SANTA CLARA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Date: June 7, 2021

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Ethan Paek
Tyler Niiyama
Justin Liu
Spencer Tsang
Kent Ngo
Jackson Tseng

ENTITLED

Real-Time Multi-Camera Traffic Intersection Analysis on IoT Devices

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

David C. Anastasiu
Digitally signed by David C. Anastasiu
Date: 2021.06.06 21:56:04 -07'00'

Thesis Advisor

Nam Ling
Nam Ling (Jun 7, 2021 08:52 PDT)

Department Chair

Real-Time Multi-Camera Traffic Intersection Analysis on IoT Devices

by

Ethan Paek
Tyler Niiyama
Justin Liu
Spencer Tsang
Kent Ngo
Jackson Tseng

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science in Computer Science and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 7, 2021

Real-Time Multi-Camera Traffic Intersection Analysis on IoT Devices

Ethan Paek
Tyler Niiyama
Justin Liu
Spencer Tsang
Kent Ngo
Jackson Tseng

Department of Computer Science and Engineering
Santa Clara University
June 7, 2021

ABSTRACT

The continuous growth of urban and suburban areas has increased the number of commuters and general transportation along central roadways, which has resulted in a trend of more traffic congestion in essential areas every year. This creates a challenge for civil engineers to improve the efficiency of the traffic intersection systems. Our team worked on this issue by building a system to improve traffic intersection analysis by video camera to count cars using multiple camera angles at a single intersection. By providing better real-time traffic network analysis, civil engineers can more accurately monitor the efficiency of traffic intersections and then make better predictions for timing changes in traffic light signaling. The novelty of our project comes from using state of the field machine learning vehicle counting algorithms for a single camera and then modifying them to work across an Internet of Things (IoT) network of devices for multiple camera angles across a single intersection. We achieved fast processing speeds on the Nvidia Jetson NX IoT devices by utilizing quantization of the object detection YOLO algorithms to be able to process video input in real time. Another improvement from current video-camera based solutions is using multiple camera angles in a network to reduce vehicle counting inaccuracy from camera obstruction from a single perspective. For future work, our project can be improved by working on the re-identification of cars when the tracking is lost, and at crowded intersections where exclusive regions of interest for intersection paths are hard to define.

ACKNOWLEDGMENTS

We would like to express our gratitude to Dr. David Anastasiu for advising us throughout the senior design process. We would also like to thank the Santa Clara University School of Engineering for providing funding for our senior design project.

Table of Contents

1 Introduction **1**
1.1 Problem and Motivation . . . 1
1.2 Solution . . . 1
1.2.1 Current Solutions . . . 1
1.2.2 Contribution . . . 2

2 Literature Survey **3**
2.1 Previous Works . . . 3
2.1.1 Vehicle Detection . . . 3
2.1.2 Vehicle Tracking . . . 4
2.1.3 Pipelined Algorithms . . . 5
2.2 AI City Challenge 2020 Top Algorithms . . . 5

3 Requirements **8**
3.1 Description . . . 8
3.2 Functional Requirements . . . 8
3.3 Nonfunctional Requirements . . . 8

4 Technologies Used **10**
4.1 Description . . . 10
4.2 IoT Devices . . . 10
4.3 PyTorch and TensorRT . . . 10
4.4 Scikit-learn and OpenCV . . . 11
4.5 RabbitMQ . . . 11
4.6 Datasets Used . . . 11
4.6.1 AI City 2020 Challenge . . . 11
4.6.2 COCO Data Set . . . 11

5 Design Rationale **12**
5.1 Architecture Diagram . . . 12
5.2 System Architecture . . . 13
5.2.1 Object Detection . . . 13
5.2.2 Quantization . . . 13
5.2.3 Autoencoder . . . 14
5.2.4 Jetson Communication . . . 15
5.2.5 Vehicle Tracking . . . 16

6 Evaluation **17**
6.1 Description . . . 17
6.2 Results . . . 17

7 Ethical Concerns **19**
7.1 Security . . . 19
7.2 Social . . . 19

8 Limitations and Challenges **20**
8.1 Quantization . . . 20
8.2 System Architectures . . . 20
8.3 Remote Collaboration . . . 21

9 Future Work **22**
9.1 Current State . . . 22
9.2 Next Steps and Long-Term Vision . . . 22

10 Conclusion **23**
10.1 What We Have Learned . . . 23
10.2 Why it is Important . . . 23

A User Manual: How to Setup **25**
A.1 Installing and Upgrading Libraries . . . 25
A.2 Installing Pytorch 1.7.0 . . . 26
A.3 Installing torchvision 0.8.0 . . . 26
A.4 Building Our Pre-trained Model . . . 26
A.4.1 Installing Pycuda . . . 26
A.4.2 Installing Tensorflow . . . 27
A.4.3 Building YOLOv3 Model . . . 27
A.5 Setting up Rabbit MQTT Server . . . 28
A.5.1 Installing Rabbit MQ . . . 28
A.5.2 Allow Other Devices on Local Network to Connect to Rabbit MQ Server . . . 29

B Source Code **30**
B.1 Evaluation Code . . . 30
B.1.1 eval_yolo.py . . . 30

List of Figures

5.1 Architecture Diagram for Real-Time Multi-Camera Traffic Intersection Analysis 12
5.2 256-dim. bottleneck reconstruction vs. 1024-dim. bottleneck reconstruction 14
5.3 Bottleneck layer comparison 15
5.4 Architecture Diagram of Communication Between Jetsons via RabbitMQ 15

Chapter 1

Introduction

1.1 Problem and Motivation

Growing populations around road-centric cities have increased the complexity and magnitude of daily traffic that commuters continuously face. This increase in traffic requires smarter traffic infrastructure in order to allow a larger volume of vehicles to travel in the area without necessarily building more roads. Civil engineers analyze traffic by recording traffic data using different advanced detection methods such as inductive loops, magnetic detection, radar detection, and more recently video imaging detection [1]. Inductive loops are one of the more common approaches for traffic measurement, however they are expensive to install and require eventual replacement of the inductive detectors which requires road closures. From a study with the Texas Department of Transportation, the cost of advanced induction detectors hardware and installation on a single lane highway is reported to cost $13,880 and is $19,680 for a two-lane highway [1]. Video cameras combined with novel machine learning technology can provide highly accurate vehicle detection and counting in real time without the need for expensive induction detectors and routine maintenance.

1.2 Solution

In this section, we explore the current solutions active in the market, and our group's contribution to future solutions.

1.2.1 Current Solutions

One of the first video image detection systems is the tripwire system, which is actively implemented in traffic infrastructures. The tripwire system allows the operator to define "limited number of detection zones in the video camera field of view" and the advantage is minimal processing cost [1]. The severe disadvantage to the tripwire system is that it only reports the vehicle counts and speeds for the detection zones. Recently there has been more research for advanced traffic analysis from video data using novel machine learning techniques. One of the competitions that has been accelerating this research is the AI City Challenge [2]. Two of the challenges from the 2020 AI City Challenge were video-based automatic vehicle counting and city-scale multi-target multi-camera vehicle tracking. From the first

challenge, competing teams submitted algorithms that counted vehicles and the paths they took from a camera view and were evaluated by the accuracy of vehicle counting and performance efficiency. The second challenge tracked vehicles across multiple camera angles at a single intersection and across a city network of intersections and were evaluated upon which teams had the best accuracy in vehicle detection over the camera network. The winners from this 2020 competition are the base for our project's contribution to traffic network analysis from video data.

1.2.2 Contribution

Our project was based upon five of the top performing algorithms from the AI City Challenge 2020: DiDi1, DiDi2, Baidu, ENGIE, and Zero-Virus. The novelty of our project comes from creating a Internet of Things (IoT) network of devices to monitor a different camera angle each for a single intersection to perform real-time vehicle counting. Two of the problems our project worked at solving are: tracking individual vehicles within an intersection, and counting vehicles based on the path they take through an intersection. The main benefit of performing vehicle tracking and counting from multiple camera angles is that it can improve the problem of vehicle obstruction which creates inaccuracies for a single camera angle perspective. Specifically, when a vehicle moves behind something like a traffic sign or another vehicle, the camera and thus the tracking algorithm loses sight due to the obstructing object resulting in the trackers to consider the same car as two unique vehicles. By having multiple camera angles tracking collaboratively in the IoT network, a fully non-obstructed perspective can be better created. The progress of multi-camera single-intersection real time traffic vehicle counting in an IoT network is an improvement for state-of-the-art machine learning traffic analysis. Our team successfully quantized 3 of the 5 algorithms, with two of them working in the IoT network. Specific performance achievements will be mentioned in the Design Rationale and Evaluation chapters.

Chapter 2

Literature Survey

2.1 Previous Works

Mirthubashini and Santhi [3] provide a comprehensive overview and comparative analysis on current approaches to vehicle detection and vehicle tracking. The vehicle detection methods surveyed are SSD, Faster R-CNN, YOLO, and Haar Cascades. The vehicle tracking methods surveyed are Centroid Tracking Algorithm, KCF, DCF-CSR, TLD, and Medianflow.

2.1.1 Vehicle Detection

Singleshot Multibox Detector (SSD) proposed by Liu et al. [4] uses a Convolutional Neural Network (CNN) image classifier like VGG-16 as a base network, then adds convolutional feature layers at the end of the base network. Each feature layer decreases in size progressively to detect objects at different scales. For each m x n feature layer, a small kernel produces output values relative to a default bounding box. The default bounding boxes tile the original image in a convolutional manner and a number of filters are applied equal to the number of classes plus the 4 directional offsets from each bounding box. The overall loss function is a weighted sum of smooth L1 loss for the localization loss and softmax loss for the confidence loss.

Faster R-CNN proposed by Ren et al. [5] consists of a CNN that creates a feature map, then uses a region proposal network (RPN) on the feature map to find regions of interest. The RPN uses a sliding n x n window over the feature map, mapped to a lower dimensional feature, then fed into a box-regression layer and a box-classification layer. The box-regression layer outputs coordinates of boxes relative to anchors, and the box-classification layer outputs probabilities that there is or isn't an object in the box proposed by the box-regression layer. A benefit to Faster R-CNN is that it's translation invariant, meaning that an image will be recognized as the same image if it's moved to another location. An improvement to Faster R-CNN, Mask R-CNN [6], improves Faster R-CNN's Kalman filter, accurately detects vehicles under a variety of complex environments. From experiments on the Common Objects in Context (COCO) dataset, Mask R-CNN achieves higher accuracy on complex environments than traditional methods, but does

not perform in real time, achieving only 2.8 frames per second.

You Only Look Once (YOLO) proposed by Redmon et al. [7] splits the input image into grids and uses anchor boxes to detect objects within each grid. Each bounding box calculates a probability that the box contains an object of a certain class. Non-max suppression is used to discard overlapping boxes and chooses the boxes with the highest probability. YOLOv3 [8] and YOLOv4 [9] have built upon the original YOLO architecture, boasting better accuracy and speed. Zhang et al. [10] proposed an improved version of YOLOv3, called K-YOLOv3. Some modifications included an improved mean-shift adaptive tracking algorithm to predict where the vehicle would move in the next image frame, having a based YcbCr color space shadow segmentation to improve the tracking effect, and having a multi-scale KCF tracking algorithm to improve the scale conversion and feature description.

Haar Cascades proposed by Viola and Jones [11] passes sub-rectangles of an input image through Haar features to extract features. Of the hundreds of thousands of calculated Haar features, Adaboost is used to select the best features that are passed to a classifier. Weak learners compute Haar features over a moving window of the image. The selected features are added individually to the moving window, where the classifier must accept or reject the window at each stage.

2.1.2 Vehicle Tracking

Centroid Tracking Algorithm is the most intuitive vehicle tracking algorithm. It calculates the centroid of each bounding box in a frame, then associates each box with the closest bounding box centroid in the next frame. Euclidean distance is used as the proximity measure.

Henriques et al. [12] proposed Kernelized Correlation Filters (KCF), which attempts to find an optimal filter to produce a desired output, centered in the form of a Gaussian shape. The Discrete Fourier transform is used to diagonalize the data matrix, greatly reducing the size and number of computations required.

Discriminative Correlation Filters with Channel and Spatial Reliability (DCF-CSR) was introduced by Lukežič et al. [13]. Building upon Discriminative Correlation Filters, which learn a filter from a pre-defined response on a training image. The spatial reliability map is used to change the filter support from the tracking frame to a part of the selected region.

Tracking Learning Detector (TLD), proposed by Kalal et al. [14], consists of tracking, learning, and detection modules that operate simultaneously. The tracking module estimates an object's location frame by frame. The detector analyzes each frame to detect all appearances of a detected object and corrects the tracker if necessary. The learning module uses P-N learning to train and correct the detector's false positives and false negatives.

Medianflow tracker, also proposed by Kalal et al. [15] tracks the target object forward and backward in time and calculates the difference in trajectories. The difference between trajectories is used to detect errors in tracking. However, this method only works when movement is predictable and small.

Siamese networks are also used for vehicle tracking. Li et al. [16] introduce a Siamese network with backward prediction-based vehicle tracking, which includes a forward position tracking to reduce interference and noise, and a backward prediction verification to prevent false positives. In the case of the forward position tracking failing to detect the correct target, the backward prediction would verify and diminish the chance of detecting the wrong target. The system merges the weights of both of these bounding boxes in order to produce a better output with more accurate detection.

2.1.3 Pipelined Algorithms

Liang et al. [17] created a vehicle counting model by training a YOLOv3 network on a dataset with multiple perspectives from highway CCTV cameras. They used an improved multi-scaled and multi-feature tracking algorithm which is based on KCF in order to avoid extracting single features and single-scale defects. For matching, they combine Intersection over Union (IoU) similarity and a proposed row-column optional association criterion. Finally, they are able to automatically determine the travelling direction of each individual vehicle based on its trajectory and update the vehicle count based on the setting position of traffic lines/lanes and trajectory. Their method achieves a high accuracy of vehicle detection while maintaining accuracy and precision in tracking multiple objects, and obtains accurate vehicle counting results which can meet real-time processing requirements.

Bui and Cho [18] proposed a comprehensive framework with multiple classes and movements for vehicle counting. They utilize YOLOv3 for vehicle detection, and used DeepSORT and a distinguished region tracking approach for vehicle tracking. They achieved an accuracy of 80% to 98% for different movements within a busy traffic intersection with only a single view of a CCTV camera. However, the determination of distinguished regions for tracking vehicles is a manual process.

Mutharpavalar and Sebastion [19] describe a system using multiple IP-based cameras to create a system used to monitor traffic in real-time. Mutharpavalar and Sebastion use multiple methods to allow for easier motion and vehicle detection including background subtraction, optical flow, RGB2GRAY Conversion, etc. In each camera, four parallel lines are drawn on the screen to mark starting and ending coordinates which are passed as arguments to a function that handles capturing and analyzing the videos. In this project, Mutharpavalar and Sebastion developed code to only monitor vehicles going North and South. Data processing is done on one camera at a time at intervals of 10 minutes to avoid overflowing the system with data or slow down when processing all cameras at the same time.

2.2 AI City Challenge 2020 Top Algorithms

In addition to the pipelines highlighted above, the AI City Challenge 2020 top algorithms provide a useful baseline for vehicle tracking and detection. The Baidu, ENGIE, DiDi1, DiDi2, and Zero-Virus algorithms have all achieved high accuracy on the challenge dataset [20].

Baidu

The Baidu [21] algorithm pipeline consists of three parts: frame-wise vehicle detection, online multiple vehicle tracking, and trajectory-based vehicle counting. For vehicle detection, Baidu used a two-stage Faster-RCNN, which uses Resnet50 as the feature extractor. Vehicle tracking uses Deepsort with Kalman filtering, and matches vehicle paths with tracks based on the Mahalanobis distance between vehicle detections. Detection re-matching and single object tracking strategies are used to reduce ID-switching errors caused by occlusion. Tracks are assigned based on the Hausdorff distance between a vehicle's track and the proposed trajectory, and the vehicle is counted once it leaves the region of interest (RoI).

ENGIE

The ENGIE [22] algorithm is divided into four parts: the motion model, Tracktor, filtering and track generation, and line crossing. The motion model uses the linear path of an object to propose possible locations. The Tracktor module uses Faster R-CNN to predict the bounding box locations given the proposed locations from the motion model. Filtering and track generation filters out any bounding boxes that are a certain percentage outside of the region of interest (ROI). Then, proposed tracks are removed or combined using non-max suppression, and remaining tracks are added to the list of tracks. The line crossing model assigns a vehicle's path to one of the tracks. Classifying freight trucks from regular vehicles is done by comparing the area of the bounding box to the average vehicle's bounding box area.

DiDi1

The DiDi1 group did not publish a paper to accompany their code, but their system uses YOLOv3 to predict bounding box locations for vehicles, which uses Darknet as the feature extractor. Vehicle tracking uses a Kalman filter with high uncertainty given to initial velocities since they are not captured on video. The predicted bounding box locations are assigned to current boxes based on IoU, then estimated trajectory paths are calculated and compared with preset tracks for each camera. Vehicles are assigned to tracks based on their trajectory's similarity and direction.

DiDi2

DiDi2's [23] system consists of five modules: vehicle perception, object tracking, trajectory modeling, trajectory matching, and vehicle counting. DiDi2 used NAS-FPN for their vehicle detection model, which is based on RetinaNet. Images are pre-processed by using Gaussian background modeling to separate moving objects from the stationary background. Then, a Kalman filter is used for motion prediction and state updates. A greedy matching algorithm is used to match predicted targets and detections, then the IoU is used to assign any remaining unmatched targets. Trajectories are generated from the object tracking and similar trajectories are aggregated using k-means clustering.

The trajectory is divided into several segments to reduce errors, then angle and distance similarity is computed segment by segment.

Zero-Virus

Zero-Virus' [24] algorithm uses Mask R-CNN with a ResNeXt-101 feature pyramid network backbone for vehicle detection. The Mask R-CNN feature representation of a detected object is given an ID and assigned to tracks based on feature similarity and spatial position. They use linear interpolation to predict the position of vehicles based on previous frames. A point-segment distance function is used as a proximity metric to compare trajectories to existing tracks, a completeness metric measures progress along a track, and a stability metric measures if the vehicle moves along the track at a constant distance. These three metrics are aggregated to determine the final confidence score of a track.

Chapter 3

Requirements

3.1 Description

This chapter lists our system's functional and nonfunctional requirements, grouped by importance. Requirements define and qualify what the system must do. In Section 3.2, the list of functional requirements are categorized into required and recommended. In Section 3.3, the nonfunctional requirements are considered in terms of performance and reliability.

3.2 Functional Requirements

Functional requirements explain what the system must do. They specify a behavior or function, which can be described as a set of inputs, the behavior, and outputs.

- Required
  - Use quantized machine learning models to accurately detect and track vehicles from recordings of active traffic intersections
  - Handle interoperability with other IoT devices
  - Automatically determine if one vehicle is the same from different camera angles
- Recommended
  - Upgrade the system to work in real-time on an actual traffic intersection
  - Keep count of the number of vehicles that pass through a traffic intersection
  - Expand the number of IoT devices for this system for enhanced accuracy

3.3 Nonfunctional Requirements

Nonfunctional requirements describe the manner in which the functional requirements are completed and achieved. In addition, nonfunctional requirements define how a system is supposed to behave.

- Performance
  - The system must provide accurate detection and tracking of vehicles
  - The system must be able to process data in real-time
- Reliability
  - The system must protect privacy, confidentiality, and integrity of data
  - The communication protocols must be secured and authenticated for sending and receiving data

Chapter 4

Technologies Used

4.1 Description

In this section, we discuss the technologies that are used in our project including the Nvidia Jetson kits, data sets, libraries, deep learning models, and communication software.

4.2 IoT Devices

Regarding the IoT devices used in our project, we decided to use three Nvidia Jetson Xavier NX kits. These Jetson kits are designed to bring super fast performance to edge systems and embedded applications. With hundreds of CUDA cores, multiple CPUs, and dedicated deep learning accelerator engines, we are able to run resource-intensive algorithms and neural networks with less power while maintaining a reasonable amount of accuracy. In the final implementation, each Jetson would be connected to a video camera to take in the video frames as input to the rest of the system. Each Jetson would also be running the quantized algorithms individually while sending vehicle data to the other Jetsons.

4.3 PyTorch and TensorRT

PyTorch is a widely used open source Python machine learning library that provides an easy way to create and deploy deep learning models. Pre-trained implementations of state-of-the-art models are also provided, including many of the object detection models mentioned in our literature survey like YOLOv3 and YOLOv4. We used these implementations and used PyTorch to build and train our image autoencoder. PyTorch's torchvision provided many image preprocessing techniques used in our video processing. TensorRT is an Nvidia software development kit used for deep learning inference. It is built upon Nvidia's parallel programming model, CUDA, providing several optimizations that allow deep learning models to run quickly on Nvidia devices like Jetsons. One of these optimizations is int8 and fp16 quantization we used to increase the inference speed of object detection models on the Jetsons.

4.4 Scikit-learn and OpenCV

Scikit-learn is an open source Python machine learning library that provides implementations of machine learning models, preprocessing, and evaluation metrics [25]. In our project, we used scikit-learn's evaluation metrics to measure and compare the performance of various modules in our algorithm. OpenCV is an open source computer vision library that provides implementations of image processing and video analysis. We used OpenCV for image transforms like resizing and normalization, as well as for reading video.

4.5 RabbitMQ

RabbitMQ is an open source message broker that transmits data via messages. We chose this library because of its simple implementation and the wide variety of messaging protocols supported. RabbitMQ provides its own frameworks for different types of services that the developer may want. It is possible to simply send and receive a message, send a singular message to all available queues, and have consumers receive messages in a round-robin fashion, etc. For the purpose of our project, we planned to use the publish/subscribe model provided by RabbitMQ which would allow messages to be sent to all the Jetson devices simultaneously. However, we later decided that it would be easier to sequentially send messages between each Jetson, because we used only three IoT devices.

4.6 Datasets Used

4.6.1 AI City 2020 Challenge

The AI City Challenge was created to push the boundaries of what AI and machine learning can do in edge systems. The conference provides a large dataset of videos of multiple intersections. For the purpose of our project, we chose sets of cameras that overlook the same intersections. After finding such camera footage, we marked the timestamps such that the cameras were synchronized with each other. We also made manual annotations to the videos detailing the region of interest, the path of each lane within the intersection, as well as the start and end points for each lane we are tracking.

4.6.2 COCO Data Set

The "Common Objects in Context" (COCO) data set is used primarily for object detection and segmentation. It contains over 300,000 images that can be broken down to detect over a million objects separated into 171 categories. Since the YOLOv3 and YOLOv4 object detection models have been pre-trained with the COCO data set, the results of the quantized versions of those algorithms should be compared on the same data set.

Chapter 5

Design Rationale

5.1 Architecture Diagram

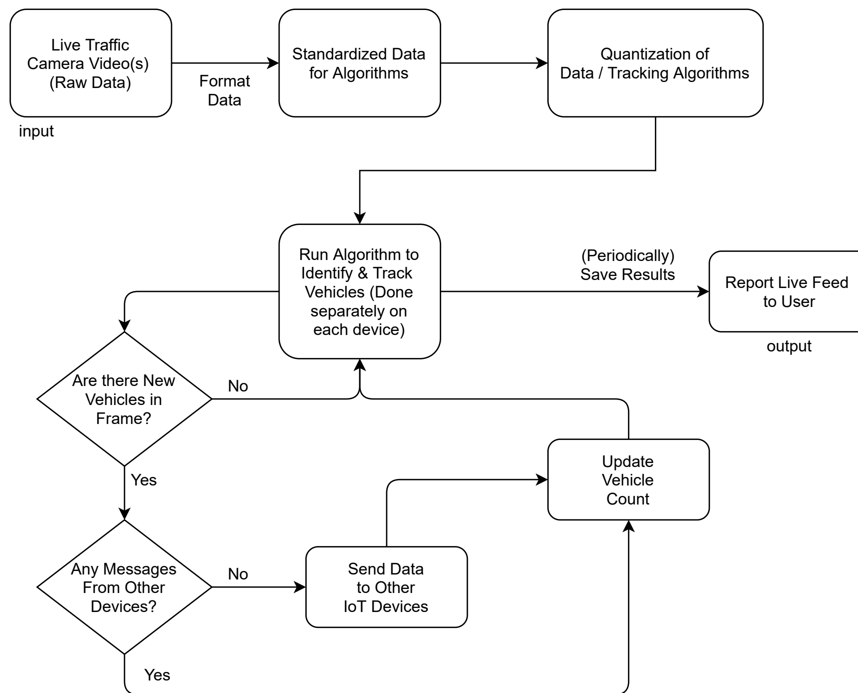

Figure 5.1: Architecture Diagram for Real-Time Multi-Camera Traffic Intersection Analysis

Each Jetson has its own camera overlooking the same intersection. Before we input data into our quantized algorithms, we had to reformat the camera footage into smaller dimensions such that it would be processed even faster in the neural networks. Each Jetson is running the algorithms locally to identify and track any vehicles it sees from its angle. If a Jetson sees new vehicles in frame and has not received any messages from the other Jetsons, it sends information of

the new vehicles to the other Jetsons and updates the vehicle count accordingly.

5.2 System Architecture

5.2.1 Object Detection

For the object detection portion of our system, we used YOLOv3, the third iteration of the YOLO model proposed by Redmon et al. [8]. Compared to previous YOLO versions, YOLOv3 uses a 53-layer Darknet for feature extraction instead of the original 19-layer Darknet. Features are extracted at the last three blocks of the Darknet, allowing for object detection at multiple scales. Anchor boxes are assigned to each of the extracted features, and predictions are calculated for offsets to the box, probability the box contains an object, and probabilities the object is of a certain class. Non-max suppression is used to remove boxes with low object probabilities or conflicting class categorizations.

The YOLOv3 network takes a frame of traffic intersection footage as an input, then outputs bounding boxes, object probabilities, and class confidence scores for all of the objects detected in the image. We discard objects that are not classified as cars or trucks, then pass the bounding boxes and confidence scores to the autoencoder.

5.2.2 Quantization

Quantization is a technique for improving the computational cost and running time for a neural network to propagate a response for a given input. The process of quantization is completed by reducing the precision of the weights and inputs in a neural network [26]. The resulting quantized neural network consumes less memory and requires less total arithmetic instructions to produce an output compared to the non-quantized variation [26]. For example, a series of 64-bit float operations representing a neural network will require more memory and CPU instructions to complete compared to a series of lower precision floats or integer operations. Having a smaller instruction count for a neural network to propagate an output allows the neural network to produce a response in less time. Having a faster neural network is important for our hardware because we want to be able to produce responses from the neural network at a rate that can keep up with real time video frame rates.

In our project, we applied quantization to our object detection models using the TensorFlow and TensorRT deeplearning frameworks. Our DiDi1 algorithm started with a YOLOv3-416 object detector implemented in TensorFlow, and then was converted into a YOLOv3-288 TensorRT quantized implementation with floating point 16 (FP16). Our DiDi2 algorithm was converted from a NAS-FPN-640 object detector model to a YOLOv4-416 quantized implementation with FP16 as well. Additionally, we were able to quantize the Baidu algorithm which was implemented with the paddledetection object detector using the paddlepaddle library. However paddledetection requires a 64-bit operating system, and is not compatible with the ARM architecture of our IoT network.

5.2.3 Autoencoder

Autoencoders were initially introduced by Rumelhart et al. [27] in 1986 and were traditionally used for dimensionality reduction. More recently, autoencoders have been used to learn generative models of data [28], and has been applied in areas such as anomaly detection and image de-noising. In our algorithm, we use an autoencoder to extract high-level features of vehicles detected by the YOLOv3 network. An autoencoder consists of two parts: an encoder and a decoder. The encoder is a convolutional neural network that takes an input image and outputs a bottleneck layer. The bottleneck layer is used as the input to the decoder, which is another convolutional neural network that outputs a re-creation of the input image. Since we are using the autoencoder for feature extraction, the bottleneck layer is much smaller than the input image.

The usage of an autoencoder has two purposes: It reduces the size of an image to make calculations and transmissions between Jetsons easier and the convolutional layers generalize high level features in a way that raw pixel data does not. Our autoencoder consists of 5 convolutional encoding layers that reduce a $112 \times 112 \times 3$ dimension input to a $4 \times 4 \times 64$ dimension bottleneck layer and 5 convolutional decoding layers that expand the bottleneck layer back to a $112 \times 112 \times 3$ dimension image. Using the bottleneck encoding as the extracted features, this results in a $36.75\times$ reduction in size. The size of the bottleneck layer determines the amount of information preserved in the reconstruction. Figure 5.2 shows how a 256-dimension encoding preserves much less information than a 1024-dimension encoding.

Figure 5.2: 256-dim. bottleneck reconstruction vs. 1024-dim. bottleneck reconstruction

Since we use convolutional layers to extract high-level features from the images, similar images have similar encodings. Figure 5.3 shows how visually different input images have noticeably different encodings. The bottom images are 32×32 representations of the 1024-dimension bottleneck encodings, with dark pixels corresponding to high values and light pixels corresponding to low values. We can see that the first two images are almost identical, so their encodings are almost identical. The third image is very different, so it produces a noticeably different encoding. These bottleneck encodings are the representations of detected vehicles that we pass between Jetsons to compare.

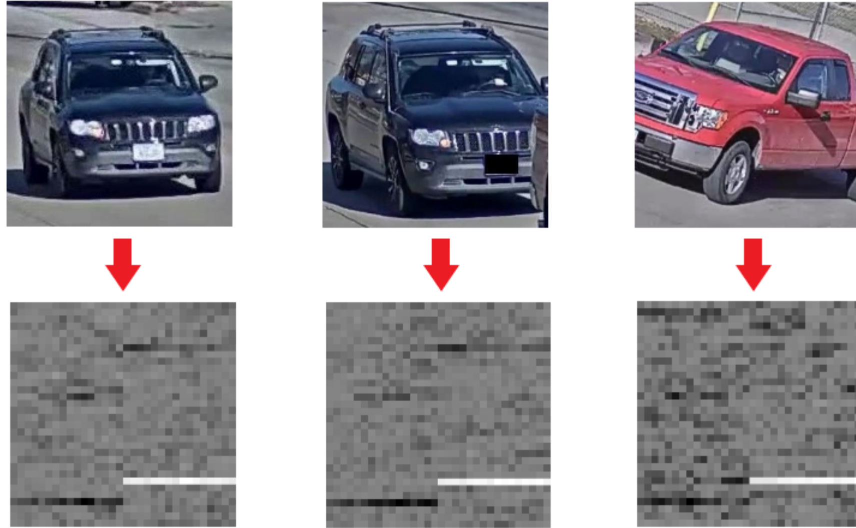

Figure 5.3: Bottleneck layer comparison

5.2.4 Jetson Communication

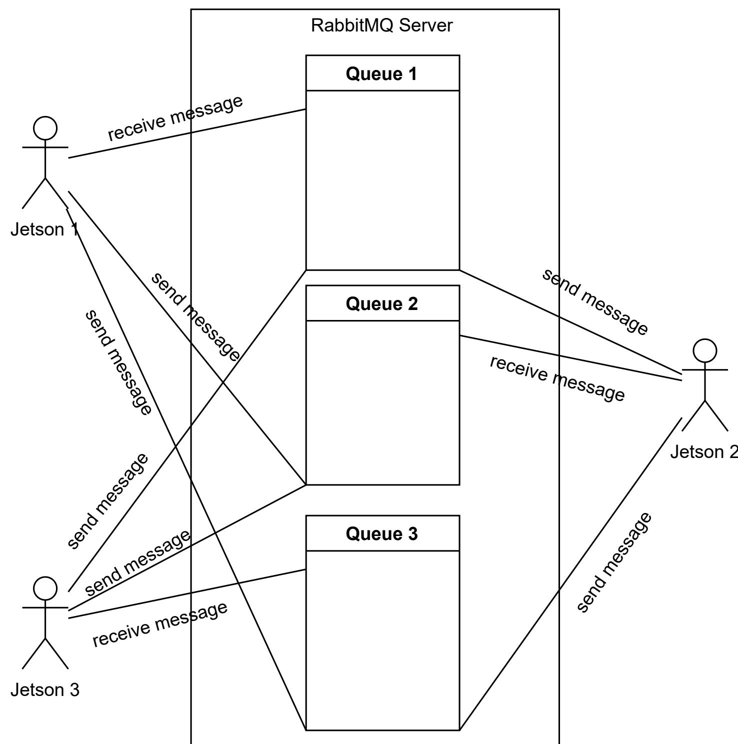

Figure 5.4: Architecture Diagram of Communication Between Jetsons via RabbitMQ

In our system, the Jetsons will all connect to one of the other corresponding Jetsons that has RabbitMQ operating as the main communication server. Each Jetson will be a assigned to its own queue while being able to send messages

to queues assigned to other Jetsons. If there is at least one message in the queue, the corresponding Jetson reads the message, compares it to the vehicle data present to far, then updates the vehicle count respectively. If there are no messages in a queue but there are new vehicles seen in the video frame, the corresponding Jetson will send the autoencoded representation of the new vehicles to the other queues.

5.2.5 Vehicle Tracking

When the autoencoded representation of a vehicle is received by a device, it is compared with every autoencoded vehicle in the current view using Euclidean distance. The vehicle with the smallest Euclidean distance is assigned the same ID as the received vehicle. This is based on the assumption that an autoencoded vehicle viewed from one angle will be similar to the same autoencoded vehicle viewed from a different angle. Once the vehicle IDs are assigned, the tracking uses the same Kalman tracker from DiDi1.

Chapter 6

Evaluation

6.1 Description

As mentioned previously, the initial goal of our project was to quantize five state-of-the-art machine learning algorithms for vehicle counting and tracking so that they can run in real-time on lower-cost IoT devices. Thus, in order to evaluate our algorithms, we wanted to ensure that we were able to reduce the amount of time it took for our algorithms to analyze data while simultaneously barring minimal loss in accuracy due to a floating-point precision drop and implementation of optimized TensorRT engines. All evaluation was completed on an Nvidia Titan Xp.

6.2 Results

As seen in Table 6.1, DiDi1 originally utilized a YOLOv3-416 model that achieved a mean average precision (mAP) of 55.3 on an Intersection over Union (IoU) of 0.5 [8]. In addition, this algorithm originally processed videos between 45-50 frames per second (FPS). DiDi2 originally ran on a NAS-FPN-640 model that is able to achieve a mAP of 33.0 on multiple IoU thresholds between 0.5 and 0.95 in increments of 0.05 (0.5, 0.55, 0.6, 0.65, ..., 0.95) [29] but analyzed videos at a much lower rate of 10-12 FPS.

To evaluate our quantized algorithms, we used the data set from the COCO 2017 validation images and the ground truths and predictions from the COCO 2017 train/validation annotations. In addition, we utilized a script from the same GitHub repository that contained the quantized YOLO models we implemented to evaluate and retrieve their respective mAPs. Please refer to Appendix B.1.1 for the source code.

To our surprise, the quantized versions of DiDi1 and DiDi2 not only performed much faster than their original models, but also happened to achieve higher accuracy as well. DiDi1 achieved nearly 5% higher accuracy than its original counterpart and roughly a 1.75x speedup in FPS. DiDi2 was able to get 6% higher mAP and over 2x faster processing than the original model. Although these achievements are not what we predicted, these can likely be explained by digging deeper into the factors that contribute to this increased performance.

For evaluating the quantized algorithms, we only used the COCO validation data set from 2017. In contrast, the

original DiDi1 model was evaluated with various Pascal VOC data sets between 2007-2012 [8] and the original DiDi2 model was evaluated on the COCO training and validation data sets from 2017 [29]. In other words, the increase in mAP for our quantized models can largely be attributed to a difference in the data sets that were used; the original models used thousands of more images and annotations to retrieve more accurate and realistic mAPs. If we were to re-evaluate the quantized algorithms on the same images and annotations of the original models, we predict that there may be a drop in detection accuracy.

In conclusion, from these results, although DiDi2 does have a higher accuracy overall, DiDi1 may be more useful in higher quality videos that have a higher FPS. In addition, it would appear that the model precision change to FP16 does not seem to cause any degradation to detection accuracy.

Algorithm	**mAP @ IoU=0.5**	**mAP @ IoU=0.5:0.95**	**FPS**
Original DiDi1 (YOLOv3-416)	55.3	NA	45-50
Quantized DiDi1 (YOLOv3-288, FP16)	60.1	33.0	75-80
Original DiDi2 (NAS-FPN-640)	NA	39.9	10-12
Quantized DiDi2 (YOLOv4-416, FP16)	69.9	45.9	22-26

Table 6.1: Evaluation of Detection for DiDi1 and DiDi2 Algorithms

Chapter 7

Ethical Concerns

7.1 Security

With any similar project, there will be data securities that need to be ensured. Since our project inherently deals with real-time traffic camera footage, we would need to protect the privacy of those who are in the camera feed if our project becomes utilized in public traffic intersections. If the our project is not properly secured, it will be exposed to various network vulnerabilities and will be susceptible to malicious attacks, resulting in a failure to protect civilians' privacy. According to the ACM Code of Ethics [30], we are responsible for respecting the privacy of all. Therefore, if our system is not able to protect civilians' or users' privacy, then we bear full responsibility for our development decisions and must face any consequences.

Since RabbitMQ is an open-source message broker, we will need to ensure that non-authorized users are not able to access our data by setting up key-based or password-based messages. Since all IoT devices are intended to be setup on the same network, if a malicious user were to hack into one or multiple devices, they could intercept all of the vehicle detection and tracking data, intentionally force errors upon the traffic analysis, and/or spy on civilians. Thus, we must take careful steps to ensure that only authorized users are able to access our system and that data sent between our devices is encrypted.

7.2 Social

Since our project is aimed to assist in providing real-time traffic signal patterns for applications and future road surveyors, we must ensure that these results are continuously accurate and sufficient. An incorrect analysis of traffic intersections could result in misinterpreting traffic patterns and confusing users of which roads are busier than others. In other words, if our vehicle detection and tracking algorithms yield poor results, our project could end up doing more harm than good for society. The hope for our project is to help other engineers to improve and create better roads and pathing to alleviate heavy traffic. This will help reduce carbon emissions created while cars sit idle at intersections and save individuals times by shortening their commute times.

Chapter 8

Limitations and Challenges

8.1 Quantization

As mentioned previously, we initially began our project with the goal to quantize five state-of-the-art algorithms: DiDi1, DiDi2, Baidu, ENGIE, and Zero-Virus. Unfortunately, as we began to work on quantization and worked to understand how to implement it in our algorithms, we faced several adversities and difficulties in successfully doing so.

Firstly, quantization is a relatively new concept in machine learning, without much development or research on it quite yet. Our team had a hard time finding research papers and resources that contained much documentation and tutorials on how to fully implement quantization into a given machine learning algorithm. Second, the algorithms that we were focusing on had little to no documentation available. Thus, we had to manually interpret thousands of lines of code by ourselves to fully understand the logic and functionality of these algorithms.

In summary, our team spent the majority of our project attempting to quantize five state-of-the-art machine learning algorithms for vehicle counting and tracking. Since the concept of quantization is relatively new and there was a large absence of documentation for these algorithms, we ended up only successfully quantizing three out of the five algorithms: DiDi1, DiDi2 and Baidu.

8.2 System Architectures

Our project can be divided into two parts of development between two different systems. The first system is a Ubuntu 18.04 server running on the X86-64 architecture. This server has two Titan Xp GPUs, and was used to quantize and test our algorithms before we transferred them to the IoT network. The second system we developed on was the IoT network of three Nvidia Jetson NX devices. The Jetson NX also runs on Ubuntu 18.04, but runs on a ARM64 architecture. This difference in system architecture led to an additional workload of having to duplicate the environment for each algorithm for both the server and the Nvidia Jetson IoT network. This entails installing the software dependencies for specific deep learning frameworks and installing required libraries for the ARM64 architecture which is completed

differently than for the X86-64 architecture. One of the problems with this is that there can be incompatibilities between the two architectures for using certain software dependencies. For example, we implemented quantization on the Baidu object detector on the X86-64 server. However, we realized that the object detection model that Baidu uses is incompatible running on ARM64 architecture, which is our desired final system, so our implementation is unusable. Part of this is due to the popularity of the Baidu object detector, which is less commonly used compared to the TensorFlow deep learning framework which is compatible on all of our system architectures.

8.3 Remote Collaboration

Due to the COVID-19 pandemic, the entire 2020-2021 academic school year was completed through remote learning. In turn, the entire design and implementation process was completed remotely. Originally, when we received our three Nvidia Jetson Xavier NX kits, we shipped them to the houses of three different members in order to better distribute the workload amongst our team for those that wanted to work directly on the devices. However, many issues arose that we did not anticipate at the start of this project.

One common issue was internet instability. In order to work on the Jetsons remotely, members of our team had to connect to the devices through a secure shell protocol (SSH), which allowed us to securely remote login from our personal computers. Unfortunately, in a handful of instances, different members of our team experienced significant internet outages in increments of one or two days. We had to pause most of our work and development during these time periods, which limited productivity.

Another issue that we did not anticipate was the need to have all Jetson devices on the same local network in order for our communication protocol (RabbitMQ) successfully working and functioning between each other. Alternatively, we could have attempted to setup a VPN and have all three Jetsons on the same private network but this would have created much more work than necessary and potentially more issues, especially considering our time constraints. Therefore, we ended up shipping all of the Jetson kits to one member of our team in order to get RabbitMQ properly sending and receiving information between our devices.

Chapter 9

Future Work

9.1 Current State

Our team was able to use traffic camera footage from multiple angles at a single intersection to identify and track cars that would have been lost and re-identified incorrectly due to obstructions. The two Jetsons are able to communicate over the same network and transfer limited data. In order for our system to work in real time, we quantized our models so that the algorithms that we used were able to run with sufficient frames per second on our Jetsons. The system was implemented with the state-of-the-art object detection model YOLOv3. This allowed for bounding boxes and confidence scores for each car or truck that enters the frame. We utilized footage from the 2020 AI City Challenge in order to train our algorithms.

9.2 Next Steps and Long-Term Vision

The next steps of the project would entail developing RabbitMQ to transfer traffic camera data from on Jetson to another, incorporating vehicle counting, and getting the Jetsons to run the system in real time out in the field. Our next step to improve our network would be to implement RabbitMQ to allow for more than two Jetsons to concurrently communicate with each other. It would be desirable to connect our Jetsons to camera traffic cameras so that they can consistently and constantly monitor and collect data over a long period of time. Another goal is to add to the project so that the Jetsons can communicate over multiple networks instead of only running together on a single network.

We also could improve upon our vehicle re-identification system, as it sometimes assigns vehicle IDs to the wrong vehicle, or assigns the same ID to multiple vehicles. Training an SVM [31] instead of using raw Euclidean distance could place emphasis on certain features extracted from the autoencoder. Other triplet loss-based methods [16, 32] could remove the need for an autoencoder altogether, potentially improving re-identification accuracy.

Chapter 10

Conclusion

10.1 What We Have Learned

Quantization of machine learning algorithms is a relatively new topic in the field of machine learning. This fact became known in the early phases of this project when we were tasked to quantize five algorithms but could only quantize two of them successfully. Not only did this provide a small window into leading-edge research in our career paths, it taught us lessons about being more adaptive in the software engineering design process. For a majority of Winter quarter, we spent our time trying to quantize all of the existing algorithms. As a result, we had fallen behind schedule of the original deadlines that we had set for ourselves. There should have been a point to abandon the algorithms we could not quantize earlier on.

Due to the coronavirus pandemic, we were limited in our ability to collaborate with each other as much as in person. Unable to have a workplace where all group members could meet, resources including the Jetson developments were also spread out, severely delaying the final implementation of our project. It became apparent that there was a lack of communication between group members where there should have been when working in a remote environment as opposed to on-campus. As the year went on, communication had certainly improved but even then it was a struggle bring the final project together.

10.2 Why it is Important

Multi-Camera Traffic Intersection Analysis in real time would benefit humanity by fixing some of the traffic problems caused by overpopulation. Due to overpopulation, many cities are overcrowded with cars and other forms of public transportation. By easing the flow of traffic, there would be faster and safer commutes for everyone. Additionally, data collected from this experiment will be used to encourage learning about traffic congestion in order to make cities safer and more efficient.

Futhermore, carbon emissions from cars, motorcycles, and trucks in the world would be reduced significantly since there would be less idle time waiting at intersections. With the increased flow of traffic in mind, cities can design roads

and intersections in highly populated and congested areas. In the end, drivers could get to places they want to go easier while saving the environment and time.

Since our projects utilizes Jetson NX development kits on a local network, we are able to send and receive data in real time, which could be used for traffic lights and to collect accurate data over long periods of time. This implementation would also save the cost of using physical sensors such as induction loops.

Appendix A

User Manual: How to Setup

A.1 Installing and Upgrading Libraries

In this section, we would be installing libraries that is necessary for running our Didi1 algorithm.

1. sudo apt-get update
2. sudo apt-get upgrade
3. sudo apt-get install python3-dev python3-pip
4. sudo apt-get install llvm-8*
5. export LLVM_CONFIG=/usr/bin/llvm-config-8
6. sudo pip3 install -U pip
7. sudo pip3 install Cython==0.29.22
8. sudo pip3 install numba==0.46.0
9. sudo pip3 install llvmlite==0.32.1
10. sudo pip3 install scikit-learn==0.21.2
11. sudo pip3 install tqdm==4.33.0
12. sudo pip3 install ffmpeg==1.4
13. sudo pip3 install ffmpeg-python==0.2.0
14. sudo pip3 install pandas==0.22.0
15. sudo pip3 install onnx==1.4.1
16. sudo pip3 install onnxruntime==1.7.0
17. sudo pip3 install numpy==1.18.1
18. sudo pip3 install setuptools==53.0.0
19. sudo pip3 install testresources==2.0.1

A.2 Installing Pytorch 1.7.0

We built Pytorch 1.7.0 from Nvidia's website (https://forums.developer.nvidia.com/t/pytorch-for-jetson-version-1-8-0-now-available/72048).

1. cd
2. export OPENBLAS_CORETYPE=ARMV8
3. wget https://nvidia.box.com/shared/static/cs3xn3td6sfgtene6jdvsxlr366m2dhq.whl -O torch-1.7.0-cp36-cp36m-linux_aarch64.whl
4. sudo apt-get install python3-pip libopenblas-base libopenmpi-dev
5. sudo pip3 install torch-1.7.0-cp36-cp36m-linux_aarch64.whl

A.3 Installing torchvision 0.8.0

We downloaded torchvision 0.8.0 from the github source (https://github.com/pytorch/vision torchvision).

1. cd
2. sudo apt-get install libjpeg-dev zlib1g-dev libpython3-dev libavcodec-dev libavformat-dev libswscale-dev
3. git clone –branch v0.8.0 https://github.com/pytorch/vision torchvision
4. export BUILD_VERSION=0.8.0
5. cd torchvision/
6. sudo python3 setup.py install

A.4 Building Our Pre-trained Model

We would be following JKJung's github (https://github.com/jkjung-avt/tensorrt_demos) and using his yolov3_288 and yolov4_416 pretrained object detection models. Note: The NX Jetson Xavier have already installed Tensorrt. If the user's local machine does not have TensorRT, it can be installed locally following NVIDIA's TensorRT webpage (https://docs.nvidia.com/deeplearning/tensorrt/install-guide/index.html).

A.4.1 Installing Pycuda

1. cd
2. mkdir project
3. cd project/
4. git clone https://github.com/jkjung-avt/jetson_nano.git
5. cd jetson_nano
6. ./install_basics.sh
7. source $HOME/.bashrc
8. sudo apt install build-essential
9. sudo apt install make
10. sudo apt install cmake

11. sudo apt install cmake-curses-gui
12. sudo apt install git
13. sudo apt install g++
14. sudo apt install pkg-config
15. sudo apt install curl
16. sudo apt install libfreetype6-dev
17. sudo apt install libcanberra-gtk-module
18. sudo apt install libcanberra-gtk3-module
19. ./install_protobuf-3.8.0.sh

A.4.2 Installing Tensorflow

1. sudo apt install y libhdf5serialdev hdf5tools libhdf5dev zlib1gdev zip libjpeg8dev liblapackdev libblasdev gfortran
2. sudo pip3 install U future==0.18.2 mock==3.0.5 h5py==2.10.0 keras_preprocessing==1.1.1 keras_applications==1.0.8 gast==0.2.2 futures pybind11
3. sudo pip3 install pre extraindexurl https://developer.download.nvidia.com/compute/redist/jp/v44 tensorflow==1.15.2

A.4.3 Building YOLOv3 Model

1. cd
2. cd project/
3. git clone https://github.com/jkjung-avt/tensorrt_demos.git
4. cd tensorrt_demos/ssd
5. ./install.sh
6. ./build_engines.sh
7. cd ../plugins
8. vi Makefile

   # Make changes in the Makefile and locate the environment variables $TENSORRT_INCS and $TENSORRT_LIBS. Change the values according to the path where the TensorRT dynamic libraries are stored in. The $TENSORRT_INCS path should have "trtexec" dynamic library and the $TENSORRT_LIBS path variable should have libnvinfer.so, libnvparsers.so, and other TensorRT dynamic libraries. Since our NX Jetson Xavier have already installed Tensorrt, our environment variables like this:

   TENSORRT_INCS=-I"/usr/src/tensorrt/bin"

   TENSORRT_LIBS=-L"/usr/lib/aarch64-linux-gnu"
9. # Move libyolo_layer.so into our Real-Time Multi-Camera Traffic project under folder plugins/.
10. make
11. Check if pycuda has successfully been installed.
12. cd tensorrt_demos/yolo

13. ./download_yolo.sh
14. python3 yolo_to_onnx.py -m yolov3-288
15. python3 onnx_to_tensorrt.py -m yolov3-288
16. # Move the yolov3-288.trt output into our Real-Time Multi-Camera Traffic project under folder yolo/.

A.5 Setting up Rabbit MQTT Server

A.5.1 Installing Rabbit MQ

This tutorial assumes that the user is installing this on Ubuntu and other Linux-based architectures.

1. sudo apt-get install apt-transport-https
2. sudo apt-key adv –keyserver ”hkps://keys.openpgp.org” –recv-keys ”0x0A9AF2115F4687BD29803A206B73A36E6026DFCA”
3. sudo apt-key adv –keyserver ”keyserver.ubuntu.com” –recv-keys ”F77F1EDA57EBB1CC”
4. curl -1sLf ’https://packagecloud.io/rabbitmq/rabbitmq-server/gpgkey’ — sudo apt-key add -
5. sudo tee /etc/apt/sources.list.d/rabbitmq.list <<EOF
6. deb http://ppa.launchpad.net/rabbitmq/rabbitmq-erlang/ubuntu bionic main
7. deb-src http://ppa.launchpad.net/rabbitmq/rabbitmq-erlang/ubuntu bionic main
8. deb https://packagecloud.io/rabbitmq/rabbitmq-server/ubuntu/ bionic main
9. deb-src https://packagecloud.io/rabbitmq/rabbitmq-server/ubuntu/ bionic main
10. EOF
11. sudo apt-get update -y
12. sudo apt-get install -y erlang-base
13. erlang-asn1 erlang-crypto erlang-eldap erlang-ftp erlang-inets
14. erlang-mnesia erlang-os-mon erlang-parsetools erlang-public-key
15. erlang-runtime-tools erlang-snmp erlang-ssl
16. erlang-syntax-tools erlang-tftp erlang-tools erlang-xmerl
17. sudo apt-get install rabbitmq-server -y –fix-missing
18. pip install pika
19. service rabbitmq-server start
20. Enter admin password for your device.

    The server should be running on your local device. One can then use the example files on RabbitMQ’s tutorials on the official website. This, however, does not allow other devices on a local network to connect to the server. This topic will be covered next.

A.5.2 Allow Other Devices on Local Network to Connect to Rabbit MQ Server

1. sudo rabbitmqctl add_user qa1 yourPassword

   This creates an authorized user with the username 'qa1' and password 'yourPassword'.

2. sudo rabbitmqctl add_vhost virtualHost1

   This creates a virtual host with the name 'virtualHost1'.

3. sudo rabbitmqctl set_permissions -p virtualHost1 qa1 ".*" ".*" ".*"

   This allows user 'qa1' permission to read and write within 'virtualHost1'.

4. The username, password, and name of the virtual host will all be used in the functions PlainCredentials, BlockingConnection, and ConnectionParameters within the pika library.

Appendix B

Source Code

B.1 Evaluation Code

This section contains the code that was used to evaluate the quantized versions of DiDi1 and DiDi2 according to their mAP on a specific dataset.

B.1.1 eval_yolo.py

```
"""
Source credit to:
https://github.com/jkjung-avt/tensorrt_demos
"""

import os
import sys
import json
import argparse
import cv2
import pycuda.autoinit  # This is needed for initializing CUDA driver

from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval
from progressbar import progressbar
from utils.yolo_with_plugins import TrtYOLO
from utils.yolo_classes import yolo_cls_to_ssd


HOME = os.environ['HOME']
VAL_IMGS_DIR = HOME + '/data/coco/images/val2017'
VAL_ANNOTATIONS = HOME + '/data/coco/annotations/instances_val2017.json'


def parse_args():
    """Parse input arguments."""
    desc = 'Evaluate mAP of YOLO model'
    parser = argparse.ArgumentParser(description=desc)
    parser.add_argument(
        '--imgs_dir', type=str, default=VAL_IMGS_DIR,
        help='directory of validation images [%s]' % VAL_IMGS_DIR)
```

```
    parser.add_argument(
        '--annotations', type=str, default=VAL_ANNOTATIONS,
        help='groundtruth annotations [%s]' % VAL_ANNOTATIONS)
    parser.add_argument(
        '--non_coco', action='store_true',
        help='don\'t do coco class translation [False]')
    parser.add_argument(
        '-c', '--category_num', type=int, default=80,
        help='number of object categories [80]')
    parser.add_argument(
        '-m', '--model', type=str, required=True,
        help=('[yolov3|yolov3-tiny|yolov3-spp|yolov4|yolov4-tiny]-'
              '[{dimension}], where dimension could be a single '
              'number (e.g. 288, 416, 608) or WxH (e.g. 416x256)'))
    parser.add_argument(
        '-l', '--letter_box', action='store_true',
        help='inference with letterboxed image [False]')
    args = parser.parse_args()
    return args


def check_args(args):
    """Check and make sure command-line arguments are valid."""
    if not os.path.isdir(args.imgs_dir):
        sys.exit('%s is not a valid directory' % args.imgs_dir)
    if not os.path.isfile(args.annotations):
        sys.exit('%s is not a valid file' % args.annotations)


def generate_results(trt_yolo, imgs_dir, jpgs, results_file, non_coco):
    """Run detection on each jpg and write results to file."""
    results = []
    for jpg in progressbar(jpgs):
        img = cv2.imread(os.path.join(imgs_dir, jpg))
        image_id = int(jpg.split('.')[0].split('_')[-1])
        boxes, confs, clss = trt_yolo.detect(img, conf_th=1e-2)
        for box, conf, cls in zip(boxes, confs, clss):
            x = float(box[0])
            y = float(box[1])
            w = float(box[2] - box[0] + 1)
            h = float(box[3] - box[1] + 1)
            cls = int(cls)
            cls = cls if non_coco else yolo_cls_to_ssd[cls]
            results.append({'image_id': image_id,
                            'category_id': cls,
                            'bbox': [x, y, w, h],
                            'score': float(conf)})
    with open(results_file, 'w') as f:
        f.write(json.dumps(results, indent=4))


def main():
    args = parse_args()
    check_args(args)
```

```
    results_file = 'yolo/results_%s.json' % args.model

    yolo_dim = args.model.split('-')[-1]
    if 'x' in yolo_dim:
        dim_split = yolo_dim.split('x')
        if len(dim_split) != 2:
            raise SystemExit('ERROR: bad yolo_dim (%s)!' % yolo_dim)
        w, h = int(dim_split[0]), int(dim_split[1])
    else:
        h = w = int(yolo_dim)
    if h % 32 != 0 or w % 32 != 0:
        raise SystemExit('ERROR: bad yolo_dim (%s)!' % yolo_dim)

    trt_yolo = TrtYOLO(args.model, (h, w), args.category_num, args.letter_box)

    jpgs = [j for j in os.listdir(args.imgs_dir) if j.endswith('.jpg')]
    generate_results(trt_yolo, args.imgs_dir, jpgs, results_file,
                     non_coco=args.non_coco)

    # Run COCO mAP evaluation
    cocoGt = COCO(args.annotations)
    cocoDt = cocoGt.loadRes(results_file)
    imgIds = sorted(cocoGt.getImgIds())
    cocoEval = COCOeval(cocoGt, cocoDt, 'bbox')
    cocoEval.params.imgIds = imgIds
    cocoEval.evaluate()
    cocoEval.accumulate()
    print(cocoEval.summarize())


if __name__ == '__main__':
    main()
```

Bibliography

[1] S. Sunkari, R. Parker, H. Charara, T. Palekar, and D. Middleton, "Evaluation of cost-effective technologies for advance detection," 09 2005.

[2] M. Naphade, S. Wang, D. C. Anastasiu, Z. Tang, M.-C. Chang, X. Yang, L. Zheng, A. Sharma, R. Chellappa, and P. Chakraborty, "The 4th ai city challenge," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, p. 2665–2674, June 2020.

[3] J. Mirthubashini and V. Santhi, "Video based vehicle counting using deep learning algorithms," in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pp. 142–147, 2020.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," *Lecture Notes in Computer Science*, p. 21–37, 2016.

[5] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015.

[6] L. Lou, Q. Zhang, C. Liu, M. Sheng, J. Liu, and H. Song, "Detecting and counting the moving vehicles using mask r-cnn," in *2019 IEEE 8th Data Driven Control and Learning Systems Conference (DDCLS)*, pp. 987–992, May 2019.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016.

[8] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018.

[9] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.

[10] K. Zhang, H. Rren, Y. Wei, and J. Gong, "Multi-target vehicle detection and tracking based on video," in *2020 Chinese Control And Decision Conference (CCDC)*, pp. 3317–3322, 2020.

[11] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," 02 2001.

[12] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "High-speed tracking with kernelized correlation filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, p. 583–596, Mar 2015.

[13] A. Lukežič, T. Vojíř, L. Čehovin Zajc, J. Matas, and M. Kristan, "Discriminative correlation filter tracker with channel and spatial reliability," *International Journal of Computer Vision*, vol. 126, p. 671–688, Jan 2018.

[14] Z. Kalal, K. Mikolajczyk, and J. Matas, "Tracking-learning-detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 7, pp. 1409–1422, 2012.

[15] Z. Kalal, K. Mikolajczyk, and J. Matas, "Forward-backward error: Automatic detection of tracking failures," in *2010 20th International Conference on Pattern Recognition*, pp. 2756–2759, 2010.

[16] A. Li, L. Luo, and S. Tang, "Real-time tracking of vehicles with siamese network and backward prediction," in *2020 IEEE International Conference on Multimedia and Expo (ICME)*, pp. 1–6, 2020.

[17] H. Liang, H. Song, H. Li, and Z. Dai, "Vehicle counting system using deep learning and multi-object tracking methods," *Transportation Research Record*, vol. 2674, no. 4, pp. 114–128, 2020.

[18] K.-H. N. Bui, H. Yi, and J. Cho, "A multi-class multi-movement vehicle counting framework for traffic analysis in complex areas using cctv systems," *Energies*, vol. 13, p. 2036, Apr 2020.

[19] A. A. P. Mutharpavalar and P. Sebastian, "Measuring of real-time traffic flow using video from multiple ip-based cameras," in *2019 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, pp. 186–191, 2019.

[20] Z. Tang, M. Naphade, M.-Y. Liu, X. Yang, S. Birchfield, S. Wang, R. Kumar, D. Anastasiu, and J.-N. Hwang, "Cityflow: A city-scale benchmark for multi-target multi-camera vehicle tracking and re-identification," 2019.

[21] Z. Liu, W. Zhang, X. Gao, H. Meng, X. Tan, X. Zhu, Z. Xue, X. Ye, H. Zhang, S. Wen, and E. Ding, "Robust movement-specific vehicle counting at crowded intersections," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 2617–2625, 2020.

[22] A. Ospina and F. Torres, "Countor: Count without bells and whistles," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[23] Z. Wang, B. Bai, Y. Xie, T. Xing, B. Zhong, Q. Zhou, Y. Meng, B. Xu, Z. Song, P. Xu, R. Hu, and H. Chai, "Robust and fast vehicle turn-counts at intersections via an integrated solution from detection, tracking and trajectory modeling," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 2598–2606, 2020.

[24] L. Yu, Q. Feng, Y. Qian, W. Liu, and A. G. Hauptmann, "Zero-virus: Zero-shot vehicle route understanding system for intelligent transportation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[26] T. Danka, "How to accelerate and compress neural networks with quantization," 2020.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Internal Representations by Error Propagation*, p. 318–362. Cambridge, MA, USA: MIT Press, 1986.

[28] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2014.

[29] G. Ghiasi, T.-Y. Lin, R. Pang, and Q. V. Le, "Nas-fpn: Learning scalable feature pyramid architecture for object detection," 2019.

[30] "Acm code of ethics." https://www.acm.org/code-of-ethics. Accessed: 2021-05-09.

[31] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, p. 273–297, Sept. 1995.

[32] X. Zhu, Z. Luo, P. Fu, and X. Ji, "Voc-reid: Vehicle re-identification based on vehicle-orientation-camera," 2020.