CPSC 599 Project Milestone Report

Sam Hoang Hong (UCID: 30088823)
Tin Le (UCID: 30089727)
Nam Nguyen Vu (UCID: 30154892)

Introduction

In the modern era, Artificial Intelligence (AI) is constantly growing in importance. We now see AI exists in all aspects of life, from automation to robotics. In all of these applications, computer vision plays a crucial role. In the context of this course, we learn how to create computer Vision models that could open up a multitude of opportunities for innovation. **For our final CPSC599 project, we aim to create a computer vision program that can detect cars, pedestrians, and traffic lights with high accuracy and performance.**

The problem of detecting cars and other street objects is a core part of self-driving vehicle technology. Autonomous vehicles must be able to reliably detect objects on the road, in order to safely navigate and maneuver. Although this field has already been explored in the modern era, our creation of such a model could be highly impactful, at least within our community. Independent teams from universities or start-ups could develop their computer vision models—or even self-driving platforms—using our model as a basis. This kind of technology normally belongs to large companies, and those companies rarely give independent teams access. Additionally, in a world where self-driving vehicles are becoming more and more prevalent, it is important to understand the mechanics behind them. This project can give us valuable insight into the modus operandi of this contemporary technology.

In the past, object detection (particularly on street images) was considered to be fictional, due to the low accuracy and the tremendous computational requirements. However, with the existence of new computer vision algorithms, higher accuracy could be achieved from a fraction of computational power. Because of that, it is now possible to create object detection models from personal computers, which have similar accuracy rates as previous models created using giant and expensive systems.

During our preliminary research, we identified YOLO (You Only Look Once) as one of the most robust pre-trained models for object detection tasks. “Object detection using YOLO: challenges, architectural successors, datasets and applications”[1] describes the architectural design of prior versions of YOLO (up to YOLOv4). It explains the loss function relating to the boxes on the detected objects. We also found several other tutorial articles[2] [3] which described how to implement car detection using YOLO. With these approaches, the first step was to collect data, by snapshotting images of dashcam footage. Then, the images were labeled with bounding boxes, with different classes representing different object types (ex. Car, pedestrian, traffic sign). The input images are afterward inputted into the YOLO model, which then outputs bounding boxes along with confidence values for the objects it detects.

---

[1] Diwan, T., Anirudh, G. & Tembhurne, J.V. Object detection using YOLO: challenges, architectural successors, datasets and applications. *Multimed Tools Appl* **82**, 9243–9275 (2023). https://doi.org/10.1007/s11042-022-13644-y

[2] https://towardsdatascience.com/guide-to-car-detection-using-yolo-48caac8e4ded

[3] https://medium.com/analytics-vidhya/object-detection-using-yolo-and-car-detection-implementation-1ec79e882875

During our demonstration to Dr. Malaki for the milestone, one of the issues he pointed out with the “out of the box” pre-trained YOLO model is glitching. Because the model only makes predictions for the current frame and does not consider the previous or next frames, the bounding boxes may occasionally glitch and pop in and out. For this CPSC599 project, our team’s next objective is to fix this issue. One of our plans is to create a new model that takes in the output of the original model as input, and then eliminates the bounding box glitches. By allowing our model to analyze the video and consider previous and next frames, the bounding box predictions will be smoother and less glitchy.

Goal

Our group’s first goal is to develop a model that can detect cars, pedestrians, and traffic lights in dash cam footage. This goal has already been accomplished as of the project milestone. Our group’s second goal, as proposed by Dr. Malaki, is to smoothen the performance by implementing temporal deep learning to prevent glitches. Our final model will be able to predict where an on-screen object will move within the next few frames. These predictions can be combined across several frames, which will prevent the glitchy effect that tends to occur with the “out of the box” YOLO model.

Here is a summary of our expected outcomes:

1. Create a program that can detect 3 types of objects: cars, pedestrians, and traffic lights, with high accuracy (aiming to 85% and up)
2. Optimize the program using temporal deep learning to prevent bounding box glitches

Data

For this project, we used one of our team members’ dashcam to collect data. We then downloaded dashcam footage across several days. Each team member then extracted screenshots from the videos to prepare for labeling. We extracted about 1 frame per second, however this number varied based on how much the objects moved.

Example of a screenshot image

After extracting the frames from a video, we would then label the objects, using a tool called LabelMe. We drew bounding boxes around each car, pedestrian, and traffic light. Cars, people, and traffic lights were assigned the label 0, 1, and 2 respectively.

Example of a labeled image

For each labeled image, LabelMe generates a JSON file containing the coordinates of the bounded boxes, in addition to their labels (object type).


```
shapes:
  0:
      label:        "0"
    points:
      0:
          0:        391.58653846153845
          1:        630.2644230769231
      1:
          0:        526.2019230769231
          1:        533.6298076923077
    group_id:       null
    description:    ""
    shape_type:     "rectangle"
    flags:          {}
```


Excerpt of a JSON file from LabelMe

However, the coordinates in the JSON represent the x-min/max and y-min/max of the boxes, which does not follow the YOLO standard. Our methodology to convert the JSON files to a suitable format is explained in the next section.

Methodology

Preprocessing and Data Augmentation Pipeline

As mentioned previously, the JSON-formatted coordinates from LabelMe are not suitable for YOLO. Thus, we developed a python script to convert the coordinates into the appropriate format. For each bounding box, we calculate the x_center, y_center, width, and height, following the YOLO requirements[4]. These coordinates are stored in a TXT file.

```
0 0.140299 0.567708 0.115885 0.123843
0 0.541016 0.537037 0.089193 0.101852
0 0.783986 0.494316 0.099222 0.094351
0 0.441890 0.506545 0.018270 0.026178
1 0.296360 0.526351 0.007722 0.042797
```

Example of YOLO coordinates in a .txt file

We repeated this process to manually generate about 300 samples. We originally planned to generate more samples, but Dr. Malaki suggested changing our focus to optimizing the frames instead, because YOLO is a pre-trained model and therefore does not need us to provide as many samples. Additionally, we picked our images such that there was a reasonable amount of pedestrians and traffic lights captured, since these were rarer than cars in the dashcam footage.

In YOLO, each image is divided into a grid, and each grid cell is responsible for predicting bounding boxes and class probabilities. The annotation format typically includes information about the objects present in an image, their bounding box coordinates, and class labels. The YOLO format for annotation usually follows a specific structure. Here's an example to illustrate the format:

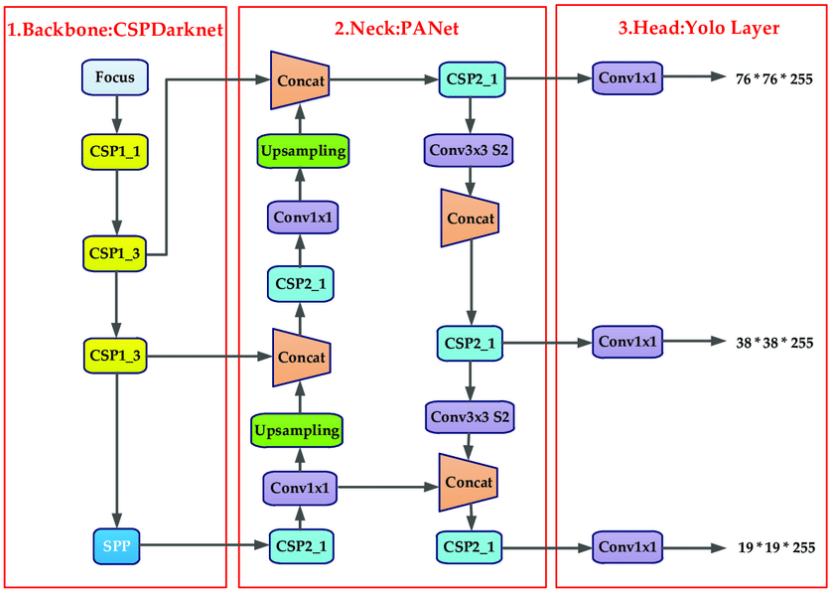
<object-class> <x> <y> <width> <height>

- <object-class>: The class label of the object present in the bounding box.
- <x>: The x-coordinate of the center of the bounding box, normalized to the image width.
- <y>: The y-coordinate of the center of the bounding box, normalized to the image height.
- <width>: The width of the bounding box, normalized to the image width.
- <height>: The height of the bounding box, normalized to the image height

[4] https://albumentations.ai/docs/getting_started/bounding_boxes_augmentation/

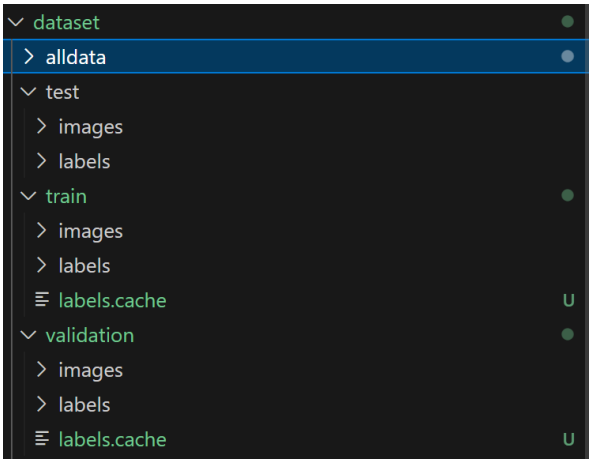
Model Architecture

For our project, we used YOLOv5, Python 3.8, and PyTorch.
Here is the architecture of YOLOv5.



Source:
https://www.researchgate.net/figure/The-network-architecture-of-YOLOv5-1-Backbone-CSPDarknet-for-feature-extraction-2_fig1_358553872

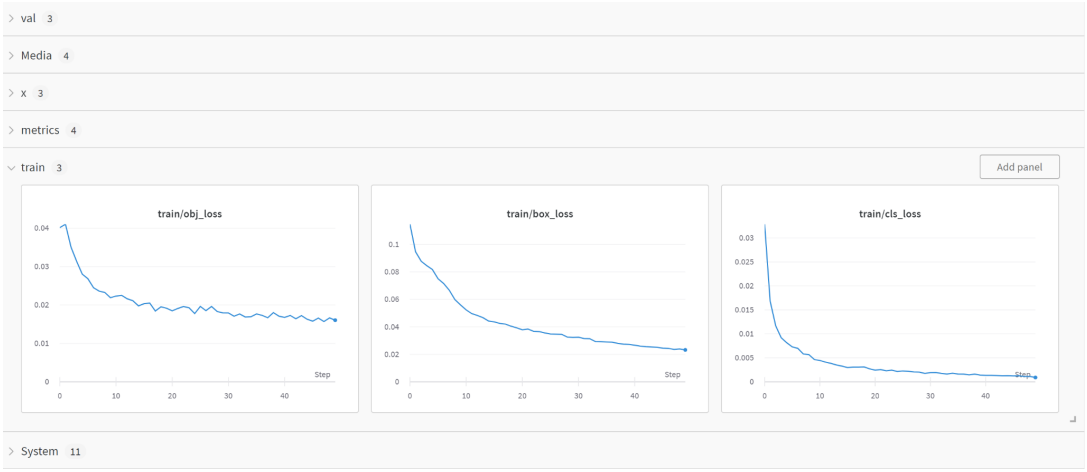
Here is an overview of our folder structure:



- **dataset**: The root folder containing the entire dataset.
    - **train**: Data for training a machine learning model.
        - images: Training images for the model.
        - labels: Annotations or labels for the training images.
    - **validation**: Data for validating the trained model.
        - images: Images used to validate the model.
        - labels: Labels or annotations for the validation images.

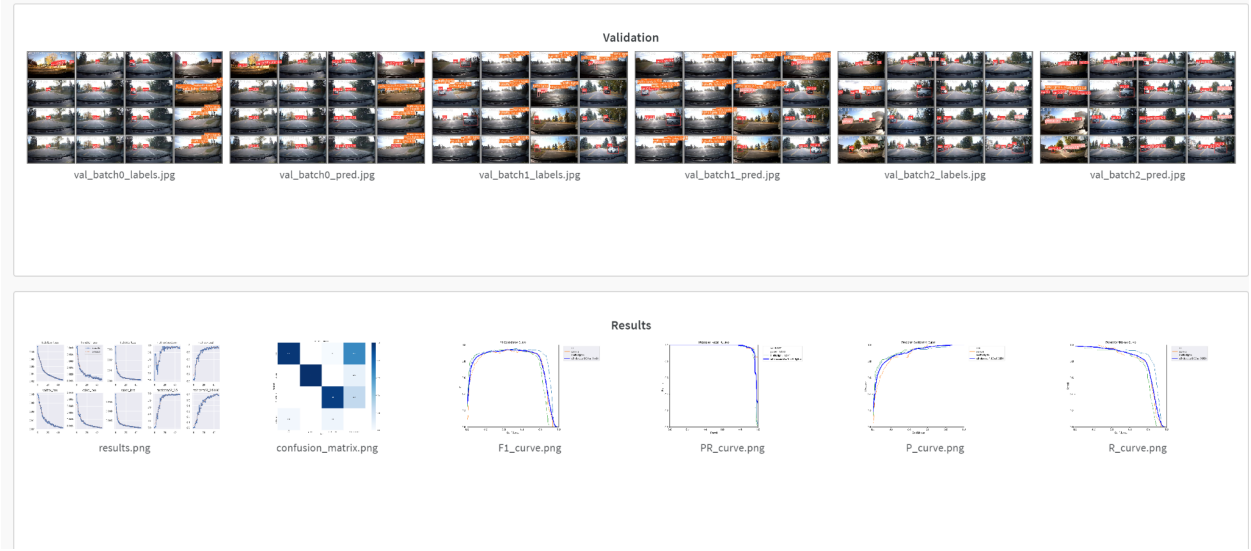
- **test**: Data for testing the trained model.
    - images: Images used to assess the model's performance.
    - labels: Labels or annotations for the test images

While training the model, we used WandB[5] to track and visualize the training process in real-time.



View of WandB

Using the tool, we were able to track some of the key YOLO metrics, including Intersection over Union (IoU), Precision and Recall, Average Precision (AP), Mean Average Precision (mAP), Confusion Matrix, and Frames Per Second (FPS)[6].



Some of the statistics as seen on WandB

[5] https://wandb.ai/site
[6] https://docs.ultralytics.com/guides/yolo-performance-metrics/#object-detection-metrics

Postprocessing

For testing, we developed a python script that can test the model on an image or video.

```
PS C:\Users\hoang\OneDrive\Desktop\University\Fourth year\CPSC599\project> python detect.py --weights runs/train/exp3/weights/best.pt --source "C:\Users\hoang\OneDrive\Desk
top\University\Fourth year\CPSC599\project\yolov5\dataset\test\images\VID_125 - frame at 2m14s.jpg"
```

Executing the test script

The result video contains bounding boxes around the detected objects, plus their labels and probability values.

Additionally, the script also provides a summary of the testing detection:

```
YOLOv5  2023-11-7 Python-3.10.9 torch-2.1.0+cpu CPU

Fusing layers...
Model summary: 157 layers, 7018216 parameters, 0 gradients, 15.8 GFLOPs
image 1/1 C:\Users\hoang\OneDrive\Desktop\University\Fourth year\CPSC599\project\yolov5\dataset\test\images\VID_125 - frame at 2m14s.jpg: 384x640 7
cars, 1 person, 237.6ms
Speed: 3.0ms pre-process, 237.6ms inference, 20.8ms NMS per image at shape (1, 3, 640, 640)
Results saved to runs\detect\exp5
```

Our final phase of the project and postprocessing however, will be a complex task. Under the advice of Dr. Malaki, we will implement another model that will take the output of the YOLO model and smoothen the bounding box glitching, which is a common issue. More details can be found in the next section.

Challenges and Solutions

It was relatively straightforward to accomplish our first goal and implement the YOLO model that detects the three types of objects. Our next major challenge will be to find a way to prevent the glitching of the bounding boxes. As mentioned previously, Dr. Malaki's suggestion is to create another model that takes the output of the first model as input. This second model will use temporal deep learning and object tracking to smoothen the boxes.

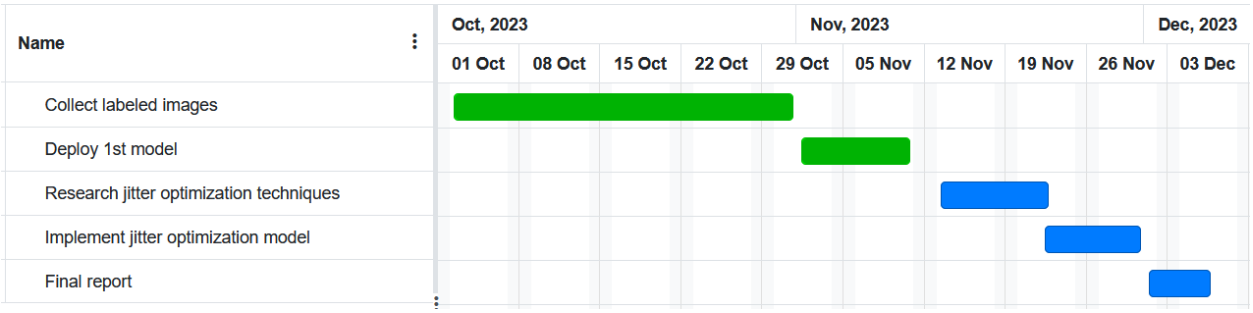
During our research, we found several papers[7] [8] detailing the glitching problem, often referred to as bounding box *jitter*. Some solutions found during our research include:

- Implementing a Long Short-Term Memory (LSTM) network, which is a type of Recurrent Neural Network (RNN). This can capture dependencies between bounding box positions over time, which can help reduce the jitter.
- Implementing a Kalman Filter, which can estimate an object's state over time. These can also operate in real-time, which might be crucial for future applications.

As of this paper, we are still working on deciding how to implement this. This part of the project will be finished and tested by the final due date.

Updated Timeline

- Phase 1: By Oct 31, collect ~300 images with objects labeled (Done)
- Phase 2: By Nov 10, deploy the first model, successfully detect cars, pedestrians, and traffic lights (Done)
- Phase 3: By Nov 22, research and decide the model to use to reduce bounding box jitter
- Phase 4: By Nov 30, implement the optimization model
- Phase 5: By Dec 6, finalize the final report



[7] Fung, Donovan, et al. *Recurrent CNNs for Bounding Box Stability in Object Detection*, https://cs230.stanford.edu/projects_winter_2019/reports/15812427.pdf.
[8] Zhang, Hong, and Naiyan Wang. *On The Stability of Video Detection and Tracking*, https://arxiv.org/pdf/1611.06467.pdf.