REAL-TIME VEHICLE DETECTION AND TRACKING FOR WORK ZONE SAFETY

by

Colin Parks

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Computer Engineering

Charlotte

2022

Approved by:

_____
Dr. Hamed Tabkhivayghan

_____
Dr. Omid Shoghli

_____
Dr. Jim Conrad

ABSTRACT

COLIN PARKS. Real-time Vehicle Detection and Tracking for Work Zone Safety.
(Under the direction of DR. HAMED TABKHIVAYGHAN)

Computer vision and deep learning are rapidly evolving fields being implemented for a variety of different applications, including object detection and tracking for autonomous vehicles. While today's technology is not at the level of full automation, many vehicles contain advanced driver assistance systems which are used to help protect drivers, passengers, and pedestrians by anticipating and avoiding incoming danger. This comes in the form of collision warning and collision intervention, which alerts the driver and performs automatic braking in the event of a forward, backward, or side collision [1].

Highway work zones are dangerous for both the workers and drivers of incoming vehicles, with the majority of deaths in work zones being attributed to incoming vehicles colliding with workers [2]. As this is similar to the autonomous vehicle problem, a comparable system could be created to address the safety concern for work zones. Anticipating an accident in a highway work zone is challenging due to the speed and volume of passing vehicles, making alerting workers of a potential risk ahead of time difficult. This requires the system to be able to detect vehicles from a great enough distance to make predictions in time, which would require an efficient hardware and software implementation.

This paper proposes a vehicle detection and tracking system operating in real-time on an embedded edge device. By tracking the position of vehicles over time, their trajectory can be estimated by a separate algorithm/pipeline. This pipeline would also be responsible for analyzing trajectories to detect anomalous behavior near the work zone. The edge device, an NVIDIA Jetson AGX Xavier, would be connected to a camera facing from the edge of the work zone towards incoming traffic from the

road/highway.

To find the optimal solution for efficient vehicle detection, a few popular object detection networks were implemented in PyTorch and tested on the edge device, with the best performing network being used in the final pipeline. Several optimizations were applied to the network, with the most significant being the implementation of TensorRT, an inference engine from NVIDIA. TensorRT aims to accelerate inference tasks on NVIDIA GPUs by applying several optimizations to the network. A tracking algorithm/network was also implemented in PyTorch and subject to the same optimizations as the detection pipeline.

The baseline of each network was benchmarked on the edge device along with each individual improvement/optimization. The optimized detection and tracking pipelines were combined to form the overall vehicle detection and tracking pipeline. The results demonstrate the major advantages of utilizing TensorRT and NVIDIA hardware for efficient inference, leading to major improvements in the end-to-end throughput of the pipeline while maintaining accuracy and power efficiency.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES ix

LIST OF ABBREVIATIONS x

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: LITERATURE REVIEW 4

2.1. Edge Computing 4

2.2. Deep Learning 4

2.2.1. Computer Vision 5

2.2.2. Neural Networks 5

2.3. Object Detection Networks 6

2.3.1. MobileNet 6

2.3.2. EfficientDet 8

2.3.3. YOLO 9

2.4. Object Tracking Networks 10

2.4.1. DeepSort 10

2.5. Machine Learning Frameworks 11

2.5.1. PyTorch 11

2.5.2. TensorRT 13

2.6. Datasets 14

2.6.1. COCO 15

2.6.2. BDD100K 16

CHAPTER 3: Approach 18

3.1. Vehicle Detection 18

3.1.1. Pre-Processing 20

3.1.2. Inference 21

3.1.3. Post-Processing 22

3.2. Vehicle Tracking 23

3.3. Methodology 23

3.3.1. Baseline Network 24

3.3.2. Utilizing the GPU 25

3.3.3. Optimizing Inference with TensorRT 26

3.3.4. Using a Specialized Dataset: BDD100K 30

3.3.5. Integration of Vehicle Tracking with DeepSort 30

CHAPTER 4: RESULTS 35

4.1. Hardware 35

4.1.1. Software Configuration 36

4.2. Benchmarks 37

4.3. Baseline Networks 37

4.3.1. Inference on the CPU 38

4.3.2. Inference on the GPU 39

4.4. Optimizing Inference with TensorRT 41

4.5. YoloV4 Trained on BDD100K 44

4.6. Integration of DeepSort for Vehicle Tracking 45

4.6.1. Optimizing DeepSort with TensorRT 46

4.7. Vehicle Detection and Tracking Pipeline 49

4.7.1. Power Consumption 51

CHAPTER 5: CONCLUSION 54

5.1. Future Work 55

REFERENCES 57

LIST OF TABLES

TABLE 4.1: Inference Latency at 512x512: CPU vs. GPU 41

TABLE 4.2: System Configuration of each Power Mode 52

LIST OF FIGURES

FIGURE 3.1: Edge Device and Camera Setup 19

FIGURE 3.2: Example Image Captured from Camera Setup 20

FIGURE 3.3: Vehicle Detection: Pre-Processing Stage 21

FIGURE 3.4: Vehicle Detection: Post-Processing Stage 23

FIGURE 3.5: PyTorch Initialization and Inference Process 25

FIGURE 3.6: TensorRT Initialization and Inference Process 29

FIGURE 3.7: Vehicle Detection Pipeline using YoloV4 and TensorRT 29

FIGURE 3.8: DeepSort Pipeline using TensorRT 32

FIGURE 3.9: Overall Vehicle Detection and Tracking Pipeline 33

FIGURE 4.1: Latency Measurements at Standard Resolution (256x256) on CPU 38

FIGURE 4.2: Inference Latency Across Multiple Resolutions on CPU 39

FIGURE 4.3: Throughput Measurements at Standard Resolution (256x256) on GPU 41

FIGURE 4.4: Inference Throughput Across Multiple Resolutions on GPU 42

FIGURE 4.5: Inference Throughput Across Multiple Resolutions for YoloV4: PyTorch vs. TensorRT 43

FIGURE 4.6: Latency Comparison of each YoloV4 Pipeline Stage: PyTorch vs. TensorRT 44

FIGURE 4.7: Latency Comparison of each YoloV4 Pipeline Stage: COCO vs. BDD100K 45

FIGURE 4.8: Distribution of the Number of Detections in Frame for Input Video 47

FIGURE 4.9: Latency Comparison of each DeepSort Stage: PyTorch vs. TensorRT 47

FIGURE 4.10: Average DeepSort Inference Latency over Number of Detections: PyTorch vs. TensorRT 48

FIGURE 4.11: Average DeepSort Inference Throughput Across Multiple Batch Sizes 49

FIGURE 4.12: Average Latency of each Pipeline Stage and End-to-End 50

FIGURE 4.13: Example Output Image from Pipeline with Only Cars 51

FIGURE 4.14: Example Output Image from Pipeline with Cars and Trucks 51

FIGURE 4.15: Power Consumption of each Power Mode 52

LIST OF ABBREVIATIONS

ANN Artificial Neural Network

CNN Convolutional Neural Network

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

FP16 16-bit Floating Point

FP32 32-bit Floating Point

FPN Feature Pyramid Network

FPS Frames Per Second

GPU Graphics Processing Unit

INT8 8-bit Integer

IOU Intersection-Over-Union

mAP Mean Average Precision

MOT Multiple Object Tracking

NMS Non-Maximum Suppression

ONNX Open Neural Network Exchange

ReLU Rectified Linear Unit

SDK Software Development Kit

SORT Simple Online and Realtime Tracking

YOLO You Only Look Once

CHAPTER 1: INTRODUCTION

Work zones can be dangerous and pose a threat to workers on-site, pedestrians, and the driver and passengers of incoming vehicles. The number of fatal work zone crashes have been on a steady rise over the last decade, increasing by 46% from 2010 to 2020, while the number of worker fatalities has remained around the same at an average of 56 deaths per year [3]. From 2011 to 2017, 76% of highway work zone fatalities were caused by a transportation event, 60% of which involved a worker being struck by an incoming vehicle [2]. Highway work zones are especially dangerous for workers due to the high speeds of passing vehicles, and in cases of heavy traffic where a significant amount of vehicles are passing by. Workers also aren't going to be as alert while they are at work, resulting in them being unable to react in time to avoid a collision.

Deep learning and computer vision have been used for many applications, and is rapidly evolving in the embedded and edge computing space. Object detection specifically is used in applications such as vehicle detection for autonomous vehicles, person detection for security and surveillance, and medical feature detection for images and scans in healthcare. A vehicle anomaly/accident prediction system could be useful in improving work zone safety by predicting accidents ahead of time and warning pedestrian workers.

This overall system would need five components: vehicle detection, vehicle tracking, trajectory and path prediction, anomaly detection, and reporting. Once a vehicle has been detected and tracked for a short duration of time, any anomalies such as high speeds or quickly changing speed or lanes can be reported and used to alert pedestrians in the work zone. This system should significantly reduce the number of work zone fatalities. This thesis will focus on the first two components: vehicle detection and

tracking.

The challenge with this approach is keeping the total system latency low enough to make predictions in time. Vehicles entering or passing work zones can be going fast, especially so for highway work zones. The NC Department of Transportation Work Zone Speed Limit Guidelines state that on freeways with speed limits exceeding 65 MPH, a typical speed limit reduction is 10 MPH around the work zone [4]. That would make the standard speed limit for the area around a highway work zone 55 MPH. If the system is to detect and track incoming vehicles, it would need to be able to accurately detect the vehicles from a sufficient distance to make a prediction in time. Even if the system is able to make fast predictions, if the vehicle isn't detected far enough away from the work zone then it would be too late by the time a prediction is made. Therefore, a trade-off must be made between accuracy of predictions and processing speed.

This thesis proposes an efficient real-time system for detecting and tracking vehicles from the perspective of a highway work zone with the purpose of alerting workers and pedestrians of potential danger from incoming vehicles. A pipeline for vehicle detection and tracking was constructed and benchmarked with several different optimizations to demonstrate an uplift in performance. Because edge computing is advantageous for real-time applications, an edge device handles all processing from video capture to risk reporting if the system predicts an accident. The edge device used in the system is the NVIDIA Jetson AGX Xavier [5], an efficient embedded device from NVIDIA. This device comes with an NVIDIA GPU, which supports the use of TensorRT for efficient inference, which is shown to be the bottleneck of the system.

By implementing an efficient vehicle tracking pipeline running on an edge device, this would facilitate the estimation of vehicle trajectories. Therefore, the overall anomaly detection system would rely heavily on fast and accurate detections of ap-

proaching vehicles as well as the ability to track their movements over time. This vehicle detection and tracking pipeline running in real-time on an edge device will be the focus of this thesis.

The major contributions of this thesis are the following:

- Implements a few different networks using PyTorch to run on an edge device and evaluates the performance of each.
- Evaluates the performance of each network with several benchmarks to see trade-off between execution speed and accuracy.
- Converts selected networks to use TensorRT for optimized inference.
- Evaluates the performance impact using TensorRT instead of PyTorch.
- Proposes optimized vehicle detection and tracking pipeline for real-time performance on edge device.
- Benchmarks the power consumption of the overall pipeline over the different edge device power modes.

CHAPTER 2: LITERATURE REVIEW

2.1 Edge Computing

The core idea of edge computing is to keep computing close to the source of data, which differs from cloud computing which offloads computing to an external device that isn't necessarily close to the source of data. With edge computing, data is stored and processed without interacting with a cloud computing platform. This gives edge computing the advantage in processing speed since it doesn't require any intermediate data transmissions, as all processing is done locally. This makes edge computing ideal for fields such as autonomous driving, which rely on rapid feedback [6].

In addition to the ability to better perform real-time processes, edge computing also has lower cost and is much more energy efficient than cloud computing, since collected data doesn't need to be uploaded to the cloud computing center. Network bandwidth pressure is also reduced for the same reason, which frees bandwidth for communication between local edge devices. Security is also an advantage with edge computing, since all processing is done on-site, no potentially critical data can be lost or leaked [6].

2.2 Deep Learning

Deep learning, sometimes used interchangeably with artificial intelligence and machine learning, is a subset of the two. AI generally involves introducing human intelligence to machines, so that the system may process or make decisions similar to how a human would. Machine learning uses collected data to build analytical models. Deep learning, similar to machine learning, uses large amounts of data to build a multi-layer neural network, which can be used for tasks like prediction, detection,

and classification [7].

The biggest difference between deep learning and traditional machine learning is the performance improvement as the amount of available data grows. Deep learning modeling excels when dealing with large amounts of data because of its multi-layer approach, allowing it to process large amounts of features to build a data-driven model. Because of the dependency on large amounts of data, deep learning algorithms require more computationally expensive operations when training and running inference on a model. These more complex operations are often easily parallelized, taking more advantage of a graphics processing unit (GPU) over a central processing unit (CPU). Thus, deep learning applications typically rely on high-performance hardware with GPUs [7].

2.2.1 Computer Vision

The field of computer vision involves the capture, manipulation, and analysis of data from images and videos. Images are made up of individual pixels with an associated intensity value. Grey scale images have only one intensity value per pixel (one channel), while color images have a red, green, and blue intensity value per pixel (three channel). This data is imperative for training and testing object detection networks, which seek to achieve object localization and classification. Locations of detected objects in an image are described by a bounding box [8].

2.2.2 Neural Networks

Artificial neural networks (ANN) consist of an input layer which takes an input usually in the form of a multidimensional vector, multiple hidden layers, and an output layer that is responsible for making a decision. Each hidden layer contains a set of neurons that are connected to the previous and next layer by weights, whose values are determined during training. Compared to ANNs, convolutional neural networks (CNN) are primarily used for pattern recognition in images. ANNs suffer with image-

focused tasks because of the computational complexity required for working with image data. Datasets with smaller input sizes like the MNIST dataset with input image size of 28 x 28 are suitable for ANNs, but images are typically much larger than this. In that example, each neuron in the first hidden layer will have 728 weights, since the image is of size 28 x 28 and is greyscale, meaning it only has one channel. When considering a network taking inputs of colored images size of 64 x 64, it's neurons will have 12,288 weights in the first layer, making it significantly more computationally complex [9].

2.3 Object Detection Networks

2.3.1 MobileNet

MobileNet is an efficient neural network designed for mobile computer vision tasks. The major innovations in MobileNetV1 were the use of depthwise separable convolution and two model shrinking hyper-parameters. Depthwise separable convolution is used to separate a convolution into a depthwise convolution and pointwise convolution. A standard convolution involves filtering features from convolutional kernels and combining these features to form a new representation. This can be factored into a depthwise convolution which applies a single filter per input channel and then a pointwise convolution, which uses a 1x1 convolution to combine the outputs of the depthwise layer. This depthwise separable convolution leads to a 8 to 9 times decrease in computational cost with a minor decrease in accuracy compared to standard convolutions for MobileNetV1, which uses 3x3 depthwise separable convolutions [10].

The two hyper-parameters that were introduced were width multiplier and resolution multiplier. The width multiplier is responsible for creating thinner models by applying a parameter to reduce the number of input channels and output channels uniformly. This reduction creates smaller and less computationally expensive models, which lead to lower latency at the trade-off of accuracy. The resolution multiplier is used to downscale the input image, which shrinks the representation of each layer

by the same amount. This also reduces the computational cost of the network by decreasing the number of parameters [10].

MobileNetV2 expanded on the first version by introducing a linear bottleneck block with inverted residuals. Bottleneck blocks are a replacement for traditional convolutional blocks which perform a 3x3 convolution for each channel and then non-linear activation function like ReLU. Instead, bottleneck blocks first use a 1x1 convolution to transform the input into a lower-dimension, then performs the 3x3 convolution over the reduced channels (known as the bottleneck layer), before finally transforming it back to the original feature dimension of the input with another 1x1 convolution. The input is also added to the output through a residual connection. They found that the use of the non-linear activation function lead to the loss of information on the affected channels, showing a drop in efficiency. To solve this, the ReLU is removed following the bottleneck layer, creating a linear bottleneck block that maintains information from channels that would have been reduced. The inverted residuals invert the bottleneck block, expanding the input to a higher-dimension instead of reducing by using a shortcut, which was shown to be more memory efficient [11, 12].

The improvements from MobileNetV2 were used in combination with the introduction of a squeeze and excitation block and a new non-linear activation function called swish to form MobileNetV3. The squeeze and excitation block is a module to emphasize important features while suppressing less important features. It does this by performing a squeeze operation, which learns global information by finding channel dependencies, then an excitation operation to emphasize the channel-wise dependencies [13]. The ReLU function was replaced with swish, another non-linear activation function shown to consistently outperform ReLU in terms of accuracy, even when replaced on models that were designed for ReLU [14, 15].

2.3.2 EfficientDet

EfficientDet is a family of object detectors that aims to achieve higher accuracy and better efficiency across a wide variety of different resource constraints by introducing a new method of multi-scale feature fusion. Early object detectors often used backbone networks for extracting feature hierarchies, with the feature pyramid network (FPN) being one of the pioneering works [16]. Feature pyramids are built upon image pyramids, which are pyramids containing an image at different resolutions or scale. The FPN takes a single-scale image and outputs feature maps at multiple levels using a CNN as a backbone, allowing for the recognition of objects at different scales. It does this by using a top-down pathway to upsample semantically stronger feature maps from higher pyramid levels, known as feature fusion, which allows lower-level feature maps to contain higher resolution features [17].

The conventional top-down FPN is limited by the one-way information flow, which path aggregation network (PANet) seeks to solve by adding a bottom-up aggregation in addition to the top-down. Going further, NAS-FPN utilizes neural architecture search to find a better multi-scale feature network topology instead of a constant aggregation path, leading to better efficiency at the cost of the computation time of the search and irregular network structure that is difficult to interpret. With the extra bottom-up aggregation path, PANet is able to achive better accuracy that FPN and NAS-FPN, but at the cost of efficiency due to having more parameters and computations. The proposed feature network is called the bi-directional feature pyramid network, or BiFPN. This network applies several optimizations, the most important being the addition of a bi-directional path. While PANet contains a top-down and bottom-up path, BiFPN combines these to form a bi-directional path that acts as a single feature network layer. This feature network layer is repeated multiple times to enable more high-level feature fusion [16].

EfficientNets is a family of CNNs that are the result of scaling a baseline model.

A mobile-size baseline model called EfficientNet-B0 was designed using a neural architecture search with a focus on accuracy and performance (in FLOPS). Larger models were created by scaling the baseline model, resulting in 7 more models from EfficientNet-B1 to B7. The models were scaled using a new compound scaling method proposed by EfficientNet, which uses a compound coefficient to uniformly scale network width, depth, and resolution [18]. The EfficientDet architecture consists of 3 different networks: EfficientNet as the backbone network for multi-scale feature extraction, BiFPN as the feature network, and a shared class and bounding box prediction network which takes input from the last BiFPN layer [16].

2.3.3 YOLO

You Only Look Once (YOLO) is an object detection system that runs a single convolutional network over an image, without the need for a complex pipeline with a backbone network like R-CNN with its region proposal method and classifier. After resizing the image, it is run through the convolutional network with the output being the bounding box and class probability for each detected object. Non-max suppression is used to filter out detections with a confidence value under the specified threshold [19].

Due to the simplicity of the pipeline, there is a massive improvement in efficiency compared to other real-time systems. Because YOLO evaluates the entire image at once, it is able to gather contextual information about each object, reducing the number of errors involving the background of the image compared to Fast R-CNN [20]. YOLO is also able to create a generalized representation of objects and is therefore able to perform well even in cases of unexpected inputs. One shortcoming of YOLO is its accuracy and ability to precisely localize smaller objects [19].

2.4 Object Tracking Networks

2.4.1 DeepSort

The problem of multiple object tracking (MOT) involves detecting objects and tracking them across each frame of a video sequence. Simple online and real-time trackign (SORT) is a proposed method for an efficient MOT solution that would be better suited to real-time applications. SORT works by using a CNN based detection network to detect each object in a frame, then propagates each object state to future frames to create associations. Once a detection has been created, the location of the object in the next frame, known as the target, is estimated using a linear constant velocity model. If a detection for the current frame can be associated with any of the targets from previous frames, then the bounding box of the object is used to update the target state in which velocity components are obtained using Kalman filtering. This is done to get a more accurate estimated bounding box for the target, but if no association can be for a given target then it's future location is determined using the linear constant velocity model [21].

For associating objects with targets from previous frames, the intersection-over-union (IOU) is calculated between each detection and existing target bounding box using the Hungarian algorithm. The IOU represents the amount of overlap between two bounding boxes. A threshold is set to reject associations in which the IOU between the object and target are below a certain value. Using the IOU distance to form associations is also shown to better handle short-term occlusion from overlapping targets. In the case of occlusion, the occluding target would have it's location updated while the covered target would be unchanged since no association can be made [21].

To create trackers for new objects, any detection with an IOU distance less than the set threshold is treated as a new untracked object. This new tracker is initialized using the predicted bounding box and its velocity set to 0 as it has been unobserved until this point. New trackers must also be associated with detections enough be-

fore being established in order to reduce false positives. Old trackers are removed when no associations have been made for a certain number of frames, preventing the accumulation of a large amount of trackers which would lead to localization errors [21].

SORT was evaluated alongside other trackers on the MOT benchmark, showing that methods with the highest accuracy also tended to be the least efficient, while methods that achieved high efficiency resulted in low accuracy. Compared to these methods, SORT is able to achieve high performance without compromising on accuracy or speed [21].

DeepSort is an extension of SORT with the introduction of a deep association metric. SORT achieved good performance, but suffered from a high number of identity switching and issues with tracking through occlusion. To overcome this drawback, the association metric used in SORT is replaced by a CNN trained on a re-identification dataset. Compared to the previous association metric, this CNN based metric is able to learn based on object location and appearance. The addition of a deep appearance descriptor helps to alleviate the occlusion and ID switching issues by taking advantage of the appearance of an object in addition to its location, making it able to more accurately distinguish between detections. Even with the addition of a deep neural network, DeepSort is able to achieve high efficiency tracking like the original SORT, but requires the use of a GPU [22].

2.5 Machine Learning Frameworks

2.5.1 PyTorch

PyTorch is a Python library that aims to provide a deep learning framework that has both high usability and high execution speed. The library was created with four major design principles in mind to achieve its high usability. First, it must abide by the standards and established design goals of the Python ecosystem. Second, it should make writing models easier by providing researchers with an intuitive API

that handles some of the complexities of machine learning. Third, it must provide an acceptable trade-off between performance and usability, and provide tools for researchers to tweak code execution to find their own performance improvements. Lastly, it should remain simple enough so that researchers can allocate more of their time to adding new features and adapting to new situations in order to keep up with the rapidly evolving field of AI and machine learning [23].

In PyTorch, network layers are defined by a Python class, with the constructor initializing the parameters of the layer. Functions such as 2d convolutions or activation functions are provided by PyTorch so that any neural network can be easily implemented. In addition to this, because PyTorch is built for the open-source Python ecosystem, it is able to take advantage of it's interoperability with support for libraries like NumPy. While ease of use is the largest priority, PyTorch also needs good performance to be useful, which it does by applying several optimizations. The majority of PyTorch is written in C++ for its high performance, and implements the tensor data structure, GPU and CPU operators, and parallel operations. The separation of control and data flow is maintained in PyTorch, so tensor operations are offloaded to the GPU to execute asynchronously using a CUDA stream, while program logic is handled by the host CPU [23].

The majority of tensor operations require the dynamic allocation of memory for the output tensor, but doing this on the GPU creates a large bottleneck as it must wait until all queued work on the GPU is done. To solve this, PyTorch uses a custom allocator which incrementally builds a cache of CUDA memory rather than all at once. The standard multiprocessing module for Python is inefficient for large arrays like tensors, so PyTorch extends it into its own multiprocessing module which acts as a drop in replacement. This allows for the movement of tensor data between processes using shared memory. The performance of PyTorch was evaluated and compared to several other deep learning frameworks, which show that its performance is within

17% of the fastest framework for all benchmarks [23].

2.5.2 TensorRT

TensorRT is an software development kit (SDK) developed by NVIDIA that enables high performance machine learning inference. The main goal of TensorRT is to take advantage of NVIDIA hardware to run fast and efficient inference on pre-trained networks from training frameworks like PyTorch or TensorFlow. There is a C++ and Python application programming interface (API) with nearly the same set of features but differences in implementation and purpose. The Python API is meant to take advantage of the interoperability of Python with libraries like NumPy and SciPy, while the C++ API is meant to be more efficient and used in applications requiring real-time processing [24].

There are two phases to TensorRT: the build phase and runtime phase. The build phase involves optimizing a model for the target GPU of the system, creating a TensorRT engine. Constructing the engine requires a model definition for the network, which should be in the open neural network exchange (ONNX) format. Models from training frameworks like PyTorch must be converted into the ONNX format before building the engine. ONNX provides a common format that models from widely used frameworks can be converted to, which enables the use of a variety of inference engines, including TensorRT. Given the model definition, an interface called the builder is used to perform certain optimizations on the model to produce the engine. The engine is created in a serialized format, which can be saved for later use. The runtime phase takes the serialized engine from the build phase and deserializes it for execution. Running inference on the engine requires the allocation input and output buffers in GPU memory [24].

Inference engines like TensorRT performs two types of optimizations: model compression and hardware mapping. Model compression involves compressing trained models for more efficient inference. It does this by removing unused neural network

layers, fusing consecutive layers into one operation, merging branches among layers, and quantizing floating-point values. Hardware mapping involves mapping the model to utilize the most optimal hardware functions by mapping individual layers to pre-implemented CUDA kernels. Compute unified device architecture (CUDA) is an API also made by NVIDIA that gives access to processes called kernels that can be executed on the GPU. This includes taking advantage of specific hardware like a tensor processing unit (TPU) or deep learning accelerator (DLA). Testing these optimizations on NVIDIA edge devices over a variety of neural network models, TensorRT is shown to maintain or slightly improve upon accuracy in classification tasks while providing a 23x-26x performance improvement in throughput [25].

2.6 Datasets

Datasets play a major role in computer vision research by providing a means of training and evaluating algorithms and networks. Datasets used for object recognition tasks fall into one of three groups: image classification, object detection, and semantic labeling. Image classification is used to indicate whether objects are present in an image with a binary flag. A recent example of an image classification dataset is ImageNet [26], which includes 22,000 categories with 500-1000 images each, containing in total over 14 million labelled images. Object detection tasks require both object classification and localization of the object in the image, described by a bounding box. The PASCAL visual object classes (VOC) dataset [27] is widely used for object detection tasks, containing 20 object categories with 27,000 labelled bounding boxes spanning 11,000 images. Because bounding boxes are limited to a rectangle, some pixels may not belong to the object. In comparison, semantic labeling involves classifying and labeling each individual pixel of an image, which enables labeling of objects with individual instances being hard to define [28].

2.6.1 COCO

The Microsoft Common Objects in Context (MS COCO) dataset is a large-scale dataset that addresses issues with scene understanding in computer vision research. These issues include detecting objects in non-iconic images, understanding context between objects in an image, and precise localization of objects in an image. COCO also seeks to capture common object categories, which are representative, relevant to practical applications, and appear frequently enough to facilitate the collection of a large dataset [28].

To accomplish this, a list of categories was compiled from the PASCAL VOC dataset, a subset of the 1200 most popular words for identifiable objects. This list of 272 categories was narrowed down to 91 by a vote from the co-authors based on how common they are, usefulness for practical applications, and diversity relative to other categories. Images are broken up into iconic images and non-iconic images. Iconic images contain a single large object from a canonical perspective and centered in the image. These images provide high quality object instances, but suffer from lack of contextual information. Non-iconic images on the other hand contain a combination of different objects and are shot from a non-canonical perspective. These images provide context between objects in a scene, enabling better generalization for datasets with more non-iconic images [28].

For annotating the collection of images, a three-stage annotation pipeline was constructed with the goal of cost efficient and high quality annotations. The pipeline consists of a category labeling stage to label all present categories, an instance spotting stage to mark locations of all instances of labeled categories, and instance segmentation stage for segmenting all object instances. With all images annotated, the MS COCO dataset contains 91 common object categories, with 2.5 million labeled instances in 328,000 images. The constructed dataset was compared to other popular datasets including ImageNet [26], PASCAL VOC [27], and SUN [29], with each

having a different focus. ImageNet was created with the goal of capturing a large number of object categories, with over 20,000 categories. The main application of PASCAL VOC is object detection for natural images, while SUN tries labeling scenes and associated objects.

Compared to ImageNet which has over 20,000 categories, COCO only has 91 but has more instances of each category which can lead to more precise 2D localization in trained models. Another advantage is that the average number of labeled instances per image for COCO (7.7) is higher than ImageNet (3) or PASCAL VOC (2.3), which aids in learning contextual information between objects. The SUN dataset, which is known for its strong contextual information, contains 17 object instances per image, but significantly less instances overall [28].

2.6.2 BDD100K

Large-scale annotated datasets like ImageNet [26] and COCO [28] have contributed heavily to advances in supervised learning, with many deep learning models depending on millions of training images to achieve high performance. This can be a challenge for applications like autonomous driving that lack such a large-scale and comprehensive dataset. Existing datasets for autonomous driving are limited in terms of the variation of scene, quality of annotations, and variation in geographical location, and typically suffer from overfitting. The BDD100K is a driving video dataset that aims to resolve these shortcomings [30].

The BDD100K dataset consists of over 100K 40 second videos of annotated driving scenes from a variey of locations in New York, San Fransisco, and other major US cities. These videos vary in the type of location, weather condition, and time of day. GPS/IMU data was also recorded to utilize driving trajectories for tracking tasks. The dataset consists of 10 categories including type of vehicle, traffic light, person, and passenger. There is also the inclusion of instances of occlusion and truncation, with half of instances being occluded and about 7% of instances trucated. For each video,

the frame at the 10th second is used for image tasks like detection and segmentation, and the entire sequence of frames are used for tracking tasks like MOT. Annotations for object detection are provided in the form of bounding boxes [30].

In addition to bounding boxes for 10 categories, there are also annotations for lane markings, drivable areas, and semantic instance segmentation. The lane markings set consists of 8 categories for curb, crosswalk, single and double lines, and white and yellow lines. Labels appear different colors to indicate direction (parallel and perpendicular) and are drawn as full or dashed lines to indicate the type of line. Driving areas are divided into two categories: directly drivable and alternatively drivable. Directly drivable areas describe the region that the vehicle is currently driving on and where the vehicle has the right of the way. Alternatively drivable areas on the other hand describe regions the vehicle can move to by changing lanes, but not the area the vehicle is currently driving.

For semantic instance segmentation, pixels are annotated for 10,000 randomly sampled images from the entire dataset. This label set contains 40 classes in order to capture diversity of objects in driving scenes as well as maximize the number of labeled pixels. For tracking tasks, a MOT dataset is also included with 2,000 videos, each one being 40 seconds and annotated at 5 FPS for a total of 400K frames. This makes it 10x larger than the popular tracking MOT17 [31] dataset. There is also a smaller dataset for multiple object tracking and segmentation (MOTS), which has 90 videos [30].

CHAPTER 3: Approach

The key component of the system is being able to detect and track incoming vehicles fast and accurately enough to assess the risk and alert pedestrians of danger. In order to be able to perform trajectory prediction, a sequence of information for each detected vehicle is needed. This includes the unique ID of each vehicle, and its corresponding location and lane number relative to the camera. An object detection network will be used for determining the current location of each vehicle, and a tracking algorithm will be responsible for determining the ID of each vehicle.

The requirements of the overall vehicle detection and tracking system are the following:

- The system must be able to detect vehicles that are over 3 seconds away from the camera setup in the work zone.
- The system must correctly identify and maintain the same identity for each passing vehicle.
- The system must achieve an average throughput of above 15 FPS end-to-end for normal traffic conditions.

3.1 Vehicle Detection

A camera was connected to the edge device for capturing incoming frames of traffic approaching the work zone. The camera is setup on a work vehicle near the edge of the work zone and pointed towards the direction of incoming traffic. Figure 3.1 shows an example of this setup on a two-lane two-way road separated by a median, with the edge device and camera represented by the black dot on top of the blue truck,

and it's approximate viewing angle represented by the dashed red lines. An example image is shown in figure 3.2, which shows a frame captured from the camera in a test environment acting as a simulated work zone.

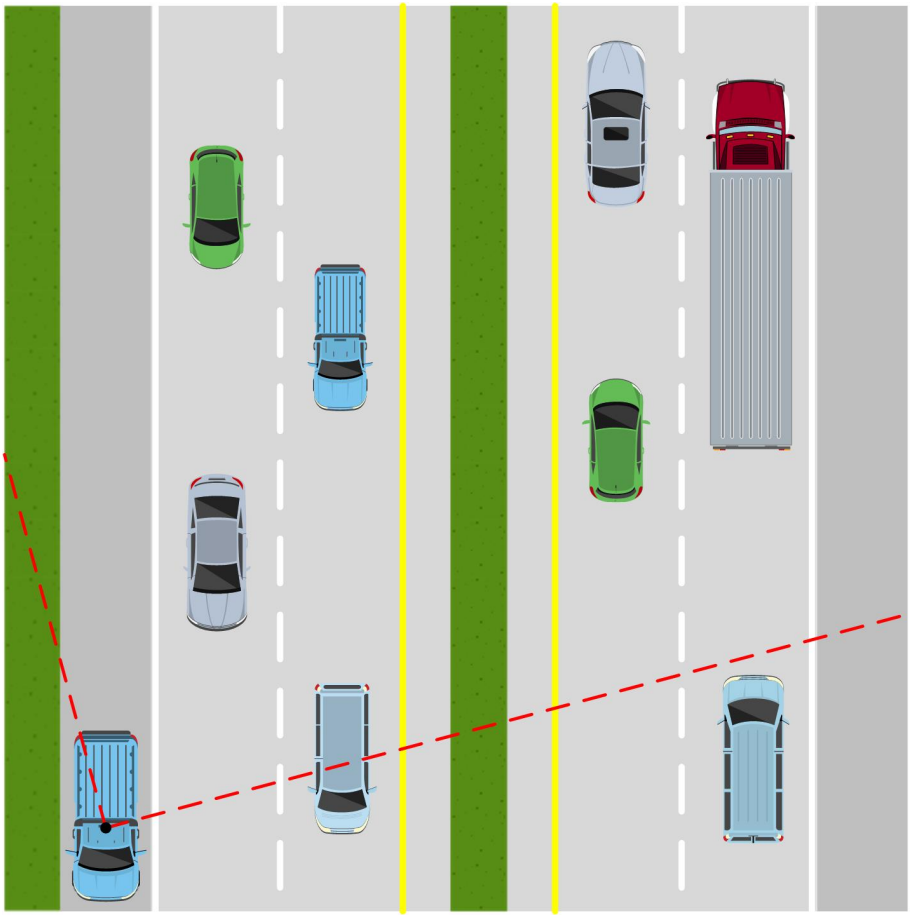

Figure 3.1: Edge Device and Camera Setup

The objective of object detection is to locate and classify specific objects in an image. For vehicle detection, this means locating vehicles on the road and classifying the type of vehicle (car, truck, bus, etc.). To accomplish this, a CNN based network is used to generate the classifications and bounding boxes for each detection in the input image. Three steps must take place to generate detections from the input images: pre-processing, inference, and post-processing.

Figure 3.2: Example Image Captured from Camera Setup

3.1.1 Pre-Processing

The input to the vehicle detection network is an image or frame from a video of a road or highway setting. Each frame is captured by the camera then the image data is saved to an array using OpenCV. Before being passed to the network, the image must be resized to fit the network input resolution and normalized by dividing each pixel value by 255, making all values fall between 0 and 1. A reshaping may also need to be done for the data to fit into the format required by the inference engine.

Image data comes in a 3D matrix of shape (height, width, channel), which for a 1080p colored image would be (1080, 1920, 3). However, both PyTorch and TensorRT use the NCHW format, which is of shape (batch, channel, height, width). The batch size is the number of input images that can be propagated through the network at once. For the same 1080p example as before and a single input image to the network at a time, the shape would be (1, 3, 1080, 1920).

For different network resolutions, the input image must be resized to the desired size, which would be (1, 3, H, W) for an input network size of HxW. Finally, the image

data must be copied to the device if using the GPU for inference. These few processes that manipulate data before going to the network are known as the pre-processing step. An example of this step is shown in figure 3.3, where an input image of shape 1920x1080 is captured by a camera then run through the pre-processing step. Steps in green show data and processes operating on the GPU, while blue denotes operations on the CPU.

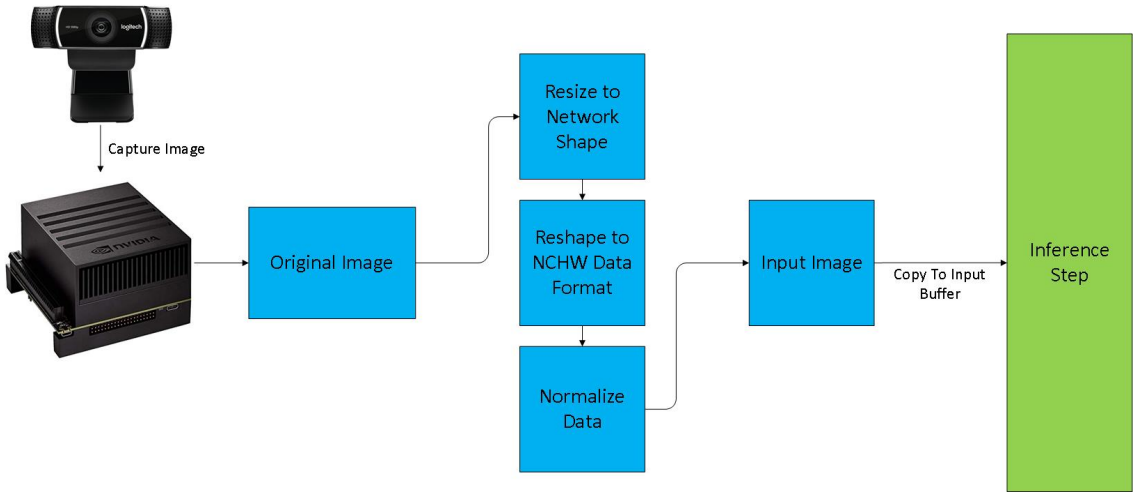

Figure 3.3: Vehicle Detection: Pre-Processing Stage

3.1.2 Inference

Once the input data is in the correct format it gets passed to the the network. The process of propagating an input through the network to generate an output is known as inference. This inference step is performed by an inference engine like PyTorch and TensorRT. For PyTorch, this inference step can be executed on the CPU or executed on the GPU using CUDA. When a GPU is available, it will almost always provide better performance for inference tasks because of its parallel nature.

TensorRT on the other hand requires the use of a supported NVIDIA GPU, but because of this is able to better optimize the engine for the hardware used. To run on the GPU, an input and output buffer of CPU and GPU memory must be allocated. The input data is copied to the input buffer in the pre-processing step, then the input

buffer is copied from the device memory (CPU) to host memory (GPU) using CUDA. Once the host has the input buffer in memory it performs inference with the data.

The outputs from inference go into the output buffers on the host, which are then copied to the output buffers on the device. In TensorRT, the result for each output is a flat array of size NxCxHxW, where N is the batch size, C is the channel size, H is the input network height, and width is the input network width.

3.1.3 Post-Processing

The outputs from the inference step contain the classification, confidence, and bounding box location for each detection. In TensorRT, because each output is a flat array, they must each be reshaped into the expected format before continuing post-processing. There may be detections that have a confidence value that isn't high enough to be considered accurate, or bounding boxes may overlap on the same vehicle. To make sure only high confidence detections and best prediction per vehicle are considered, non-maximum suppression is used to filter the list of detections.

Non-maximum suppression (NMS) is a post-processing algorithm that filters out low confidence detections and keeps only the best bounding box for each detection. Object detectors may predict multiple overlapping bounding boxes for a single object, when in reality only the best and most accurate box should be considered. The first step in NMS is to remove all bounding boxes with a confidence lower than a confidence threshold. Then for each class, the bounding box with the highest confidence value is kept and boxes with high overlap belonging to the same class are removed.

To determine the overlap between bounding boxes, the IOU is calculated and compared to an IOU threshold. If the IOU value is greater than the IOU threshold then it is determined to belong to the same object and that bounding box is removed. This process is repeated for each class until only the best detection for each object remains. The confidence and IOU threshold values can be adjusted to either keep more uncertain detections, or aggressively filter only the absolute most certain detec-

tions. Figure 3.4 shows the post-processing stage for the vehicle detection pipeline in TensorRT.

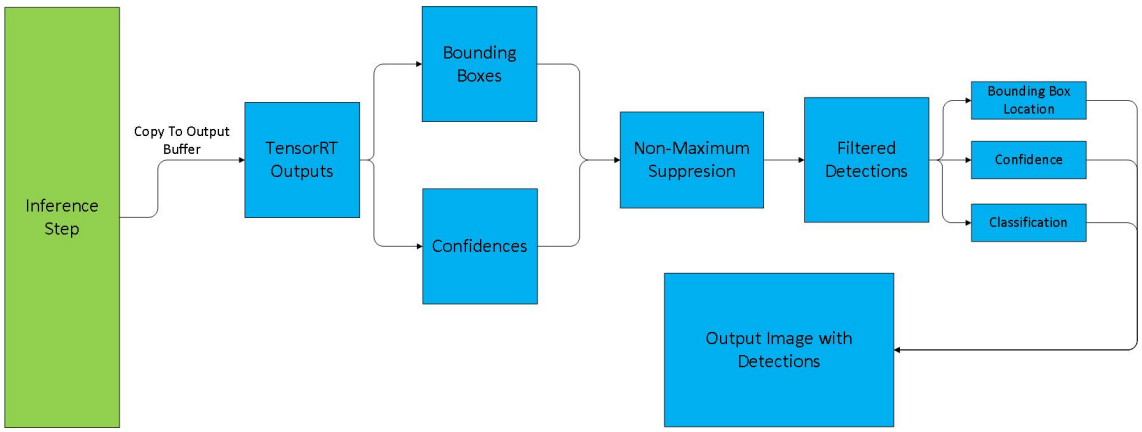

Figure 3.4: Vehicle Detection: Post-Processing Stage

3.2 Vehicle Tracking

Once a list of high confidence and unique bounding boxes are identified by the post-processing step, these bounding box locations can be used to track vehicles and other objects over time. To track a specific vehicle, a unique identity (ID) must be assigned to each detection when it first appears, then that ID must be maintained in future frames. The process of reassigning an ID to a previously seen object is called object re-identification (Re-ID).

This can be accomplished using an algorithm that associates object in the current frame with objects in previous frames. This association is used to match current detections with objects that have assigned IDs, or create a new ID if no association can be made. The purpose of the Re-ID process is to track the same vehicle over time and estimate its trajectory.

3.3 Methodology

This thesis proposes a system that achieves real-time performance for detecting and tracking vehicles on a highway. The system pipeline must be efficient enough but also have good enough accuracy to detect vehicles from a far distance. To satisfy these

requirements, a few lightweight networks were tested on an edge device and separately benchmarked across several different optimizations. These benchmarks demonstrate the performance improvement from each optimization and how it impacts the overall system.

3.3.1 Baseline Network

To determine the best combination of networks and algorithms for real-time vehicle detection and tracking, a baseline with no optimizations was constructed. This baseline consists of two backbone image classification networks and two object detection networks that emphasize efficiency and real-time performance: MobileNetV3 backbone [15], EfficientDet-D0 with an EfficientNet-D0 backbone [16], and YoloV4 with a CSPNet based backbone called CSPDarkNet53 [32, 33]. Each selected object detection network uses an image classification network trained on ImageNet as a backbone for performing feature extraction, and the object detection networks themselves, known as the head of the network, were trained on the COCO dataset.

These networks were selected for their state-of-the-art performance in cases where efficiency is important, like operating on an edge device where resources are limited. Each network was implemented in PyTorch and ran using weights trained on the COCO dataset. Inference in PyTorch requires a network definition, consisting of the PyTorch model defined in a Python class, and a checkpoint file with pre-trained weights. PyTorch loads the checkpoint file into the network definition to create the model, which can then be used for inference by passing in an input image. Figure 3.5 shows the process of initializing a PyTorch model and executing inference on an input image.

Inference was ran only on the CPU, benchmarking both the inference latency and end-to-end latency. The inference latency is the latency of the inference step, while the end-to-end latency is the latency from the pre-processing step where an image is captured to the post-processing step where bounding boxes are generated. This es-

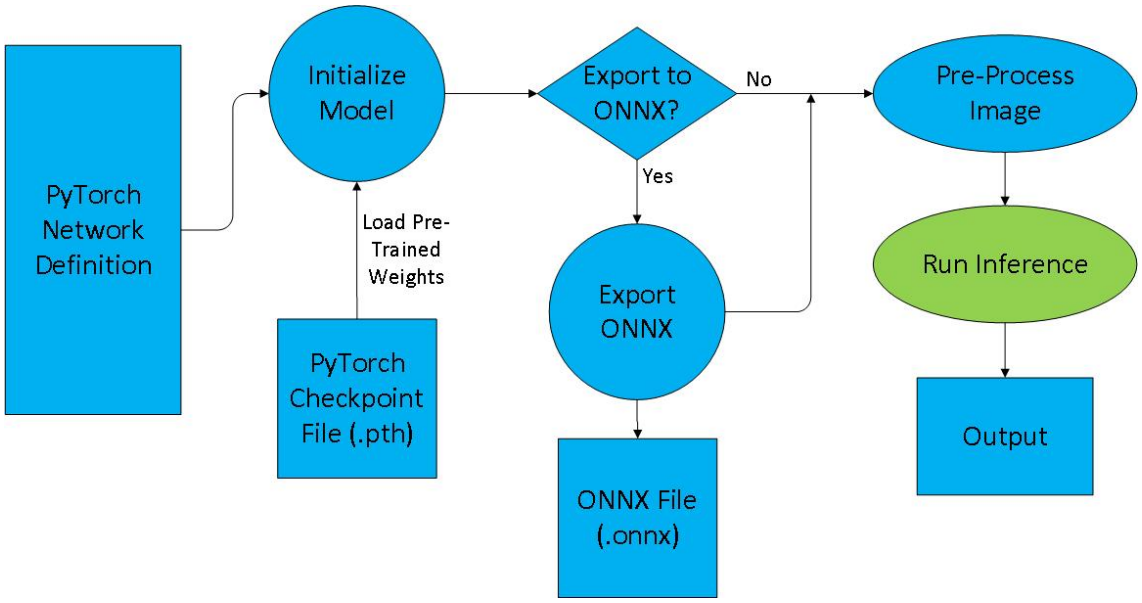

Figure 3.5: PyTorch Initialization and Inference Process

tablishes the baseline performance for each network to compare with the performance uplift with each additional optimization.

3.3.2 Utilizing the GPU

To show the significance of using an edge device equipped with a GPU, the first optimization is to perform the same benchmarks but using the GPU for inference. For this to happen, the model must be configured to use CUDA and the input image data must be copied to the GPU memory, which happens in the pre-processing step. To get the most accurate timing, the CPU and GPU must be in sync or else the CPU thread will execute before the GPU finishes executing the inference, resulting in an inaccurate latency measurement. The CUDA API has a function for recording events in code, including the timing information for measuring latency, and a synchronize event that is used to achieve CPU/GPU synchronization.

By measuring the latency separately for the inference step and end-to-end pipeline, the percentage of time that inference takes up in the pipeline can be and shown for each network. This illustrates how significant of a bottleneck inference is for object classification and detection networks. Latency is only a part of the overall system

performance, with the detection accuracy being just as important. Each network is capable of taking a variety of input resolutions due to the implementations of an FPN in the backbone.

The standard input size for each network is (224, 224, 3), but can be resized by increasing or decreasing the height and width by 32, allowing the use of higher resolution inputs. EfficientDet is the exception to this, with the EfficientDet-D0 model having a standard input size of (512, 512, 3) but can be scaled up or down by increments of 128. Using larger input sizes to the network comes with a benefit in accuracy, but with increased computational cost and latency. To analyze the trade-off between accuracy and efficiency, the latency benchmarks were done over multiple input resolutions.

Using smaller input sizes may result in lower system latency, but with the decrease in accuracy, the system would struggle with detecting vehicles from farther distances. To demonstrate the significance of the selected input network resolution, benchmarks on each network were done across multiple resolutions. The selected resolutions are 256x256, 512x512, 768x768, and 1024x1024. These resolutions were chosen to be equally spaced between a base resolution that works for each network and an unrealistic resolution for real-time processing like 1024x1024.

3.3.3 Optimizing Inference with TensorRT

While the appeal of PyTorch is its ease of use and interoperability, and is a method of getting a quick implementation of a model, it's execution speed is a downside in real-time applications. The TensorRT Python API provides a drop-in replacement to inference in PyTorch, allowing for both fast model creation and execution speed. The API cannot, however load the PyTorch model directly to perform inference but instead requires the use of the Open Neural Network Exchange (ONNX) format.

ONNX is an open format that allows for models to be interchanged between different frameworks. This enables designing models in PyTorch and then converting it

to any framework or inference engine supported by ONNX. The PyTorch library includes an ONNX module that can be used to export a PyTorch model to the ONNX format. The export module takes in the trained PyTorch model, a sample tensor with the network input size in the NCHW format, all input/output names, and other optional parameters and exports it to an ONNX file.

The operation of TensorRT can be broken down into two phases: the build phase and the runtime phase. In the build phase, the TensorRT network definition is created from the ONNX model and used to create the engine. For this, TensorRT provides a built-in ONNX parser for populating the network definition from an ONNX file. Another option for the conversion to an engine is the use of the trtexec command line tool, which automatically parses the ONNX file to convert to a TensorRT engine, then serializes it to memory for future use. This tool is useful for removing the overhead of parsing the ONNX model from the beginning of code execution by instead deserializing the TensorRT engine file into a network definition directly [24].

Once the TensorRT network definition is established, a builder is configured and used to create the TensorRT engine. The builder can be configured to perform several optimizations on the model such as reducing the precision for calculations, modifying available memory to control execution speed, and restricting available CUDA kernels. TensorRT uses FP32 (32-bit floating point) for calculations by default, but can be reduced to FP16 or quantized to INT8 for faster processing and reduced memory demand at the cost of accuracy. The builder also performs automatic optimizations like removing dead computations, combining and reordering operations for more efficient execution on the GPU, constant folding, and creating an optimized schedule for executing the model. This schedule is obtained by evaluating the latency of each layer using various data formats to minimize kernel execution and format transformation cost [24]. The resulting engine can only be used on the device it was built on, due to the platform and software version specific optimizations applied.

After the builder creates the optimized TensorRT engine, the runtime phase uses that engine to run inference. Before inference can be ran, the engine file must be deserialized and then used to create the execution context which is the interface used to execute inference. The context contains bindings for each input and output tensor of the network, which are used to allocate buffers for each binding on the host (CPU) memory and device (GPU) memory [24]. The CUDA stream, a queue of operations to be executed on the GPU, is initialized for use in inference. Buffer sizes are set at the start of execution after deserializing the engine since it will stay a constant size for each instance of execution.

Once the execution context is created and all input/output buffers are allocated, the engine is ready to run inference on a given input. To avoid the extra overhead at the beginning of each code execution and unnecessary repetition of ONNX conversions, the trtexec tool was used to convert each network from ONNX format to a TensorRT engine. The engine was set to have a batch size of 1 as the pipeline is purely sequential, and FP16 mode was enabled, which reduces all computations from FP32 precision to FP16 precision for an overall reduction in computational cost. While FP32 precision may enable better vehicle detection accuracy, the main goal of utilizing TensorRT is a reduction in inference latency.

To run the inference on a given input, it must be pre-processed the same way as in PyTorch and transformed to the NCHW format. Once the pre-processed input is ready, it must be transferred from the host memory to device memory using CUDA, then the context is used to execute inference given the list of bindings and the CUDA stream. The results from inference are then copied back over from device memory to host memory and the CUDA stream is synchronized so the host waits until the copy has completed before proceeding. The outputs from TensorRT are in the form of a flattened 1-D array and must be reshaped to the same format as PyTorch, which is the first step in the post-processing stage. The overall TensorRT process broken

down by phase is shown in figure 3.6.

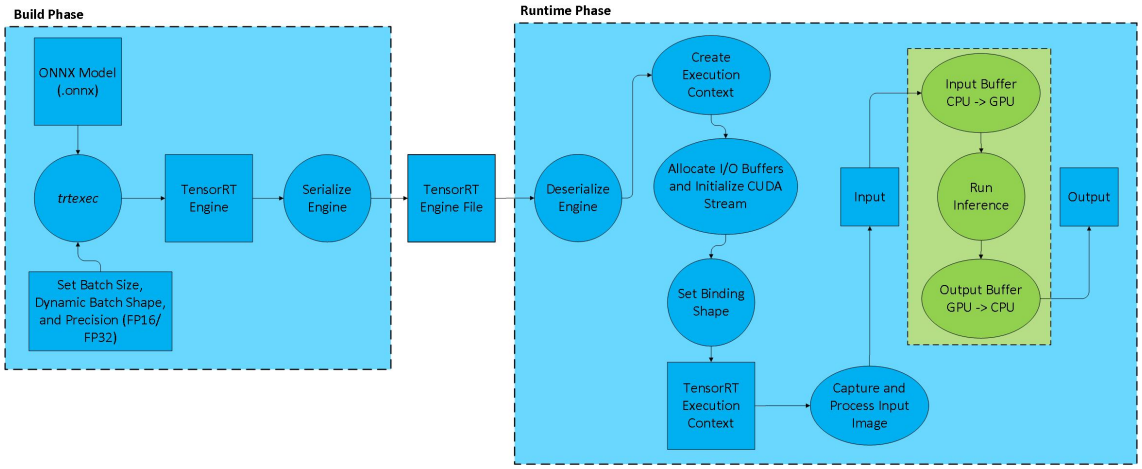

Figure 3.6: TensorRT Initialization and Inference Process

Out of the baseline networks, YoloV4 was selected to continue testing with due to the results of the benchmarks presented in the Results chapter. Figure 3.7 shows the vehicle detection pipeline using YoloV4 with TensorRT for inference. Benchmarks were ran measuring the latency and throughput of the different stages to evaluate the performance impact of using TensorRT. Like the previous sections, tests will be ran over different resolutions to find the best balance between speed and accuracy, and by pipeline stage to see how TensorRT affects the bottleneck caused by inference.

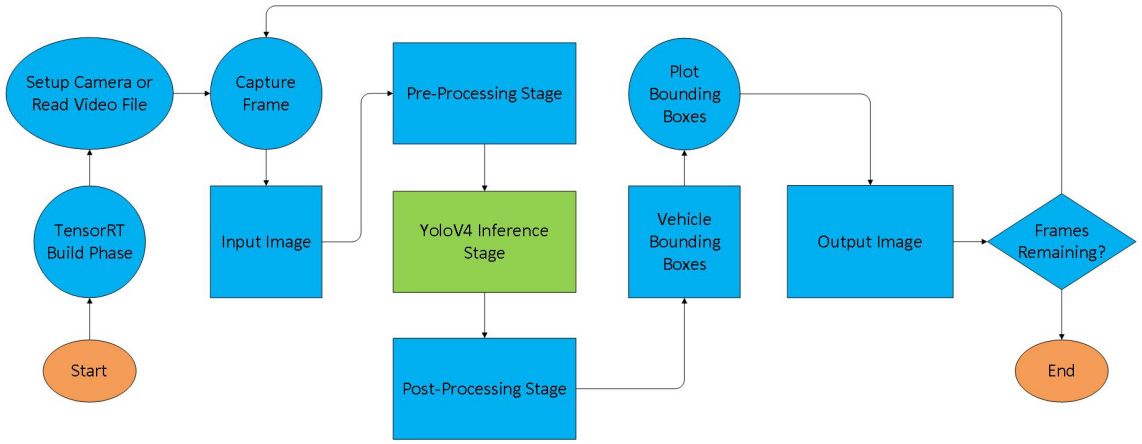

Figure 3.7: Vehicle Detection Pipeline using YoloV4 and TensorRT

3.3.4 Using a Specialized Dataset: BDD100K

Because this application is centered around a highway work zone, there are many classes of objects that the system shouldn't care about. The COCO dataset [28] contains 80 object classes prioritizing objects that are most common. Because the focus of the system is on capturing objects around a work zone, most classes in the COCO dataset are not applicable to this application. In order for the model to specialize in detecting vehicles, the YoloV4 model was trained again using the BDD100K dataset [30], which contains video of annotated driving scenes captured from a variety of locations.

The dataset contains 10 classes - pedestrian, rider, car, truck, bus, train, motorcycle, bicycle, traffic light, and traffic sign - which is an eighth of the classes that COCO has and are better suited to the highway environment. The detection accuracy was evaluated after training the BDD100K model to compare it's accuracy to the COCO model. In addition, the latency of each pipeline stage was benchmarked for each model to evaluate the performance improvement, if any, with using a more specialized dataset.

3.3.5 Integration of Vehicle Tracking with DeepSort

The vision portion of the system is responsible for detecting and tracking vehicles over time from a camera. While the networks discussed above are meant to handle detecting vehicles in each frame, another stage is necessary for tracking detected vehicles over time. This algorithm or network must be able to assign each detected vehicle a unique identity and maintain that identity for each frame that vehicle is captured by the camera. This requires the capability to associate current detections with detections from previous frames to determine if it is the same vehicle, or a new one that hasn't been seen yet.

The vehicle tracking stage can be integrated at the end of the existing pipeline

after the post-processing stage in which the final list of approved vehicle bounding boxes are established. The tracking stage will take the list of bounding boxes and assign an ID to each, then continuously reassign that ID to the same vehicle in the following frames. This allows the position of each vehicle to be tracked over time by analyzing the displacement of the bounding boxes between frames. Another pipeline will take this information to estimate vehicle trajectories over time as they approach the work zone in order to detect anomalous behavior.

A popular existing tracking algorithm, SORT [21], was created as a solution to the multiple object tracking (MOT) problem. DeepSort [22] expands on SORT by providing a deep association metric to improve the association of objects between frames in order to reduce the amount of ID switching. The network used was trained on the Market1501 person re-identification dataset [34], which contains image crops of people with an assigned identity captured from a multi-camera setup. The proposed vehicle detection pipeline produces bounding boxes of vehicles, which can be used to create image crops for each. DeepSort takes these image crops along with their corresponding confidence values and the original image, and returns back each bounding box and it's assigned ID.

After performing feature extraction on the image crops and correctly associating the same vehicles between present and past frames, DeepSort updates the tracking information for each vehicle. A list of trackers, containing all necessary tracking information such as assigned ID, are maintained for each detected vehicle. New detections with no association to previous detections are given a new tracker with a newly generated ID, while vehicles that do have an association are reassigned that ID of the previously seen vehicle. After a certain number of frames have passed without a tracker ID being reassigned, it's tracker is removed from the list. This number is kept relatively low since stale IDs likely belong to vehicles that have already passed the work zone, which we are no longer concerned about. The overall operation of

DeepSort is shown in figure 3.8.

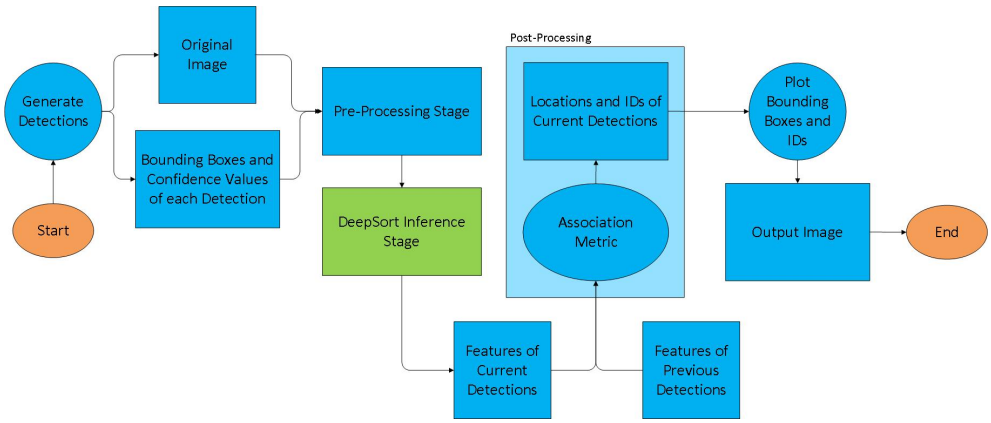

Figure 3.8: DeepSort Pipeline using TensorRT

The vehicle tracking stage contains the same steps as vehicle detection, meaning DeepSort has it's own pre-processing, inference, and post-processing steps. Because DeepSort is not as computationally expensive as YoloV4, it's operation is consolidated into one stage called DeepSort. Like the evaluated object detection networks, this feature extraction network can be ran on the CPU or GPU, and using PyTorch or converting to TensorRT for inference. After implementing DeepSort in PyTorch and integrating it into the vehicle detection pipeline, the DeepSort model was converted to ONNX and then to a TensorRT engine using the same process as previously described. The performance impact of the integration of DeepSort into the pipeline will be benchmarked, evaluating it's individual performance by step and it's contribution to the pipeline as a whole. Figure 3.9 shows the operation of the overall vehicle detection and tracking pipeline with YoloV4 and DeepSort.

3.3.5.1 Optimizing DeepSort with TensorRT

The process for converting inference in DeepSort from PyTorch to TensorRT is nearly the same as the process for YoloV4, with the only major difference being the use of a batch size greater than 1. YoloV4 uses a batch size of 1, since only one frame of data will be going into the network at one time, but DeepSort must run inference

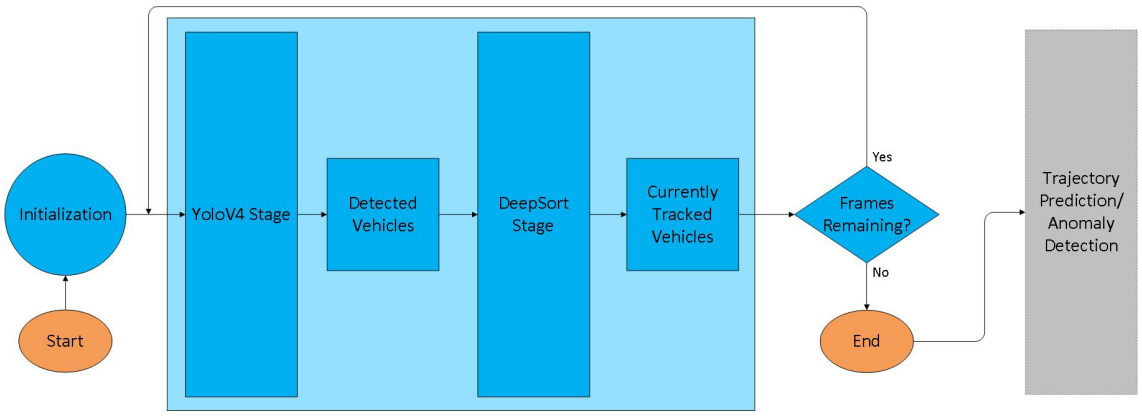

Figure 3.9: Overall Vehicle Detection and Tracking Pipeline

across each image crop. In a highway setting, this could easily reach more than 10 vehicles detected at once, which would be a significant bottleneck if ran sequentially. TensorRT allows for dynamic batch sizes, which allows for batch sizes to be set at runtime. This means that different batch sizes could be applied depending on the scenario.

To enable dynamic batching in TensorRT, it first must be configured in the ONNX conversion. When converting from PyTorch to ONNX, an optional parameter called dynamic axes is used to specify which dimension should be dynamic for each input and output tensor. So in this case only the batch size needs to be specified as all other dimensions (channel, height, width) are fixed. For TensorRT, setting a dynamic batch size requires a modification to the arguments passed to the trtexec tool, and explicitly setting the batch size during runtime when setting up the execution context. To convert from an ONNX model with dynamic axes, three new parameters must be specified when running trtexec: minimum shapes, optimal shapes, and maximum shapes.

The minimum shapes specifies the minimum tensor size the engine can be ran for, the optimal shapes specifies the tensor size that should see the most optimization, and the maximum shapes specifies the maximum tensor size the engine can be ran for. Each parameter must specify the shape for each input and output tensor, the input

tensor being the data from image crops, and the output tensor being the extracted features. The DeepSort TensorRT engine was configured using a minimum batch size of 1, optimal batch size of 16, and maximum batch size of 32. These sizes were selected to optimize performance for heavy traffic situations, and also allow plenty of headroom for extreme traffic situations. Other configurations were also used and tested to evaluate the impact of these optimizations on performance. After converting DeepSort to TensorRT, it's performance impact was evaluated using the same benchmarks used for the PyTorch implementation.

CHAPTER 4: RESULTS

As discussed in the Approach chapter, the main part of the system is the vehicle detection and tracking pipeline. The requirements for the pipeline are efficient real-time performance with a throughput above 15 FPS end-to-end, the ability to detect vehicles from a far enough distance, and consistent and reliable vehicle identifications between frames. Vehicles must be detected more than 3 seconds before reaching the work zone in order to give pedestrian workers time to react to alerts from the system, and collect enough tracking data to predict future trajectories.

For tracking vehicles to work, IDs assigned to detected vehicles must remain consistent for future frames. This means that once a unique ID has been assigned to a vehicle, the system must assign that same ID to that vehicle in the frames following. A common issue with this is ID switching, in which a vehicle is continuously assigned a new ID conflicting with it's previously assigned ID. This can include assigning a brand new unique ID, or the ID of a vehicle that has already passed by the work zone.

To ensure that each requirement has been met by the system, the different implementations discussed in the Approach will be ran and tested on an edge device.

4.1 Hardware

As this is a real-time application, it must take advantage of the low latency properties of edge computing. The core device that will be performing all processing and facilitate communication between devices must be powerful but highly efficient. As one of the leaders in AI computing, NVIDIA has hardware and software solutions for applications from data centers to edge devices.

NVIDIA's Jetson platform is meant for edge and embedded computing applications, with the Jetson AGX Xavier [5] striking a good balance between performance and cost. The Xavier contains a dedicated NVIDIA GPU with Tensor cores, an 8-core ARM CPU, deep learning and vision accelerators, 30W max power consumption, and a small form factor allowing for ease of deployment. In addition to the 30W max power, additional power modes are available to restrict power to 15W or 10W for for power efficient processing. Along with its high performance and efficiency, the Xavier comes with the JetPack software development kit (SDK). This SDK contains a version of Ubuntu OS, all drivers and firmware, AI libraries including TensorRT and CUDA, computer vision libraries including OpenCV, and developer tools for debugging and system profiling, allowing for quick startup and rapid deployment.

For capturing incoming video, a camera is connected to the Xavier via one of its USB ports. The only requirements for the camera are a reasonable image resolution and capture frame rate. In order for the camera to not bottleneck the system, it's capture resolution should be greater than the input resolution required by the object detection network and it's capture frame rate should be higher than the end-to-end system frames per second (FPS). This is because if the capture frame rate is lower, the system will have already finished making all predictions by the time the camera has captured the next frame, introducing a bottleneck. A realistic camera resolution and frame rate for this scenario is 1080p and 30 FPS.

4.1.1 Software Configuration

The NVIDIA Jetson AGX Xavier was flashed with the JetPack SDK version 4.4.1, which includes TensorRT version 7.1.3, CUDA version 10.2, and OpenCV version 4.1.1. For the PyTorch implementations, PyTorch version 1.6.0 and its dependency Torchvision version 0.7.0 were installed.

4.2 Benchmarks

The benchmarks ran on the NVIDIA Jetson board will evaluate the implementation of each baseline network running on the CPU, then move into the different optimizations and additional features added to the pipeline. This will cover the baseline object detection networks running on the CPU and GPU, the selected detection network with TensorRT integration for optimizing inference, and the integration of an object tracker. The primary metric these benchmarks will aim to evaluate is the average pipeline latency when running on the CPU, and average throughput when running on the GPU. Additionally, because this is running on an edge device, efficiency and power consumption must be considered.

4.3 Baseline Networks

The baseline networks introduced in the Approach - MobileNetV3, EfficientDet, and YoloV4 - were implemented on the NVIDIA Jetson board and benchmarked to evaluate the difference in performance achieved by each. While EfficientDet and YoloV4 are object detection networks with an image classification network as a backbone, MobileNetV3 is an image classification network meant to be used as a backbone to an object detection network. MobileNet introduced an efficient single shot object detector called SSDLite [11] meant to be used with a MobileNet backbone network, however this implementation was not able to work with the software configuration on the Jetson Xavier, so only the backbone network was included in the following benchmarks.

To demonstrate the difference in performance between these networks and make a fair comparison with MobileNetV3, the benchmarks separate the backbone network (image classifier) and head network (object detector) into two different measurements. Because the aim is to find which network will best satisfy the requirements of the pipeline, the metrics that will be focused on in this section are average inference and

end-to-end latency. These metrics will also be evaluated across multiple resolutions to determine how well they scale to a various network input sizes.

4.3.1 Inference on the CPU

The baseline performance is established by running the entire pipeline, including inference, on the CPU. The first benchmark measures the inference latency and end-to-end latency for each network using the standard resolution 256x256. The sample input consisted of a sequence of 1000 images taken from a video capturing traffic approaching a simulated work zone. Because latency is higher in the beginning when the device is just starting execution, the first 10 images didn't contribute to collected data. Figure 4.1 shows the results collected from the first benchmark. The data shows that on the CPU, the inference step in each network takes up on average 98.3% of the entire processing time of the pipeline, making it a massive bottleneck.

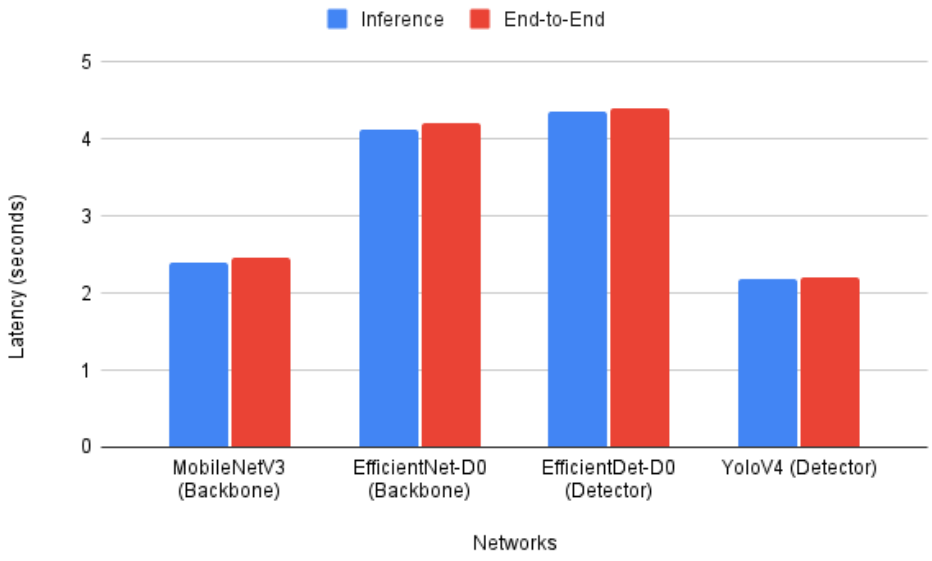

Figure 4.1: Latency Measurements at Standard Resolution (256x256) on CPU

The latency measurements at the standard resolution indicate that MobileNetV3 performs better as a backbone network than EfficientNet-D0, and YoloV4 performs substantially better as an object detector than EfficientDet-D0 with it taking almost

3 times longer for inference. While this may be the case at lower resolutions, not every network is able to scale well to higher resolution network inputs, which is essential for improving accuracy. The second benchmark measures only the inference latency across 4 resolutions - 256x256, 512x512, 768x768, and 1024x1024 - while still running entirely on the CPU. Figure 4.2 shows the results of this benchmark. The results indicate that each network is able to scale well to increasing network resolutions with the exception of YoloV4, whose inference latency grows at a much higher rate than the other 3 networks. However, YoloV4 does have the performance advantage at resolutions lower than 512x512, making it the best choice so long as the accuracy at those resolutions are acceptable.

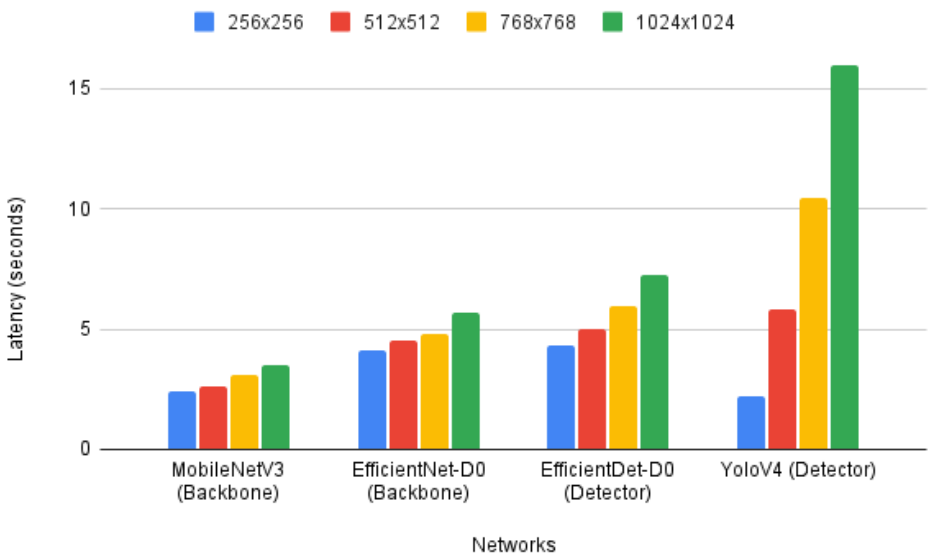

Figure 4.2: Inference Latency Across Multiple Resolutions on CPU

4.3.2 Inference on the GPU

Machines with a GPU are able to perform parallel processing much faster than on the CPU, while also taking some of the computational burden off of the CPU. The NVIDIA Jetson AGX Xavier that was used as the edge device has an NVIDIA GPU with 512 CUDA cores and 64 Tensor cores, which handle processing data in parallel

and performing matrix multiplications and mixed precision calculations. By leveraging the GPU for inference tasks, there should be a massive uplift in performance and result in a more balanced pipeline by alleviating the major bottleneck.

The same benchmarks ran on the CPU were reproduced using the GPU for inference. Rather than using latency as the metric, these benchmarks will measure throughput instead. Latency measures the time from an input going into the network to the output being available to the host, with lower latency being better. Throughput however, is the number of inferences that can be completed in a fixed amount of time (usually per second), with higher throughput being better [24]. Throughput is measured in frames per second (FPS), which represents the number of frames the system is able to run inference on in one second.

Figure 4.3 shows the average inference and end-to-end throughput in FPS at input resolution 256x256 for each network, and figure 4.4 shows the average inference throughput across the same resolutions used in the CPU benchmark. The results show that when utilizing the GPU in PyTorch, the inference step becomes much less of a bottleneck in the backbone networks, and shows a slight improvement in the object detection networks. When evaluating inference throughput with varying input resolutions, roughly the same pattern is present as the CPU benchmark. For each network, increasing the input network resolution resulted in a decrease in throughput, with the performance hit increasing with each resolution bump. In order to maintain real-time performance, the system should avoid going beyond a network size of 512x512, as there are also diminishing returns on accuracy with each increase.

The overall performance uplift by utilizing the GPU can be seen by comparing to the inference latency when using the CPU at a common resolution. Table 4.1 shows the comparison between CPU and GPU inference latency for each network evaluated at network size 512x512, and the corresponding percent performance uplift for each. These results show an average performance uplift of 98.45% when moving inference

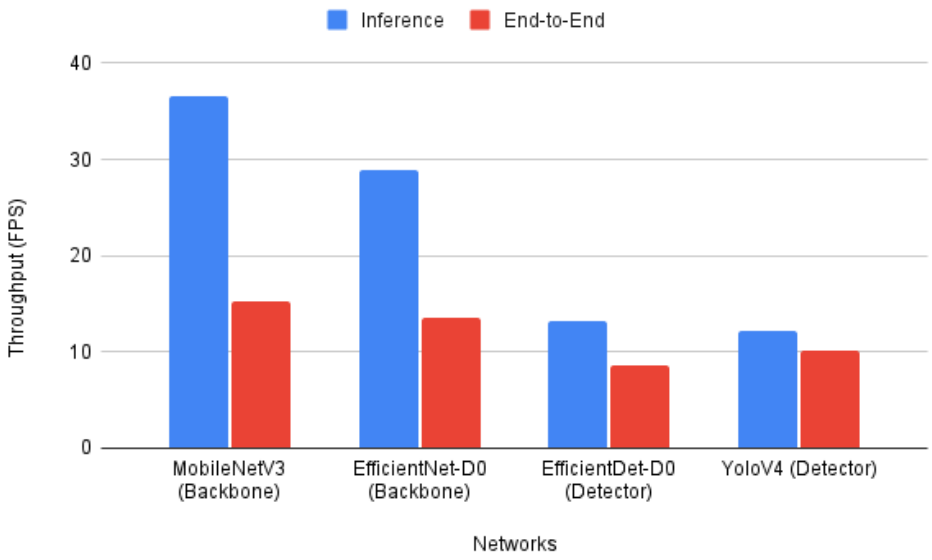

Figure 4.3: Throughput Measurements at Standard Resolution (256x256) on GPU

from the CPU to the GPU.

Table 4.1: Inference Latency at 512x512: CPU vs. GPU

	CPU Latency (ms)	GPU Latency (ms)	Perf. Uplift (%)
MobileNetV3 (BackBone)	2651.5	27.77	98.95
EfficientNet (BackBone)	4501.4	35.945	99.20
EfficientDet (Detector)	5007.5	102.145	97.96
YoloV4 (Detector)	5835.0	133.333	97.71

4.4 Optimizing Inference with TensorRT

While running inference on the GPU yields significantly better performance, using the PyTorch engine leaves performance on the table. By converting to a TensorRT for running inference, several automatic optimizations are applied to the model. Optional optimizations include changing the batch size, using FP16 or INT8 precision rather than FP32, and utilizing available DLAs, which can be configured when building the engine from the ONNX file. To evaluate the performance uplift with using TensorRT, the YoloV4 implementation was selected due to it's close performance to EfficientDet

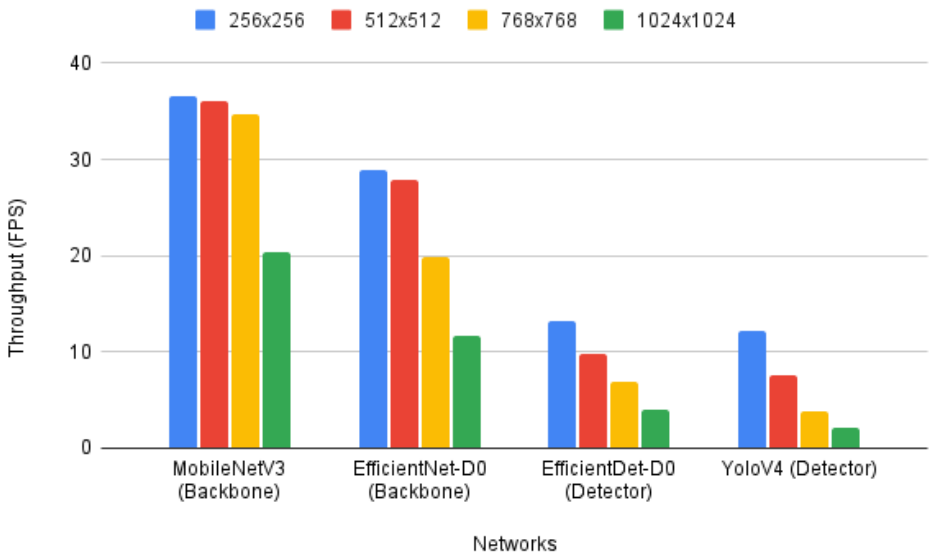

Figure 4.4: Inference Throughput Across Multiple Resolutions on GPU

while also having layers more compatible to TensorRT.

The first benchmark will look at the increase in inference throughput going from PyTorch to TensorRT for the YoloV4 implementation. This will run across 5 resolutions from 320x320 to 608x608 increasing by 96x96, and an extra resolution 352x608 for a rectangular aspect ratio which may allow for better accuracy. The only optional optimization used was using FP16 instead of FP32 for reduced computational complexity. Figure 4.5 shows the results of this benchmark, which demonstrates that the performance uplift is roughly the same at each input resolution tested, with the average improvement to throughput being approximately 4x. For a balance between inference speed and accuracy, the resolution 352x608 was used going forward.

The next benchmark breaks down the performance improvement by pipeline stage for PyTorch and TensorRT, evaluating the latency of each to see how they contributes to the overall performance of the pipeline. The results of this benchmark are shown in figure 4.6. The change in pre-processing latency was negligible, which is expected since the edge device uses the same method of capturing and processing frames for both implementations. The post-processing shows a very slight decrease of sub-1 ms,

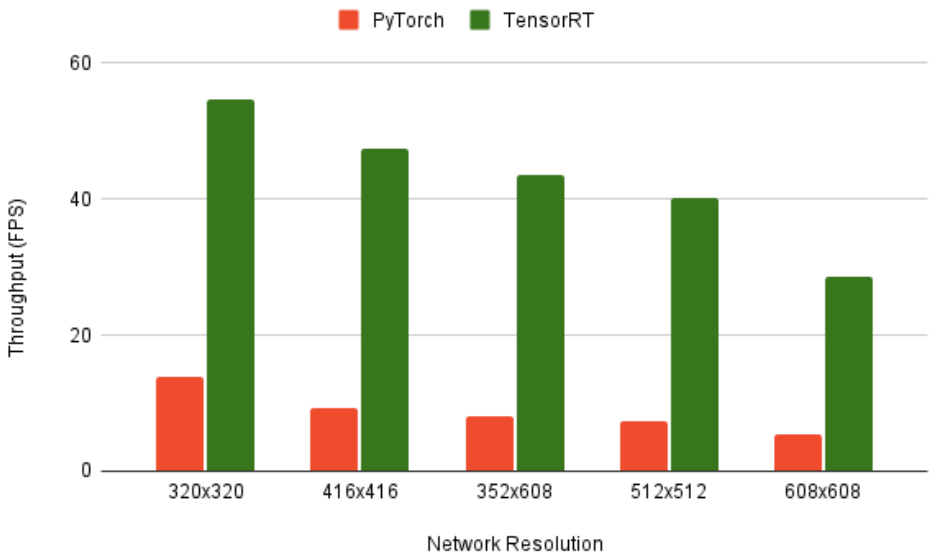

Figure 4.5: Inference Throughput Across Multiple Resolutions for YoloV4: PyTorch vs. TensorRT

which comes from the need of PyTorch to convert the output of inference from a PyTorch tensor to a NumPy array. This is avoided in TensorRT as the output from inference is a flat array which only needs to be reshaped which is computationally cheaper.

The inference step in TensorRT is where the main improvement occurs, with it becoming significantly less of a bottleneck on the overall pipeline. In the PyTorch implementation, inference accounted for approximately 85.6% of the total latency of the pipeline, while only accounting for 57.7% in the TensorRT implementation. This indicates that the implementation using TensorRT results in a much more balanced pipeline, with less CPU idle time waiting for a result from the GPU. In addition to creating a more balanced pipeline, there was a 5.3x increase in inference throughput and 3.6x increase in the end-to-end throughput, leading to significantly faster overall execution.

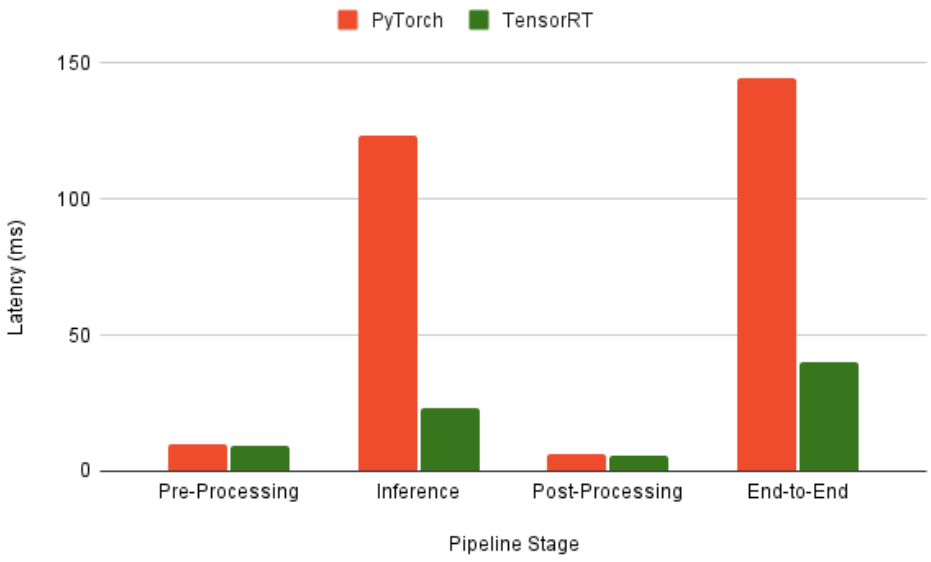

Figure 4.6: Latency Comparison of each YoloV4 Pipeline Stage: PyTorch vs. TensorRT

4.5 YoloV4 Trained on BDD100K

To further improve the performance of the vehicle detector, the model was retrained on a more specialized dataset. While COCO has 80 classes, most of which aren't useful to this application, while the BDD100K dataset has 10 classes that are much more applicable to driving scenes. Using this dataset should improve detection accuracy and reduce misclassifications. The pre-trained model using COCO reports a 43.5% mAP at resolution 608x608, while the new model trained on BDD100K resulted in a 45.1% mAP at resolution 352x608, which shows a marginal improvement in detection accuracy at a lower resolution, meaning faster execution speed as well.

To evaluate the performance impact, the latency of each pipeline stage was measured when using the BDD100K model compared to the COCO model. Figure 4.7 shows the results of this benchmark, which show a negligible change to pre-processing latency, a slight drop in inference latency, and a massive drop in post-processing latency. The reason for the sharp drop in post-processing latency is that the latency of NMS is proportional to the number of classes, since NMS must be performed for each

class. Because the BDD100K has 10 classes compared to the 80 for COCO, NMS has less computations to perform for each frame.

This resulted in a marginally faster end-to-end execution speed, going from approximately 39 ms to 35 ms. Overall, the usage of the BDD100K dataset in place of COCO resulted in a marginal increase in detection accuracy while also seeing a marginal improvement to the end-to-end latency of the pipeline.

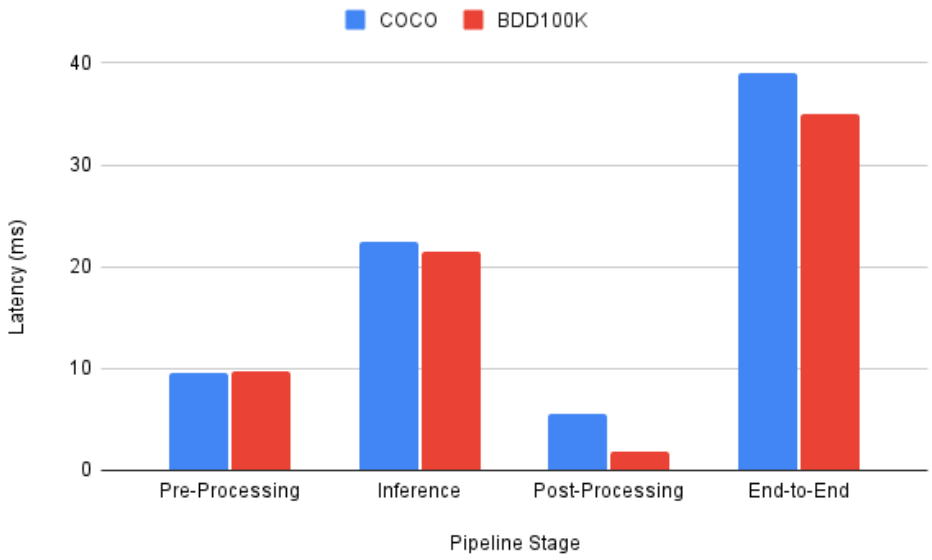

Figure 4.7: Latency Comparison of each YoloV4 Pipeline Stage: COCO vs. BDD100K

4.6 Integration of DeepSort for Vehicle Tracking

After completing the vehicle detection pipeline using YoloV4 and implementing TensorRT for inference, the vehicle tracking stage must be integrated to the pipeline. This vehicle tracking will be handled by using DeepSort for vehicle re-identification, allowing the tracking of vehicles over time by their assigned ID. DeepSort uses a feature extraction network to generate features from image crops, which are used for associating identified objects between frames. The result from the post-processing step in the detection pipeline is a list of bounding box locations, which can be passed to DeepSort along with the original image to create the necessary image crops.

This puts the DeepSort stage directly after the post-processing stage, with the new output from the pipeline being a list of bounding boxes and their assigned IDs. The first benchmark will evaluate the individual performance of DeepSort over each step (pre-processing, inference, post-processing) and end-to-end by measuring the average latency of each. The second benchmark will evaluate the throughput of DeepSort inference by the number of detections in the frame. Because DeepSort must process each image crop individually, the computation cost should go up as the number of detections increases, so this benchmark will evaluate how well DeepSort scales to number of objects in a single frame. These benchmarks were run on the PyTorch implementation as well as the TensorRT implementation to compare the overall performance uplift from the TensorRT engine.

4.6.1 Optimizing DeepSort with TensorRT

DeepSort was converted from PyTorch to TensorRT the same way as YoloV4, with the exception being the usage of a dynamic batch size. The same benchmarks used for the PyTorch implementation were reproduced again in TensorRT in order to see it's impact to the DeepSort pipeline as well as the overall pipeline (YoloV4 + DeepSort), and if there is any change in the ability for DeepSort to scale to an increasing number of detections in a single frame. The input used for these benchmarks was a longer version of the original video capturing a wider variety of traffic conditions.

The benchmarks ran over the first 5000 frames, with the distribution of the number of detections shown in figure 4.8. The distribution shows that most frames contains between 1 and 4 detections, but do include frames with more detections in smaller quantities. This is likely due to the fact that the video captures traffic from a two-lane two-way road with four lanes in total. The results of the latency benchmark across DeepSort steps and benchmark for inference throughput over the number of detections in frame are shown in figure 4.9 and 4.10 respectively.

The benchmark over DeepSort stages steps shows that like YoloV4, the inference

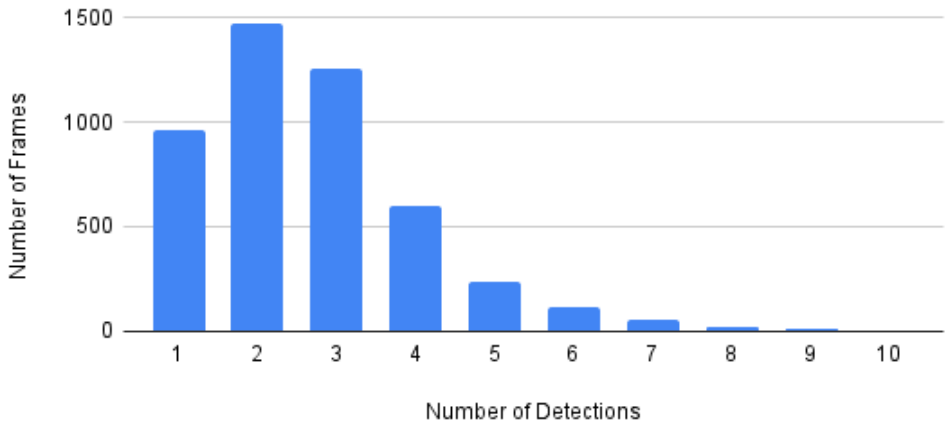

Figure 4.8: Distribution of the Number of Detections in Frame for Input Video

step is the step with the highest latency, taking up about 46.7% of the total latency of the DeepSort stage. After implementing TensorRT for inference, the post-processing step becomes the step with the highest latency, while the inference latency only accounts for around 20% of the total latency. The results show approximately a 8.9x and 3.8x improvement to the inference step and overall DeepSort stage respectively. The pre-processing step shows negligible change (less than 0.5 ms) as expected since there should be no optimization going on there.

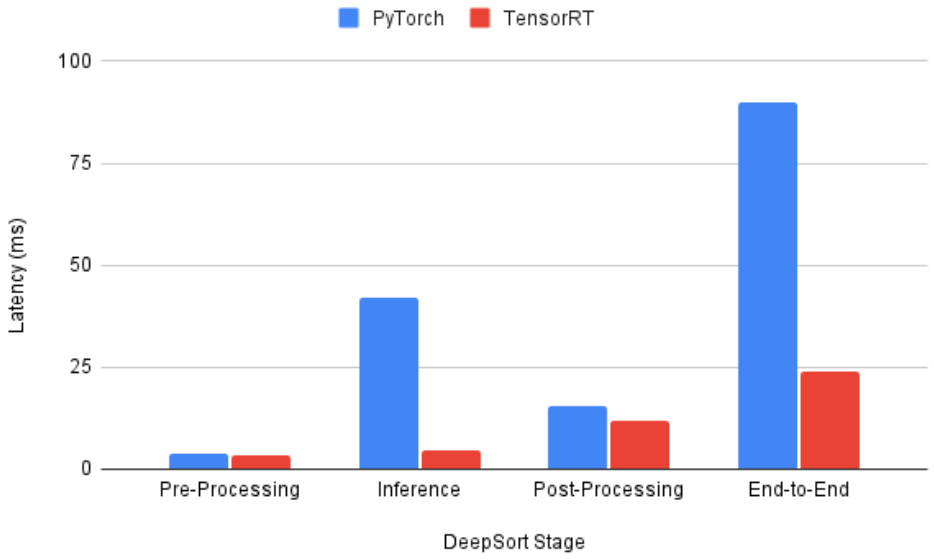

Figure 4.9: Latency Comparison of each DeepSort Stage: PyTorch vs. TensorRT

The overall improvement to DeepSort by using TensorRT is comparable to YoloV4, with a 3.8x and 3.4x latency improvement respectively. The new bottleneck, the post-processing step, increased from around 17% of the total latency before TensorRT to 49.5% after TensorRT. The benchmark over the number of detections in frame shows that the latency fluctuates heavily with the number of detections in PyTorch, being significantly higher when there were less than 4 detections, but at its lowest at 4 detections. The TensorRT inference on the other hand maintained a steady inference latency regardless of the number of detections, further supporting the reduction in the bottleneck of the inference step.

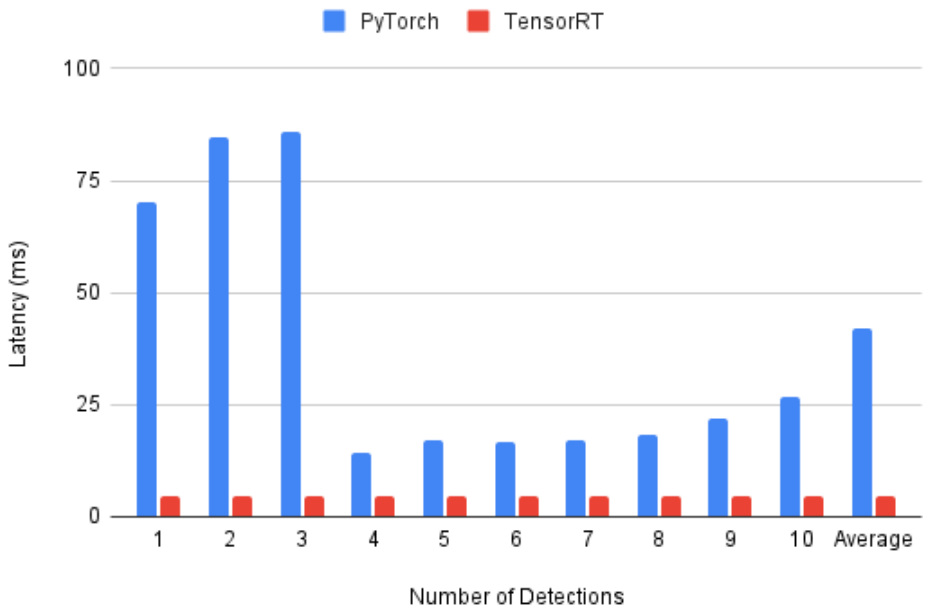

Figure 4.10: Average DeepSort Inference Latency over Number of Detections: PyTorch vs. TensorRT

The TensorRT engine was configured to use a minimum batch size of 1, optimal batch size of 16, and maximum batch size of 32. During runtime, the batch size was set to 10 because the video input contained a maximum of 10 potential detections in a single frame. In order to run inference with a lower batch size in this case, multiple inferences would need to be run which is far from ideal. However, in lighter traffic

situation, the system may benefit from a smaller batch size if the average number of cars per frame is lower. Conversely, the batch size must be higher in cases of heavy traffic where there are more detections per frame.

To evaluate the difference in performance when using different batch sizes, the average inference throughput was measured across several batch sizes between 10 as used in the benchmarks above up to 32, the max batch size configured for TensorRT. This data, shown in figure 4.11, demonstrates a steady drop in throughput as the batch size setup for inference increases. The change in throughput from batch size 12 to 16 and 16 to 20 is noticeably higher than the change in higher batch sizes, which could be the result of configuring the engine for an optimal batch size of 16.

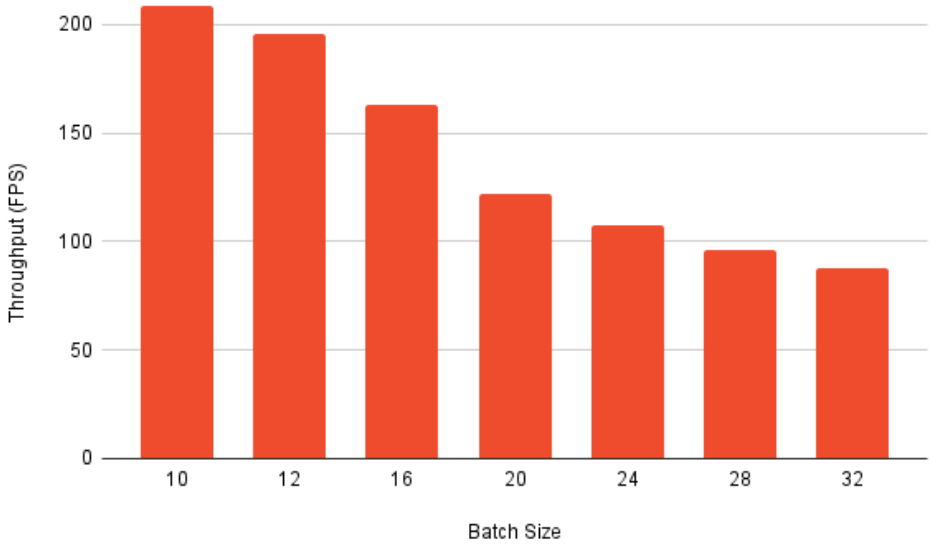

Figure 4.11: Average DeepSort Inference Throughput Across Multiple Batch Sizes

4.7 Vehicle Detection and Tracking Pipeline

The DeepSort stage was added right after the overall YoloV4 pipeline, which now simplifies to the YoloV4 stage. The average throughput of the vehicle detection stage using YoloV4 and vehicle tracking stage using DeepSort were measured along with the end-to-end throughput to see the contribution of each to the overall vehicle

detection and tracking pipeline. These measurements were taken using both PyTorch and TensorRT to see the performance uplift for each stage and end-to-end. The same video input used for previous DeepSort benchmarks was used in this case, taking the average over 5000 frames.

The results of this are shown in figure 4.12, indicating a similar performance uplift after implementing TensorRT for inference in YoloV4 and DeepSort. The approximate uplift to throughput was 4x and 3.76x for the YoloV4 and DeepSort stages respectively, and 3.9x to the end-to-end throughput of the overall pipeline. This brought the average end-to-end throughput up from 4.38 FPS with PyTorch to 17.10 FPS with TensorRT. The throughput is above the 15 FPS goal for real-time processing, and is able to achieve this on a video with moderate levels of traffic. Figure 4.13 shows an output image from the system, which shows that each visible car in frame has an accurate bounding box, unique ID, and correct classification. Figure 4.14 shows another example with multiple vehicle types.

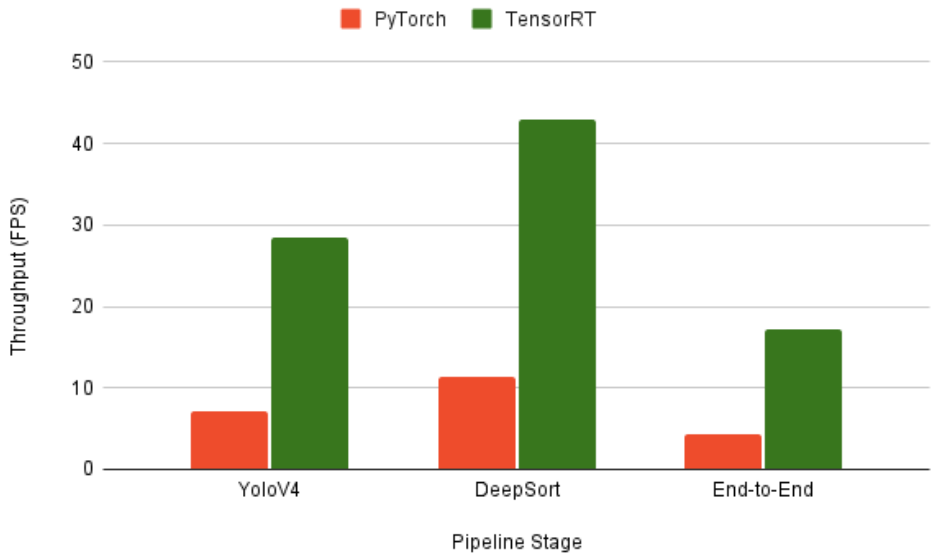

Figure 4.12: Average Latency of each Pipeline Stage and End-to-End

Figure 4.13: Example Output Image from Pipeline with Only Cars

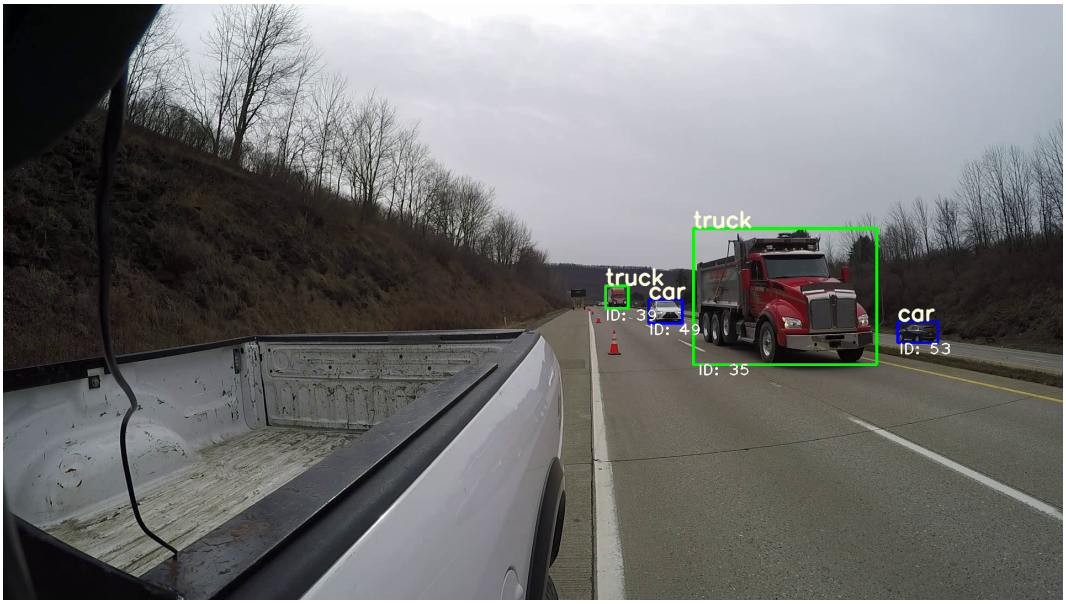

Figure 4.14: Example Output Image from Pipeline with Cars and Trucks

4.7.1 Power Consumption

The NVIDIA Jetson Xavier provides 3 power modes: 10W, 15W, and 30W. This gives additional flexibility to control the trade-off between performance and power consumption. This is especially important considering that the edge device must be powered entirely by a power source local to the work zone like a generator or work

vehicle. To evaluate the performance impact when using the lower power modes, the complete pipeline was benchmarked at each power mode, measuring the end-to-end throughput and power consumption. The total system power consumption was broken down into GPU, CPU, and then all other system components.

Table 4.2 shows the number of active CPU cores and CPU/GPU clock frequencies configured for each power mode. Figure 4.15 shows the results of the benchmark for power consumption over the power modes. Because the system is sequential, the CPU and GPU will never be fully utilized since they constantly have to wait on each other to finish their task. This is one reason why the average power consumption measured for each power mode was less than it's allowed max power.

Table 4.2: System Configuration of each Power Mode

Power Mode	CPU Cores Active	CPU Clock (MHz)	GPU Clock (MHz)
30 W	8	2300	1400
15 W	4	1200	675
10 W	2	1200	522

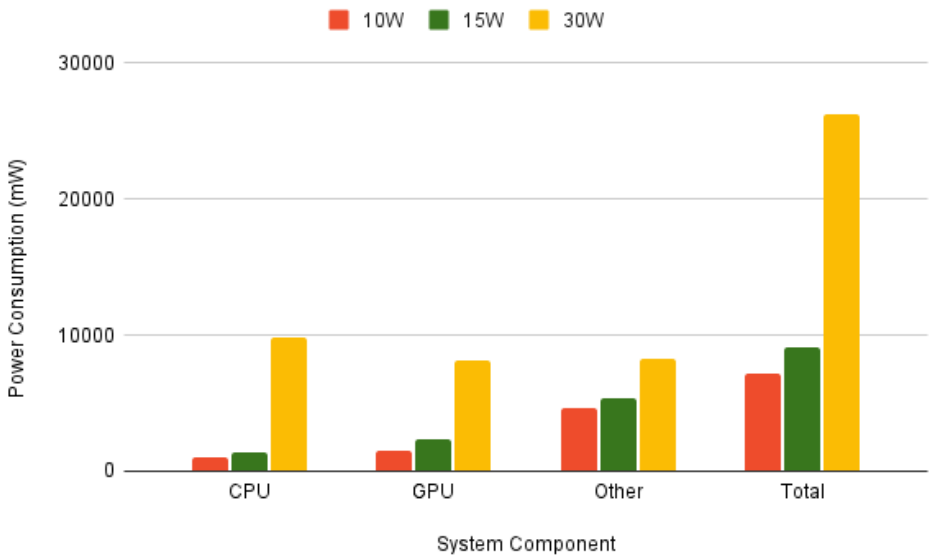

Figure 4.15: Power Consumption of each Power Mode

The average end-to-end throughput measured for each power mode was the follow-

ing: 6 FPS for 10W, 9.32 FPS for 15W, and 15.52 FPS for 30W. For now, only the 30W power mode is able to achieve the required real-time performance, but future improvements to the pipeline will impact the performance at each power mode.

CHAPTER 5: CONCLUSION

The safety of highway work zones is a growing concern, with a consistent rise in work zone fatalities over the past decade caused by vehicle collisions. While new vehicles are showing an improvement in safety features such as collision avoidance, there have been little to no improvements to the safety of work zones and workers themselves. In order to increase the safety of highway work zones, anomalies in vehicles approaching a work zone can be detected by predicting the trajectories of the incoming vehicles. To facilitate vehicle trajectory and path prediction, all vehicles must first be detected and tracked using a camera pointed from the work zone towards incoming traffic.

This thesis has proposed an efficient vehicle detection and tracking pipeline with all processing being done on an edge device. A YoloV4 network trained on the BDD100K dataset was paired with DeepSort to track detected vehicles over a sequence of frames. This pipeline, consisting of the YoloV4 stage and DeepSort stage, requires a separate instance of inference for each network per frame. Running a heavy tasks like inference on an edge device, where resources are constrained, leads to a significant bottleneck in the pipeline. Each network was implemented using PyTorch and ran using an NVIDIA Jetson AGX Xavier as the edge device.

Even using the GPU for inference, the performance of the overall pipeline is far from the required performance for real-time processing. In addition to using the GPU, the use of the PyTorch engine for inference was replaced with the TensorRT engine, which optimizes models for NVIDIA hardware and applies several other optimizations to the network. The performance uplift gained from swapping from PyTorch to TensorRT was around a 4 times improvement to the end-to-end throughput of the pipeline. The final pipeline was benchmarked using a 1080p video capturing simulated work zone

traffic of varying levels and was able to achieve an average of 17 FPS on the NVIDIA Jetson AGX Xavier while consuming an average of around 26 W.

5.1 Future Work

The current implementation of the pipeline is in Python, using the TensorRT Python API for inference and the OpenCV Python API for computer vision tasks. While Python's strength is it's ability for rapid development and creating quick and simple implementations, it suffers in computation speed. On the other hand, C++ is known for it's speed but requires more complex implementations. Both OpenCV and TensorRT have a C++ API, meaning the proposed pipeline could be fully implemented in C++, resulting in a more efficient system that can achieve a much higher throughput. Besides it's complexity, another downside with this approach is that future pipeline stages like trajectory prediction would need to be implemented in C++ to be compatible.

The proposed pipeline has 2 stages, one for each network, with it's own pre-processing, inference, and post-processing steps. The pipeline is fully sequential, which limits the overall throughput as each stage must be completed one at a time. Along with converting the implementation to C++, the pipeline could also be implemented concurrently, with each stage running on its own CPU thread. The Xavier has an 8-core CPU, providing plenty of room for concurrent processing of the pipeline. While the end-to-end latency of a sequential pipeline is the sum of it's stages, the latency for a concurrent pipeline is the latency of the slowest stage. By constructing a balanced pipeline in which each stage handles roughly the same amount of computation, the concurrent implementation would be far faster than the sequential version.

The proposed pipeline with it's two stages, running at approximately 28 FPS and 42 FPS for the YoloV4 stage and DeepSort stage respectively, resulted in an end-to-end throughput of around 17 FPS. If implemented concurrently, with the YoloV4

and DeepSort stages running on separate threads, the end-to-end throughput would be roughly 28 FPS. This would make the concurrent version of the pipeline around 64.7% faster. The downside with the concurrent implementation is it's complexity in implementation and maintenance.

This thesis only focused on two of the five components of the overall system. While this system is able to track vehicles in real-time, to accomplish anomaly detection, this tracking data must be used to estimate current and predict future vehicle trajectories. In addition, there must be a way to report detected anomalies to workers/pedestrians. By using existing WiFi-enabled wearable technology - such as AR goggles or smart watches - reports of danger can be alerted to workers in real-time. For this to be possible, it would require low latency communication between the edge device and wearables. Each of these challenges that weren't addressed in this thesis are a part of the future work of this project.

REFERENCES

[1] N. H. T. S. Administration, "Automated vehicles for safety." https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety, 2022.

[2] N. I. for Occupational Safety and Health, "Highway work zone safety." https://www.cdc.gov/niosh/topics/highwayworkzones/default.html, August 2022.

[3] N. S. Council, "Motor vehicle safety issues." https://injuryfacts.nsc.org/motor-vehicle/motor-vehicle-safety-issues/work-zones/#:~:text=Work\%20zone\%20deaths\%20reached\%20a,zone\%20deaths\%20have\%20increased\%2046\%25., 2022.

[4] N. C. D. of Transportation, "Work zone speed limit guidelines for nc highway construction and maintenance activities on high speed facilities." https://connect.ncdot.gov/resources/safety/Teppl/Pages/Teppl-Topic-Original.aspx?Topic_List=W28, 2011.

[5] N. Corporation, "Jetson agx xavier developer kit." https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit, 2022.

[6] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," in *IEEE Access*, May 2020.

[7] I. H. Sarker1, "Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions," in *SN Computer Science*, August 2021.

[8] Z.-Q. Zhao, P. Zheng, S. tao Xu, and X. Wu, "Object detection with deep learning: A review," July 2018.

[9] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," November 2015.

[10] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," April 2017.

[11] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 4510-4520*, January 2018.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," tech. rep., Microsoft, December 2015.

[13] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, "Squeeze-and-excitation networks," in *CVPR 2018*, September 2017.

[14] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," October 2017.

[15] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," in *ICCV 2019*, May 2019.

[16] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2020)*, November 2019.

[17] T.-Y. Lin, P. DollÃ¡r, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," December 2016.

[18] M. Tan, R. Pang, and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *2019 International Conference on Machine Learning*, May 2019.

[19] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," June 2015.

[20] R. Girshick, "Fast r-cnn," in *ICCV 2015*, April 2015.

[21] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, "Simple online and realtime tracking," in *ICIP 2016*, February 2016.

[22] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," March 2017.

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS 2019*, December 2019.

[24] N. Corporation, "Nvidia tensorrt developer guide." `https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html`.

[25] O. Shafi, C. Rai, R. Sen, and G. Ananthanarayanan, "Demystifying tensorrt: Characterizing neural network inference engine on nvidia edge devices," in *2021 IEEE International Workshop/Symposium on Workload Characterization*, November 2021.

[26] J. D. W. D. R. S. L.-J. L. K. L. L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009.

[27] M. Everingham, L. V. Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," in *2010 International Journal of Computer Vision*, September 2009. pp 303–338.

[28] T.-Y. Lin, M. Mairen, S. Belongien, L. Bourdevn, R. Girshickn, J. Haysn, P. Peronan, D. Ramanann, C. L. Zitnickn, and P. DollÃ¡r, "Microsoft coco: Common objects in context," May 2014.

[29] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba, "Sun database: Large-scale scene recognition from abbey to zoo," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, August 2016.

[30] F. Yun, H. Chenn, X. Wangn, W. Xiann, Y. Chenn, F. Liun, V. Madhavann, and T. Darrell, "Bdd100k: A diverse driving dataset for heterogeneous multitask learning," in *CVPR 2020*, May 2018.

[31] A. Milan, L. Leal-Taixe, I. Reid, S. Roth, and K. Schindler, "Mot16: A benchmark for multi-object tracking," March 2016.

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

[33] C.-Y. Wang, H.-Y. M. Liao, I.-H. Yeh, Y.-H. Wu, P.-Y. Chen, and J.-W. Hsieh, "Cspnet: A new backbone that can enhance learning capability of cnn," November 2019.

[34] L. Zheng, Z. Bie, Y. Sun, J. Wang, C. Su, S. Wang, and Q. Tian, "Mars: A video benchmark for large-scale person re-identification," in *ECCV 2016*, September 2016. pp 868–884.