Handwritten digits tracking using YOLO

Javier Montes Pérez
Emilio Rodrigo Carreira Villalta

Thursday 1st 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 handwritten digits in videos. The objective is to build a system capable of automatically identifying and following handwritten digits within video frames and performing arithmetic operations based on these digits. This system has potential applications in educational tools, automated grading systems, and other areas.

The project, accessible in this GitHub repository [1], includes various features aimed at achieving this goal. It encompasses the detection and tracking of handwritten digits in video using YOLO, preprocessing and augmentation of training data, scripts for training custom YOLO models, and evaluation scripts to assess model performance. Additionally, it provides functionality to perform arithmetic operations on the detected digits.

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

The purpose of this document is to provide a step-by-step guide on how to obtain your dataset, successfully train your YOLO model, and finally, conduct tests to evaluate how well it detects handwritten digits in video.

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.

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 yolov8x.pt?

The model yolov8x.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.
- **x**: This letter stands for "extra-large" and signifies that the model is designed with a larger capacity compared to its smaller counterparts like yolov8s (small) or yolov8m (medium). In the context of YOLOv8, the x variant offers more parameters, which can lead to better performance in terms of 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 yolov8x is an "extra-large" model, it is designed to be more efficient and lightweight compared to previous models with similar capacities. In fact yolov8x.pt is an excellent choice, especially when computing resources are limited because 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.

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 "yolov8x.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:

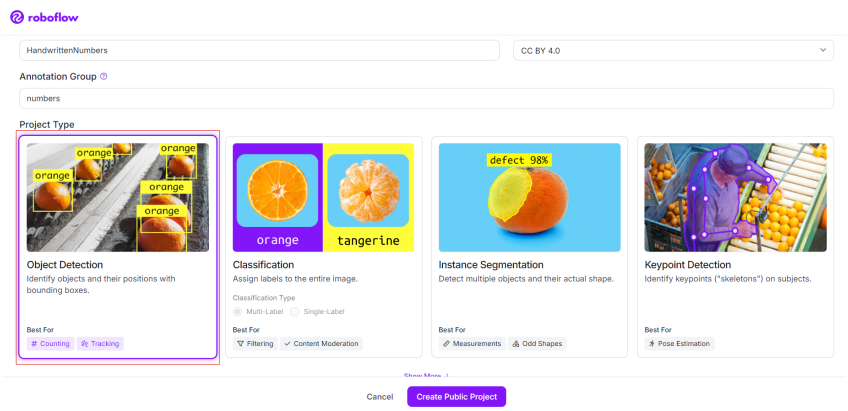

Figure 2.1: Create a project in Roboflow

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.

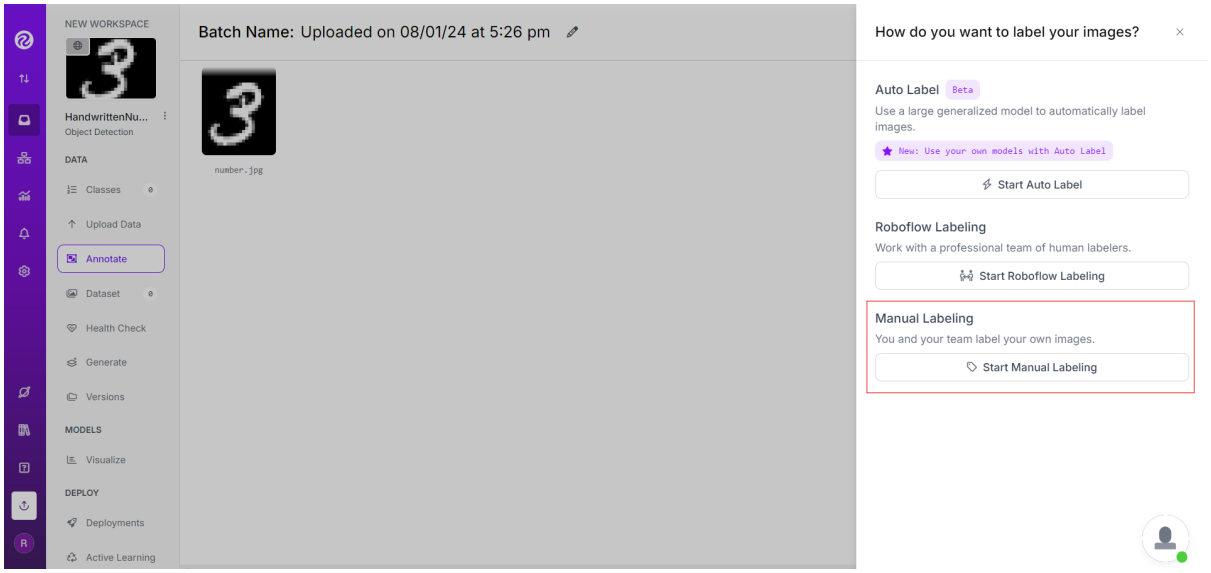

Figure 2.2: How to label the images?

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:

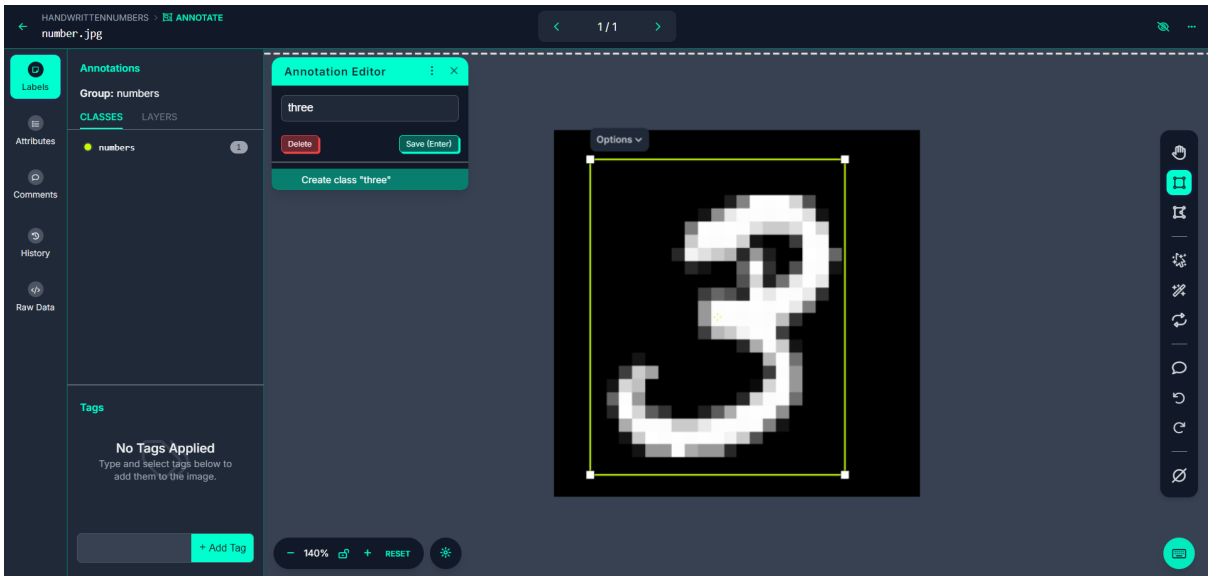

Figure 2.3: Label of a 3

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.

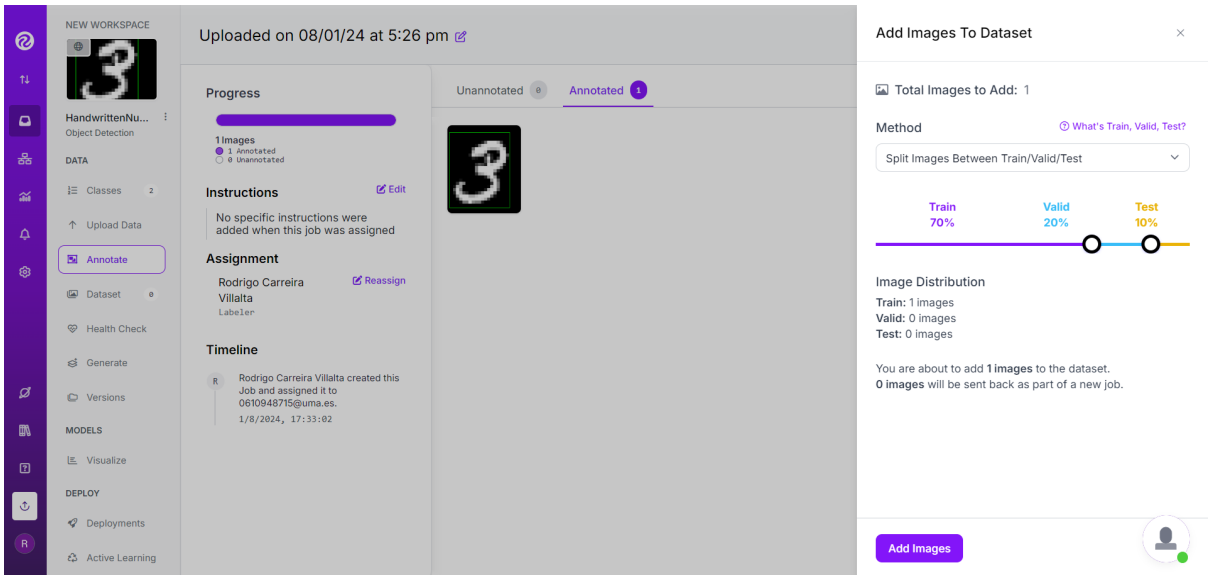

Figure 2.4: Percentages for train, validate and test

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:

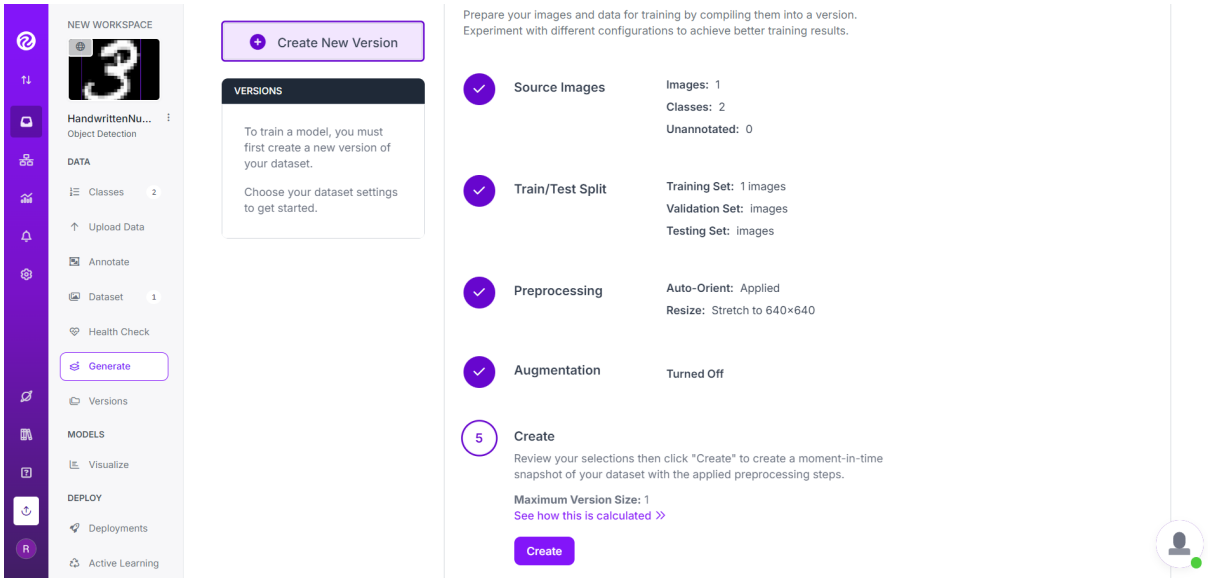

Figure 2.5: Obtaining Our YOLO-Formatted Dataset

2.2.3 Configuration files

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, since our goal is to learn from this project, we have a great resource available: the MNIST dataset. The MNIST institute provides a large dataset containing 60,000 training images of handwritten digits and arithmetic operators. Additionally, there are resources available where the MNIST dataset has already been labeled in YOLO format, saving us the effort of manual labeling.

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 MNIST dataset for training our YOLO model. First of all we start with the necessary imports and we save our path and the path where our dataset will be organized:

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

cd = os.getcwd()
base_path = 'MNISTyolov8_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 = 'MNISTyolov8.zip'
base_extract_path = 'MNISTyolov8'

# Unzip the file
with ZipFile(f"MNIST.v5i.yolov8/{zip_file_path}", 'r') as
   ↪ zip_ref:
    zip_ref.extractall(base_extract_path)

# Paths for extracted images and labels
images_path = os.path.join(base_extract_path,
   ↪ 'all_images')
labels_path = os.path.join(base_extract_path,
   ↪ 'all_labels')

# Gather image and label file names and sort them in
# ascending order (in this case string order)
image_files = sorted(os.listdir(images_path))
label_files = sorted(os.listdir(labels_path))
assert len(image_files) == len(label_files) == 7128
# (already known number of items in dataset)

# Shuffle and split data
data = list(zip(image_files, label_files))
random.shuffle(data)

train_data = data[:int(0.6 * len(data))]
val_data = data[int(0.6 * len(data)):int(0.9 * len(data))]
test_data = data[int(0.9 * len(data)):]

for split in ['train', 'val', 'test']:
    os.makedirs(os.path.join(base_path, split, 'images'),
       ↪ exist_ok=True)
    os.makedirs(os.path.join(base_path, split, 'labels'),
       ↪ exist_ok=True)

move_files(train_data, 'train', images_path, labels_path)
move_files(val_data, 'val', images_path, labels_path)
move_files(test_data, 'test', images_path, labels_path)

print("Data␣distribution␣completed␣successfully␣...")
```

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 handwritten digits 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

4. Image Tracking on Video

In this chapter, we will use the already trained model to recognize handwritten digits not only in images but also in videos. We will process the video information, perform the necessary arithmetic operations, and display the results to demonstrate that the model performs accurately in our tests.

5. Bibliography

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

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