Mini Project of EE5104

Deployment of the YOLOv5 on Jetson-nano

Yifan Chen
2007060205
School of Engineering
University of Galway
Y.Chen25@universityofgalway.ie

Abstract—

The research of this mini project optimizes and deploys the target detection task using the YOLOv5 model and achieves significant results. This mini project is based on the Ultralytics YOLOv5 framework, and the YOLOv5n model is trained on the Kaggle platform using the bdd10k dataset, restricting the detection labels to three categories: 'person', 'car' and 'bike'. The model was trained on the Kaggle platform using the bdd10k dataset, restricting the detection labels to 'person', 'car', and 'bike', and generating the model in .pt format. Subsequently, the model was deployed on a Jetson Nano device for real-time object classification using a CSI camera. Due to the performance limitations of Jetson Nano, the research also used TensorRT for model optimization to improve inference speed. The innovation of this work is to take the lightweight YOLOv5 and optimize it with label customization and TensorRT for better operation on the jetson nano, which significantly improves the efficiency and applicability of real-time target detection, and provides a new solution for intelligent surveillance on edge devices, a series of issues that arose during the model optimization process were diagnosed and resolved through the model visualization approach.

Keywords—YOLOv5, Jetson Nano, bdd10k, TensorRT, FPS improvement, Compatibility fix, Model visualization

I. INTRODUCTION

Research Problem & Challenge

1. **Data Format Incompatibility:**
   The YOLOv5 model requires a specific input format, a label file (TXT) in YOLO format. Each TXT file corresponds to an image containing normalized category IDs and bounding box coordinates (x_center, y_center, width, height). In contrast, the COCO format (JSON) provided by the bdd10k dataset contains image meta-information and bounding boxes in absolute coordinates (x, y, width, height), which need to be converted to adapt to the YOLOv5 training process. In addition, COCO format files contain a large amount of metadata (e.g., image IDs, category meta-information, etc.), which may increase the workload during model training. [1]
2. **Category Management:**
   Initially bdd10k.yaml contains 10 categories (person, rider, car, etc.), to improve the performance of the model on jetson nano I used label filtering method and bdd10k_filtered. yaml retains only 3 categories after filtering through labels (person, car, bike)
3. **Hardware Limitations:**
   The Jetson Nano has limited hardware resources (4GB RAM, 128-core Maxwell GPU), and if the frame rate of the camera output screen is set too high, it may cause Argus communication to fail [1-2], the GStreamer pipeline to crash, and GPU or memory resources to run out.
4. **Inference Speed:**
   The jetson nano for this project uses a special system image file with memory and other limitations. On a Jetson device (e.g., Jetson Nano), the raw PyTorch model of YOLOv5 may have a low frame rate (e.g., 5-10 FPS) due to limited computational resources. [1-2]
5. **Model Conversion Issues:**
   When converting a model file in .pt format to .onnx format, if the best.pt file is trained on kaggle and contains paths in PosixPath format, the conversion command will not be successful when run by jetson. [3]

Importance of These Problems

1. **Data Format Incompatibility:**
   If the data format is not converted, YOLOv5 is unable to read the bdd10k dataset, resulting in the training not being able to proceed and the project directly stalling[3].In addition, if the coordinates are calculated incorrectly, it will directly lead to the training data error, the model may learn the wrong pattern, the performance of target detection and classification is directly degraded, and in the actual demonstration, there will be a missed detection or false detection.
2. **Category Management:**
   Without label filtering, the model needs to process all 10 categories, the inference is too computationally intensive, exceeding the computational capacity of the jetson nano, and the frame rate may be lower than 10 FPS, which cannot meet the real-time detection requirements, in practice, autonomous driving scenarios require at least 15-30 FPS. If the categories are not properly managed, it will lead to the model overfitting or underfitting certain categories phenomenon, reducing the detection accuracy on key categories.
3. **Hardware Limitations:**
   As I mentioned before, the system memory of the jetson nano used in this project is limited, if the frame rate and

resolution are not optimized, the Jetson Nano may crash frequently due to resource depletion, the detection task is interrupted, continuous real-time detection and classification cannot be achieved, and the high resource usage may lead to system overheating or performance degradation, shortening the service life of the Jetson Nano and increasing the hardware maintenance costs.

4. **Inference Speed:**
   High-intensity computational overheads can lead to Jetson Nano resource exhaustion, system instability, or even crashes, affecting the continuity of target detection and classification tasks [3-4]. Inefficient inference performance may limit the application scenarios of the model, such as the inability to support highly dynamic scenarios (high-speed moving objects), which directly leads to a decrease in the utility and competitiveness of the system.
5. **Model Conversion Issues:**
   If the PosixPath problem is not solved, the model cannot be converted to the ONNX format, which in turn prevents the generation of the TensorRT engine, resulting in inference optimization not taking place and the frame rate remaining at a low level (5-10 FPS)

Proposed Solution or Approach

1. **Data Conversion**
   To convert the COCO format labels (JSON) of the bdd10k dataset to YOLO format labels (TXT), you can generate a TXT file by calculating the normalized coordinates of the YOLO format based on the absolute coordinates of the data in COCO format.
2. **Label Filtering**
   Dynamically generating embedded test set information through code automation can reduce human errors and improve efficiency. In particular, after label filtering, the number and names of target categories change, and the dynamic update method avoids the hassle of manual modification as well as unexpected problems that may occur.
3. **Resource Optimization**
   To reduce the resource load, you can modify the pipeline in the python code to force a lower resolution and frame rate. Before modifying the Python code, you can test the GStreamer pipeline individually to make sure it works properly. [4]
4. **Model Conversion Fix:**
   To solve this problem, the script convert.py needs to be written to repair the best.pt file, convert PosixPath to a string, and continue the conversion using the repaired .pt model file.

Contributions:

This research paper summarizes the frequently faced problems during the deployment of the Yolo model on the jetson nano, suggests possible solutions by reviewing the literature, and then practices and solves this set of problems to achieve a ideal runtime performance.

Structure:

1. Section1: Analyzing problems and proposing solutions.
2. Section2: Review of relevant literature and summary of findings.
3. Section3: Discuss the methods I have used to successfully solve these problems.
4. Section4: Describe the preparations prior to the formal experiment
5. Section5: Analysis and summary of experimental results
6. Section6: Conclusions and possible ways for optimization.

II. Related Work

To accomplish the goal of model optimization, by reviewing the literature in Google Scholar I found that TensorRT is an ideal solution to increase inference speed, especially in real-time detection tasks. It supports mixed-precision inference (FP32, FP16, INT8) and can leverage Jetson's Tensor Core to accelerate computation and reduce memory bandwidth and computational overhead [5]. TensorRT leverages the CUDA and Tensor Core of NVIDIA GPUs, optimized for the hardware characteristics of the Jetson device. TensorRT's hardware acceleration maximizes the computational power of Jetson compared to PyTorch's general-purpose inference. [5]

Taking the **TensorRT-Based Framework and Optimisation Methodology for Deep Learning Inference on Jetson Boards** published by Eunjin Jeong, Jangryul Kim, and Soonhoi Ha in 2022 as a reference, they proposed to present a TensorRT-based framework called JEDI (Jetson-aware Embedded Deep Learning Inference acceleration), which aims to improve inference by optimizing through network-level optimization (network-level optimization) to Increase inference speed, This document provides background on optimizing deep learning inference on the NVIDIA Jetson platform, research progress using TensorRT and heterogeneous accelerators on the Jetson AGX Xavier. [5]

However, this literature focuses on the hardware Jetson AGX Xavier, which has a DLA, whereas my research addresses the Jetson Nano, which has no DLA and only a GPU, even though both are capable of using the TensorRT technique, the specific optimization methods may be different [5]. Depending on my project environment, my plan is to firstly convert the trained model files in .pt format into onnx files, then enable the FP16 hybrid accuracy suitable for Jetson Nano, which reduces memory usage and improves performance, and finally write the YOLOv5 inference code for experimentation.

III. METHODOLOGY

1. Data Conversion

To solve the first problem I mentioned earlier, I installed the pycocotools library, which loads JSON files in COCO format, extracts image meta-information and annotations and computes normalized coordinates then write category IDs and normalized coordinates to a TXT file in YOLO format. This approach extracts only the necessary annotation information such as category IDs and bounding boxes,

minimizing the computational workload during model training.

```
def convert_coco_to_yolo(coco_json_path, image_dir, output_dir):
    coco = COCO(coco_json_path)
    categories = coco.loadCats(coco.getCatIds())
    category_map = {cat['id']: idx for idx, cat in enumerate(categories)}
    image_ids = coco.getImgIds()

    for img_id in image_ids:
        img_info = coco.loadImgs(img_id)[0]
        img_width = img_info['width']
        img_height = img_info['height']
        img_filename = img_info['file_name']
        ann_ids = coco.getAnnIds(imgIds=img_id)
        annotations = coco.loadAnns(ann_ids)
        yolo_labels = []
        for ann in annotations:
            bbox = ann['bbox']
            x_center = (bbox[0] + bbox[2] / 2) / img_width
            y_center = (bbox[1] + bbox[3] / 2) / img_height
            width = bbox[2] / img_width
            height = bbox[3] / img_height
            class_id = category_map[ann['category_id']]
            yolo_labels.append(f"{class_id} {x_center} {y_center} {width} {height}")
        txt_filename = Path(img_filename).stem + ".txt"
        txt_path = os.path.join(output_dir, txt_filename)
        with open(txt_path, 'w') as f:
            f.write("\n".join(yolo_labels))
```

2. Label Filtering

After several controlled tests, I found that although implementing label filtering can effectively improve the inference speed, if it is implemented before the model is trained, it does not give the desired recognition results in the final test on jetson. Therefore, I transferred the label filtering technique to the inference code.

```
# Define allowed labels
allowed_labels = ['person', 'car', 'bike']
```

As shown in the figure, the code first defines a specific list of labels, only three categories, person, car and bike, will be displayed while the others will be ignored.

```
# Draw detection boxes
for det in pred:
    if len(det):
        det[:, :4] = scale_boxes(img.shape[2:], det[:, :4], img0.shape).round()
        for *xyxy, conf, cls in det:
            label_name = names[int(cls)]
            # Only display allowed labels: person, car, bike
            if label_name in allowed_labels:
                label = f'{label_name} {conf:.2f}'
                # Select color based on class index (cycling through Red, Green, Blue)
                color = fixed_colors[int(cls) % len(fixed_colors)]  # Use modulo to cycle colors
                cv2.rectangle(img0, (int(xyxy[0]), int(xyxy[1])), (int(xyxy[2]), int(xyxy[3])), color, 2)
                cv2.putText(img0, label, (int(xyxy[0]), int(xyxy[1]) - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
```

During inference, the code is able to check all the detections of the model and determine if they belong to the allowed categories by passing if label_name in allowed_labels, which on a resource-constrained device such as the Jetson Nano, drawing the bounding boxes and labels takes up CPU and GPU resources. By drawing only 3 categories of bounding boxes, the number of calls to cv2.rectangle and cv2.putText is reduced, thus reducing the rendering workload.

3. Resource Optimization

Problems caused by hardware limitations can also be solved by improving this implementation code. Firstly, I adjusted the GStreamer pipeline parameters from framerate=60/1 to framerate=30/1 to reduce the number of frames processed per second by the YOLO model, thus reducing the load on jetson's GPU and memory.

```
# Define the working GStreamer pipeline
gst_pipeline = (
    "nvarguscamerasrc ! "
    "video/x-raw(memory:NVMM), width=640, height=480, framerate=30/1, format=NV12 ! "
    "nvvidconv ! video/x-raw, format=BGRx ! "
    "videoconvert ! video/x-raw, format=BGR ! "
    "appsink"
)
```

I reduced the resolution to 640x480 (width=640, height=480), 640x480 is a medium resolution, lower than the common 1080p (1920x1080), reducing the amount of data per frame and lowering GPU and memory stress. And I will further reduce the processing resolution for subsequent inference.

```
# Load YOLOv5 model
weights = 'best2.pt'  # Path to your trained YOLOv5 weights
device = select_device('0')  # Change to 'cpu' if you are not using GPU
model = attempt_load(weights, device=device)
model.eval()
stride = int(model.stride.max())
names = model.module.names if hasattr(model, 'module') else model.names
```

Considering the hardware characteristics of the jetson nano, I used the code select_device('0') to select the GPU device, and GPU acceleration utilizes the Jetson Nano's 128-core Maxwell GPU to perform YOLOv5 inference, reducing the CPU load, which is this is a core feature of the Jetson Nano, taking full advantage of its hardware strengths.

```
# Preprocess image function
def preprocess_image(img, img_size=640):
    img0 = img.copy()
    img = cv2.resize(img, (img_size, img_size))
    img = img[:, :, ::-1].transpose(2, 0, 1)  # Convert BGR to RGB and rearrange
    img = np.ascontiguousarray(img)
    img = torch.from_numpy(img).to(device).float()
    img /= 255.0
    if img.ndimension() == 3:
        img = img.unsqueeze(0)
    return img, img0
```

The purpose of the code preprocess_image is to move the data to the GPU for processing. I have adjusted the image resolution from 1280x1280 to 640x640 in this function and optimized the input range of the model using normalization (dividing by 255) to improve the inference efficiency.

```
# FPS calculation
fps_list = []
smooth_window = 10
last_time = time.time()

# Compute FPS
current_time = time.time()
fps = 1 / (current_time - last_time)
last_time = current_time

fps_list.append(fps)
if len(fps_list) > smooth_window:
    fps_list.pop(0)

smooth_fps = sum(fps_list) / len(fps_list)
```

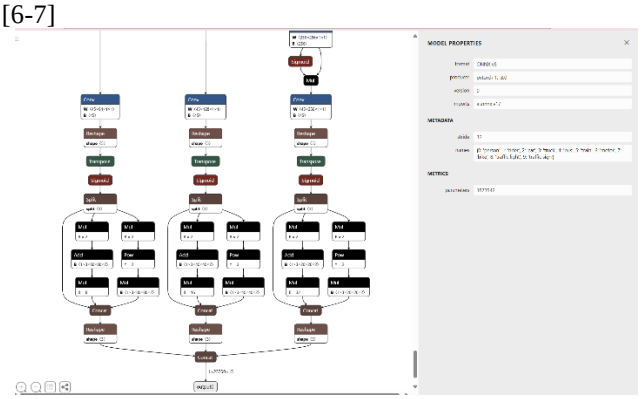
As shown in the figure, in this snippet I use fps_list and smooth_window=10 to calculate the smoothed FPS (smooth_fps = sum(fps_list) / len(fps_list), smooth FPS reduces the impact of frame rate fluctuations and allows users to monitor the performance of the device in real time, which provides a reference basis to detect whether the hardware resources are close to their limits. provides a reference for users to check whether the hardware resources are close to the limit and further adjust the parameters.

4. TensorRT

As mentioned before, by reviewing the literature, in order to convert the PyTorch model to a TensorRT model, I saved the

trained model in the specified path, opened a terminal window and executed the command python3 export.py --weights /home/jetson/Miniproject/best2.pt -- imgsz 640 --include onnx, this command is used to generate an ONNX model based on an existing PyTorch model. Since TensorRT requires a standardized input format, it acts as an intermediate format, bridging PyTorch and TensorRT. It is worth noting that installing the libraries required by ONNX will cause version conflicts with libraries we already have installed, such as numpy, and result in the python code not running, a compatibility fix will be explained later.

The netron tool can be used to visualize ONNX models. This is an open source neural network model visualization tool that supports a variety of model formats (including ONNX, PyTorch, TensorFlow, etc.), and can display the computational graph structure of a model either locally or via a browser, helping the user to intuitively understand the model's network structure, layer connections and parameters. [6-7]



As shown above, this structural image demonstrates the typical flow of the YOLOv5 detection head: convolution → activation → decoding → output, which is consistent with the design of YOLOv5. It shows the complete flow of the detection head, with no broken connections or missing layers, suggesting that the PyTorch to ONNX conversion was successful. The basic information provided on the right side of the image, format: ONNX v8, stride: 32, suggests that the ONNX model is a valid model [6-7], however stride: 32 indicates that the model supports detection of large targets, and is weak at detecting very small targets, such as distant persons. [7-8]

Save the ONNX model file in a worthwhile path and run the command /usr/src/tensorrt/bin/trtexec--onnx=/home/jetson/Miniproject/best2.onnx--saveEngine=/home/jetson/Model/best. trt --fp16 to get the TensorRT model. Combine this with FP16 optimization (using 16-bit floats compared to traditional 32-bit floats FP32) to reduce precision, reduce computation and memory usage, and avoid Jetson Nano's resource exhaustion issues such as the GStreamer pipeline crash.

Back to the NumPy version change issue mentioned earlier, in NumPy 1.20 and above, np.bool was deprecated and renamed to np.nool_, and PyCUDA would fail when trying to access np.bool because np.bool no longer existed. As shown in the figure, in the TensorRT inference code, I solved this problem with Monkey Patch. np.bool = np.bool_ ensures that np.bool still exists even in newer versions of NumPy, and PyCUDA will find np.bool when it calls trt.nptype to avoid the AttributeError. [9]

```
import cv2
import numpy as np
np.bool = np.bool_  # ? ?? PyCUDA ?? np.bool ?????(monkey patch)
import time
import sys
import os                    (import) trt: Module("tensorrt")
import tensorrt as trt
import pycuda.driver as cuda
import pycuda.autoinit
import torch
```

5. Model Conversion Fix

As for Model Conversion Issues I mentioned in Section 1, as shown below, the code loads best2.pt using torch.load, checks and fixes the PosixPath type, and in the repaired model file, all PosixPath objects are converted to strings, modifying only the paths in the metadata without affecting the model structure so that the PyTorch model can be converted successfully to ONNX format.

```
# Define paths
input_weights = 'best2.pt'  # Path to the original model file
output_weights = 'best2_fixed.pt'  # Path to save the fixed model file

# Load and fix the model
print(f"Fixing PosixPath in {input_weights}...")  # Log the start of the repair process
ckpt = torch.load(input_weights, map_location="cpu")  # Load the PyTorch model to CPU
for key in ckpt:  # Iterate through the model dictionary
    if isinstance(ckpt[key], Path):  # Check if the value is a PosixPath
        ckpt[key] = str(ckpt[key])  # Convert PosixPath to string
    elif isinstance(ckpt[key], dict):  # Handle nested dictionaries
        for subkey in ckpt[key]:  # Iterate through the nested dictionary
            if isinstance(ckpt[key][subkey], Path):  # Check if the nested value is a PosixPath
                ckpt[key][subkey] = str(ckpt[key][subkey])  # Convert PosixPath to string

# Save the fixed model
torch.save(ckpt, output_weights)  # Save the modified model to a new file
print(f"Fixed model saved as {output_weights}")  # Confirm the fixed model has been saved
```

IV. EXPERIMENTAL SETUP

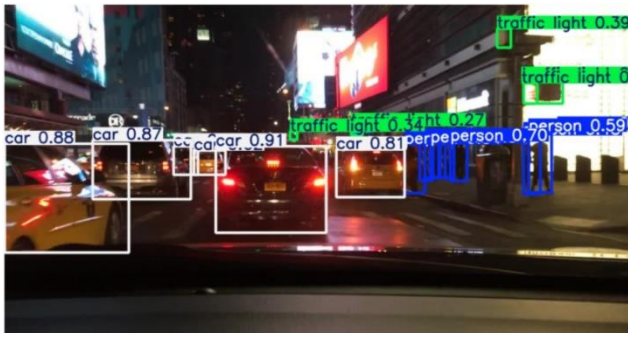
1. Model Training:

Install the necessary dependent libraries (e.g. PyTorch, YOLOv5, etc.) on the Kaggle Notebook, upload the bdd10k dataset, and organize it into the format required by YOLOv5. Train the model using YOLOv5's train.py script to generate the best.pt file

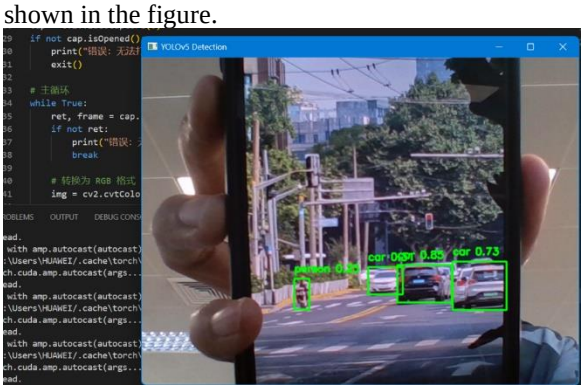
```
!python train.py \
    --img 640 \
    --batch 16 \
    --epochs 20 \
    --data /kaggle/working/bdd10k.yaml \
    --cfg /kaggle/working/yolov5n.yaml \
    --weights /kaggle/input/yolo/pytorch/default/1/yolov5n.pt \
    --name bdd10k_yolov5n \
    --project /kaggle/working/runs/train
```

Lastly, the model is validated and checked to check the mAP (average precision) and detection results.

```
Fusing layers...
YOLOv5n summary: 157 layers, 1772695 parameters, 0 gradients, 4.2 GFLOPs
val: Scanning /kaggle/input/bdd10k/bdd10k/labels/val... 1000 images, 0 backgroun
val: WARNING ⚠ Cache directory /kaggle/input/bdd10k/bdd10k/labels is not writeable: [Errno 30] Read-only file system:
                 Class     Images  Instances          P          R      mAP50
                   all       1000      18290      0.633      0.203      0.186     0.0852
                person       1000       1234      0.439      0.329      0.313      0.126
                 rider       1000         60          1          0     0.0475     0.0113
                   car       1000       9987      0.481      0.591      0.574      0.286
                 truck       1000        416      0.454       0.31      0.277      0.164
                   bus       1000        170      0.266      0.162      0.117     0.0772
                 train       1000          2          1          0          0          0
                 motor       1000         39          1          0    0.00469     0.0018
                  bike       1000        111          1     0.0175     0.0507     0.0153
         traffic light       1000       2740       0.31      0.308      0.207     0.0577
          traffic sign       1000       3531      0.377      0.315      0.273      0.112
Speed: 0.1ms pre-process, 1.0ms inference, 3.3ms NMS per image at shape (32, 3, 640, 640)
/usr/local/lib/python3.10/dist-packages/matplotlib/colors.py:721: RuntimeWarning: invalid value encountered in less
  xa[xa < 0] = -1
Results saved to /kaggle/working/runs/val/exp3
```



2. **Jetson Nano Set up**
   Burn the system image file to SD card, start Jetson and install OpenCV (with CUDA support), GStreamer, configure the CUDA environment, test the integration of OpenCV and GStreamer.
3. **Inference Code**
   Due to the greater hardware performance of the PC and the fact that it is easier for the user to compile the code, firstly I wrote the inference code on PC, and it runs as shown in the figure.

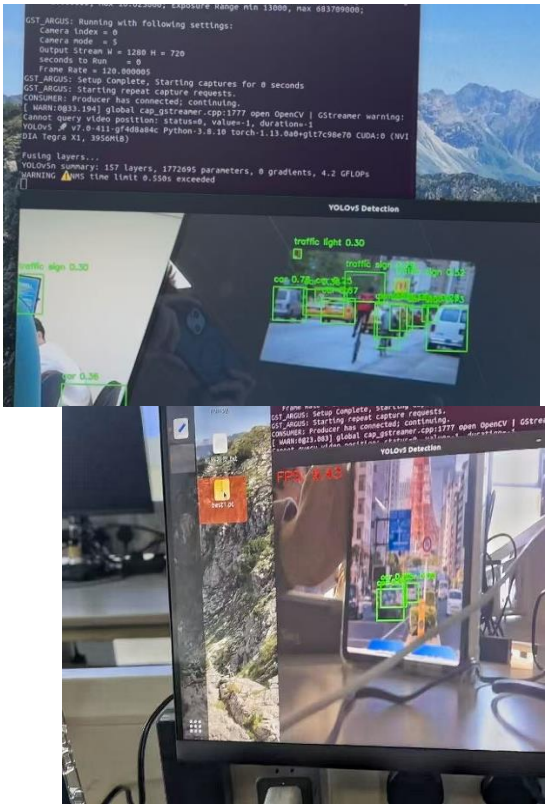


4. **Integration and Optimization of Inference Code and Jetson:**
   After the trained model achieves the desired results on the PC, modify the inference code to work on jetson, and adjust the resolution and frame rate limits to make sure it will not consume too much computing resources. Record the results before and after the modification
5. **Convert Model Format to Record Experimental Results**
   Convert the model file to a TensorRT model, verify the performance of the model after conversion, and record the results.

V. RESULTS AND DISCUSSION

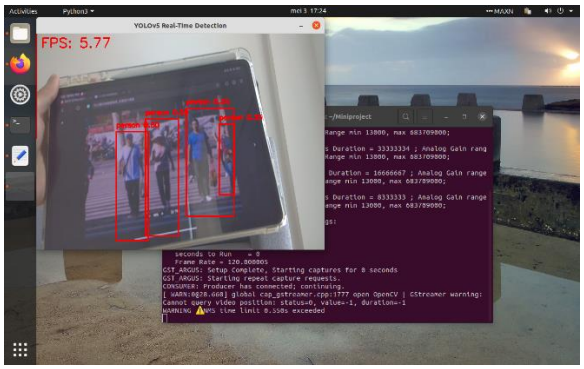
As described in Section 4, I record the results of three groups of experiments: Original Experiment (running the PyTorch model without any optimization of the inference code), Resource Optimization (optimization methods such as adjusting the resolution, FPS limits, label filtering, etc.), and TensorRT Optimization (running the optimised TensorRT model)

1. **Original Experiment**
   As shown in the figure, the model detection performance is satisfactory, but it takes a long time to run the inference code to start the camera, and running the model for a long time will cause the Jetson host to heat up seriously, and the camera will not be able to be turned on if the inference code is run repeatedly for many times, and the jetson will experience lagging or even crash. FPS close to 10 but can't maintain above 10.



2. **Resource Optimization**
   After implementing the method I mentioned in section 3 about Resource Optimization, the time for the inference code to start the camera was reduced to 5 minutes and the FPS was maintained above 10, which meets the project requirements, in addition, I edited the boxes with different colors for distinguishing between different categories of objects. Equipment heating problems improved, but not completely resolved.

3. **TensorRT Optimization**
   After running the TensorRT model, it took only 1 minute to start the camera, and the device no longer overheated. However, unexpectedly, the FPS dropped instead, and only the label ‘person’ was classified better.



VI. CONCLUSION AND FUTURE WORK

I summarized the results of the experiments in a table as follows.

Experiment	Time to Start Camera (min)	FPS	Device Heating	Detection Performance
Original Experiment	7-8	9-10	high level	Ideal
Resource Optimization	5	10-15	medium level	Ideal
TensorRT Optimization	1	5-8	no overheating	Only for ‘person’

Comparing the results of these three experiments, it can be concluded that both Resource Optimization and TensorRT Optimization are positive in improving the efficiency of the YOLO model running on the Jetson Nano. Despite the limited computational power of the hardware, these optimizations allowed me to see the desired results of the demonstration and gave me a better understanding of the significance of machine vision as a course.

By analyzing the results, there are several possible reasons for the unexpected occurrence of TensorRT experiments as follows.

1. When converting PyTorch models to TensorRT models, there may be conversion errors or incompatible layers.
2. The Jetson platform has limited resources, and TensorRT uses FP16, which may cause the model to be less sensitive to subtle features in certain categories (non- ‘person’ categories).

In order to fix this unexpected problem, here are possible approaches for the future.

1. Refactor PyTorch models to improve TensorRT compatibility, which can be tested for compatibility during refactoring using NVIDIA's torch2trt. [9]
2. Collect or enhance training datasets for non- ‘person’ categories, using data augmentation techniques (e.g. random cropping, rotation, color dithering) to increase feature diversity. [10]

REFERENCES

[1] de Carvalho A C, Freitas A A. A tutorial on multi-label classification techniques[J]. Foundations of Computational Intelligence Volume 5: Function Approximation and Classification, 2009: 177-195.

[2] Zhang Y, Guo Z, Wu J, et al. Real-time vehicle detection based on improved yolo v5[J]. Sustainability, 2022, 14(19): 12274.

[3] Diamond, S. and Boyd, S. (2016) ‘CVXPY: A Python-embedded modeling language for convex optimization’, Journal of Machine Learning Research, 17(83), pp. 1–5.

[4] Jeong E J, Kim J, Ha S. Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards[J]. ACM Transactions on Embedded Computing Systems (TECS), 2022, 21(5): 1-26.

[5] Valladares S, Toscano M, Tufiño R, et al. Performance evaluation of the Nvidia Jetson Nano through a real-time machine learning application[C]//Intelligent Human Systems Integration 2021: Proceedings of the 4th International Conference on Intelligent Human Systems Integration (IHSI 2021): Integrating People and Intelligent Systems, February 22-24, 2021, Palermo, Italy. Springer International Publishing, 2021: 343-349.

[6] Zhang Z D, Tan M L, Lan Z C, et al. CDNet: A real-time and robust crosswalk detection network on Jetson nano based on YOLOv5[J]. Neural Computing and Applications, 2022, 34(13): 10719-10730.

[7] Gopiktrishna P B, Rishekeeshan A. Accelerating native inference model performance in edge devices using TensorRT[C]//2024 IEEE Recent Advances in Intelligent Computational Systems (RAICS). IEEE, 2024: 1-7.

[8] Fang J, Liu Q, Li J. A deployment scheme of YOLOv5 with inference optimizations based on the triton inference server[C]//2021 IEEE 6th International Conference on cloud computing and big data analytics (ICCCBDA). IEEE, 2021: 441-445.

[9] Ren J, Wang Z, Zhang Y, et al. YOLOv5-R: lightweight real-time detection based on improved YOLOv5[J]. Journal of Electronic Imaging, 2022, 31(3): 033033-033033.

[10] Gupta D, Lee R D, Wynter L. On Efficient Object-Detection NAS for ADAS on Edge devices[C]//2024 IEEE Conference on Artificial Intelligence (CAI). IEEE, 2024: 1005-1010.