Rock paper scissors tracking using YOLO

Javier Montes Pérez
Emilio Rodrigo Carreira Villalta

Friday 2nd August, 2024

Table of Contents

List of Figures

1. Introduction

This document presents the process of developing a computer vision model using the YOLO (You Only Look Once) architecture to detect and track rock-paper-scissors gestures in images and videos. The objective is to build a system capable of automatically identifying and following these gestures within images or video frames, demonstrating the model's effectiveness and potential applications in various interactive systems and educational tools.

Figure 1.1: Rock paper scissors game

The project, accessible in this GitHub repository [1], includes various features aimed at achieving this goal. It encompasses the detection and tracking of rock-paper-scissors gestures in images or video using YOLO, preprocessing and augmentation of training data, scripts for training custom YOLO models, and evaluation scripts to assess model performance.

This document details the steps involved in setting up and running the project, explaining the preprocessing techniques used for the training data, the architecture of the YOLO model, and the training and evaluation procedures. Each aspect of the implementation is thoroughly described, ensuring that readers can understand both the practical implementation and the underlying concepts.

This work is part of a broader effort to develop advanced computer vision applications and is inspired by the documentation and resources provided by Ultralytics [1]. The goal is to create a comprehensive guide that not only facilitates the understanding of the techniques used but also encourages experimentation and learning in the field of computer vision and deep learning.

[1] https://github.com/rorro6787/ImageTracking

2. The Setup of the Project

Contents

2.1 YOLO, the model

YOLO, which stands for "You Only Look Once," is a popular real-time object detection system. It is widely used in computer vision tasks because of its high speed and accuracy in detecting objects within images and videos. Here's a detailed overview of YOLO:

- **Single Neural Network Approach**: Unlike traditional object detection systems that use a pipeline of separate models for region proposal and classification, YOLO uses a single convolutional neural network (CNN) to predict multiple bounding boxes and class probabilities directly from the full images in one evaluation.

- **Real-time Detection**: YOLO is designed to process images extremely quickly, making it suitable for real-time applications. For instance, YOLO can achieve frame rates of up to 45 frames per second (fps) on a modest GPU.

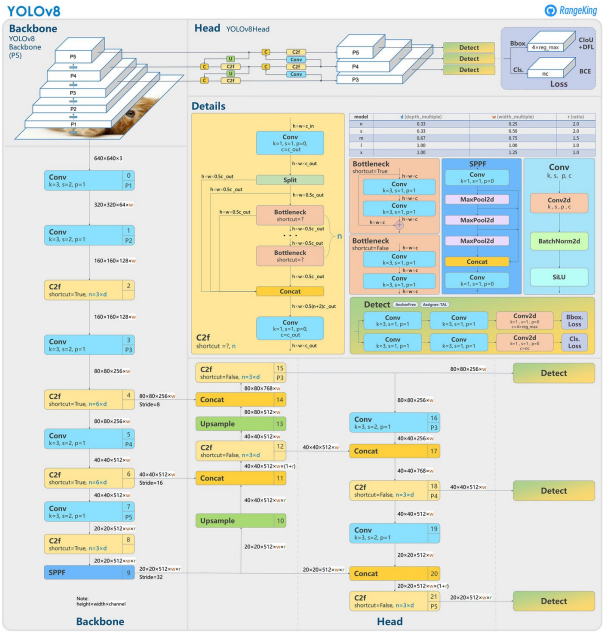

Figure 2.1: YOLO inside architecture

YOLOv8 represents the latest iteration in the YOLO (You Only Look Once) series of object detection models. As a continuation of the YOLO family, YOLOv8 builds upon the advancements made by its predecessors, enhancing both speed and accuracy. Here's a detailed look at what makes YOLOv8 stand out, particularly the yolov8x.pt model.

2.1.1 Why yolov8n.pt?

The model yolov8n.pt is one of the variants in the YOLOv8 family, and the name conveys specific information about its characteristics:

- **yolov8**: This prefix denotes that the model is part of the YOLO version 8 series. It indicates the architecture and the improvements over earlier YOLO versions.
- **n**: This letter stands for "nano" and signifies that the model is designed with a smaller capacity compared to its larger counterparts like yolov8s (small) or yolov8m (medium). In the context of YOLOv8, the n variant offers fewer parameters, which can lead to faster inference and lower resource consumption while maintaining reasonable detection accuracy and robustness.
- **.pt**: This suffix indicates that the model is saved in PyTorch format. PyTorch is a popular deep learning framework that provides flexibility and ease of use for training and deploying models.

Even though yolov8n is a "nano" model, it is designed to be highly efficient and lightweight. In fact, yolov8n.pt is an excellent choice, especially when computing resources are limited because it is a well-balanced model that provides strong performance for object detection tasks without being overly demanding on computing resources. Its advanced architecture and optimizations make it an excellent choice for projects where efficiency and effectiveness are key, especially when working with constrained computational environments. In the case of training for hand detection in a Rock-Paper-Scissors application, yolov8n.pt offers a pragmatic solution by delivering

the necessary accuracy and speed to recognize hand gestures while being well-suited for deployment on devices with limited hardware capabilities.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Figure 2.2: Rock paper scissors game

Moreover, the choice of yolov8n.pt for training a hand detection model in a Rock-Paper-Scissors application underscores a strategic balance between model efficiency and performance. The YOLOv8n variant's reduced parameter count and optimized architecture are particularly advantageous for real-time applications, where processing speed and responsiveness are crucial. Its lightweight design ensures that the model can be deployed effectively on edge devices with constrained resources, such as mobile phones or embedded systems, without compromising on detection accuracy. By leveraging the PyTorch framework's capabilities, yolov8n.pt facilitates streamlined development and deployment workflows, enabling rapid iterations and adjustments as needed. This makes it an ideal candidate for practical implementations where maintaining a high level of performance while minimizing computational overhead is essential.

2.2 How to prepare the dataset

Regardless of whether you are using YOLO 5, 7, or 8, the format of the dataset remains the same. In other words, if you have already prepared your dataset for training an older YOLO model, you can reuse it to train a newer version. As we already said, we are going to train the YOLO model "yolov8n.pt" because it is lightweight with fewer parameters, allowing the training process to be faster while still maintaining good performance.

If you don't already have a dataset prepared to train your model, you can use the services of Roboflow to create one from scratch. However, be aware that if you are working alone, the process of obtaining your custom dataset can be extremely tedious.

2.2.1 Structure of the Dataset

Before we dive into how to obtain your own custom dataset, let's first explain the file structure you need and the configuration files required. Firstly, we need to organize

our dataset into a specific directory structure. A common structure could look like this:

```
/name_dataset
    /all_images
        image1.jpg
        image2.jpg
        ...
        image.png

    /all_labels
        image1.txt
        image2.txt
        ...
        image.txt
```

In the all_images folder, you'll find each image you plan to use in your dataset. In the all_labels folder, you'll find a .txt file for each corresponding image. Let's examine the contents of one of these .txt files:

```
6 0.72475961538 0.3209134615 0.3846153846 0.59134615384
```

At first glance, it might not be clear what these numbers represent. To clarify, here's the general format for a .txt file:

```
<class_id> <x_center> <y_center> <width> <height>
```

Here's what each value represents:

- <**class_id**>: Integer representing the class of the object (starting from 0).
- <**x_center**>: Normalized x-coordinate of the center of the bounding box (value between 0 and 1).
- <**y_center**>: Normalized y-coordinate of the center of the bounding box (value between 0 and 1).
- <**width**>: Normalized width of the bounding box (value between 0 and 1).
- <**height**>: Normalized height of the bounding box (value between 0 and 1).

2.2.2 Obtain your dataset with Roboflow

Roboflow [2] is a platform designed to help users build, manage, and deploy computer vision models. It provides tools and services that simplify the process of preparing data, training models, and integrating those models into applications. Once you have an account, you can create a new project and specify the classes you want the model to identify:

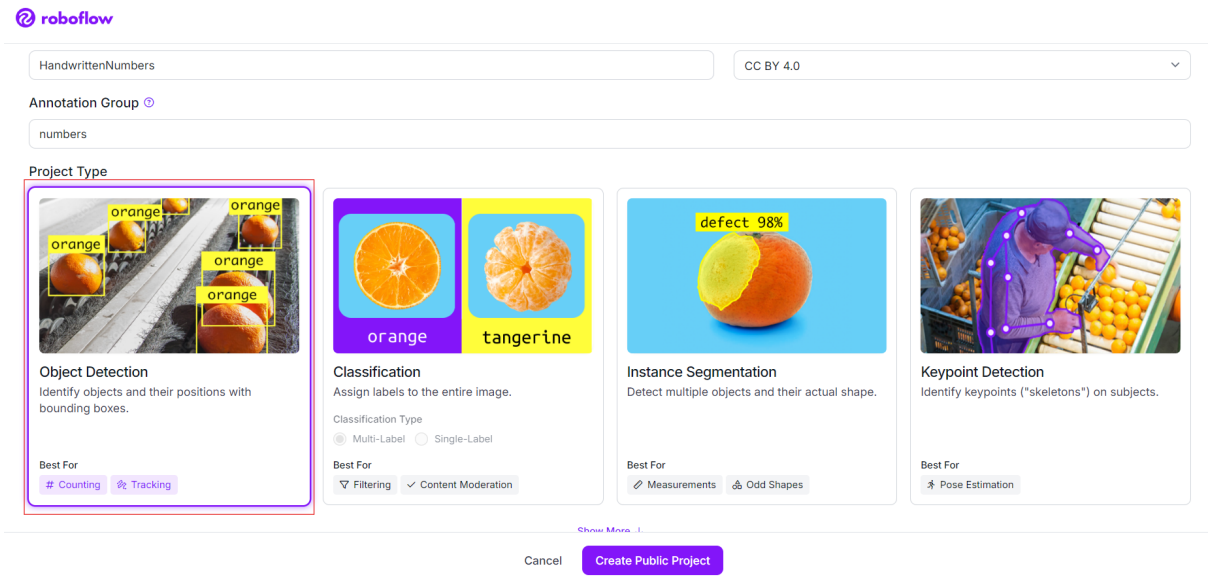

Figure 2.3: Create a project in Roboflow

Once you have created the project, you can upload or drag and drop all the images you want to use for your dataset. Depending on the number and quality of the images, this may take a moment. After uploading, you'll have the option to manually label the images, collaborate with others to label them, or use a custom model to assist with labeling. Since we do not have a custom model, the alternative will be to manually label the images.

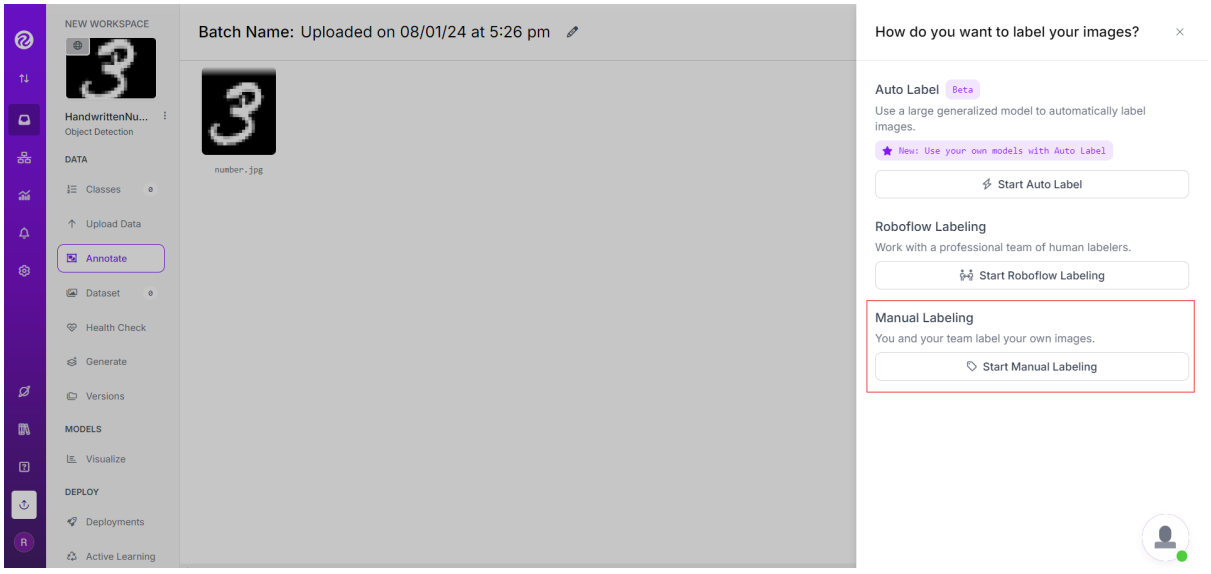

Figure 2.4: How to label the images?

Once you have done so, the next step is straightforward: to manually label each image and determine the class it belongs to. Here's an example of how this is done:

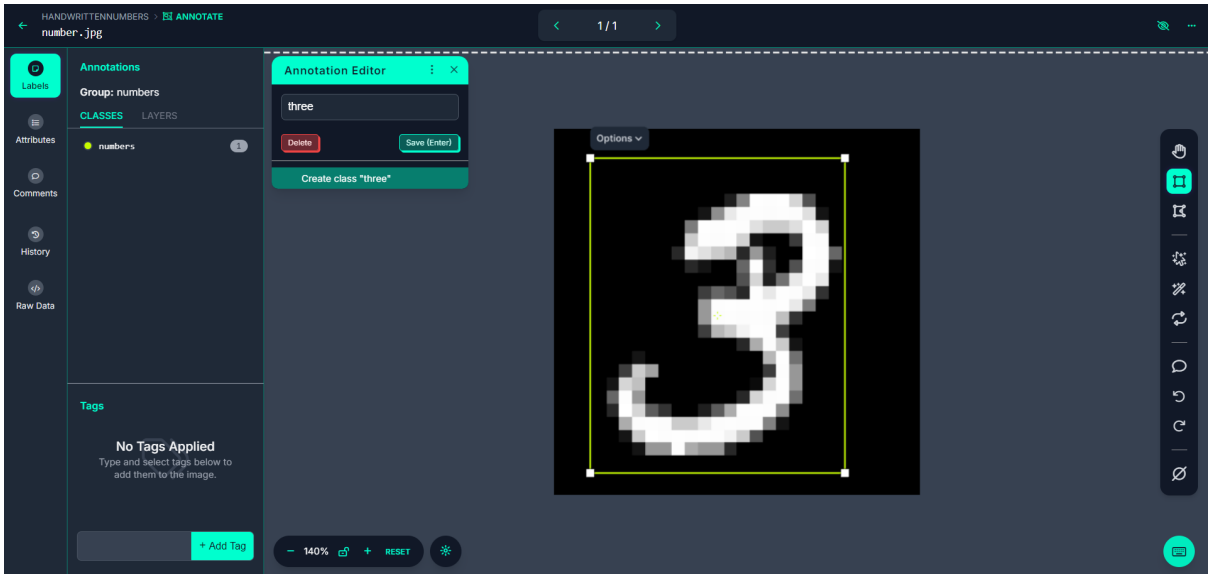

Figure 2.5: Label of a 3

Once we have labeled all the images correctly (for a large dataset, we can manually annotate the initial images, train a model, and then use that model to annotate the rest. After several iterations, we will end up with a robust model), the next step is to choose the percentages for splitting the dataset into training, validation, and testing sets.

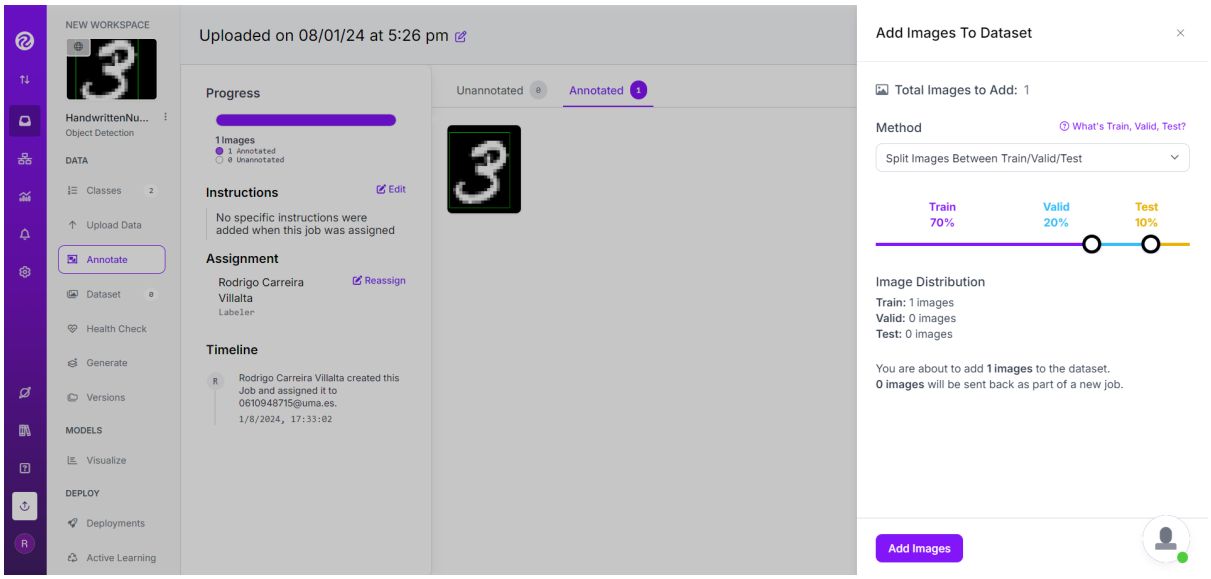

Figure 2.6: Percentages for train, validate and test

Once everything is done, it's time to create the dataset in YOLO format. Roboflow will ask if you want to add any preprocessing, augmentation, or other modifications. This decision is up to you. After configuring these options, click on "Create" to generate your dataset in the YOLO format:

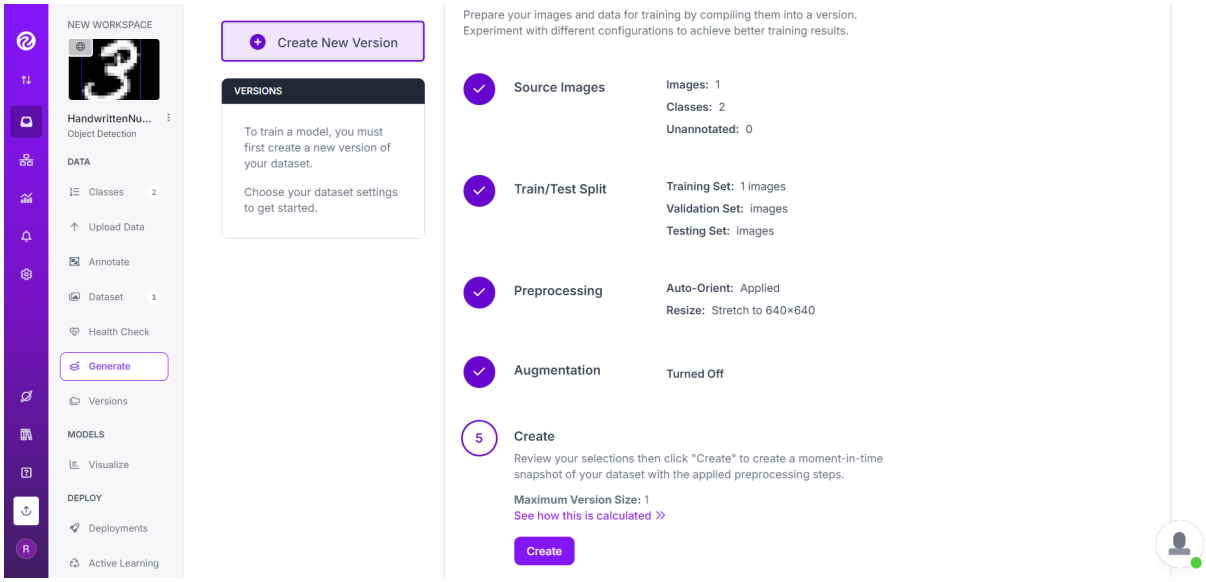

Figure 2.7: Obtaining Our YOLO-Formatted Dataset

2.2.3 Configuration files

The real importance of getting a .txt file for every image in our dataset is to be able to finally obtain a bag of files indicating the geometrical position of the detections in each of the images. At the end of this whole process, the data structure is composed of two folders: ”images” and ”labels”. In our case, they both are included in a ”dataset.zip” file, which is unzipped by our Python script.

Once the ”dataset.zip” file is unzipped, we obtain two folders, but what should we do with them? The answer is in the Ultralytics Documentation. YOLO models are designed to be trained with .yaml files. YAML is a human-readable data serialization language. It is commonly used for configuration files and in applications where data are being stored or transmitted. In our case, our configuration.yaml file will specify where the training split, validation split, and testing split are located. That’s all! Our Python script will take care of both making these splits into the three different folders needed, as well as creating the .yaml file. Its format will be as follows:

```
path: "your_computer_path\dataset_split"
train: train
val: val
test: test

nc: 3
names: ['paper', 'rock', 'scissors']
```

Finally, the whole structure of the folders that our ‘yolov8n.pt‘ model will use to train itself will be something like this:

```
/dataset
    /train
        /images
            image1.jpg
```

```
            image2.jpg
            ...
        /labels
            image1.txt
            image2.txt
            ...
/val
        /images
            image11.jpg
            image22.jpg
            ...
        /labels
            image11.txt
            image22.txt
            ...
/test
        /images
            image111.jpg
            image222.jpg
            ...
        /labels
            image111.txt
            image222.txt
            ...
```

Some of the images found in the ”images” folder will look like the following, and as you can see, not all the images will be easy for our YOLO model. For example, Figure 1.7 shows a restaurant image where YOLOv8n has to detect nothing:

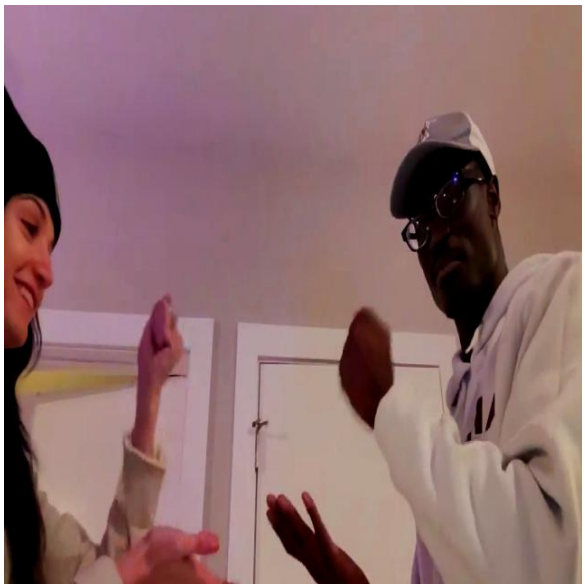

Figure 2.8: Dataset image: Example 1

Figure 2.9: Dataset image: Example 2

2.2.4 The dataset of this project

As most readers might have already deduced, the most tedious and time-consuming process is obtaining the dataset for training the model. Fortunately, for this project, we have a great resource available: a public dataset from Roboflow for rock-paper-scissors image recognition. This dataset provides a large collection of images already labeled and ready for use, saving us the effort of manual labeling. Using this dataset will allow us to focus on the learning aspects of the project and developing the image recognition model.

This significantly shortens the time we would have spent manually labeling each image using tools like Roboflow. Instead, we are left with the tasks of setting up the file structure, preparing the configuration files, and writing the Python script needed to prepare the dataset for model training.

2.2.4.1 Script to prepare the Dataset

Now it's time to break down the code used in the project to prepare the public Roboflow dataset for rock-paper-scissors image recognition and train our YOLO model. First, we start with the necessary imports and define the paths where our dataset is located and where it will be organized:

```
import cv2
import os
import sys
from ultralytics import YOLO
import shutil
import random
from zipfile import ZipFile

cd = os.getcwd()                    # it's gonna be /src
base_path = 'dataset_split'
```

Secondly, we have an auxiliary function that moves files to their respective directories based on the dataset split.

```
def move_files(data, split, images_path, labels_path):
    for img_file, lbl_file in data:
        shutil.move(os.path.join(images_path, img_file),
          ↪ os.path.join(base_path, split, 'images',
          ↪ img_file))

        shutil.move(os.path.join(labels_path, lbl_file),
          ↪ os.path.join(base_path, split, 'labels',
          ↪ lbl_file))
```

Lastly, we have a function that sets up the directory structure, splits the dataset into training, validation, and test sets, and moves the files accordingly using the auxiliary function described earlier:

```
def prepare_structure():
    # Define paths
    zip_file_path = 'dataset.zip'
    base_extract_path = 'dataset'
    url = 'drive_url_with_dataset'
    ...
    with open(os.path.join(base_path, 'config.yaml'), 'w') as
       ↪ file:
        file.write(config_content)

    print("Data distribution and yaml creation completed
       ↪ successfully ...")
```

Lastly, we implemented two auxiliary functions to remove the entire folder structure if the user decides to do so:

```
def remove_empty_dirs(path):
    for dirpath, dirnames, filenames in os.walk(path,
       ↪ topdown=False):
        if not dirnames and not filenames:
            os.rmdir(dirpath)

    # Attempt to remove the base directory
    try:
        os.rmdir(path)
    except OSError as e:
        print()

def removeAll():
    if os.path.exists(f"{cd}/dataset_split"):
        shutil.rmtree(f'{cd}/dataset_split',
           ↪ ignore_errors=True)
        print("Data distribution deleted successfully ...")
    ...
    else:
        print(f"The directory {cd}/dataset does not exist.")
```

3. Fine-Tunning the Model

Contents

In this chapter, we will demonstrate, through code and explanation, the steps needed to successfully train our YOLO model to recognize rock-paper-scissors gestures using the dataset prepared in the previous chapter. We will also cover how to utilize powerful NVIDIA GPUs in the cloud for free to meet the computational requirements of this process.

3.1 Nvidia's GPUs to do our work

To train a model of this scale, the power of our computer's CPU is insufficient. Therefore, we use Google Colab for training the model with our generated dataset. Google Colab provides us with an Nvidia T4 GPU with 12GB of VRAM. Given that we plan to train the YOLOv8n.pt model, which has approximately 3 million parameters, this GPU will be adequate for the task. After creating the account and setting up the Jupyter notebook, we select the T4 GPU and execute the following command:

```
!nvidia-smi
!pip install ultralytics
```

After that, we run the Python script we created to set up the folder structure. Next, we execute the Python script in our notebook that will be used to train our model:

```
def train_model(yaml_file, epochs, project):
    model = YOLO(model="yolov8n.pt", task="detect")
    model.to('cuda')
    model.train(data = yaml_file,
                epochs = epochs,
            project = project,
                batch = 8,
            name = "train")
    # batch = 8 is neccesary because GPU does not support
       ↪ higher

    results2 = model.val(data = yaml_file,
                         project = project,
                         batch = 8,
```

```
                    name = "test",
                    split = "test")

# Create the full path for the text file
metrics_file_path2 = os.path.join(project,
    ↪ "testing_metrics.txt")

# Write the metricts in the text file
with open(metrics_file_path2, 'w') as f:
    ...
```

Now, we set up the entire dataset structure, and immediately afterward, we train our model for the desired number of epochs (in my case, I will choose 32) and we wait for the results:

```
prepare_structure()
train_model(f"{cd}/dataset_split/config.yaml", 32,
    ↪ "Hands_Tracking_Model")
```

3.2 Model's training results

Once the training process has finished (which took around 2 hours), we obtain a folder containing the trained model. Inside we can obtain plenty of information as well as results.png that contains the graphs with the different metrics of the output model:

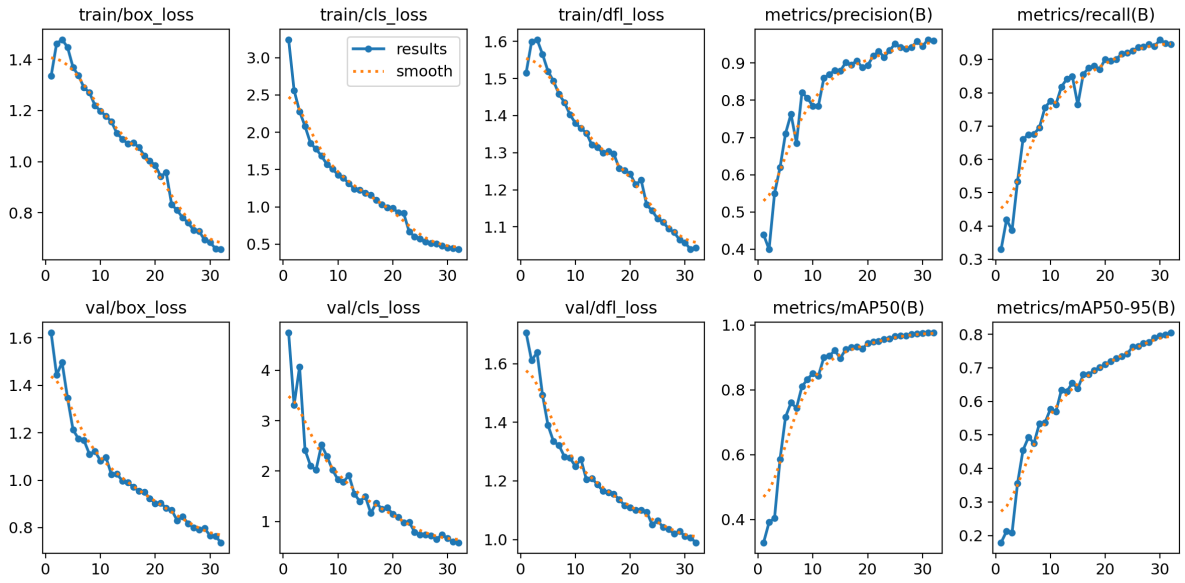

Figure 3.1: Trained Model Metrics

Performance metrics are essential tools for assessing the accuracy and efficiency of object detection models. They provide valuable insights into a model's ability to correctly identify and localize objects within images. Additionally, these metrics help us understand how well the model manages false positives (incorrectly identifying an

object that is not there) and false negatives (failing to detect an object that is present). Such insights are indispensable for evaluating the model's overall performance and identifying areas for improvement. In this guide, we will delve into the various performance metrics used in the context of $YOLOv8$, discuss their importance, and explain how to interpret them effectively.

Metrics such as $Precision$, $Recall$, and the $F1\ score$ are commonly used to measure the performance of object detection models. $Precision$ quantifies the accuracy of the positive predictions made by the model, while $Recall$ measures the ability of the model to identify all relevant objects in the images. The $F1\ score$, a harmonic mean of $Precision$ and $Recall$, provides a single metric that balances both concerns. Additionally, metrics like Average Precision (AP) and Mean Average Precision (mAP) are particularly significant in object detection. They summarize the model's performance across different thresholds, offering a comprehensive view of its effectiveness.

The $IntersectionoverUnion(IoU)$ metric is another crucial measure. It evaluates the overlap between the predicted bounding boxes and the ground truth bounding boxes. A higher IoU indicates better accuracy in object localization. By examining these metrics, one can gauge the strengths and weaknesses of the model in different scenarios, enabling targeted improvements and optimizations.

As we know, in a model development pipeline, the process doesn't stop at training and validating the model. Another crucial step is testing the model we have just trained. Testing the model is essential to evaluate its performance on unseen data, which simulates real-world scenarios and ensures that the model generalizes well beyond the training and validation datasets. Testing serves multiple purposes:

- **Performance Evaluation**: It provides a final assessment of the model's accuracy, precision, recall, and other relevant metrics. This step helps determine if the model meets the desired performance criteria and is ready for deployment.

- **Generalization Capability**: By testing on unseen data, we can measure how well the model generalizes to new, real-world data. This helps identify any overfitting that may have occurred during training, where the model performs well on training data but poorly on new data.

- **Robustness Analysis**: Testing allows us to analyze the model's robustness under various conditions. This includes evaluating the model's performance across different lighting conditions, object occlusions, and other challenging scenarios that it might encounter in real-world applications.

- **Error Analysis**: During testing, we can identify specific cases where the model fails or performs suboptimally. This analysis can provide valuable insights into the model's weaknesses and guide further improvements and refinements.

- **Benchmarking**: Testing provides a standardized way to compare the performance of different models or approaches. By using consistent testing protocols and datasets, we can objectively evaluate the strengths and weaknesses of various models and select the best one for the task at hand.

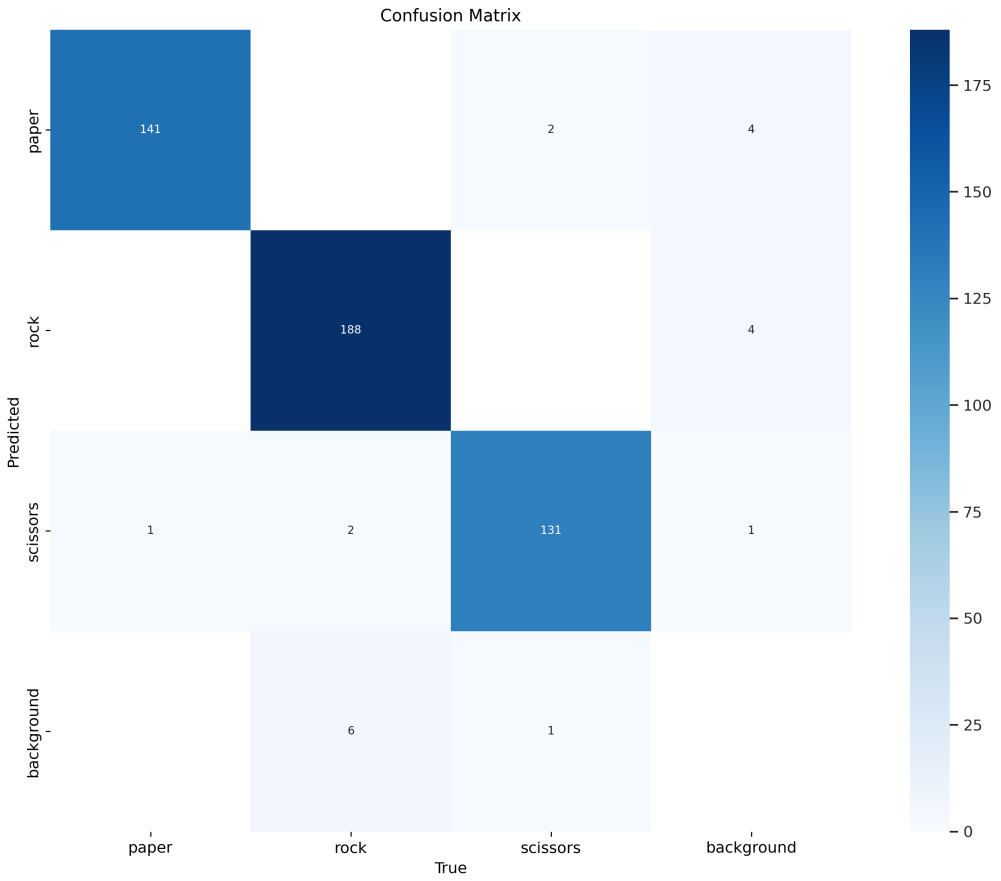

Figure 3.2: Confusion matrix during testing period

As we can see in the confusion matrix the testing period of our *YOLO* model has created, the number of *"paper"*, *"rock"* and *"scissors"* the model is detecting is really high. It says us everything is going as we expected.

3.3 Tests to the Model

We would like to test how well the model can recognize objects using our own prepared images. Here is an example of what the model detects:

Figure 3.3: Test Image

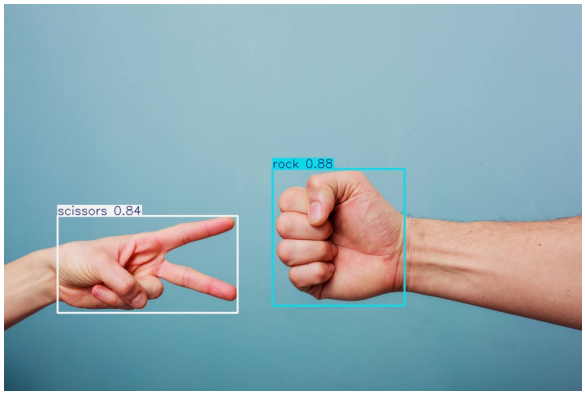

Figure 3.4: Test Image Detection

4. Image Tracking on Video

Contents

In this chapter, we will use the already trained model to recognize rock-paper-scissors gestures not only in images but also in videos. We will process the video information, identify the gestures, and display the results to demonstrate that the model performs accurately in our tests.

4.1 Rock, Paper, Scissors video tracking

To evaluate our real use cases, we have two types of tasks. The first task, which I am describing now, is to create a script that analyzes a video frame by frame to detect every instance of paper, rock, and scissors. The code in the repository implements all the necessary functionality. We will now provide pseudocode that illustrates what the function does (you can check the actual implementation in the repository [1]):

```
def process_video(video_path, buffer_size)
download_model()
open_video()
iniciate_empty_buffer()
    while video_open():
        read_frame()
        if valid_frame:
            model.analize(frame)
            if valid_detection():
                update_buffer(detection)
                if buffer_full():
                    if consistent_data_buffer():
                        process_results(buffer)
                        clean_buffer()
                    else:
                        show_inconsistency_message
            show_frame_with_results()

        if key_to_stop()
            break
```

[1] https://github.com/rorro6787/ImageTracking

```
    close_video()
    free_resources()
```

With the current code implementation, we can already use the trained model to track our desired images in any type of video, as previously mentioned.

4.2 Rock, Paper, Scissors versus

The second feature we implemented extends our existing script. Now that we have a script that can track classes in a video, we've created a new one that, given a match in this game, can detect what each competitor has chosen and provide the result of the game. The repository contains a variety of auxiliary functions, but the one responsible for determining the outcome of the match is the following function, which works in conjunction with the one described above.

```
def determine_winner(hand1, hand2):
    hand1 = classes[int(hand1.item())]
    hand2 = classes[int(hand2.item())]

    print(f"Facing␣{hand1}␣vs␣{hand2}\U0001f600")
    time.sleep(1)

    if hand1 == hand2:
        print("It's␣a␣draw!")
        print(f"Both␣hands␣are␣{hand1}.")
    elif (hand1 == "rock" and hand2 == "scissors") or \
            (hand1 == "scissors" and hand2 == "paper") or \
            (hand1 == "paper" and hand2 == "rock"):
        print("WE␣HAVE␣A␣WINNER!")
        print(f"{hand1}␣beats␣{hand2}␣:)".upper())
        print(emoji.emojize(":fire:"))
    else:
        print("WE␣HAVE␣A␣WINNER!")
        print(f"{hand2}␣beats␣{hand1}␣:)".upper())
        print(emoji.emojize(":fire:"))
```

As you can see the $hand1$ and $hand2$ variables that $determine_winner$ receives are tensors or numpy arrays. They are just single number array indicating the index of the class list that the model has detected in the frame which is being analyzed. That's why we use $handx.item()$. The rest of the method is just a simply comprobation of wheter the $hand1$ is winning $hand2$ or vice versa.

5. Bibliography

[1] **Ultralytics Documentation**, *Ultralytics Documentation*, Aug 2024, https://docs.ultralytics.com/ pages

[1] **Ultralytics Documentation for metrics**, *Ultralytics Documentation for metrics*, Aug 2024, https://docs.ultralytics.com/guides/yolo-performance-metrics/#introduction pages

[2] **Roboflow**, *Roboflow Platform*, Aug 2024, https://app.roboflow.com/ pages