of this approach in accurately classifying various bird species from audiovisual data while ensuring ethical considerations around responsible deployment practices being followed leading towards long-term sustainability goals at the workplace
3. To provide real-time insights into how avian populations are changing over time by monitoring their behaviors using advanced AI-powered tools thereby enhancing overall productivity levels benefiting both individuals/teams involved and the organization as a whole, ultimately resulting in better outcomes achieved faster than expected otherwise.

Our proposed research will make several contributions to the field of ecology and conservation biology, including:

1. Developing a new method for collecting comprehensive data on birds' behaviors that is more efficient than traditional survey methods.
2. Enabling researchers to monitor changes in bird populations over time more effectively through real-time insights generated via integrated AI-powered technologies from diverse sources such as acoustic signals, images/videos, etc.
3. Providing an innovative tool that can be used to assess the impact of environmental factors like climate change or habitat loss on avian biodiversity thereby providing advance warning signs enabling proactive measures taken addressing issues raised by stakeholders involved.

Overall, our proposed study has significant potential in helping us better understand avian ecology while driving innovation forward across multiple domains like environment, health, etc., leading towards better ROI over time. In this paper, we provide a comprehensive study of a machine-learning approach for bird species detection and classification using a CNN algorithm on images. We begin with a brief introduction

to the problem and its significance in the context of bird conservation. We then review the related work in the field, including traditional methods and machine learning-based techniques, with a focus on CNNs for image recognition.

Next, we outline the methodology employed in our study, detailing the data collection and pre-processing steps, the architecture of the CNN, and the model training and validation procedures. We also discuss the performance evaluation metrics used, including accuracy, precision, recall, and F1-score, as well as the confusion matrix.

II. Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is a type of artificial neural network inspired by the visual cortex of the human brain. It consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers. CNNs are specifically designed to efficiently analyze visual data by extracting features hierarchically. Convolutional layers, the core component of CNNs, perform convolution operations on the input image with learnable filters, allowing the network to learn meaningful features and patterns. Pooling layers downsample the spatial dimensions of the feature maps, reducing computational complexity and improving robustness. Fully connected layers connect the extracted features to the final output layer for classification or regression. CNNs incorporate various essential components. Activation functions, such as ReLU (Rectified Linear Unit), introduce non-linearity into the network, enabling better modeling of complex relationships. Batch normalization normalizes the outputs of previous layers, improving the stability and speed of training. Dropout regularizes the network by randomly dropping out units during training, reducing over-fitting. Training a CNN involves two key steps: forward propagation and back propagation. In forward propagation, the network processes the input data through its layers, extracting and transforming features. The final output is compared with the ground truth labels to compute the loss. In back propagation, the gradients of the loss with respect to the network parameters are computed and used to update the weights and biases through optimization algorithms like stochastic gradient descent (SGD). CNNs have had a significant impact on various fields. In computer vision, they excel in tasks like object recognition, image classification, object detection, and segmentation. They enable advancements in autonomous vehicles, facial recognition systems, medical image analysis, and more. CNNs have also found applications in natural language processing, speech recognition, and recommendation systems, demonstrating their versatility beyond image analysis.

III. Literature Review

Bird species detection plays a crucial role in biodiversity monitoring, ecological research, and conservation efforts. With advancements in machine learning, automated methods for bird species detection have gained significant attention. This literature review aims to provide an overview of

the current state of research in bird species detection using machine learning techniques, including the datasets, methodologies, and performance metrics employed in various studies. Several datasets have been utilized for bird species detection, enabling researchers to train and evaluate machine learning models effectively. The most commonly used dataset is the Cornell Lab of Ornithology’s eBird dataset, which provides extensive bird sighting records and

associated metadata. Other datasets, such as the XenoCanto dataset, BirdCLEF dataset, and BirdNET dataset, have also been employed, each offering unique bird species recordings and annotations. A wide range of machine learning algorithms and methodologies have been explored for bird species detection. Among the popular approaches are convolutional neural networks (CNNs), which have shown remarkable performance in image-based classification tasks. CNN-based architectures, such as ResNet, VGG, and Inception, have been adapted and fine-tuned for bird species detection. Other techniques, such as support vector machines (SVMs), random forests, and hidden Markov models (HMMs), have also been utilized for audio-based classification and bird sound analysis. Feature extraction and representation are critical in capturing relevant information from bird species data. For image-based approaches, features like local binary patterns (LBPs), color histograms, and spectrogram representations have been employed to capture the visual characteristics of birds. In audio-based approaches, Melfrequency cepstral coefficients (MFCCs), mel-spectrograms, and wavelet transform features have been used to extract relevant acoustic information. Hybrid approaches that combine visual and acoustic features have also been explored to improve the accuracy of bird species detection. Evaluation of bird species detection models involves the use of various performance metrics. Accuracy, precision, recall, and F1-score are commonly used to measure the overall performance of the classification models. In addition, confusion matrices and receiver operating characteristic (ROC) curves provide insights into the model's ability to distinguish between different bird species. Some studies have also focused on evaluating the transferability of models across different geographical regions and seasons.

IV. Methodology

To deal with numerous datasets involving different kinds of bird species, we have made use of the deep neural network known as Convolutional Neural Network to get accurate outputs. A Convolutional Neural Network is a type of neural network that specializes in processing data that has topology in the format of grids. As such, a simple example of a grid-like topology would be that of an image. To define an image in layman's terms, an image is basically the binary representation of visual data that we can perceive with our eyes or can be captured using devices like a camera, a video recorder, etc. An image consists of pixels arranged in a grid-type format that contains pixel values to denote how bright and what color each pixel should be. Our minds, or to be more scientifically accurate,

the human brain can understand, interpret, and classify a large amount of information when we lay our eyes on any object or image. From grasping simple things like shape or color to forming distinctions between similar-looking objects based on unique properties, the human brain acts like a machine, where each neuron has its own receptive field that it works on and then passes on the data or information gathered by it to the other neurons that are connected to it, leading to the coverage of the entire visual field. Convolutional Neural Network works in a similar manner. In fact, it is so similar to the human brain that just as a biological neuron responds to any signals in the restrictive region of the visual field, each neuron in CNN can process the data limited to its own receptive field. Now based on the requirement of training the model, the CNN layers may consist of many different things like shape, size, color, lines, curves, objects, faces, sceneries, etc. In a certain sense, with the proper usage of Convolutional Neural Networks, we are basically providing computers the

ability to perceive using sight. The Convolutional Neural Network model is built to work on data in the form of images. The data can be one-dimensional, two-dimensional, or sometimes while dealing with more features, it can be three-dimensional too. There are various applications of Convolutional Neural Networks including image classification, object detection, medical image analysis, natural language processing, live detection of objects in motion, and filtering out necessary information from junk data. Finally, in the various forms of computer vision projects. The Convolutional Network works in two parts, i.e., Feature Extraction and Classification, where feature extraction is the feature mapping of the dataset based on the application of the activation function on the dataset, which is followed by the pooling of the classes (as per the number of classes present in the dataset), which leads to the classification phase consisting of the final pooling layer before leading it to the output. The convolutional neural network consists of four layers: -

A. Convolutional Layer

This is the major layer that is put to use in convolutional neural networks. Convolution is the basic function of applying a filter on input that results in activation. When we repeatedly apply the same filter to an input, it results in a series of activations known as a feature map. This helps us in finding the locations and the strengths of an identified feature in input, such as an image. The main aim of convolutional neural networks is to automatically learn about a large number of filters based on a training dataset, where the learning, or as we term it “training” can finally be used to identify any type of image that is being provided as an input to test the functioning of the machine learning model. The filters can be handcrafted, namely line detectors or as we see in this case, the labels that have been used to help in identifying any image that is used to test the model in the future. The learnable filters are also sometimes referred to as kernels, and these kernels have a width and a length and are mostly squares in nature. For the inputs that we take in CNN, the depth is essentially the number of channels in the image, for example, while working

with RGB images, the depth is 3 as it’s 1 for each channel. For any existing volumes present deeper in the network, the resulting depth will be the number of filters applied in the previous layer.

B. Activision Layer

After each filter application layer in CNN, we apply an activation function such as ReLU, ELU, etc. We denote the activation layers with the correct terminologies such as RELU for ReLU, thus making it clear that an activation state is being applied inside the architecture of any said network. The purpose of the activation function is to make the output as nonlinear in nature as possible. The neurons in a neural network work with the weight, the bias, and the activation function that they are in correspondence with. As per the process, we update the weight and the bias of each individual neuron in a neural network. This is done based on the error that may occur at the output. This method is known as Backpropagation, which is the essence of our neural network training. It basically updates the weights and the bias of a neuron (say neuron number b) based on the rate of the error that is found in the previous epoch of neuron (say neuron number a), or the iteration of the model training before the previously mentioned neuron(b). This is done to finely tune the weights to minimize error rates, thus making the machine-learning model more accurate.

C. Pooling Layer

It is done to reduce the volume of the input to make the process of machine learning more tuned. The primary function of this layer is to minimize the spatial size of the input, thus reducing the number of parameters and computations present inside the network. Pooling also helps us to control overfitting, i.e., when the accuracy of the dataset used to train the model (training dataset) is greater than the accuracy of the dataset used for testing the model (testing dataset) which results in lower error rates in the training dataset and higher error outputs in the testing dataset essentially showing the signs of a flawed model. Pooling layers operate on all the depth slices of an input using the max function or the average function. Max pooling is the type of pooling that is typically done in the middle of the architecture of the convolutional neural network. Although we wish to avoid fully connected layers altogether, which is being researched with the introduction of the exotic micro-architectures in this field.

D. Fully Connected Layers

These layers are connected to all the activations that have taken place in the previous epoch, which is the standard for all feed-forward neural networks. These are placed at the end of the network.

E. Batch Normalization

This layer of the convolutional neural network is done to normalize the activations of an input volume before passing it to the next layer of the Neural Network. At testing time,

the mini-batches are replaced by the average that is computed during the training process. This helps us ensure the passage of images seamlessly through our network and yet obtain very accurate results without retaining any biases from the final mini-batch passed through the neural network during the training phase. Batch Normalization has been shown to be extremely effective at reducing the number of epochs iterating during the training of the neural network. It also helps in stabilizing the training, by focusing on a larger variety of learning rates and strengths of regularization. It doesn’t help us in tuning the working of the neural network, although making the learning rates and regularization strengths less volatile, it makes the entire process pretty straightforward. This layer hasn’t been used in our project.

F. Dropout

This is the last layer of the neural network. It is basically just a form of regularization that helps prevent instances of overfitting while increasing testing accuracy, sometimes at the expense of the cost of the training process. Dropouts randomly disconnect input layers from the previous layer to the upcoming layer in the neural network.

V. Selection and Usage of Yolov4 to get Hands-on Experience of How Image Detection Works and Using the Yolov5 CNN Model to Train our Dataset

Our project began with the simple steps of understanding the basics of Neural networks, which consisted of learning to train custom data using the Yolov4 CNN model on regular images and obtaining results. This gave us a direction to proceed with the You-Only-Look-Once algorithm version 5 to train the datasets and thus get much more accurate results on unique data. Yolo is an algorithm that detects and recognizes patterns from various objects in a picture in real-standard time. Object-based image detection is done as a regression problem that provides the probabilities of the classes of the detected images. By employing convolutional neural networks to detect objects in real time, the algorithm works based on a single forward propagation approach through a neural network to detect objects based on the dataset of the images provided to it. There are currently many versions of the YOLO algorithm like tiny YOLO, YOLOv3, YOLOv4, YOLOv5, YOLOv6, etc. The most recent version of YOLO is YOLONAS, an extremely efficient object-detecting model.

A. Importance

Out of many algorithms present that can work on our datasets, we singled out the YOLO algorithm primarily because of its speed. It improves the speed of object detection after every epoch as it predicts objects in real time. Accuracy is one of the main factors we chose for this model: being an extremely efficient prediction technique that minimizes background errors. A trained model is of no use if it can’t perform its basic functions with the optimal desired accuracy

and thus can result in a huge waste of time, resources, and energy. Our main goal was to build a highly accurate model that runs seamlessly with the correct format of data being provided to it, and we have achieved so using the You-OnlyLook-Once algorithm. Adding to the feathers in its hat, YOLO is a state-of-the-art machine learning model that is extremely efficient in learning each individual representation of an object and applying them in real-time object detection.

B. The Working Process of the YOLO Algorithm:

The YOLO algorithm works using three techniques, which are forming Residual Blocks which are then put through Bounding Box Regression and finally, the Intersection over Union technique is applied. 1. Residual Blocks: Firstly, the images are divided into various grids of dimension (NxN). In the below image, there are various grid cells of equal dimensions. Every grid is then used to detect any objects that might be present inside of them. Say if a particular object is detected in a single grid cell of an entire image during the training phase, then that grid will be responsible for detecting the object during the testing phase. 2. Bounding Box Regression: A bounding box is an outline that is applied to an object in an image to basically mark it as unique for future detection purposes. Every bounding box consists of Width, Height, Class, and the bounding box center. YOLO uses a single bounding box regression technique to predict the height, width, center, and class of the various objects. In the image below, it represents the probability of any object that may appear in the bounding box. 3. Intersection over Union: Finally, we make use of the IOU phenomenon in object detection that describes how the boxes overlap. YOLO basically makes use of the Intersection over Union phenomenon to provide an output box that surrounds the object perfectly. Every grid cell is used to predict the bounding boxes and helps in calculating the confidence scores. If the value of IOU is 1, it means that the predicted bounding box is the same as the real box, thus meaning that the algorithm has essentially eliminated the boxes that are not like the real box. In the image

given below, there are two bounding boxes present: the green bounding box is used as the predicted box and the blue bounding box represents the real box. The YOLO algorithm ensures that the two bounding boxes are always equal in all manners. Thus, YOLO divides the images into grid cells. Then each grid cell forecasts bounding boxes and provides their confidence scores. Finally, the cells predict the class probabilities to find out the class of each object efficiently.

VI. The Working Process: Experimentation, Data Collection, Code, and Output

We began our experiment by deciding to first try, experiment and understand how YOLOv4 works. So, we initially used the darknet repository. Google Colab was our choice of software, which turned out quite a nice platform to easily access our required files to kickstart our project. So the first thing we did

was to go on Google Colab, open a new notebook, and load the darknet repository. However, it is not recommended at all to load the Colab book without connecting the GPU units for the usage of our choice of Machine Learning Model.

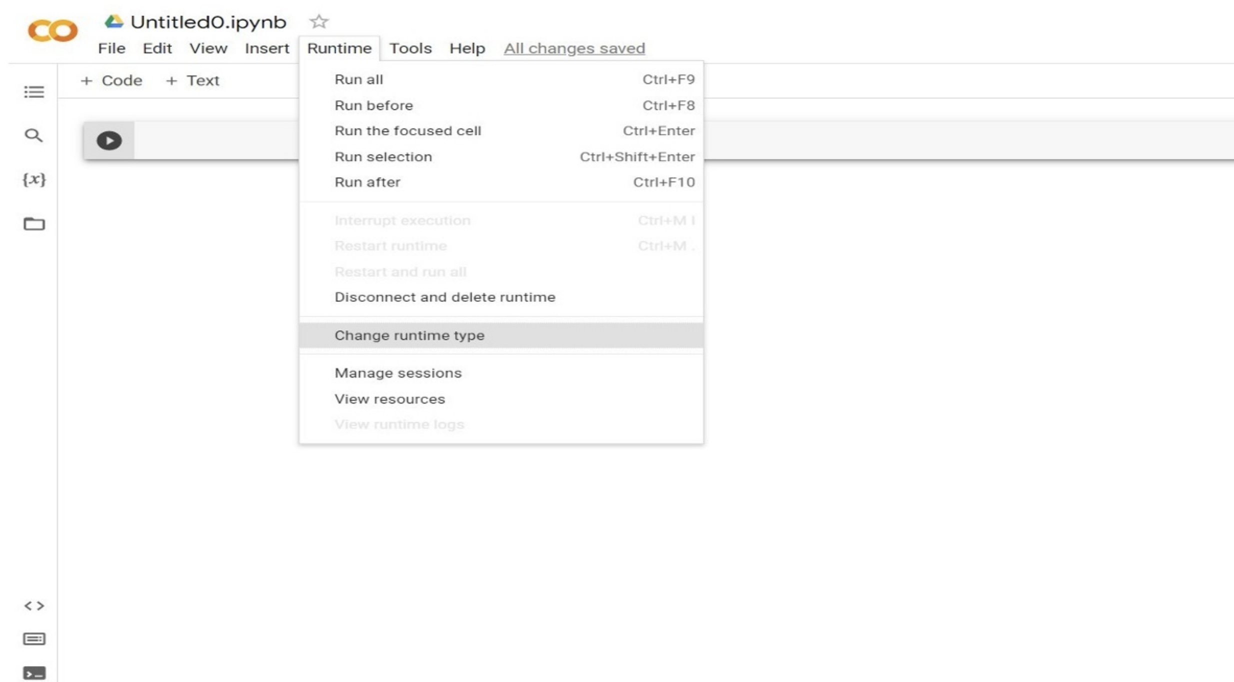

Fig. 1. Setting up Google Colaboratory

Then we select the type of processor for the machine learning model training. For better efficiency, we switch to Google's GPU for faster epochs or batch iterations.

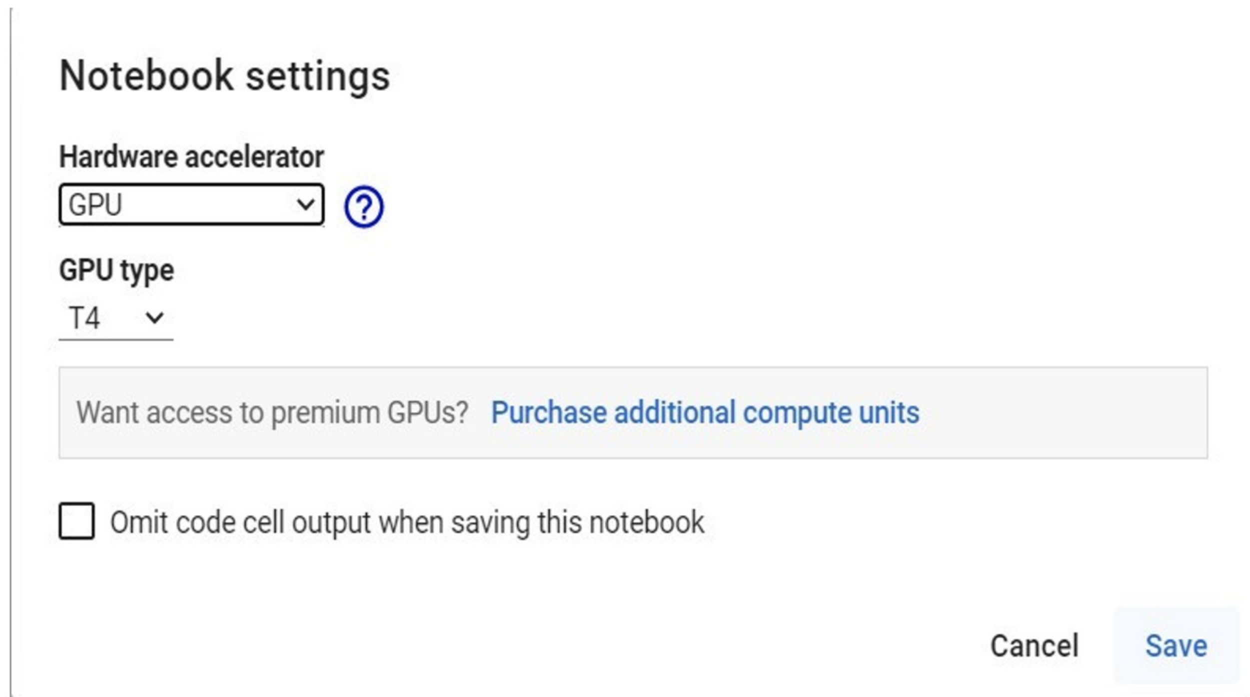

Fig. 2. Changing the Run-time settings

Then we click on the connect button to set up the notebook to begin our operations.

Fig. 3. Confirming the Internet Connectivity Status

A. Experimenting with the Darknet Repositories

We clone the darknet repositories using the following commands.

```
[2] !git clone https://github.com/AlexeyAB/darknet.git

Cloning into 'darknet'...
remote: Enumerating objects: 15521, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 15521 (delta 0), reused 5 (delta 0), pack-reused 15514
Receiving objects: 100% (15521/15521), 14.19 MiB | 20.30 MiB/s, done.
Resolving deltas: 100% (10412/10412), done.
```

Fig. 4. Cloning the Darknet Repositories

We check the NVIDIA systems of our machine and see if everything is at par for the upcoming activities.

```
[3] !nvidia-smi

Sun May 21 16:45:39 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI              Driver Version:              CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   53C    P8    12W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Fig. 5. Checking the Nvidia Systems

We run a simple command to check the contents of the darknet file that we loaded in our laboratory.

```
[5] !ls /content/darknet

3rdparty               darknet_video.py        net_cam_v4.sh
build                  data                    README.md
build.ps1              image_yolov3.sh         results
cfg                    image_yolov4.sh         scripts
cmake                  include                 src
CMakeLists.txt         json_mjpeg_streams.sh   vcpkg.json
DarknetConfig.cmake.in LICENSE                 vcpkg.json.opencv23
darknet_images.py      Makefile                video_yolov3.sh
darknet.py             net_cam_v3.sh           video_yolov4.sh
```

Fig. 6. Checking the Darknet Repositories

Now we make use of the make automation tool to ease out our task. The make tool requires a file, termed as Makefile to fulfill a set of requirements by running the needed tasks.

```
[7] !cat Makefile
ifeq ($(ZED_CAMERA), 1)
CFLAGS+= -DZED_STEREO -I/usr/local/zed/include
ifeq ($(ZED_CAMERA_v2_8), 1)
LDFLAGS+= -L/usr/local/zed/lib -lsl_core -lsl_input -lsl_zed
#-lstdc++ -D_GLIBCXX_USE_CXX11_ABI=0
else
LDFLAGS+= -L/usr/local/zed/lib -lsl_zed
#-lstdc++ -D_GLIBCXX_USE_CXX11_ABI=0
endif
endif

OBJ=image_opencv.o http_stream.o gemm.o utils.o dark_cuda.o convolutional_layer.o list.o image.o activations.o im2col.o col2im.o blas.o crop_la
ifeq ($(GPU), 1)
LDFLAGS+= -lstdc++
OBJ+=convolutional_kernels.o activation_kernels.o im2col_kernels.o col2im_kernels.o blas_kernels.o crop_layer_kernels.o dropout_layer_kernels.(
endif

OBJS = $(addprefix $(OBJDIR), $(OBJ))
DEPS = $(wildcard src/*.h) Makefile include/darknet.h

all: $(OBJDIR) backup results setchmod $(EXEC) $(LIBNAMESO) $(APPNAMESO)

ifeq ($(LIBSO), 1)
CFLAGS+= -fPIC

$(LIBNAMESO): $(OBJDIR) $(OBJS) include/yolo_v2_class.hpp src/yolo_v2_class.cpp
	$(CPP) -shared -std=c++11 -fvisibility=hidden -DLIB_EXPORTS $(COMMON) $(CFLAGS) $(OBJS) src/yolo_v2_class.cpp -o $@ $(LDFLAGS)
```

Fig. 7. Making use of the Make Automation Tool

In the above make file, there exist four important parameters whose values need to be updated manually. We do so for the build to use the CUDA (Compute Unified Device Architecture) on our GPUs. CUDA has several benefits as it provides

us with parallel hardware to run code, with the drivers most suitable for the process of carrying out the executions. It is essentially a software development kit that has libraries, debuggers, compiling tools, etc. These are used to invoke CPUbased programming into GPU-based programming. The main point of CUDA is to be able to write code and be able to run on compatible multiple parallel architectures, which includes non-GPU hardware too. This massively parallel hardware can be used to run a large number of operations much faster as compared to the CPU, yielding improvements in performance by almost a half. Along with that, the values of cuDNN, OpenCV, and tensor cores are also changed to 1 (from 0). Tensor cores are only applicable when Google Colab allocates GPUs like Titan V. The following commands are used to update the values:

```
[ ] !sed -i 's/GPU=0/GPU=1/' Makefile
    !sed -i 's/CUDNN=0/CUDNN=1/' Makefile
    !sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
    !sed -i 's/OPENCV=0/OPENCV=1/' Makefile
```

Fig. 8. Updating the values manually

Once the necessary updating for the GPU compilation in the Makefile has been completed, we can proceed on with our builds. It can be done so by simply invoking the “make” command.

```
[8] !make

gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/rnn_vid.c -(
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/crnn_layer.(
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/demo.c -o ol
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/tag.c -o obj
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/cifar.c -o (
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/go.c -o obj/
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/batchnorm_la
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/art.c -o obj
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/region_layer
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/reorg_layer.
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/reorg_old_la
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/super.c -o (
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/voxel.c -o (
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/tree.c -o ol
gcc -Iinclude/ -I3rdparty/stb/include -Wall -Wfatal-errors -Wno-unused-result -Wno-unknown-pragmas -fPIC -rdynamic -Ofast -c ./src/yolo_layer.(
./src/yolo_layer.c: In function 'process_batch':
./src/yolo_layer.c:426:25: warning: variable 'best_match_t' set but not used [-Wunused-but-set-variable]
  426 |                     int best_match_t = 0;
      |                         ^~~~~~~~~~~~
./src/yolo_layer.c: In function 'forward_yolo_layer':
./src/yolo_layer.c:707:11: warning: unused variable 'avg_anyobj' [-Wunused-variable]
  707 |     float avg_anyobj = 0;
      |           ^~~~~~~~~~
./src/yolo_layer.c:706:11: warning: unused variable 'avg_obj' [-Wunused-variable]
  706 |     float avg_obj = 0;
      |           ^~~~~~~
```

Fig. 9. Invoking the Make Command

Now to test the workings of the YOLOv4 model, we downloaded some pre-trained weights. This is done after the darknet framework has been completely in tune with the machine learning algorithm.

```
[9] !wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.weights

--2023-05-21 16:46:43--  https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.weights
Resolving github.com (github.com)... 140.82.121.4
Connecting to github.com (github.com)|140.82.121.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-asset-2e65be/75388965/ba4b6380-889c-11ea-9751-f994f5961796?X-Amz-Algori
--2023-05-21 16:46:43--  https://objects.githubusercontent.com/github-production-release-asset-2e65be/75388965/ba4b6380-889c-11ea-9751-f994f59617
Resolving objects.githubusercontent.com (objects.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 257717640 (246M) [application/octet-stream]
Saving to: 'yolov4.weights'

yolov4.weights      100%[==================>] 245.78M  63.5MB/s    in 3.9s

2023-05-21 16:46:47 (63.1 MB/s) - 'yolov4.weights' saved [257717640/257717640]
```

Fig. 10. Downloading the pre-trained weights

To display the images being taken as input and the output that is being produced, we have written a simple function.

```
[11] def showimage(path):
       %matplotlib inline
       import matplotlib.pyplot as plt
       import matplotlib.image as mpimg
       img = mpimg.imread(path)
       plt.axis("off")
       dpi = plt.rcParams['figure.dpi']
       height, width, depth = img.shape
       figsize = width / float(dpi), height / float(dpi)
       figsize = width / float(dpi), height / float(dpi)
       plt.figure(figsize=figsize)
       plt.imshow(img)
```

Fig. 11. Display Function

We have made use of our custom input in the machine learning process to check the accuracy of the model, and it was satisfactory.

```
[14] showimage('/content/city1.jpg')
```

Fig. 12. Command to display the custom image

Fig. 13. Custom Image

```
[15] !./darknet detector test ./cfg/coco.data ./cfg/yolov4.cfg yolov4.weights /content/city1.jpg -dont_show
[yolo] params: iou loss: ciou (4), iou_norm: 0.07, obj_norm: 1.00, cls_norm: 1.00, delta_norm: 1.00, scale_x_y: 1.20
nms_kind: greedynms (1), beta = 0.600000
 140 route  136                                      ->   76 x  76 x 128
 141 conv    256       3 x 3/ 2     76 x  76 x 128 ->   38 x  38 x 256 0.852 BF
 142 route  141 126                                ->   38 x  38 x 512
 143 conv    256       1 x 1/ 1     38 x  38 x 512 ->   38 x  38 x 256 0.379 BF
 144 conv    512       3 x 3/ 1     38 x  38 x 256 ->   38 x  38 x 512 3.407 BF
 145 conv    256       1 x 1/ 1     38 x  38 x 512 ->   38 x  38 x 256 0.379 BF
 146 conv    512       3 x 3/ 1     38 x  38 x 256 ->   38 x  38 x 512 3.407 BF
 147 conv    256       1 x 1/ 1     38 x  38 x 512 ->   38 x  38 x 256 0.379 BF
 148 conv    512       3 x 3/ 1     38 x  38 x 256 ->   38 x  38 x 512 3.407 BF
 149 conv    255       1 x 1/ 1     38 x  38 x 512 ->   38 x  38 x 255 0.377 BF
 150 yolo
[yolo] params: iou loss: ciou (4), iou_norm: 0.07, obj_norm: 1.00, cls_norm: 1.00, delta_norm: 1.00, scale_x_y: 1.10
nms_kind: greedynms (1), beta = 0.600000
 151 route  147                                      ->   38 x  38 x 256
 152 conv    512       3 x 3/ 2     38 x  38 x 256 ->   19 x  19 x 512 0.852 BF
 153 route  152 116                                ->   19 x  19 x1024
 154 conv    512       1 x 1/ 1     19 x  19 x1024 ->   19 x  19 x 512 0.379 BF
 155 conv   1024       3 x 3/ 1     19 x  19 x 512 ->   19 x  19 x1024 3.407 BF
 156 conv    512       1 x 1/ 1     19 x  19 x1024 ->   19 x  19 x 512 0.379 BF
 157 conv   1024       3 x 3/ 1     19 x  19 x 512 ->   19 x  19 x1024 3.407 BF
 158 conv    512       1 x 1/ 1     19 x  19 x1024 ->   19 x  19 x 512 0.379 BF
 159 conv   1024       3 x 3/ 1     19 x  19 x 512 ->   19 x  19 x1024 3.407 BF
 160 conv    255       1 x 1/ 1     19 x  19 x1024 ->   19 x  19 x 255 0.189 BF
 161 yolo
[yolo] params: iou loss: ciou (4), iou_norm: 0.07, obj_norm: 1.00, cls_norm: 1.00, delta_norm: 1.00, scale_x_y: 1.05
nms_kind: greedynms (1), beta = 0.600000
Total BFLOPS 128.459
avg_outputs = 1068395
Loading weights from yolov4.weights...
```

Fig. 14. Running the Darknet Detector on our custom image

```
[13] showimage('/content/darknet/predictions.jpg')
```

Fig. 15. Command to display the predicted output

Fig. 16. Displaying the predicted output

B. Data Collection

After we got hands-on experience with how YOLO works, we started to collect data for our machine-learning model, which consisted of birds of different species. In the following machine-learning model training and output version, we have made use of some birds by using numerous images of each one of them and labelling all the images manually. All of the labelling happened manually, so it indeed turned out to be a very tedious task. At first, we downloaded the labelImg file from the terminal and then we proceeded to create a virtual environment in Anaconda Command Shell. We proceed to access the folder in which the labelImg resources have been downloaded and by running the necessary commands, we activate the labelImg image labelling software.

```
(base) C:\Users\ayonc>conda env list
# conda environments:
#
base                  *  C:\Users\ayonc\anaconda3


(base) C:\Users\ayonc>conda activate base

(base) C:\Users\ayonc>conda install pyqt=5
Retrieving notices: ...working... done
Collecting package metadata (current_repodata.json): done
Solving environment: done

# All requested packages already installed.


(base) C:\Users\ayonc>dir
 Volume in drive C is Windows
 Volume Serial Number is D87D-3DAA

 Directory of C:\Users\ayonc
```

Fig. 17. Activating the Anaconda Virtual Environment

```
(base) C:\Users\ayonc>cd LabelImg

(base) C:\Users\ayonc\LabelImg>pyrcc5 -o libs/resources.py resources.qrc

(base) C:\Users\ayonc\LabelImg>python labelImg.py
```

Fig. 18. Accessing the labelImg files

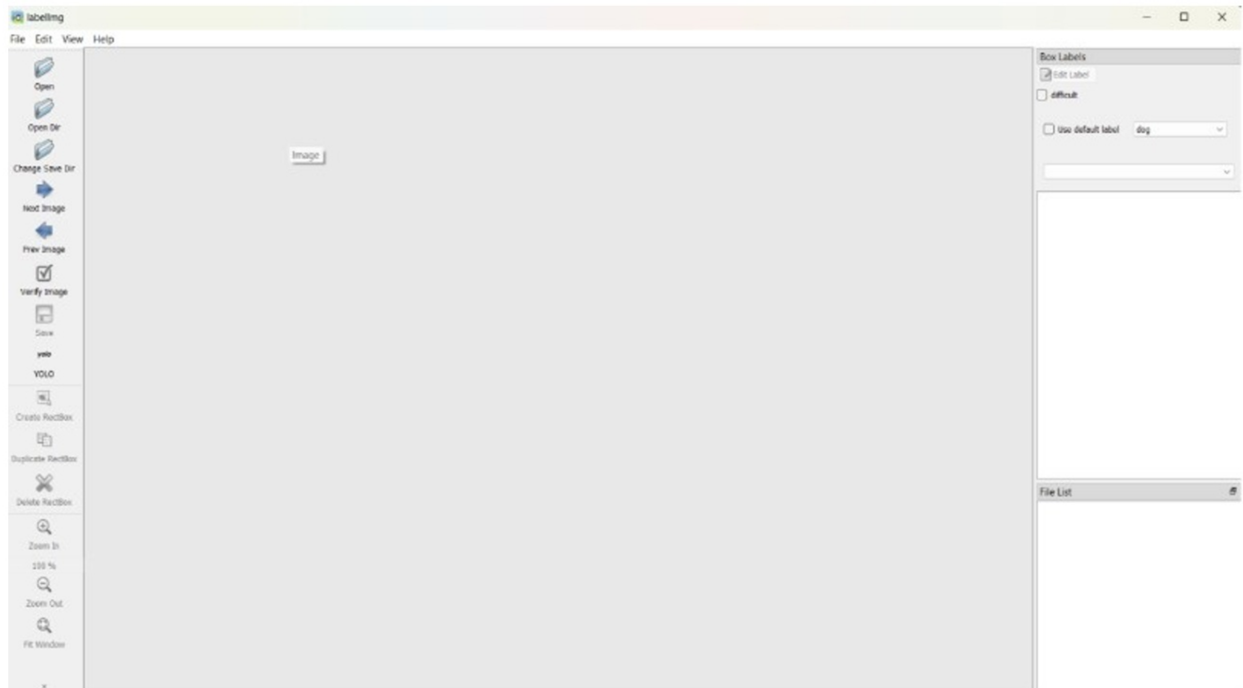

Fig. 19. Starting up the labeling software

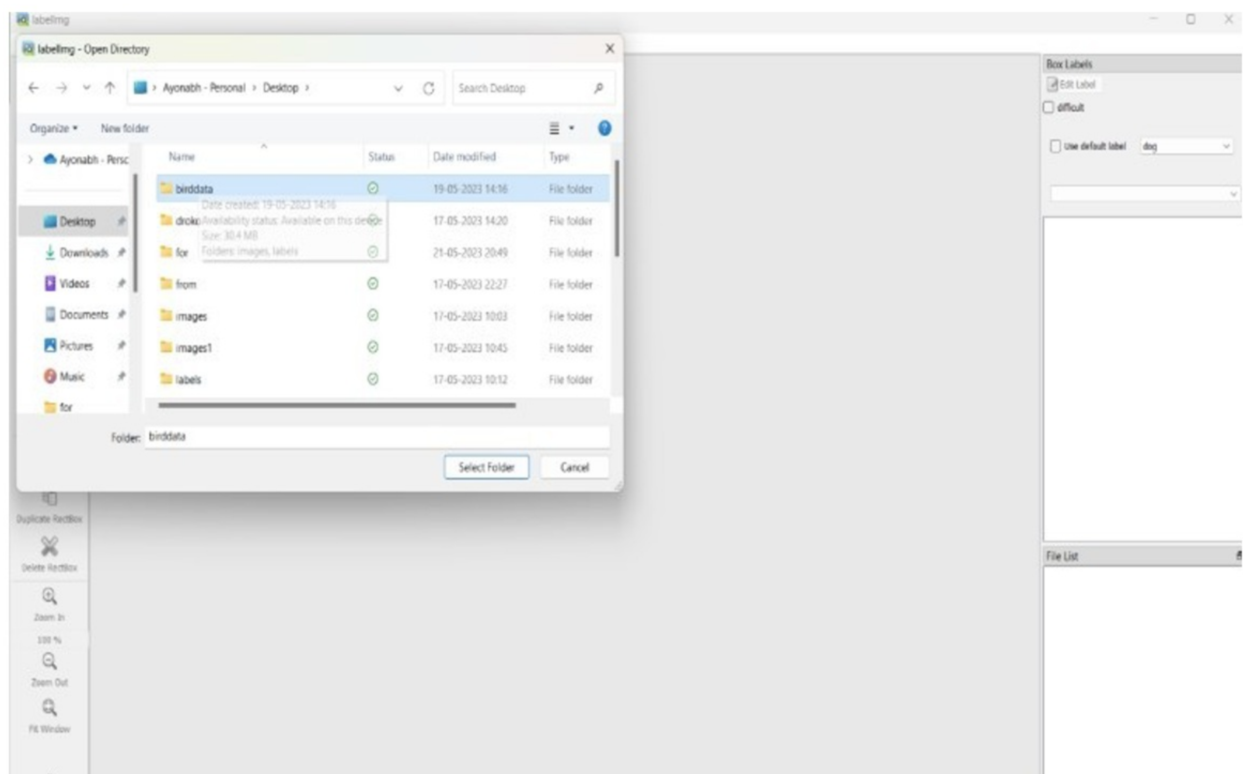

Fig. 20. Opening the folder containing the images

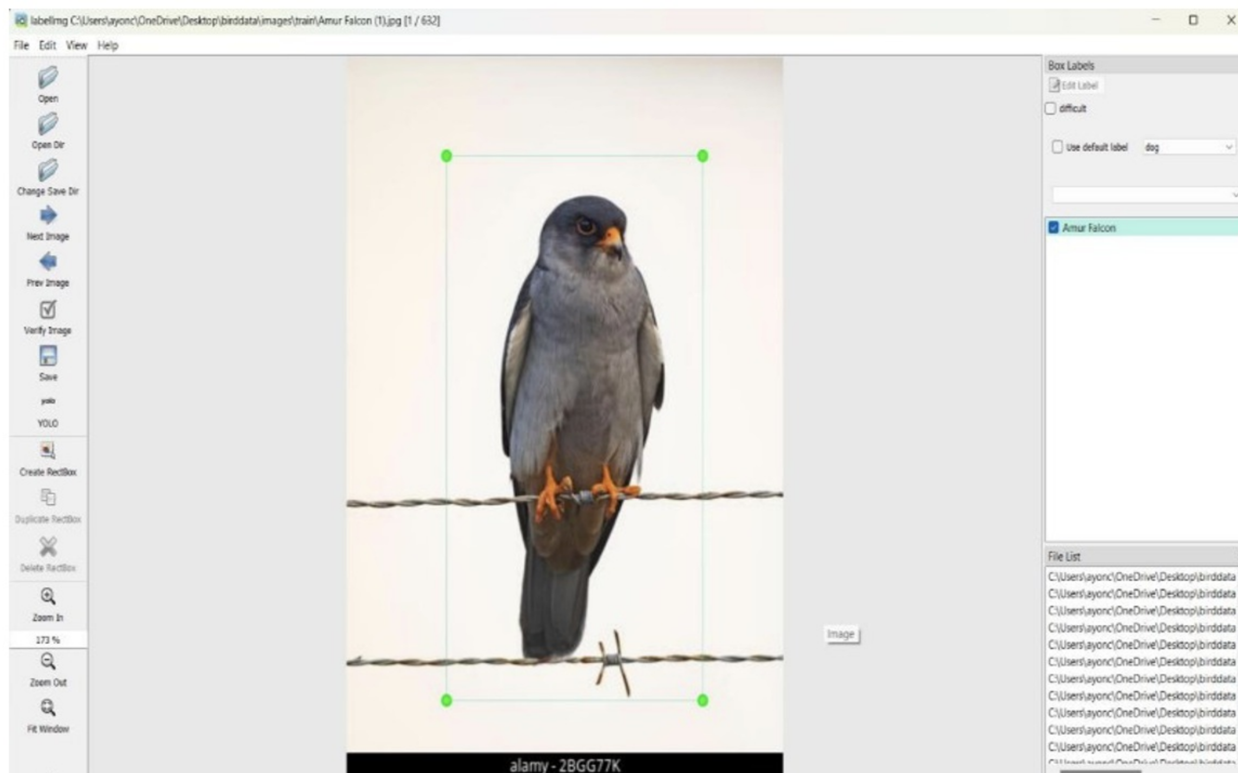

Fig. 21. Labeling each image manually as per their respective class

Finally, we have to delete all the incompatible image files that cannot be used for the model-training process. Since this task has to be done manually, hence it can be tedious sometimes.

C. Code

Thus, by forming a dataset after collecting and labelling over 600 images, we begin to train our model based on the said dataset. We make use of Google Colaboratory again, and this time we load the YOLOv5 files that are needed for the training of our model.

```
!git clone https://github.com/ultralytics/yolov5
%cd yolov5
%pip install -qr requirements.txt

import torch
import utils
display = utils.notebook_init()

YOLOv5 🚀 v7.0-169-geef637c Python-3.10.11 torch-2.0.1+cu118 CUDA:0 (Tesla T4, 15102MiB)
Setup complete ✅ (2 CPUs, 12.7 GB RAM, 23.3/78.2 GB disk)
```

```
[2] !unzip -q /content/birddata.zip -d ../
```

Fig. 22. (Above) Loading the Yolov5 files (Below) Unzipping the uploaded dataset containing labeled data

After uploading a zipped folder of our dataset onto Google Colab, we run some commands to unzip it for the training of our model to begin. But before we can proceed to do so, there is a very important task, and that is to form a YAML file that would contain the locations of the training images and the validation images and contains the total number of classes taken into consideration while forming the dataset. Here, we term the YAML file as ‘yx.yaml’.

```

train: /content/birddata/images/train
val: /content/birddata/images/val

# Classes
nc: 6
names:
  0: Amur Falcon
  1: Bald Eagle
  2: Emu
  3: Golden Eagle
  4: Great Indian Bustard
  5: Wooper Swan

```

Fig. 23. Making the YAML file that contains the classes

After we have formed the YAML file and uploaded it on Colab, we begin the training process by iterating the dataset through 300 epochs to get an efficient and accurate object prediction model using image data.

```
!python train.py --img 640 --batch 3 --epochs 300 --data yx.yaml --weights yolov5x.pt --cache

AutoAnchor: 3.18 anchors/target, 1.000 Best Possible Recall (BPR). Current anchors are a good fit to dataset ✅
Plotting labels to runs/train/exp/labels.jpg...
Image sizes 640 train, 640 val
Using 2 dataloader workers
Logging results to runs/train/exp
Starting training for 300 epochs...

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
      0/299      3.88G    0.06344    0.02899    0.04094          7        640: 100% 161/161 [00:43<00:00,  3.73it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:05<00:00,  4.42it/s]
                   all        149        148      0.281      0.495      0.218      0.104

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
      1/299      4.05G     0.0453    0.02164    0.03533          4        640: 100% 161/161 [00:37<00:00,  4.34it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.21it/s]
                   all        149        148      0.581      0.336      0.304      0.129

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
      2/299      4.05G    0.04516    0.01818    0.03249          7        640: 100% 161/161 [00:37<00:00,  4.29it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.21it/s]
                   all        149        148      0.416      0.344      0.277     0.0984

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
      3/299      4.05G     0.0455    0.01611    0.02885          6        640: 100% 161/161 [00:37<00:00,  4.26it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.09it/s]
                   all        149        148      0.448      0.447      0.364      0.209

✓ 0s   completed at 9:46 PM
```

Fig. 24. Model-Training Process (start)

```
     12/299      4.06G    0.02874    0.01335    0.01266          6        640: 100% 161/161 [00:37<00:00,  4.31it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  6.43it/s]
                   all        149        148      0.681      0.717      0.693       0.38

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     13/299      4.06G    0.02888     0.0145    0.01252         10        640: 100% 161/161 [00:37<00:00,  4.29it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  6.58it/s]
                   all        149        148       0.76      0.701      0.731      0.377

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     14/299      4.06G    0.02999    0.01422    0.01234          9        640: 100% 161/161 [00:37<00:00,  4.29it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.07it/s]
                   all        149        148      0.539      0.594      0.602      0.295

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     15/299      4.06G     0.0274    0.01366    0.01088          7        640: 100% 161/161 [00:38<00:00,  4.23it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  6.73it/s]
                   all        149        148      0.764      0.632      0.735      0.391

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     16/299      4.06G    0.02852    0.01379    0.01231         10        640: 100% 161/161 [00:37<00:00,  4.28it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.16it/s]
                   all        149        148      0.678      0.702      0.744      0.418

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     17/299      4.06G    0.02952    0.01428    0.01264          9        640: 100% 161/161 [00:37<00:00,  4.27it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  6.61it/s]
                   all        149        148      0.649      0.673      0.714       0.43
```

Fig. 25. Model-Training Process (continuation)

```
                 all       149        148      0.776      0.763      0.826      0.479

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     60/299      4.06G    0.02246    0.01122   0.005249         10        640: 100% 161/161 [00:37<00:00,  4.24it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  6.59it/s]
                   all        149        148      0.797       0.79      0.836      0.499

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     61/299      4.06G     0.0225    0.01213   0.006766          8        640: 100% 161/161 [00:37<00:00,  4.27it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.18it/s]
                   all        149        148      0.852       0.78      0.851      0.514

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     62/299      4.06G    0.02171     0.0121   0.005687          7        640: 100% 161/161 [00:37<00:00,  4.27it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.19it/s]
                   all        149        148      0.776      0.758      0.842      0.528

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     63/299      4.06G    0.02179    0.01191   0.006062          6        640: 100% 161/161 [00:37<00:00,  4.27it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.22it/s]
                   all        149        148      0.772      0.799      0.847      0.504

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
     64/299      4.06G    0.02108    0.01184   0.004925         10        640: 100% 161/161 [00:37<00:00,  4.27it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.27it/s]
                   all        149        148      0.787      0.874      0.884      0.559
```

Fig. 26. Model-Training Process (continuation)

```
      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
    148/299      4.06G    0.01551   0.009138   0.002682          3        640: 100% 161/161 [00:38<00:00,  4.23it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.26it/s]
                   all        149        148      0.783      0.799      0.834       0.53

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
    149/299      4.06G    0.01439   0.009054   0.002612          5        640: 100% 161/161 [00:37<00:00,  4.28it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  6.66it/s]
                   all        149        148      0.798      0.764      0.838       0.52

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
    150/299      4.06G    0.01642   0.009437    0.00285          5        640: 100% 161/161 [00:37<00:00,  4.25it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.28it/s]
                   all        149        148      0.867      0.693       0.81       0.52

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
    151/299      4.06G    0.01506   0.008818   0.003313          6        640: 100% 161/161 [00:37<00:00,  4.29it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  6.50it/s]
                   all        149        148      0.799       0.76      0.828      0.517

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
    152/299      4.06G    0.01567   0.009637   0.002879          8        640: 100% 161/161 [00:37<00:00,  4.25it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.22it/s]
                   all        149        148      0.905      0.752       0.84      0.543

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
    153/299      4.06G    0.01466   0.009094   0.002574          6        640: 100% 161/161 [00:38<00:00,  4.21it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.26it/s]
                   all        149        148      0.858      0.793       0.85      0.536

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
    154/299      4.06G    0.01516    0.00877     0.0036         10        640: 100% 161/161 [00:37<00:00,  4.26it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:03<00:00,  7.23it/s]
                   all        149        148       0.83      0.816      0.869      0.547
```

Fig. 27. Model-Training Process (continuation)

```
Validating runs/train/exp/weights/best.pt...
Fusing layers...
Model summary: 322 layers, 86207059 parameters, 0 gradients, 203.9 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 25/25 [00:04<00:00,  5.13it/s]
                   all        149        148      0.781      0.868      0.884      0.559
           Amur Falcon        149         33      0.968      0.929      0.942       0.58
            Bald Eagle        149         31      0.706      0.903      0.893      0.602
                   Emu        149         28      0.752      0.964      0.955      0.634
          Golden Eagle        149         25      0.727       0.84      0.875      0.551
   Great Indian Bustard        149          9          1      0.845      0.995      0.672
           Wooper Swan        149         22      0.534      0.727      0.645      0.312
Results saved to runs/train/exp
```

Fig. 28. Model-Training Process (end)

After the training gets completed, it shows the destination folder of the best.pt and the last.pt that contains the activated kernels of the birds present inside of the images. The overall result of the machine learning process is given in the data mentioned below, and a score of mAP50 nearer to 1 means a more recognized class among any particular set of classes.

D. Output

Now, for the final step in our bird recognition training, we supply the model with different test data of birds which it was able to predict with near efficiency almost every time.

```
!python detect.py --weights /content/yolov5/runs/train/exp/weights/last.pt --img 640 --conf 0.25 --source /content/baldeagle.mp4

video 1/1 (30/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.7ms
video 1/1 (31/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.7ms
video 1/1 (32/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.2ms
video 1/1 (33/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.1ms
video 1/1 (34/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.5ms
video 1/1 (35/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.9ms
video 1/1 (36/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.0ms
video 1/1 (37/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 31.3ms
video 1/1 (38/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 30.6ms
video 1/1 (39/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 31.0ms
video 1/1 (40/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 31.1ms
video 1/1 (41/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 31.7ms
video 1/1 (42/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.2ms
video 1/1 (43/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.6ms
video 1/1 (44/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.5ms
video 1/1 (45/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 33.1ms
video 1/1 (46/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 32.0ms
video 1/1 (47/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 30.9ms
video 1/1 (48/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 30.8ms
video 1/1 (49/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 30.9ms
video 1/1 (50/277) /content/baldeagle.mp4: 384x640 1 Bald Eagle, 31.5ms

Executing (10s) <cell line: 1> > system() > _system_compat() > _run_command() > _monitor_process() > _poll_process()
```

Fig. 29. Model detecting for 'Bald Eagle'

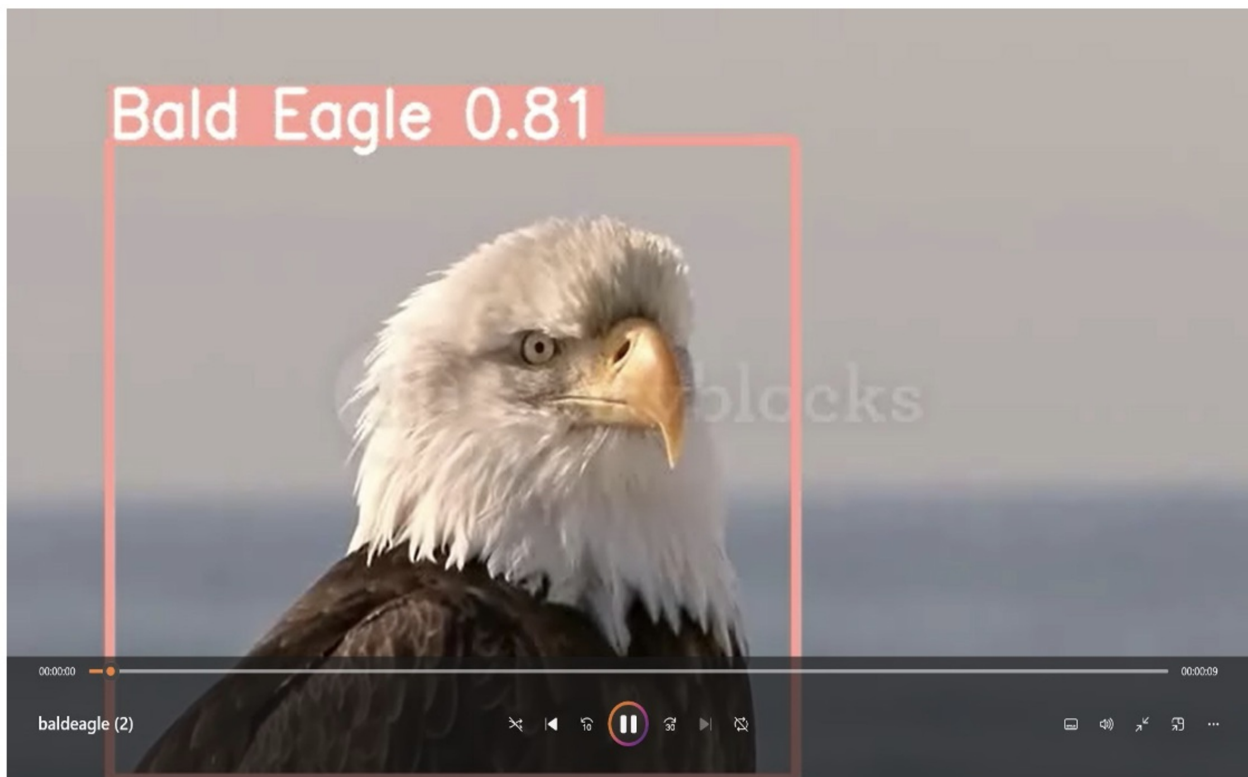

Fig. 30. Model predicting the presence of 'Bald Eagle'

```
!python detect.py --weights /content/yolov5/runs/train/exp/weights/last.pt --img 640 --conf 0.25 --source /content/WooperSwan1.mp4

video 1/1 (95/251) /content/WooperSwan1.mp4: 384x640 (no detections), 32.7ms
video 1/1 (96/251) /content/WooperSwan1.mp4: 384x640 1 Wooper Swan, 32.9ms
video 1/1 (97/251) /content/WooperSwan1.mp4: 384x640 1 Wooper Swan, 31.8ms
video 1/1 (98/251) /content/WooperSwan1.mp4: 384x640 1 Wooper Swan, 31.3ms
video 1/1 (99/251) /content/WooperSwan1.mp4: 384x640 1 Wooper Swan, 31.0ms
video 1/1 (100/251) /content/WooperSwan1.mp4: 384x640 1 Wooper Swan, 31.6ms
video 1/1 (101/251) /content/WooperSwan1.mp4: 384x640 1 Wooper Swan, 31.4ms
video 1/1 (102/251) /content/WooperSwan1.mp4: 384x640 1 Wooper Swan, 32.4ms
video 1/1 (103/251) /content/WooperSwan1.mp4: 384x640 1 Wooper Swan, 33.0ms
video 1/1 (104/251) /content/WooperSwan1.mp4: 384x640 1 Wooper Swan, 32.7ms
video 1/1 (105/251) /content/WooperSwan1.mp4: 384x640 2 Wooper Swans, 32.6ms
```

Fig. 31. Model detecting for 'Wooper Swan'

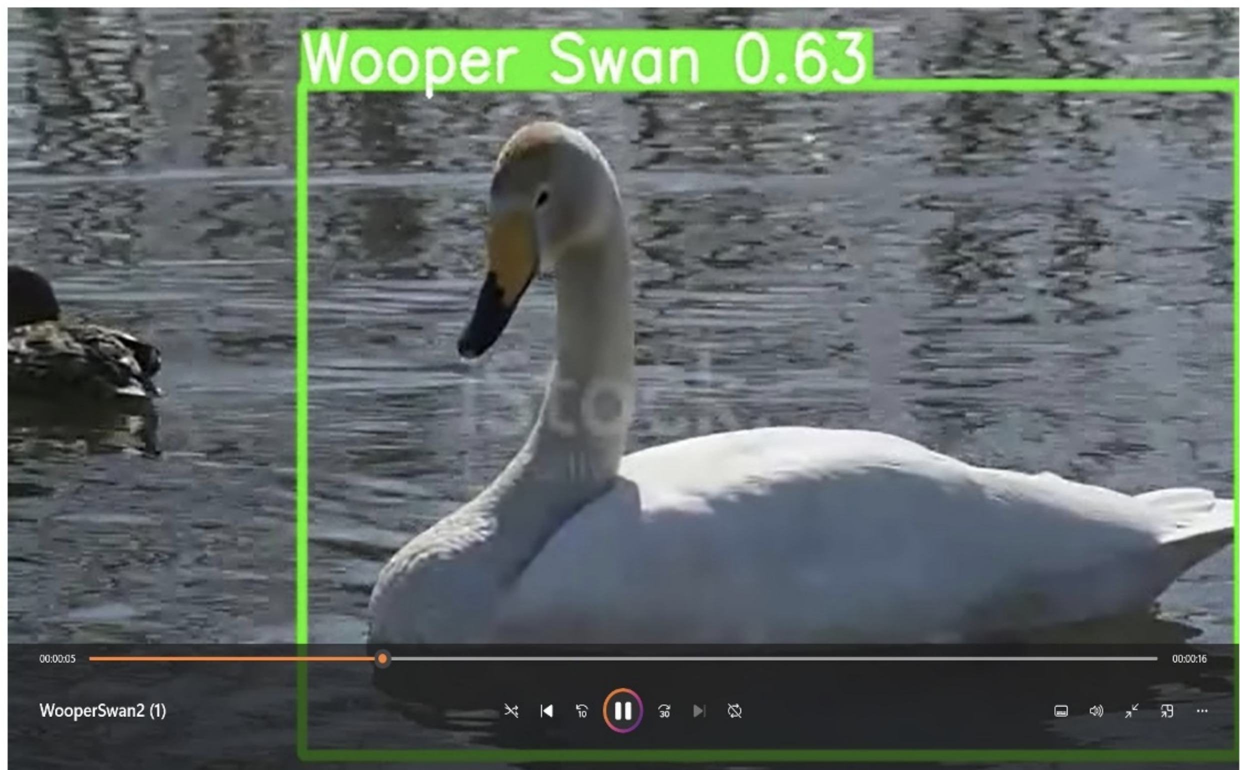

Fig. 32. Model predicting the presence of 'Wooper Swan'

```
!python detect.py --weights /content/yolov5/runs/train/exp/weights/last.pt --img 640 --conf 0.25 --source /content/WooperSwan2.mp4
video 1/1 (153/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 30.9ms
video 1/1 (154/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.5ms
video 1/1 (155/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 30.1ms
video 1/1 (156/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 29.8ms
video 1/1 (157/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 29.5ms
video 1/1 (158/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 30.5ms
video 1/1 (159/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 30.5ms
video 1/1 (160/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.0ms
video 1/1 (161/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 31.2ms
video 1/1 (162/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 29.8ms
video 1/1 (163/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 30.3ms
video 1/1 (164/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 30.4ms
video 1/1 (165/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 30.8ms
video 1/1 (166/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 29.3ms
video 1/1 (167/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 30.1ms
video 1/1 (168/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 29.6ms
video 1/1 (169/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.0ms
video 1/1 (170/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 31.2ms
video 1/1 (171/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.6ms
video 1/1 (172/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 33.0ms
video 1/1 (173/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.7ms
video 1/1 (174/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.7ms
video 1/1 (175/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.9ms
video 1/1 (176/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.7ms
video 1/1 (177/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 33.0ms
video 1/1 (178/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.2ms
video 1/1 (179/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.3ms
video 1/1 (180/649) /content/WooperSwan2.mp4: 384x640 1 Wooper Swan, 32.4ms
video 1/1 (181/649) /content/WooperSwan2.mp4: 384x640 (no detections), 33.1ms
```

Fig. 33. Model detecting for 'Wooper Swan'

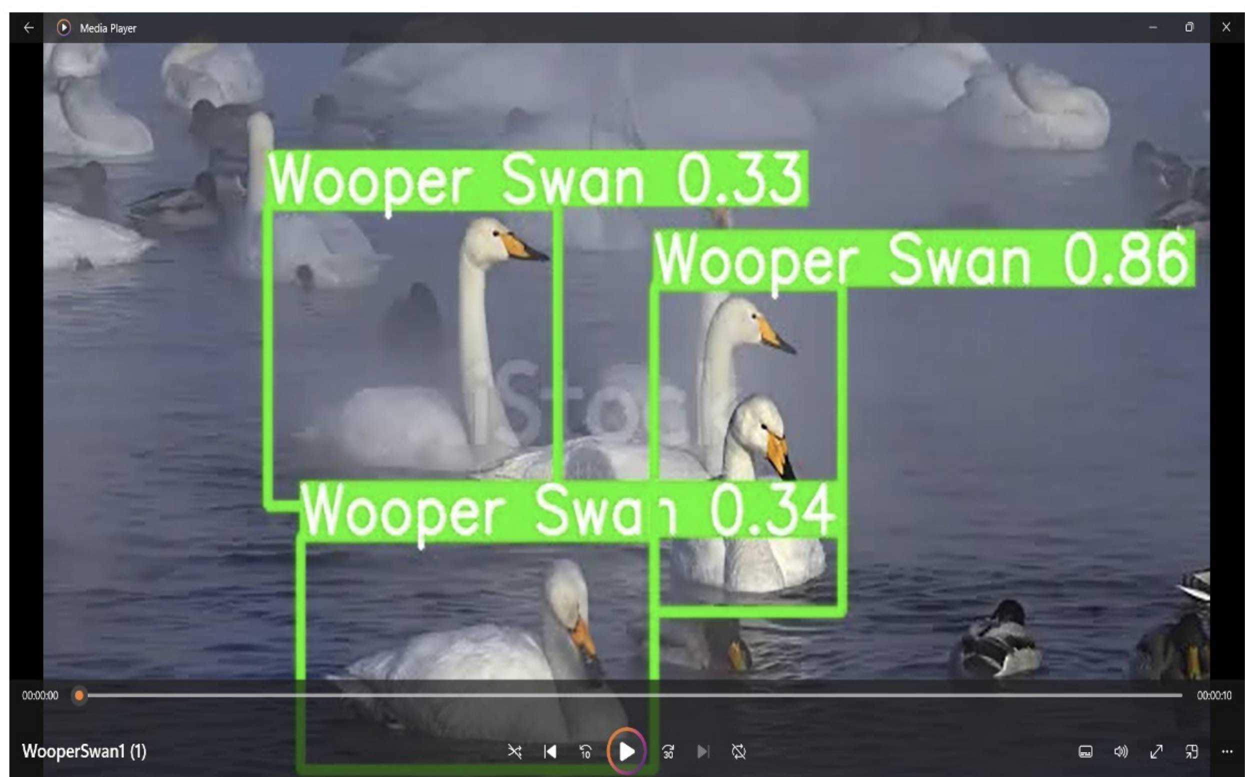

Fig. 34. Model predicting the presence of 'Wooper Swan'

```
!python detect.py --weights /content/yolov5/runs/train/exp/weights/last.pt --img 640 --conf 0.25 --source /content/AmurFalconVideo.mp4

video 1/1 (48/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 30.2ms
video 1/1 (49/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.5ms
video 1/1 (50/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 27.3ms
video 1/1 (51/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 27.4ms
video 1/1 (52/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 27.2ms
video 1/1 (53/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.1ms
video 1/1 (54/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 27.1ms
video 1/1 (55/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.0ms
video 1/1 (56/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.7ms
video 1/1 (57/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.8ms
video 1/1 (58/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 30.3ms
video 1/1 (59/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 29.7ms
video 1/1 (60/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.6ms
video 1/1 (61/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.2ms
video 1/1 (62/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.5ms
video 1/1 (63/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.0ms
video 1/1 (64/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 28.8ms
video 1/1 (65/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 29.0ms
video 1/1 (66/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 29.0ms
video 1/1 (67/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 30.4ms
video 1/1 (68/351) /content/AmurFalconVideo.mp4: 352x640 1 Amur Falcon, 29.1ms
```

Fig. 35. Model detecting for ’Amur Falcon’

Fig. 36. Model predicting the presence of ’Amur Falcon’

```
!python detect.py --weights /content/yolov5/runs/train/exp/weights/last.pt --img 640 --conf 0.25 --source /content/EmuVideo1.mp4

video 1/1 (173/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 30.9ms
video 1/1 (174/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 32.3ms
video 1/1 (175/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 29.9ms
video 1/1 (176/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 29.4ms
video 1/1 (177/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 29.3ms
video 1/1 (178/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 29.4ms
video 1/1 (179/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 29.8ms
video 1/1 (180/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 29.4ms
video 1/1 (181/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 29.1ms
video 1/1 (182/367) /content/EmuVideo1.mp4: 384x640 2 Emus, 29.1ms
video 1/1 (183/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 29.8ms
video 1/1 (184/367) /content/EmuVideo1.mp4: 384x640 1 Emu, 30.0ms
video 1/1 (185/367) /content/EmuVideo1.mp4: 384x640 2 Emus, 31.4ms
video 1/1 (186/367) /content/EmuVideo1.mp4: 384x640 2 Emus, 30.6ms
video 1/1 (187/367) /content/EmuVideo1.mp4: 384x640 2 Emus, 29.8ms
video 1/1 (188/367) /content/EmuVideo1.mp4: 384x640 2 Emus, 30.0ms
video 1/1 (189/367) /content/EmuVideo1.mp4: 384x640 2 Emus, 29.1ms
```

Fig. 37. Model detecting for 'Emu'

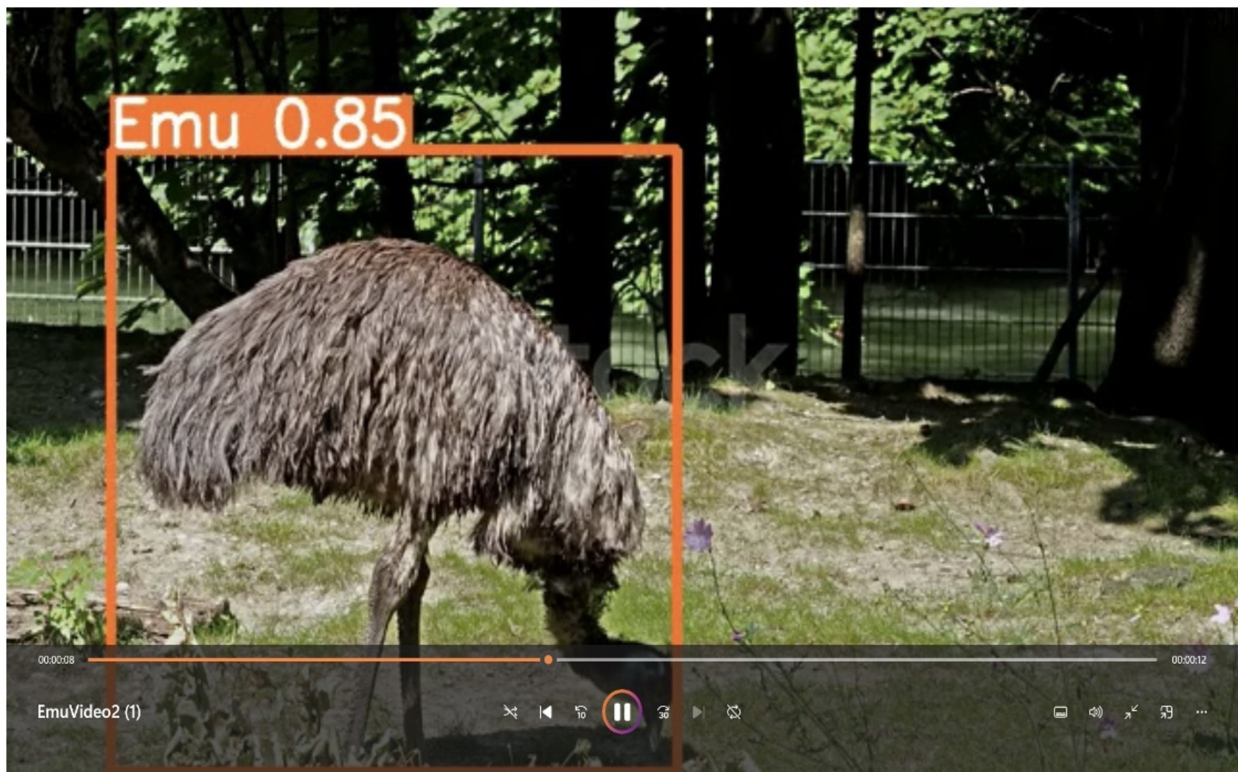

Fig. 38. Model predicting the presence of 'Emu'

```
!python detect.py --weights /content/yolov5/runs/train/exp/weights/last.pt --img 640 --conf 0.25 --source /content/EmuVideo2.mp4

video 1/1 (553/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.4ms
video 1/1 (554/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.0ms
video 1/1 (555/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 31.0ms
video 1/1 (556/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.8ms
video 1/1 (557/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 31.2ms
video 1/1 (558/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 31.5ms
video 1/1 (559/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 29.8ms
video 1/1 (560/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.3ms
video 1/1 (561/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.6ms
video 1/1 (562/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.6ms
video 1/1 (563/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.5ms
video 1/1 (564/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 31.7ms
video 1/1 (565/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 32.4ms
video 1/1 (566/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.2ms
video 1/1 (567/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 29.7ms
video 1/1 (568/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.3ms
video 1/1 (569/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.9ms
video 1/1 (570/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.5ms
video 1/1 (571/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.1ms
video 1/1 (572/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.4ms
video 1/1 (573/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 29.7ms
video 1/1 (574/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 31.9ms
video 1/1 (575/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 31.3ms
video 1/1 (576/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 31.0ms
video 1/1 (577/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 29.7ms
video 1/1 (578/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.0ms
video 1/1 (579/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.9ms
video 1/1 (580/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.7ms
video 1/1 (581/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 30.4ms
video 1/1 (582/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 32.2ms
video 1/1 (583/607) /content/EmuVideo2.mp4: 384x640 1 Emu, 32.4ms

✓ 32s completed at 9:25PM
```

Fig. 39. Model detecting for ’Emu’

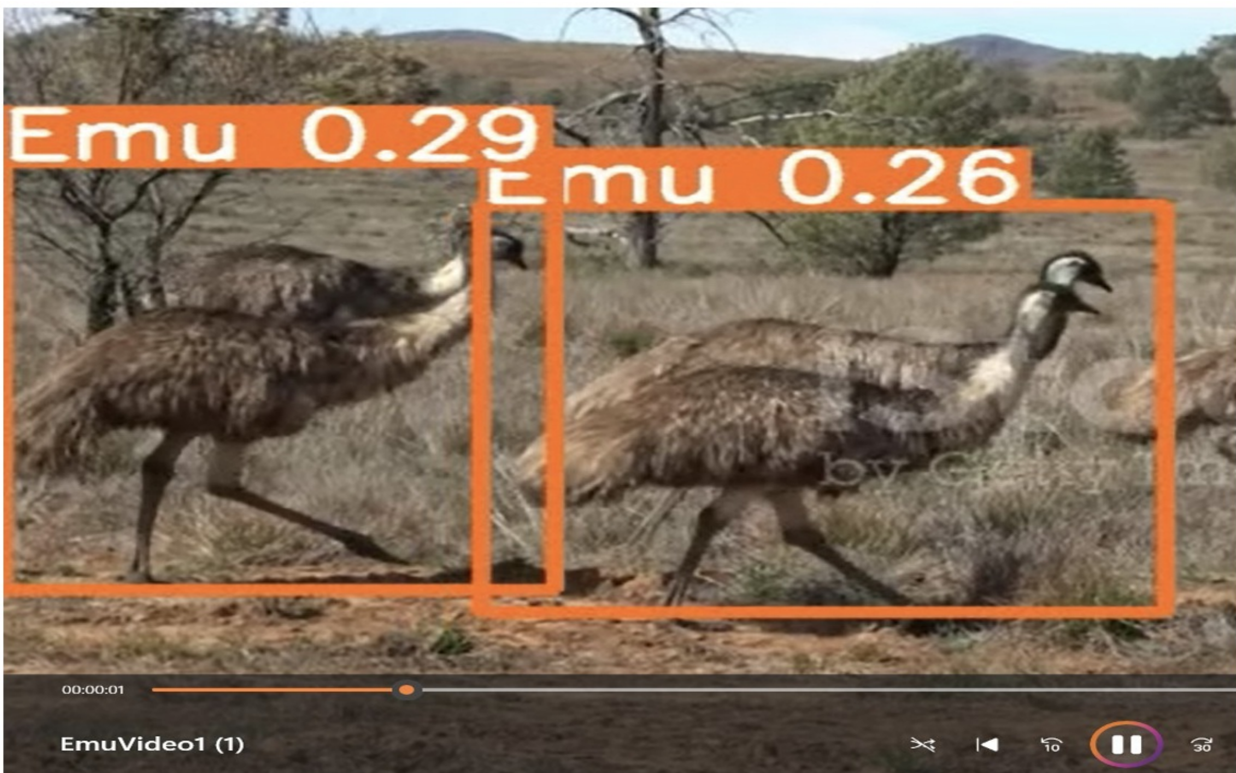

Fig. 40. Model predicting the presence of ’Emu’

```
!python detect.py --weights /content/yolov5/runs/train/exp/weights/last.pt --img 640 --conf 0.25 --source /content/GreatIndianBustardPic.jpg

detect: weights=['/content/yolov5/runs/train/exp/weights/last.pt'], source=/content/GreatIndianBustardPic.jpg, data=data/coco128.yaml, imgsz=[640
YOLOv5 🚀 v7.0-169-geef637c Python-3.10.11 torch-2.0.1+cu118 CUDA:0 (Tesla T4, 15102MiB)

Fusing layers...
Model summary: 322 layers, 86207059 parameters, 0 gradients, 203.9 GFLOPs
image 1/1 /content/GreatIndianBustardPic.jpg: 640x576 1 Great Indian Bustard, 63.2ms
Speed: 0.7ms pre-process, 63.2ms inference, 82.7ms NMS per image at shape (1, 3, 640, 640)
Results saved to runs/detect/exp5
```

Fig. 41. Model detecting for 'Great Indian Bustard'

Fig. 42. Model predicting the presence of 'Great Indian Bustard'

```
!python detect.py --weights /content/yolov5/runs/train/exp/weights/last.pt --img 640 --conf 0.25 --source /content/GoldenEagleVideo.mp4

video 1/1 (263/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.0ms
video 1/1 (264/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.5ms
video 1/1 (265/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.0ms
video 1/1 (266/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.1ms
video 1/1 (267/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.1ms
video 1/1 (268/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.4ms
video 1/1 (269/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.9ms
video 1/1 (270/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 28.2ms
video 1/1 (271/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.4ms
video 1/1 (272/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.7ms
video 1/1 (273/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.6ms
video 1/1 (274/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.0ms
video 1/1 (275/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.5ms
video 1/1 (276/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.3ms
video 1/1 (277/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.4ms
video 1/1 (278/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.3ms
video 1/1 (279/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.6ms
video 1/1 (280/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.5ms
video 1/1 (281/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.1ms
video 1/1 (282/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.5ms
video 1/1 (283/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.7ms
video 1/1 (284/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.4ms
video 1/1 (285/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.0ms
video 1/1 (286/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.7ms
video 1/1 (287/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.0ms
video 1/1 (288/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 30.2ms
video 1/1 (289/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.8ms
video 1/1 (290/317) /content/GoldenEagleVideo.mp4: 384x640 1 Golden Eagle, 29.1ms

✓ 23s completed at 9:24 PM
```

Fig. 43. Model detecting for 'Black Kite'

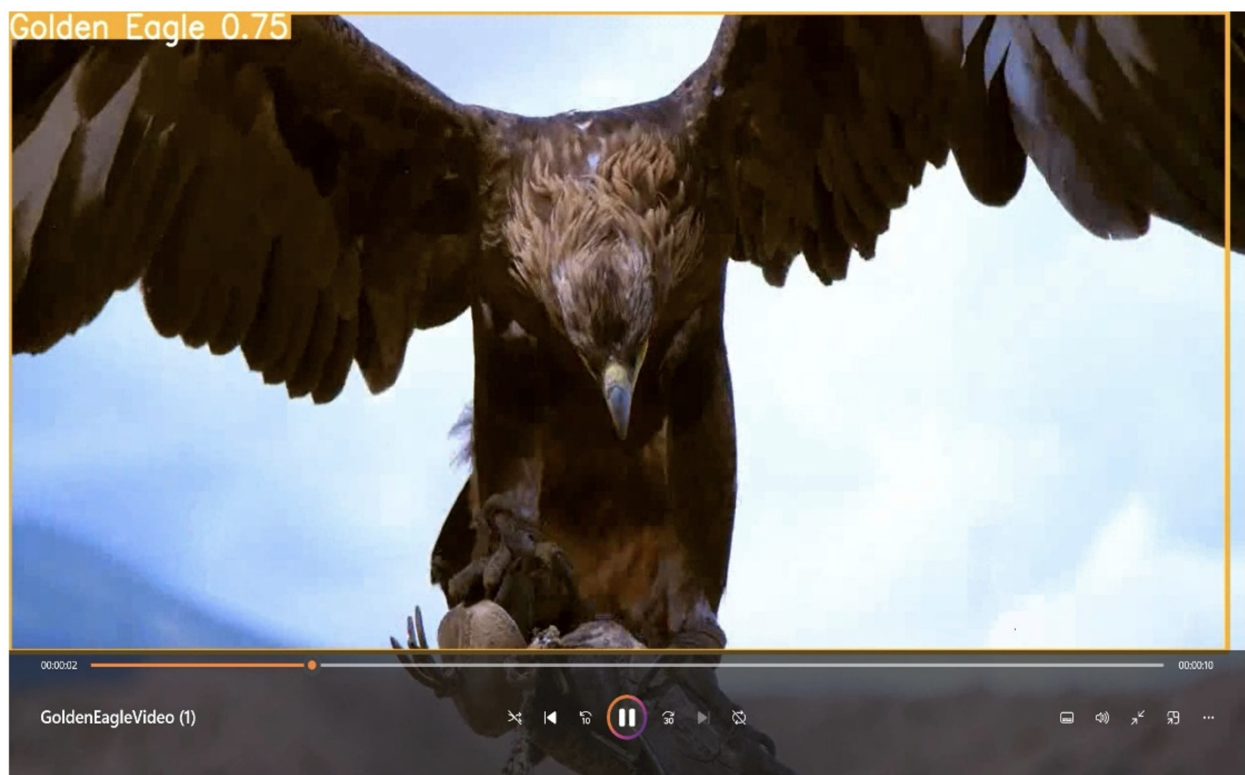

Fig. 44. Model predicting the presence of 'Black Kite'

VII. Conclusion and Future Works

We analyze the performance of our proposed approach and compare it with existing methods in the literature. We also address the challenges and limitations encountered, such as variability in image quality and bird poses, and the limited training data available for rare species. Finally, we suggest future work and potential improvements to the proposed

framework, including the incorporation of additional features, the use of transfer learning and pre-trained models, and integration with citizen science platforms. The findings of this study indicate that the proposed CNN-based approach can effectively detect and classify bird species from images, offering a valuable tool for conservationists and researchers working to protect and monitor bird populations. The field of bird species detection using machine learning has witnessed significant advancements, leveraging diverse datasets, methodologies, and performance metrics. These developments have the potential to revolutionize avian biodiversity monitoring, ecological research, and conservation efforts. While substantial progress has been made, there are still challenges to overcome and future directions to explore. Future research should focus on addressing data limitations by collecting and annotating comprehensive data- sets, involving collaborations between researchers and citizen scientists. The integration of multiple sensors and the development of deep learning architectures tailored for audio-based classification hold promise for improving the accuracy and robustness of bird species detection systems. Transfer learning, generalization, and model adaptation techniques should be investigated to ensure the applicability of models across different geographical regions and environmental conditions. Additionally, the development of real-time monitoring systems using lightweight models and IoT technologies can enable timely detection and response to conservation challenges. Future research directions in bird species detection using machine learning offer exciting avenues for advancements in the field. While significant progress has been made, several challenges remain, and further exploration is needed to improve the accuracy, robustness, and applicability of bird species detection systems. The following are key future directions that researchers can pursue:

A. Addressing Data Limitations

One of the main challenges in bird species detection is the scarcity of labeled data, especially for rare and endangered species. Future research should focus on collecting and annotating more diverse and comprehensive datasets, covering various geographical regions and seasons. Collaborative efforts between researchers, citizen scientists, and birding communities can significantly contribute to the availability of labeled data.

B. Multi-Sensor Fusion

Integrating data from multiple sensors, such as images, audio recordings, and environmental variables, holds promise for improving bird species detection accuracy. Future research should explore methods to effectively combine and fuse information from different modalities. This would allow for a more comprehensive understanding of bird presence, behavior, and habitat preferences.

C. Transfer Learning and Generalization

Assessing the transferability of models across different geographical regions, habitats, and recording conditions is an

important aspect of bird species detection. Future research should focus on developing models that can generalize well to unseen species and adapt to varying environmental factors. Transfer learning techniques, domain adaptation, and data augmentation strategies can aid in improving the generalization capabilities of bird species detection models.

D. Real-time Monitoring Systems

The development of real-time bird species detection systems can significantly contribute to conservation efforts, allowing for immediate detection and response to potential threats. Future research should explore lightweight and efficient models suitable for deployment on edge devices or in remote monitoring stations. Integration with IoT (Internet of Things) technologies and acoustic sensor networks can enable real-time monitoring of bird populations and their habitats.

VIII. Citations

- [Jmour et al., 2018] [Terven and Cordova-Esparza, 2023] [Bisong et al., 2019]

References

- [Bisong et al., 2019] Bisong, E. et al. (2019). Building machine learning and deep learning models on Google cloud platform. Springer.
- [Jmour et al., 2018] Jmour, N., Zayen, S., and Abdelkrim, A. (2018). Convolutional neural networks for image classification. In 2018 international conference on advanced systems and electric technologies (IC ASET), pages 397–402. IEEE.
- [Terven and Cordova-Esparza, 2023] Terven, J. and Cordova-Esparza, D. (2023). A comprehensive review of yolo: From yolov1 to yolov8 and beyond. arXiv preprint arXiv:2304.00501.