Deep Vision for Detecting Road Objects

CPSC599 Final Project Report

Hoang Hong
University of Calgary
hoang.hong@ucalgary.ca

Nam Nguyen Vu
University of Calgary
namnguyen.vu1@ucalgary.ca

Tin Le
University of Calgary
tin.le@ucalgary.ca

ABSTRACT

The problem of detecting cars and other street objects is a core part of self-driving vehicle technology. This project focuses on developing a computer vision program using YOLOv5 for detecting cars, pedestrians, and traffic lights in dash cam footage. The initiative addresses common challenges in object detection, particularly bounding box glitches, through the utilization of two pre-trained YOLOv5 models (YOLOv5l and YOLOv5x). The dataset was collected from dashcam footage in Calgary. Training involved 50 epochs and included techniques such as data augmentation and early stopping to prevent overfitting. The evaluation, based on mean Average Precision (mAP) and various other metrics, demonstrated improvements in accuracy. Despite the achievements, there remains room for improvement, emphasizing the need for future work involving glitch prevention, temporal deep learning, and dataset refinement. Our team's findings underscore the power of model ensembling and the prioritization of accurate data. Overall, this work contributes to advancing computer vision applications, with implications for developing more robust and reliable systems.

INTRODUCTION

Modern day autonomous vehicles must be able to reliably detect objects on the road, in order to safely navigate and maneuver. The objective for this project is to develop a computer vision program that can detect several different types of road objects with high accuracy and performance.

Our group's first goal is to develop a model that can detect cars, pedestrians, and traffic lights in dash cam footage with at least 85% accuracy. Our second goal, as proposed by Dr. Malaki, is to augment and optimize the program to prevent bounding box glitches, a common issue with models detecting objects in videos.

In the past, object detection (particularly on street images) was considered to be fictional, due to the low accuracy and tremendous computational requirements. However, with the existence of new computer vision algorithms, higher accuracy could be achieved from a fraction of computational power. With these new developments, self-driving vehicles have become more prominent. Computer vision is one of the most important tools behind self-driving technology, and in this modern era, it is important to understand the mechanics behind how the technology works.

Our preliminary research 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. We also found several other tutorial articles[2][3] which described how to implement car detection using YOLO.

MODEL ARCHITECTURE

You Only Look Once (YOLO) is a popular object detection model used across many different applications. Our team chose to use YOLOv5, a prevalent version of the model built on the PyTorch framework, due to the ample documentation available online.

A common issue with video object detection using "out of the box" pre-trained YOLO models 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. One of the methods we utilized to solve this problem was to utilize two pre-trained models, YOLOv5l and YOLOv5x, to improve our predictions' accuracy. We also implemented several augmentations and optimizations to the models, which will be detailed in the training procedure section of this paper.

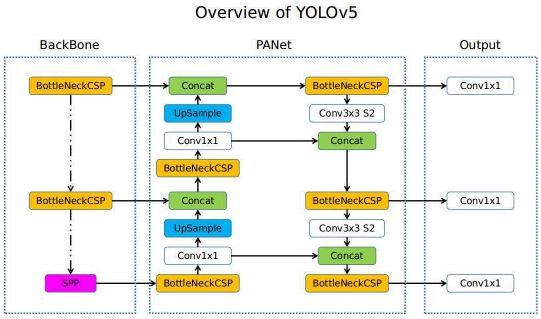

Figure 1: **Overview of the YOLOv5 architecture [4]**

DATASET

For this project, we used a 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, all of which were 1920x1080px. We extracted about 1 frame per second, however this number varied based on how much the objects moved.

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.

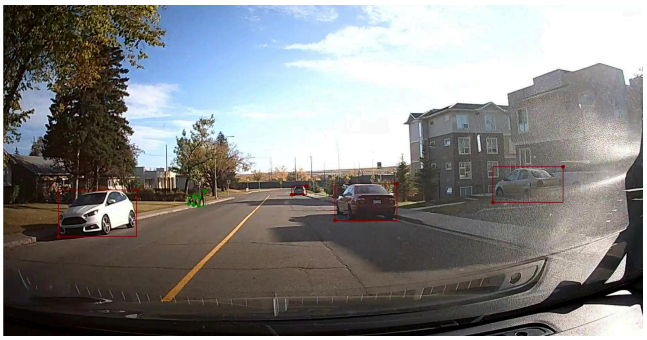

Figure 2: **Example of a labeled image**

For each labeled image, LabelMe generates a JSON file containing the coordinates of the bounded boxes, in addition to their labels (object type). 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. Thus, we created 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. These coordinates are stored in a TXT file.

This process was repeated to manually generate around 300 samples, which were then used for training. In order to achieve diversity, we tried to select more images with pedestrians and traffic lights. However, since these objects are less common than cars on the roads, this is a potential limitation of our dataset that might reduce our model's ability to detect people and traffic lights. Additionally, there were naturally issues with labeling when there were too many cars in a single image, so sometimes not every object in an image was labeled.

TRAINING PROCEDURE

The summary of our training process was as follows:

1. Mount Google Drive to Google Colab to save the best model later on
2. Extract the dataset
3. Train 2 Ultralytics models: YOLOv5l and YOLOv5x. Using 2 pre-trained models enhanced the performance of our model
4. Perform an object detection test. comparing the old trial model and the new model
5. Test using a short 17-second video.

Our training procedure involved a variety of software resources. We used Visual Studio Code data collection and preparation, but switched to Google Colab for training for the improved speed when training the model. In addition, we used wandb to visualize the process metrics after each running session.

The hardware used was a powerful Tesla T4 GPU on Google Colab with CUDA support, which helped a lot during the training process. For instance, training the first model with a trial dataset using M1 processor on a Macbook took us 7.37 hours and 8.77 hours for YOLOv5m and YOLOv5l respectively. However, by leveraging the power of the Tesla T4 GPU, we are able to achieve those training sessions in just 25 and 30 minutes. The frameworks used include YOLOv5 as the backbone, as well as PyTorch and cv2.

Our training process involved 50 epochs. Our optimization algorithm uses several parameters, including:

- **NBS**: The nominal batch size, represents the desired batch size for training.
- **accumulate**: Effective batch size for optimization, determined based on ratio of NBS to the actual batch size used in training. Used for gradient accumulation.
- **hyp**: This is the dictionary containing hyperparameters, including learning rate (lr0), momentum (momentum), and weight decay (weight_decay)
- **smart_optimizer**: This function initializes an optimizer with the specified hyperparameters. The optimizer type (opt.optimizer) and other hyperparameters are determined by the training configuration.

Our learning rate was determined by several factors:

- **opt.cos_lr**: Boolean flag indicating whether to use a cosine learning rate schedule
- **one_cycle**: This function generates a one-cycle learning rate schedule
- **lf**: Lambda function representing the learning rate schedule.

- **lr_scheduler.LambdaLR**: Initializes a scheduler with the lf function. This scheduler adjusts the learning rate during the training epochs.

Instead of relying on the default hyperparameters which are not optimized for our specific case, we generate them as part of our process. This involves three steps:

1. Initial training: The initial training with default settings provides a baseline performance to compare with. Additionally, if we customize the parameters from the start, there is a risk of overfitting hyperparameters to the dataset. Therefore, we run a few epochs of training with the default hyperparameters.
2. Hyperparameter evolution: This evolution helps the model find the best hyperparameters for the use case.
3. Training with the newly evolved hyperparameters

As mentioned previously, we used two pre-trained models, YOLOv5l and YOLOv5x, to improve our accuracy. This improved the bounding box predictions and reduced the glitching bounding box issue.

We mainly used 2 techniques to prevent overfitting. The first is data augmentation: some images in the dataset were transformed and rotated, along with their labels. The second is early stopping; we implemented code to halt training when there is no longer improvement. These techniques also seemed to help fix the glitching problem.

Using wandb, we were able to analyze several performance metrics while training, which are shown below:

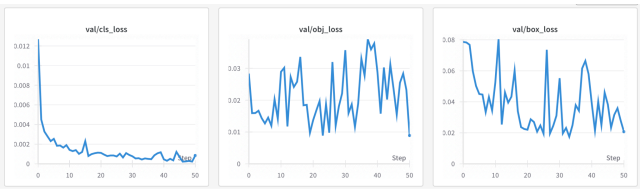

Figure 3: **Loss curve graphs**

EVALUATION PROCEDURE

We use several metrics in order to save the best model and to present the results. The primary metric we use to judge accuracy is mean Average Precision (mAP). This is a popular metric in object detection tasks mainly for measuring the quality of bounding boxes predictions.

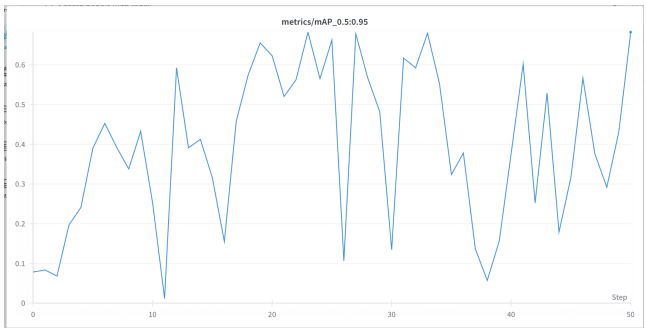

Figure 4: **mAP graphs**

We analyzed the quantitative results using several standard metrics such as precision, recall, and F1-score. Average Precision (AP) and mAP were also used to evaluate the model's performance across different classes and thresholds. This is used to visualize the performance of the model using 4 metrics: True Positive, True Negative, False Positive and False Negative. This metric is applied for each class. Lastly, we plot the precision-recall curve to analyze the trade off between precision and recall for different thresholds.

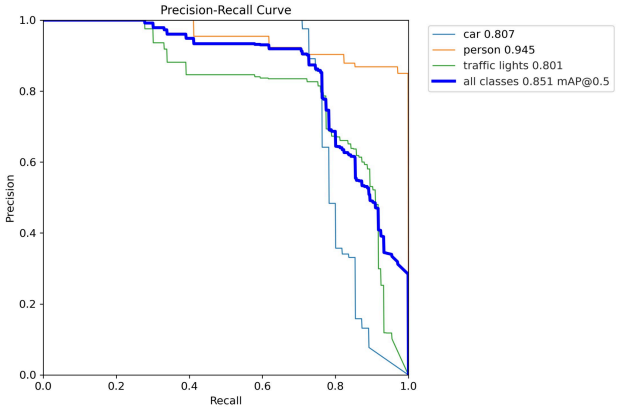

Figure 5: **Precision-recall curve**

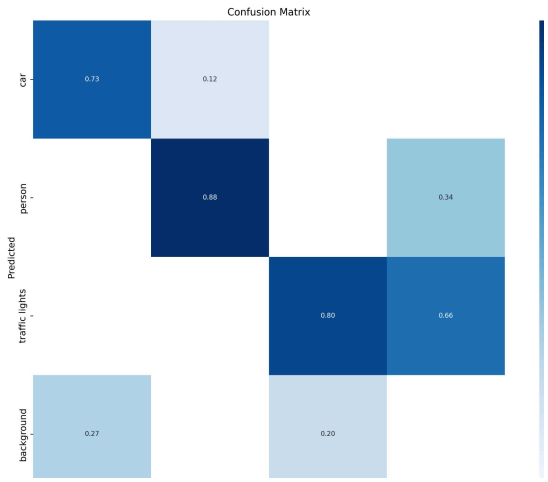

Figure 6: **Confusion matrix**

For the qualitative assessment, we assess the accuracy of the bounding boxes predicted by the model. This also involves visually inspecting the bounding boxes to see how well they align with the ground truth boxes. Additionally, we tested our trial model and final model on videos, and visually verified that the final model reduced bounding box glitching.

Figure 7: **Screenshot of trial model's predictions**

Figure 8: **Screenshot of final model's predictions**

Despite the rigorous training and evaluation procedures, our model still has some limitations and there are rooms for potential improvements and future work. Although the glitching problem was reduced, we could not rectify it completely as we did not have time to implement any temporal deep learning models. Finally, the lack of a powerful GPU for training can also limit the ability to apply more complex improvements which could directly affect the performance of the model. Some possible solutions include obtaining a more balanced and accurate dataset, upgrading the hardware, or by updating the output generating process to better create a model which is more robust and reliable.

DISCUSSION SECTION

The result of our fine-tuning process of the YOLOv5 model for car detection has significant implications in the context of computer vision. The successful detection of car and on-the-road objects could play a crucial role in the development of softwares for self-driving cars and collision-avoiding systems. In addition, this project could be further developed to be used for traffic control. This could help monitor and manage the traffic flow more effectively to save precious time for many traffic participants. In addition, its application could also enhance the management of car parking spaces and parking lots by monitoring the number of vehicles in the lots and the inflow and outflow of those vehicles. If better developed, it can enhance security in car parking spaces and parking lots by detecting unauthorized vehicles.

A major lesson our project highlighted is that having less but accurate data is more important than having ample data with lots of noise. Furthermore, we also discovered that using two models simultaneously to detect objects will generally result in better accuracy and lower loss compared to using only one. This has demonstrated the benefits of model ensembling in improving prediction performance, and our model has improved upon some previous studies and work. For example, compared to paper [5], we were able to implement hyperparameter tuning.

Despite the successes, there are several improvements that could be made. One is to save the output .txt files and then go through a preprocessing layer before generating the result. Although the glitching has been significantly reduced by using 2 models, this could further prevent the problem. Additionally, future work could implement temporal deep learning to consider object positions between frames. These upgrades could highly increase the reliability of our model.

CONCLUSION

To summarize, our project leveraged YOLOv5 to develop a computer vision program for car detection in dash cam footage. Through extensive training and evaluation, we

achieved substantial improvements in accuracy, as shown by the reduction in glitches and enhanced mAP. The potential applications of our findings extend to the real-world fields of self-driving technology, traffic control, and parking management. Our project highlights the significance of model ensembling and the importance of accurate data over quantity. Moving forward, further advancements could involve pre-processing layers for glitch prevention, implementation of temporal deep learning for enhanced object tracking, and considerations for a more balanced dataset. Despite current limitations, our work contributes to the ongoing development of computer vision applications, paving the way for more reliable and effective systems in the future.

SUPPLEMENTAL MATERIAL: **REFERENCES**

[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] Tan, B. (2020, August 6). *Guide to car detection using Yolo*. Medium. https://towardsdatascience.com/guide-to-car-detection-using-yolo-48caac8e4ded

[3] Bisht, B. S. (2020, November 19). *Object detection using Yolo and car detection implementation*. Medium. https://medium.com/analytics-vidhya/object-detection-using-yolo-and-car-detection-implementation-1ec79e882875

[4] *Overview of model structure about Yolov5 · issue #280 · ultralytics/yolov5*. GitHub. (n.d.). https://github.com/ultralytics/yolov5/issues/280

[5] Fung, Donovan, et al. Recurrent CNNs for Bounding Box Stability in Object Detection, https://cs230.stanford.edu/projects_winter_2019/reports/15812427.pdf.