Edge-Computing Deep Learning-Based Computer Vision Systems

Edwin Lee Weill
Clemson University, eweill@g.clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Recommended Citation
https://tigerprints.clemson.edu/all_dissertations/2285

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
EDGE-COMPUTING DEEP LEARNING-BASED COMPUTER VISION SYSTEMS

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

by
Edwin Weill
December 2018

Accepted by:
Dr. Melissa Smith, Committee Chair
Dr. Robert J. Schalkoff
Dr. Harlan Russell
Dr. Amy Apon
Abstract

Computer vision has become ubiquitous in today’s society, with applications ranging from medical imaging to visual diagnostics to aerial monitoring to self-driving vehicles and many more. Common to many of these applications are visual perception systems which consist of classification, localization, detection, and segmentation components, just to name a few. Recently, the development of deep neural networks (DNN) have led to great advancements in pushing state-of-the-art performance in each of these areas. Unlike traditional computer vision algorithms, DNNs have the ability to generalize features previously hand-crafted by engineers specific to the application; this assumption models the human visual system’s ability to generalize its surroundings. Moreover, convolutional neural networks (CNN) have been shown to not only match, but exceed performance of traditional computer vision algorithms as the filters of the network are able to learn "important" features present in the data.

In this research we aim to develop numerous applications including visual warehouse diagnostics and shipping yard management systems, aerial monitoring and tracking from the perspective of the drone, perception system model for an autonomous vehicle, and vehicle re-identification for surveillance and security. The deep learning models developed for each application attempt to match or exceed state-of-the-art performance in both accuracy and inference time; however, this is typically a trade-off when designing a network where one or the other can be maximized. We investigate numerous object-detection architectures including Faster R-CNN [1, 2], SSD [3], YOLO [4, 5], and a few other variations in an attempt to determine the best architecture for each application. We constrain performance metrics to only investigate inference times rather than training times as none of the optimizations performed in this research have any effect on training time. Further, we will also investigate re-identification of vehicles as a separate application add-on to the object-detection pipeline. Re-identification will allow for a more robust representation of the data while leveraging
techniques for security and surveillance.

We also investigate comparisons between architectures that could possibly lead to the development of new architectures with the ability to not only perform inference relatively quickly (or in close-to real-time), but also match the state-of-the-art in accuracy performance. New architecture development, however, depends on the application and its requirements; some applications need to run on edge-computing (EC) devices, while others have slightly larger inference windows which allow for cloud computing with powerful accelerators.
Dedication

This dissertation is dedicated to my academic advisor without whose constant support and guidance this research would not have been possible. This dissertation is also dedicated to my wife, parents, and sister, who have always been supportive and an inspiration to pursue interests and aspirations.
Acknowledgments

This dissertation was made possible by the tremendous help and support of the following faculty members, family members, and colleagues.

I begin by thanking my academic advisor, Dr. Melissa C. Smith, who graciously accepted me as her pupil. Dr. Smith’s words of wisdom, immense experience, constant encouragement, insightful suggestions, and limitless networking ability allowed for the completion of this research and dissertation.

Secondly, I would like to thank Dr. Robert Schalkoff for his support through the entire doctoral process as well as the instruction he has provided as a professor and mentor. His immense knowledge in the fields of machine learning and artificial intelligence have not only aided me in my studies and research but his enjoyment of the field has motivated me to pursue this topic in a way that would not be possible without constant stimulation.

I would also like to thank Dr. Amy Apon and Dr. Harlan Russell for serving on my doctoral committee and instructing me in various fields of computer architecture, computer vision, and high performance computing.

I am also thankful for my wife, Hiliary, my parents, Edwin and Jeanine, and my sister, Shannon, who provided constant support during the research process and writing of this manuscript.

I also owe a debt of gratitude to my fellow colleagues from the Future Computing Technologies Laboratory (FCTL) at Clemson University: Jesse, Varun, Ankit, Jamar, Karan, and many others whom I fail to mention for providing me a stimulating work environment during this process. I also owe thanks to Ashraf and Vivek (previous Ph.D. students of Dr. Smith’s FCTL) without whom I may not have pursued graduate school. A special thanks goes to Jesse for assisting greatly in helping get our current research to a phase where growth was imminent after this document has been received.
I would also like to mention a few members of my team at NVIDIA (Ratnesh, Arihant, and Varun) with whom I discussed many of the ideas presented in this thesis (especially the topics involving re-identification) as well as submitted a paper and patent to go along with the work.

Lastly, I would like to acknowledge the Engineering and Science Education department at Clemson University for providing the Graduate Assistants in Areas of National Need (GAANN) Fellowship. This fellowship has provided learning experiences in not only education research but in correct teaching practices and methods. I would also like to acknowledge André Luckow at BMW for providing funding for computing hardware as well as travel expenses for attending top-tier computing conference, without which I would not have been able to see first hand the research being performed in the subject areas.
# Table of Contents

Title Page ................................................................. i
Abstract ................................................................. ii
Dedication ............................................................... iv
Acknowledgments ....................................................... v
List of Tables ........................................................... ix
List of Figures .......................................................... x

1 Introduction .......................................................... 1

2 Background & Literature Review ......................................... 4
   2.1 Machine Learning .................................................. 4
      2.1.1 Machine Learning Basics .................................. 4
      2.1.2 Neural Networks ............................................. 7
         2.1.2.1 A Single Neuron ...................................... 7
         2.1.2.2 Feed-forward Network ............................... 9
         2.1.2.3 Multi-Layer Perceptron ............................. 9
         2.1.2.4 Training and Inference ............................ 10
      2.1.3 Deep Learning ............................................... 13
         2.1.3.1 Convolution and Convolutional Layers ............. 14
         2.1.3.2 Deep Learning Frameworks ......................... 16
   2.2 Computer Vision with Deep Learning ............................. 17
      2.2.1 Image Classification ..................................... 17
      2.2.1.1 Datasets .............................................. 18
      2.2.1.2 Architectures ....................................... 19
      2.2.2 Object Detection .......................................... 21
      2.2.2.1 Datasets ............................................ 22
      2.2.2.2 Architectures ....................................... 22
      2.2.3 Re-Identification ......................................... 25
      2.2.3.1 Datasets ............................................. 28
      2.2.3.2 Architecture and Loss Function .................... 30
   2.2.4 Hardware ...................................................... 32
      2.2.4.1 Training ............................................... 32
      2.2.4.2 Inference ............................................ 32
   2.3 Summary .......................................................... 33

3 Research Design and Experiments .................................... 34
List of Tables

3.1 NVIDIA Drive PX2 Specifications ........................................ 38
3.2 Titan X Inference Times ...................................................... 44
3.3 NVIDIA Jetson TX2 Specifications ........................................ 45

4.1 Faster-RCNN Inference Times ............................................. 64
4.2 TensorRT Classification Model Optimizations on Jetson AGX Xavier (time in ms) . 81
4.3 TensorRT Detection Model Optimizations on Jetson AGX Xavier (time in ms) . . 82
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>AI Overview</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Single Neuron</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Activation Functions</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Shallow Feed Forward Network Architecture</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Multi-layer perceptron</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>Convolution [6]</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>Convolution Filters [7]</td>
<td>15</td>
</tr>
<tr>
<td>2.8</td>
<td>Receptive Field [8]</td>
<td>16</td>
</tr>
<tr>
<td>2.9</td>
<td>Image Classification</td>
<td>18</td>
</tr>
<tr>
<td>2.10</td>
<td>AlexNet Architecture</td>
<td>19</td>
</tr>
<tr>
<td>2.11</td>
<td>VGG Architecture</td>
<td>20</td>
</tr>
<tr>
<td>2.12</td>
<td>GoogleNet Architecture</td>
<td>20</td>
</tr>
<tr>
<td>2.13</td>
<td>ResNet Unit</td>
<td>21</td>
</tr>
<tr>
<td>2.14</td>
<td>Faster RCNN Architecture</td>
<td>24</td>
</tr>
<tr>
<td>2.15</td>
<td>YOLO Example</td>
<td>25</td>
</tr>
<tr>
<td>2.16</td>
<td>SSD Architecture</td>
<td>25</td>
</tr>
<tr>
<td>2.17</td>
<td>Samples from VeRi [9] Dataset (Each Row is Separate Identity)</td>
<td>27</td>
</tr>
<tr>
<td>2.18</td>
<td>Re-Identification Process for Creating Embeddings for Objects</td>
<td>29</td>
</tr>
<tr>
<td>2.19</td>
<td>Vehicle Re-Identification Datasets</td>
<td>29</td>
</tr>
<tr>
<td>2.20</td>
<td>Triplet Loss Optimization</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Autonomous Driving Architecture</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>NVIDIA Drive PX2 Architecture</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Perception Module</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>Example Images from the KITTI dataset</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>Example Images from the LISA dataset</td>
<td>40</td>
</tr>
<tr>
<td>3.6</td>
<td>DetectNet Architecture [10]</td>
<td>41</td>
</tr>
<tr>
<td>3.7</td>
<td>DetectNet Inference Architecture [10]</td>
<td>42</td>
</tr>
<tr>
<td>3.8</td>
<td>DetectNet-Depth Architecture</td>
<td>42</td>
</tr>
<tr>
<td>3.9</td>
<td>DAC Pipeline</td>
<td>45</td>
</tr>
<tr>
<td>3.10</td>
<td>DAC Dataset Samples</td>
<td>46</td>
</tr>
<tr>
<td>3.11</td>
<td>Embedded SSD (eSSD) Architecture</td>
<td>48</td>
</tr>
<tr>
<td>3.12</td>
<td>Barcode Label Synthetic Dataset Samples</td>
<td>51</td>
</tr>
<tr>
<td>3.13</td>
<td>Barcode Label Detection Pipeline</td>
<td>52</td>
</tr>
<tr>
<td>3.14</td>
<td>YOLO Architecture</td>
<td>53</td>
</tr>
<tr>
<td>3.15</td>
<td>Tiny-YOLO Architecture</td>
<td>54</td>
</tr>
<tr>
<td>3.16</td>
<td>Trailer ID/Parking Lot Detection Pipeline</td>
<td>54</td>
</tr>
<tr>
<td>3.17</td>
<td>Trailer ID/Parking ID Dataset Samples</td>
<td>55</td>
</tr>
<tr>
<td>3.18</td>
<td>Batch Data Selection Techniques</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Intersection Over Union (IoU)</td>
<td>63</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Gallery Image Ranking Based on Probe Image</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Faster-RCNN Vehicle/Pedestrian Detection</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Example 1: DetectNet Depth Image</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Example 2: DetectNet Depth Image</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Jetson TX2 Inference Times on KITTI Images</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Inference on KITTI Cars</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>Inference on KITTI Pedestrians</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>DAC mAP Results</td>
<td></td>
</tr>
<tr>
<td>4.11</td>
<td>Evaluation Results of eSSD Architecture</td>
<td></td>
</tr>
<tr>
<td>4.12</td>
<td>DAC Challenge Output 1</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>DAC Challenge Output 2</td>
<td></td>
</tr>
<tr>
<td>4.14</td>
<td>Barcode Label Detection Performance</td>
<td></td>
</tr>
<tr>
<td>4.15</td>
<td>Synthetic Inference</td>
<td></td>
</tr>
<tr>
<td>4.16</td>
<td>Real Warehouse Barcode Detection</td>
<td></td>
</tr>
<tr>
<td>4.17</td>
<td>iPhone 7 Plus Label Detection Inference</td>
<td></td>
</tr>
<tr>
<td>4.18</td>
<td>Trailer ID/Parking Number Detection Performance</td>
<td></td>
</tr>
<tr>
<td>4.19</td>
<td>Trailer ID Detection and Preliminary OCR</td>
<td></td>
</tr>
<tr>
<td>4.20</td>
<td>Visual Results for Vehicle Re-Identification with VeRi Dataset. Green are Correct Retrievals while Red are Incorrect Retrievals (All from Different Cameras)</td>
<td></td>
</tr>
<tr>
<td>4.21</td>
<td>TensorRT Inference Optimizations [12]</td>
<td></td>
</tr>
</tbody>
</table>
Nomenclature

$AD$  Autonomous Driving
$ADAS$  Advanced Driver Assistant Systems
$ANN$  Artificial Neural Networks
$CNN$  Convolutional Neural Network
$DL$  Deep Learning
$DNN$  Deep Neural Network
$EC$  Edge Computing
$eSSD$  Embedded Single-Shot Multi-box Detector
$FPS$  Frames Per Second
$IoU$  Intersection over Union
$mAP$  Mean Average Precision
$MLP$  Multi-layer Perceptron
$ReLU$  Rectified Linear Unit
$RPN$  Region Proposal Network
$SSE$  Sum of Squared Errors
$UAV$  Unmanned Aerial Vehicle
In recent years, deep learning has become one of the largest contenders in many different fields of research. In particular, fields such as computer vision, robotics, and natural language processing have been effected greatly. With the success of deep learning architecture like CNNs, developers now have the ability to no longer hand craft features or algorithms for their specialized applications. Deep learning has shown itself to be efficacious in learning abstract patterns in data rather than hand-designing them for a specific application. DNNs (in particular CNNs) have become the state-of-the-art in their ability to not only learn representations of the data but also construct features at many scales, orientations, etc. from a trainable dataset. An algorithm that has the ability to learn on its own rather than be instructed through the learning process has been the impetus for the growth of deep learning in a wide range of applications. Through extraction of features enumerated in the learned weights of a DNN, they are able to compete with (if not exceed) the accuracy and speed of traditional computer vision tasks with the added advantage of not having to hand-craft algorithmic-specific and data-specific features.

Many of the cutting-edge breakthroughs that have occurred in recent years have been either optimizing the training and inference process, novel frameworks and architectures, or using and existing architecture to tackle a real-world application or problem (the focus of most of this research). Some of the heaviest players in the field of deep learning are start-ups, companies, and university teams focusing on automation and autonomous systems. For example self-driving cars or self-flying drones are defined by their ability to act on demand rather than having a human control them. For this they need a computer vision system capable of not only executing in real-time, but also...
providing useful information to the subsystems which control the motion and other components. Even though these systems attempt to use cutting-edge technology, they are often hindered by the safety constraints imposed on them by obstacles in the environment. For this reason, systems that are deployed on mobile robots like drones or self-driving cars must not only go through rigorous testing but also be able to execute in real-time to combat issues that may occur in the environment.

When comparing all autonomous systems, to some degree, they all utilize the same technology for motion in an environment. First, there must be some subsystem which can reason about the objects or important artifacts of an environment and then decisions must be made based on these artifacts. For autonomous driving, some of these artifacts could include location of all vehicles and pedestrians in the local vicinity as well as information about the road itself including lane markings, turns, etc. The main object of any supervised autonomous system is to retrieve some information from its sensors, extract and process information about the scene, and the pass along that information to a planning module which can make decisions about proposed path and speed for the system. Due to the complexity of these systems, developers look for every portion which can be accelerated or re-factored in an effort to learn a better representation of its environment while also making it as efficient and fast as possible.

In this work, we set out to tackle four different tasks: autonomous driving, drone surveillance detection, warehouse management systems, and vehicle re-identification. With each of these tasks, we will be working towards a real-time solution (being able to run inference on models in real-time). In collaboration with CUICAR, we have developed a perception module for autonomous driving with the ability to not only estimate the position of important objects in the scene such as vehicles, pedestrians, and signs but also estimate their distance from the camera (or autonomous system mounted camera). We have also developed an inference engine with the ability to detect objects in a drone-vision (aerial) setting including vehicles, pedestrians, boats, and many more objects. Further, we have developed a two DNN-based system for BMW: one for localizing and scanning barcodes on shipping labels and a second for detecting trailers in a train yard as well as the parking spots in which they are currently located. Lastly, we have developed a solution for identifying vehicles within a multi-camera system without the need for camera-to-camera tracking; rather establishing a signature for each of the vehicles.

The remainder of this dissertation is organized as follows. Chapter 2 provides background information on topics from computer vision to neural networks to deep learning to re-identification.
Also, throughout Chapter 2, we have described some of the relevant literature that has been developed in fields like CNNs, deep learning, object detection, and re-identification. The combination of background and literature review amalgamate to provide the motivating force for designing the numerous applications in this dissertation. Chapter 3 presents each of the applications as well as the overall system design and DNN architecture designs (and trade-offs) specific to each application. Also Chapter 3 enumerates the experiments that were completed with developing each application.

We wrap up this dissertation with Chapter 4 presenting the results for each application and experiments as well as conclusions for each. Chapter 5 concludes by enumerating the contributions that this work has provided (including best practices for creating an edge computing system) as well as provide a few future work steps that could be taken which were not in the scope of this research.
Chapter 2

Background & Literature Review

In this chapter, basic machine learning and computer vision concepts will be discussed as an introduction to the applications to be discussed as part of the work provided for this dissertation. First, we will discuss machine learning basics including machine learning algorithms, neural networks, deep learning and optimization techniques for these algorithms (i.e. "training"). Next, we will discuss traditional computer vision and how it has played a role in recent history in the birth of deep learning and feature extraction networks. In this section we will also discuss traditional architectures for each of the computer vision-based deep learning techniques. Finally, we will discuss the concept of metric or representation learning, which will allow for a discussion of triplet loss and the re-identification problem.

2.1 Machine Learning

In this section, we will discuss machine learning basics as well as provide more in-depth discussions of both traditional neural networks and their evolution into deep neural network models. A discussion of deep learning frameworks as well as how to utilize them for training and inference are also discussed in this section.

2.1.1 Machine Learning Basics

Machine learning (often interchanged with the terms artificial intelligence and deep learning) is actually a subset of artificial intelligence and a superset of deep learning. To be clear, artificial
intelligence is typically used for a much broader concept of machines that have the ability to carry out tasks in a manner by which they are considered "intelligent". Machine learning, on the other hand, is built around the idea that providing a machine with a certain amount of data, it should be able to learn something meaningful on its own. Goodfellow, et al. [13] illustrate this through a very helpful diagram and flowchart, shown in Figure 2.1.

As the Venn diagram alludes to, deep learning can be thought of as a type of representation learning, which is a type of machine learning. Following this, machine learning is shown as a subset of AI; however, it should be mentioned that not all AI approaches utilize machine learning as a solution. Machine learning can be thought of as a system that is provided some sort of data $D$, placed in some "environment" $E$, is designed for a specific set of tasks $T$ and is evaluated on some performance metric $P$. A mathematical interpretation of this is shown below as an optimization problem.

$$
\begin{align*}
\maximize_D & \quad P(D) \\
\text{subject to} & \quad T_i(D) \subset E, \ i = 1, \ldots, m.
\end{align*}
$$

This equation states that with any given machine learning (also deep learning algorithm), the goal is
to maximize the performance metric, $P$, given some data $D$ by training on $m$ tasks $T_i$ contained in the training environment $E$. For example, we may be asked to train a clustering algorithm to determine the differences in certain types of clothing (i.e., shirts, shorts, jeans, shoes, etc.). Our clustering algorithm (machine learning algorithm) will take as input a set of data $D$ which is representative of the data we would like to cluster. We will then perform distance calculations and mean updates (tasks $T_i$) inside the environment (i.e., group of data for clustering) that will group all like objects together. The performance metric we will be maximizing is the number of correct examples that are placed in each group after training (assuming we have labeled data).

When compared to traditional algorithms, machine learning is able to aide developers and scientists in creating easier solutions to higher dimensional problems which before needed hand crafted algorithms developed by experts in the field. As can be seen in the Venn diagram in Figure 2.1a, logistic regression is a perfect example of a traditional machine learning algorithm. Logistic regression is involves solving for some output $y$ given some set of inputs $x$ and which are linearly combined with equation coefficients (later called weights in neural networks) forming the equation $y = w^T x$ where $y$ is the output, $x$ is the input, and $w$ are the equation coefficients. Later we will discuss the $w$ terms as the weights of a neural network rather than coefficients of an equation. Figure 2.1b illustrates that for classical machine learning techniques, there are hand-designed features that are used for converting the inputs to features which the machine learning algorithm can use.

A subset of machine learning is representation learning. In representation learning, we have removed the dependency on hand-crafted features by the designer and allow the algorithm (network or other) to develop its own interpretation of the data, in turn developing its own set of features. This can be seen in the second from right column of the flow chart in Figure 2.1b. An example of this would be a simple neural network which will be discussed in the next section in more detail. However, in short, the neural network is able to use the data provided to create a "representation" of the data (normally in a smaller dimensionality than the original data). The goal of representation learning is to allow the system to learn representations of the data rather than the scientist or developer hand-crafting each feature; most of the time, this yields to much higher performance. Numerous algorithms including neural networks, deep learning, and numerous unsupervised techniques utilize the idea of representation learning to learn from a large set of data.

In the next section, we will go more in depth into artificial neural networks and how they are designed to learn from a dataset.
2.1.2 Neural Networks

An Artificial Neural Networks (ANN) is a computational model which is slightly modeled by the way biological neural networks in the human brain process information. With that said, ANNs (at least those designed at the moment) are nowhere near as complex as those found in the human brain. Over the past decade, ANNs have generated large amounts of excitement due to their groundbreaking results in fields such as computer vision, speech recognition, and many more. In this section we will discuss the building blocks of a neural network (i.e. a single neuron), the feed-forward architecture and then discuss the multi-layer perceptron.

2.1.2.1 A Single Neuron

The basic unit of computation in a neural network (also in deep neural networks) is a neuron (sometimes also called a unit or a node depending on the architecture and framework that is used). Overall, the function of a single neuron is quite straightforward; it receives input from either the outside world or other neurons, combines them with weights (which are learned through a process of training), then passed through a function $f$. Figure 2.2 illustrates a typical model for a single neuron. Again, there are many other models of a neuron; however, this model is used in most implementations of (deep) neural networks.

The model of a single neuron shown in Figure 2.2 can be functionally defined as a non-linearity ($f$) of an affine transformation. Given the set of inputs $x$ and the network weights $w_i$, we
can calculate the output of a single neuron $i$ with the following equation:

$$o_i = f(x; w, b) = f\left(\sum_j w_{ij}x_j + b\right) = f(w^T x + b) \tag{2.2}$$

Equation 2.2 illustrates the linear combination of the weights and inputs of the neuron with an added bias term $b$. The bias term is a term added to each neuron that gives the neuron the ability to reason with an additional dimension as well as avoid an input vector of all zeros. Although we are adding an additional input to each neuron, we are not losing any generality; in the best case, by adding dimensionality, we are adding generality to the network and giving it the ability to map to a larger range of values.

Typically, the set of all parameters (including weights and biases) are condensed into a single parameter $\theta$. Equation 2.3 illustrates how $\theta$ represents the parameters of the network in a more general manner.

$$o_i = f(x; \theta) = f(\theta x) \tag{2.3}$$

The non-linearity function $f$ (also called an "activation function" or "transfer function") is usually defined as a differentiable function like hyperbolic tangent or a rectified linear unit (ReLU). Figure 2.3 illustrates examples of the hyperbolic tangent and ReLU activation functions. The purpose of using non-linear transfer/activation functions is to create a model that is more generalizable with the ability to adapt to a variety of data (rather than simply having a linear mapping from input to output).
2.1.2.2 Feed-forward Network

Single neurons create a basis for modeling a mathematical function; however, with a single neuron, the complexity of the function which it is able to model is limited. For this reason, layers of neurons are created in an attempt to teach the system a better representation of the governing mathematical equation. A traditional feed forward neural network consists of an input layer, an output layer, and a numerous hidden layers. For this particular example, we will presume a single hidden layer (and expand it in Section 2.1.2.3). Figure 2.4 illustrates a generic model for a feed-forward network (note the input, output, and single hidden layers). The input layer consists of an input vector $\mathbf{x} = x_1, ..., x_k$, the hidden layer consists of a vector of $N$ neurons $\mathbf{h} = h_1, ..., h_N$, and an output layer consisting of an output vector $\mathbf{y} = y_1, ..., y_M$. A staple feature of the feed-forward network is that every element in the input layer is connected to every neuron in the hidden layer with weights $w_{ki}$; this indicates the network weight between the $k^{th}$ input elements and the $i^{th}$ hidden neuron. Similarly, the weights from the hidden layer to the output layer can be defined as $w_{ij}$ illustrating the connection between the $i^{th}$ hidden layer neuron and the $j^{th}$ output neuron. The weights of the network ($w_{ki}$ and $w_{ij}$) can be solved for by "training" which will be discussed for the general case in Section 2.1.2.4.

2.1.2.3 Multi-Layer Perceptron

The multi-layer perceptron (MLP) is an extension of the concepts derived for a feed-forward network, however, an MLP consists of multiple hidden layers (rather than just a single hidden layer).
By extending the neural network architecture to contain more layers, in essence, we are allowing the network to learn a more complex representation of the mathematical equation to map from input to output. Figure 2.5 illustrates an MLP architecture with multiple hidden layers. As with the traditional feed-forward architecture, each node in a given layer is connected to all previous and all subsequent nodes. This network architecture is known as fully connected; as this network architecture begins to grow, the trainable parameters grow drastically leading to a training time that increases dramatically as the network grows. As discussed in Section 2.1.2.2, the function for a single hidden layer network (containing weights \( w \), bias \( b \), and inputs \( x \)) is given by the non-linear mapping \( y = f(x; \theta) \); furthermore, to extend this mapping to multiple hidden layers, multiple non-linear functions can be used in the form \( y = f_2(f_1(x; \theta)) \), where \( f_1 \) represents the mapping for the first hidden layer and \( f_2 \) represents the mapping of the second hidden layer. Again, we will discuss training in general in Section 2.1.2.4. In Section 2.1.3, we will discuss how this architecture was expanded even further as well as different style architectures built to model computer vision techniques (as well as improve training time).

2.1.2.4 Training and Inference

**Training:** Before discussing deep learning, we will first go through an introduction of the training and inference process or neural networks (as they are similar, if not the same, for deep learning models as well). To successfully create a neural network that is able to map inputs to outputs correctly, the weights of the neural network must be iteratively trained or optimized. Typically, this optimization process is called back-propagation and uses a technique called gradient descent. Gradient descent is an optimization approach which attempts to find function parameters (in this case, neural network weights) that will minimize the cost function (i.e. minimize the error in which
the neural network performs). Gradient descent utilizes the negative gradient of the optimization (cost/error) function to optimize the network parameters that minimize the loss computed with input examples. The following technique discusses the process by which neural network parameters are optimized (trained). Most of the derivation provided has been slightly modified from information presented in Chapter 12 of [14] by Robert Schalkoff.

First, we would like to calculate how much error has been accumulated as the network is presented training examples. For this, we will use the metric sum of squared errors (SSE) which computes the error between the output of the network \( o^p \) and the expected output (target/label) \( t^p \) where \( p \) is the input pattern given to the network. If we assume to be using sigmoid-type activation functions in each our the neurons, we can utilize the Generalized Delta Rule (GDR) in order to update the weights in each neuron of each layer. Information regarding the derivation of the GDR can be found in [14]. Assuming we develop an error definition as shown in Equation 2.4, we can define the error metric for the output layer as well as any hidden layers in the network.

\[
E_p = \frac{1}{2} \sum_j (t^p_j - o^p_j)^2 \quad (2.4)
\]

If we presume activation functions of type sigmoid, we can derive the output equation for a given pattern as well as the derivative computation that will allow us to "follow the negative gradient" to optimize the model. Equation 2.5 illustrates the output equation assuming a sigmoid activation and Equation 2.6 illustrates the derivative computation to be used for back-propagation. In Equations 2.5 and 2.6, \( net_j \) represents the output of the neuron (unit) before the activation function has been applied.

\[
o_j = f(net_j) = \frac{1}{1 + e^{-net_j}} \quad (2.5)
\]

\[
f'(net_j) = o_j(1 - o_j) \quad (2.6)
\]

Next, for the output neuron, we would like to compute the error metric associated with the provided pattern. First, we will use the error definition given in 2.4 to derive 2.7. Following, we compute the weight correction for the hidden layer to output layer neuron weight \( w_{ji} \), shown in Equation 2.8. \( \epsilon \) in Equation 2.8 illustrates the learning rate for the back-propagation algorithm.
The learning rate is a hyper-parameter of the network which helps control how much weights are adjusted in an attempt to account for overshooting a local minima value. The parameter $\delta^p_i$ is a generalized output for any of the neurons in the network and its formulation can be seen in Equation 2.9.

\[
\frac{\partial E^p}{\partial o^p_j} = -(t^p_j - o^p_j) \\
\delta^p_j = (t^p_j - o^p_j)f'_j(n^p_j) \\
\Delta^p w_{ji} = \epsilon(t^p_j - o^p_j)f'_j(n^p_j)\tilde{o}^p_i
\]

(2.7)

(2.8)

\[
\tilde{o}^p_i = \begin{cases} 
\delta^p_i, & \text{if input is the output of a neuron in a previous layer (hidden & output layers)} \\
i_i, & \text{if input is a direct input to the network (input layer)} 
\end{cases}
\]

(2.9)

The above derivations work well for computing errors and weight updates for those neurons associated with the output layer, however, due to the 'indirect' effects of the weights in the hidden layer to $E_p$ shown in Equation 2.4, GDR must be derived for weight updates of hidden layer weights. Rumelhart et al. derive GDR in [15] illustrating the use of the chain rule to back-propagate errors through a multi-layer network recursively. With that said, we must generate a general equation (recursive formulation) which will update any weight in the network. Again, a full derivation can be found in [14] leading to the weight update equation shown in Equation 2.10. The difference in this equation and that of the output neuron weight update is the $\delta^p_n w_{nk}$ term which is obtained by solving for the weight updates on the output layer.

\[
\delta^p_k = f_k^i(n^k_k) \sum_n \delta_n^p w_{nk}
\]

(2.10)

To conclude the discussion of training, we will provide simple step enumerating the steps described in detail above (assume that we have initialized the weights of the network using some random fashion).

1. Provide an input $i_p$ to the network and calculate output $o_i$ for each neuron in the network.

2. Use Equation 2.8 to update all output layer weights.
3. Use Equation 2.10 to update all weight in hidden layers.

4. Iterate until either an iteration threshold has been reached or the changes in weights are insignificant.

As GDR and back-propagation began to gain popularity, it became clear that training larger networks would be very time-consuming and would require massive amounts of computational resources. One of the major reasons for the large requirements are the fully-connected layers which comprise MLPs. For this reason, many developers and deep learning patrons decided to switch gears and develop newer models which are not only easier to train but also much more computationally efficient due to sparseness in the networks. Section 2.1.3 will go more into depth about this concept and how it has framed this research endeavor.

Inference: Inference is the process of taking an input (whether that be a simple pattern, an image, a video, a speech sample, etc.) and passing it through your machine learning algorithm (in this case, we will focus on neural networks) in what is called a "forward pass". This is the same process that is done when training (before the weights are updated). We can assume that we have already gone through some training process as described above and now we want to make sure our model performs adequately. Unlike training, there is no need to include a backward pass to compute errors and update weights. Usually, the inference phase of a pipeline can also be called deployment as one is using the trained model to predict outcomes on real-world data.

2.1.3 Deep Learning

Two of the pioneers of DL (Yann LeCun and Yoshia Bengio) drew from the limitations described above in the training and architecture aspects of MLP and researched new methods for using machine learning. LeCun et al. in [16, 17, 18] developed the idea of Convolutional Neural Networks (CNN) which sparked the new field known as "Deep Learning". The architecture which was proposed in [18] is called LeNet and was used to perform the task of image classification on the MNIST dataset (a dataset of handwritten digits created by LeCun and associates also derived in [18]). One of the key points of this research that created an abundance of further research directions was LeNet and its convolutional layer’s ability to extract meaningful features and pass those onto fully connected layers for classification. This paradigm leads to what was referred to previously in this document as "representation learning" where the CNN is able to create its own set of learned
features without the intervention of the developer or scientists. The main building block of CNNs is the concept of convolution (and the convolutional layer); this concept will be discussed in the Section 2.1.3.1.

2.1.3.1 Convolution and Convolutional Layers

The main building block of a CNN is the convolutional layer (containing learned convolutional filters). As with its counterpart in computer vision, the convolutional filters that are learned (similar to convolution kernels) are used to develop feature maps which provide the network with useful information about classifying the intended objects.

Before exploring convolutional filters in a neural network context, first we must understand how convolutions work in traditional computer vision (and also why we use them). On the left of Figure 2.6, we can see our input pattern (in this case, let’s assume each value represents a pixel value), in the center we can find the convolutional filter, and on the right we can see the output (destination pixel value). The process of convolution is simple, we will use a sliding window approach and slide the convolutional kernel across the input image from top left to bottom right. At each location, we will compute a weighted sum of the input pixels and the convolution kernel values producing a single value. Each weighted sum will produce a different portion of the output matrix which, as a whole, defines the output "feature map".

Some desired output from traditional convolutional filters (which have been hand designed) can be seen in Figure 2.7. The filters designed for this experiment were particularly designed to
pick up on edges (both horizontal and vertical) in the image. However, for real-world applications, there is an inherent need to not have to hand create every filter for an image classification problem. CNNs were designed for exactly this task. The filters that are learned in a CNN directly correlate to the filters that are used in a traditional convolutional computer vision problem.

Convolutional layers in a CNN learn these filters through the process of training and are then able to generate feature maps similar to that of the output shown in Figure 2.7. However, one main difference is that these features are completely learned and they encode specific details within the image at each location. Like traditional convolution, the filters that are learned are typically 3x3 or 5x5. In Section 2.2, we will discuss more in depth different architectures which employ convolution for the tasks of image classification, object detection, and segmentation. One last concept that needs to be discussed is the concept of receptive fields in a CNN. Presuming a 3x3 filter in a given convolutional layer, the receptive field of a particular neuron is the region of space in the previous filters which cause the neuron to fire or not. Figure 2.8 illustrates the receptive field of a single neuron looking backward in the network. As you proceed deeper into a network, the receptive field grows; in essence, smaller features are being created in the filter maps in earlier layers of the network and larger features from the image are being created in later layers.

Pooling is also a large contributor to the success of CNNs. Pooling layers are typically used to progressively reduce the spatial size of the feature map representation, thus reducing the number

Figure 2.7: Convolution Filters [7]
of parameters in effect reducing the amount of computation in the network. Another problem with deep learning is the concept of overfitting. While training a neural network (if not done properly), it is easy to over-train or over-fit to the training dataset. This will cause comparatively worse performance on a test set which the network has not been exposed to when compared to the training set. By inserting a "pooling" layer after convolutional layers in a network, we are effectively providing the model with the ability to be invariant to scale, shifts, and distortions of the objects in the images; this will also help control the negative effects which cause overfitting. There are a few methods for pooling including max-pooling, average-pooling, and a few others. These pooling methods simply take the max or average, respectively, over the area in question.

2.1.3.2 Deep Learning Frameworks

This section will briefly discuss the different frameworks that are available for development of deep learning architectures. Only the frameworks that have previously been used or plan to be used will be described in detail.

**Caffe** [19] is a deep learning framework developed with modularity and speed in mind by Berkeley AI Research (BAIR) and begun by Yangqing Jia during his PhD at UC Berkeley.

**Darknet** [20] is an open source deep learning framework completely written in C and CUDA delivering a fast and easy interface for model creation and was developed by Joseph Redmon at the University of Washington.

**Tensorflow** [21] is an open source software developed by Google for numerical computations using data flow graphs. These graphs are typically neural networks, however, any computation graph can
be derived using Tensorflow.

**PyTorch** [22] is a Python package developed by Facebook replacing all Numpy calculations with GPU computations as well as providing an automatic gradient calculation mechanism.

Other frameworks (that had no part in the development of this dissertation, but are valid choices as frameworks) include Chainer [23], MXNet [24], and many others that are failed to be mentioned.

### 2.2 Computer Vision with Deep Learning

Deep learning, in all essence of the term, technically means any ANN that has multiple layers of non-linear transformations. However, typically deep learning refers to networks that perform feature extraction with the ability to learn the representations of the data rather than being taught. LeCun et al.’s success with CNNs provided a solid base for further research in deep learning. Most architectures today can draw their ancestral roots to LeNet or a modified-LeNet architecture. In the following sections, we will discuss the history and of CNN architectures as they apply to computer vision tasks like image classification, object detection and re-identification. In each section, we will elaborate on the well known dataset developed for each task as well as the incremental developments in architectural design that have lead to the work completed for this dissertation.

#### 2.2.1 Image Classification

The first task that we will discuss is image classification, which consequently was the first computer vision-type task solved with deep learning. This task can be easily described as presentation of an image to a neural network where the output is given as the prediction of which object is in the image. Figure 2.9 illustrates a typical image classification pipeline; provide the network with an image of a dog and then network makes a prediction of "dog" with its highest probability. The softmax layer at the end of the network allows for classification of the objects in the dataset. It provides probability values for each of the classes being in the image (i.e., a higher probability for class *dog* if there is a dog in the image).
Figure 2.9: Image Classification

2.2.1.1 Datasets

As one of the first (as well as simpler) problems in deep learning for computer vision, there are many datasets that are used as baselines for new architectural designs and optimization.

**MNIST** is a handwritten digits dataset (i.e. containing the numbers 0-9) developed by LeCun et al. for use with their architecture LeNet. This dataset is used as a benchmark for classification performance when new loss functions or architectures are being developed. Many modern networks have been able to reach upwards of 99% accuracy due to its small resolution of 32x32 pixels per image and small size of only 10 classes; therefore, this dataset is being used now as a beginning dataset to test the correctness of a model.

**CIFAR10/CIFAR100** are datasets containing 60,000 32x32 color images containing natural objects such as airplanes, automobiles, dogs, etc. CIFAR-10 is split into 10 general classes while CIFAR-100 is split further into subcategories totaling 100. This is a smaller natural image dataset that can be used to train relatively quickly for network testing and evaluation.

**ImageNet** is one of the world’s largest public image datasets containing 1000 classes of images. The ImageNet Large Scale Visual Recognition Competition (ILSVRC) allows contestants to train their networks on the 1000 class dataset and provides a leaderboard for the winners of the competition. This challenge lead to the ANN architecture called AlexNet [25] (discussed more in depth in the next section) which began the deep learning hype cycle by outperforming all other computer vision-based techniques by a significant margin.
2.2.1.2 Architectures

There are many architectures presently used for image classification. For purposes of this document, we will discuss the most profound accomplishments to the field of image classification. Following the example of LeNet, one of the pioneer architectures of the field of computer vision (image classification, in general) is AlexNet [25] created by Alex Krizhevsky while working under Geoffrey Hinton. AlexNet is composed of 5 convolutional layers that perform feature extraction and then are followed by 3 fully-connected layers whose purpose is to output a classification percentage for each of the 1000 classes in the form of a 1000-way softmax function. The output with the highest classification percentage is taken as the object which is present in the image. Figure 2.10 illustrates the architecture developed by Krizhevsky et al.; notice that the architecture is split into two separate portions. The architecture is split due to computational limitations of the GPUs at the time of this networks creation. Each half of the model was placed on difference GPUs and trained separately (oddly enough, both sections learned their own independent features). The 11x11 blocks shown on the far left of the image illustrate the convolutional filters that are learned throughout the training process with an input image, in this case, of size 224x224x3 (i.e., an RGB image). The first layer depicted (the bottom half of the first layer, to be more exact) is of size 48x55x5. This means that there are 48 independent filters in this layer created by sliding the 11x11 convolutional kernel with a stride of 4 (skipping 4 pixels every slide). Further the max pooling after this layer creates a smaller dimensionality layer as well as attempts to filter out features that are not needed (i.e., feature that aren’t edges, corners, etc.)

In 2014, Simonyan and Zisserman from the University of Oxford proposed an architecture
for classification called VGG [26]. The key features that made this architecture revolutionary are keeping the convolutional filters as simple as possible as well as growing the number of layers significantly to 19 layers. Another main idea of this paper is to stack convolutional layers before pooling rather than pooling after each layer (as AlexNet has done), while still being able to improve the error rate to 7.3% on the ImageNet challenge. Figure 2.11 illustrates the VGG architecture (notice the much smaller filters sizes creating much smaller convolutional layers).

Also in 2014, Szegedy et al. [27] went with a different strategy which was creating a more complex architecture, naming it GoogleNet (depicted in Figure 2.12). One of the main contributions of this architecture is its Inception module which is able to process inputs in parallel through multiple 1x1 and 3x3 convolutions. Although the architecture is much bigger and more complex than previous architectures, it minimizes the total number of parameters, therefore speeding up inference time.

Lastly, one of the most important innovations in recent years in image classification is the development of ResNet [28] by Kaiming He et al. One of the major problems as models grow
deeper is the problem of vanishing gradients. In short, the problem is that using the chain rule for back-propagation only works through a finite number of layers before the errors being propagated are non-negligible. To combat this problem, Kaiming He et al. showed that by inserting a skip-connection in the architecture, the optimization is able to learn a residual mapping instead of the mapping itself. This architecture won the ImageNet 2015 Challenge with 152 layers and a residual unit is shown in Figure 2.13.

2.2.2 Object Detection

While image classification has been well studied over the years and architectures have been developed to perform extraordinarily well with this task, there are many instances where simply identifying the objects in an image is not enough. The bulk of this dissertation is based on this concept. Rather than simply stating that a particular image contains a "dog", we would like a system that both says there is a "dog" and the dog is located at \((x_1, y_1), (x_2, y_2)\) position in the image. However, we can not simply use the same architectures that were used for image classification because the output for these networks are softmax probabilities which include no location information. Although it is not done this way in practice, we can think of object detection as an extension of image classification where we develop a heat map where objects could potentially be located in the image and then classify each of the areas as either an object or not. Another way to think of the object detection problem is to have a sliding window go over the entire image and pass each window through a classifier. In this section we will discuss in detail a few architectures which have come to be the state-of-the-art architectures for object detection as well as mention a few datasets that are available for particular applications.
2.2.2.1 Datasets

COCO is an object detection dataset developed by Microsoft standing for Common Objects in Context. This dataset contains complex everyday scenes containing 91 objects in natural environments with about 2.5 million images. There are many APIs that provide pre-trained models that have been trained using the COCO dataset. These models are useful for using pre-learned features as starting points for custom datasets.

Pascal VOC is another object detection dataset developed for "large scale" image classification and detection. There are 500K images in this dataset consisting of about 20 classes of normal objects (including vehicles, buildings, etc.) Again, there are many APIs that provide pre-trained model trained on this dataset as a starting point for training with custom data.

KITTI is an autonomous driving detection dataset developed in a collaboration between University of Toronto and KIT. This dataset contains objects that would be relevant for autonomous driving models such as different types of vehicles, pedestrians, roads, etc. Not only does this dataset contain object detection labels, but there are many other types of data including road segmentation, stereo vision and lidar data for depth mapping, and optical flow dataset to track motion of objects. For our purposes, we will only be using the detection aspects of this dataset.

There are many other datasets for object detection depending on the task at hand including LISA (a traffic sign database), WIDER (a facial detection database), and ALPR (for license plate recognition).

2.2.2.2 Architectures

Architectures that are currently being used for object detection are much different than those being used for image classification (aside from the inherent backbone of convolutional layers). Again, the main difference is that we are required to predict not only the class of the object but also its location in the image which inherently involves more than a softmax probability layer.

The first recent technique that has been used for object detection does not involve a convolutional network at all. In 2005, Dalal and Triggs [29] introduced the concepts of Histogram Oriented Gradients (HOG) features which used a sliding window approach on a pyramid of scaled images. For each scaled image, HOG features were calculated and then fed into a support vector machine (SVM) to create the classifiers.
The development of HOG methods led directly to the creation of the first deep learning based object detection, Region-based Convolutional Neural Networks (R-CNN) [30]. First, the same strategy was employed as HOG, however, the SVM classifiers were replaced with CNNs. It was impossible, however, at this time to run CNNs on so many image patches due to computational limits. To circumvent this, R-CNN uses a technique called Selective Search which reduces the number of bounding boxes that are given to the classifier significantly. Once the patches are fed to the convolutional networks for feature extraction, they are then fed to SVMs to perform classification as well as a bounding box regressor.

Next, Girshick developed a technique called Fast R-CNN [1] to alleviate one of the main problems facing both R-CNN as well as other similar techniques: it was able to be trained in an end-to-end fashion. One other addition that was made was combining the bounding box regression into the neural network itself. This was accomplished by having two heads on the network: one classification head and one bounding box regression head.

Still further improvements could be made by Girshick et al. when developing the newest of the 2-phase networks, Faster R-CNN [2]. By far the slowest portion of the Fast R-CNN pipeline was the Selective Search algorithm. To speed up the training process (as well as the inference process), Faster R-CNN replaces the selective search component with a smaller CNN called the Region Proposal Network (RPN). Figure 2.14 illustrates the Faster R-CNN architecture which is still currently one of the highest performing architectures in terms of accuracy on datasets like Pascal VOC while being 10 times faster than Fast R-CNN.

For all the aforementioned architectures (R-CNN, Fast R-CNN, and Faster R-CNN), the object detection problem is broken down into a two-stage pipeline where object proposals are generated throughout the image and then a classifier and regressor are run on each proposal to determine its efficacy in the final output. However, on specific architectures (namely, embedded architectures), this heavy pipeline is not suitable for running CNNs in real-time. For this reason, there have been a few architectures developed that have been coined "single shot detectors" which look at the detection problem as a regression problem as a whole instead of a classification and regression problem separately.

The first architecture which attempted a "single-shot" mentality was YOLO [4]. Standing for "You Only Look Once" this detector was able to learn the class probabilities as well as bounding box coordinates together. The difference in two-stage techniques and the single-shot techniques is
that the former uses a set of object proposals to perform classification on while the latter uses a set of grid boxes overlaid on the image with different aspect ratios to localize objects. In short, YOLO divides the input image into a grid where each grid predicts N bounding boxes and corresponding confidence values. Techniques such as non-maximal suppression and thresholding are then used to remove extraneous boxes leaving behind the boxes which are best fit for the objects in the image. Figure 2.15 illustrates the full YOLO pipeline. One advantage of YOLO over R-CNN methods is seeing the entire image allows for contextual information to hinder the amount of false positives. However, due to the limitation of the architecture, sometimes it struggles to localize smaller objects. More recently, [5] was published extending the YOLO model to not only execute faster but also generalize much better; the new architecture design was able to detect up to 9000 classes of objects.

Following YOLO, the second single-shot architecture was developed, namely Single Shot Detection (SSD) [3]. SSD utilizes numerous great features from the YOLO architecture including predicting boxes based on a grid cell system. Two of the major differences between YOLO and SSD are that SSD predict off-sets based on grid cells rather than learning the box itself as well as predicting boxes at multiple scales by taking outputs from many subsequent convolutional layers. In Figure 2.16 it can easily be seen that the output is created by a combination of the outputs.
from many different convolutional layers. The initial layers (VGG-16 layers) are used for feature extraction while the following 5 convolutional layers are used for localizing objects of different aspect ratios (layers closer to the input looking for smaller objects).

One of the major problems in developing and deploying deep learning architectures for object detection in general is choosing the correct architecture to begin developing with. There is always a trade-off when it comes to deep learning models between accuracy and speed. If an application requires the fastest inference possible (for example, autonomous driving), an architecture that needs to make object proposals for every image and then classify each one of those may not be fast enough. On the other hand, for applications such as medical imaging where speed is not so important but the more accurate prediction can lead to a better medical outcome may lend itself to a deeper architecture.

2.2.3 Re-Identification

In contrast with image classification and object detection, the task of re-identification has not been studied as thoroughly and, therefore, there are fewer techniques that are appropriate when
designing a system. Image classification involves simply stating the presence of an object or item-of-interest in a given image. Object detection takes this one step further and not only identifies the object but also localizes it within the image. The task of re-identification involves extracting metadata (or a feature) from an already localized object for use with further processing (i.e. tracking, surveillance, security, etc.). However, this is already what a traditional image classification network is performing; feature extraction so that each class can be separated into its own portion of the embedding space. The major difference in the task of classification and re-identification can be illustrated by a problem involving hundreds of objects versus a problem containing millions of objects.

If there are only hundreds of classes (i.e. a dataset such as ImageNet), current networks have the ability to use traditional softmax layers to create a model with the ability to decipher between the classes easily. However, in the case of the million-class dataset (number of cars entering an airport in a period of time), it is nearly impossible to find a current model capable of using softmax classification to perform well. For this reason there are techniques that can be used to force objects of different classes (even if they are almost the same object) to be separate in embedding space representation. As most re-identification tasks utilize image classification networks, in this section we will focus on the changes made to a network as well as the special training methods used for creating a model capable of the aforementioned abilities as well as illustrate a few dataset that were used in the later section to solidify our findings. The specific task of vehicle re-identification will be studied in this work as the person re-identification is a much easier problem and is a well-studied problem. The problem we will discuss is creating a feature (signature) for each of the vehicles in a given dataset.

In video analysis, most higher level algorithms like action recognition and anomaly detection rely on traditional methods like Multiple Camera Multiple Object Tracking (MC-MOT); this method employs object verification (or re-identification) for gaining a confidence value to associate objects across multiple videos [31]. All of the techniques utilized for re-identification can also be utilized in single camera scenarios where the objective is determine if the same object appeared in the scene more than once [32, 33, 34].

As we will be discussing vehicle re-identification as our application of choice, we will first give a brief description followed by some of the unique characteristics that this application entails. The task of vehicle re-identification is to identify the same vehicle as it travels across a camera network. Vehicle search has become a much more necessary application as smart cities and smart
management systems have become more prevalent \cite{35}. Previous works \cite{36, 37} have shown the ability to create a unique identifier based on license place information; however, in many scenes that involve vehicles, the cameras are not placed in a manner in which the license plate is in view from all angles. Therefore, vision-based re-identification is necessary to create a unique identifier based on other aspects of the vehicle like appearance including viewpoint shifts/rotations, lighting variations, and different poses. Figure 2.17 illustrates some of these aforementioned challenges in a given dataset.

As mentioned before, vehicle re-identification brings about a few unique challenges when compared to a traditional person re-identification application. A few of these challenges are as follows:

1. In a system for vehicle re-identification, the labels are much more fine-grained than person or face labels. Given that there are a finite number of colors and types of vehicles, the diversity in the datasets is much less than other re-identification problems. This causes problems when defining the difference between objects or vehicles.

2. Multiple views of the same vehicle must be semantically correlated and, therefore, the identity of the given vehicle should be correctly decided aside from the viewpoint. Any approach to vehicle re-identification should be viewpoint agnostic. Some previous approaches \cite{38, 39} utilize
all of this information separately to make decision about identity.

3. In any real-world environment, a re-identification system should be able to extract subtle physical cues (or differences) in objects (in particular, things like dust, dents, etc. when it comes to vehicles) to help distinguish between vehicles which have the same characteristics otherwise (color, type, etc.). However, due to location of these anomalies, it could be difficult to see them given the viewpoint from the camera. For this reason, in practice, there is also a spatio-temporal matching piece that is employed to introduce a new parameter to distinguish objects that are similar [40, 39].

In short, we first need to obtain an embedding for each of the objects. The embedding is then used to perform a distance metrics which expresses the closeness of the objects in embedding space. Any good embedding should be invariant to illumination, scale, and viewpoint changes, just to name a few. Prior to advances in deep learning, most embeddings were handcrafted using a mixture of multiple types of feature extractors or learning a ranking system for the objects of similar identities [41, 42, 43, 44, 45, 46, 47].

Figure 2.18 illustrates the overall pipeline for a vehicle re-identification problem. First we have a signature extractor or creator which creates the embedding discussed above. This embedding is then compared to the other objects which the system knows about (a.k.a. that gallery). The gallery is made up of all other images in the systems world. The probe image is then compared to each one to decide which other vehicle it is closest (hopefully the same vehicle identity). The order of the gallery images in Figure 2.18 shows an optimal ranking of the gallery images after a correctly trained feature extractor has been created. Notice that the gallery images with the smallest distance from the probe image are the same vehicle and even close to the same pose while the larger distance ones are of other vehicles.

2.2.3.1 Datasets

VeRi is proposed in [9] is one of the first datasets created solely for the task of vehicle re-identification. This dataset encompasses 40,000 bounding box annotations of 619 vehicle (identities) across 20 cameras in traffic surveillance scenes. Each vehicle is captured in 2-18 cameras in various viewpoints and varying illuminations. Notably, the viewpoints are not restricted to only front/rear but also side views, thereby making it one of the more challenging datasets. The annotations per vehicle include
Figure 2.18: Re-Identification Process for Creating Embeddings for Objects

Figure 2.19: Vehicle Re-Identification Datasets

make and model of vehicles, color, and inter-camera relations and trajectory information. A few example images from the VeRi dataset can be seen in Figure 2.19a.

**vID** is proposed in [39] comprises 221,763 bounding box annotations with a much larger group of identities (26,267) and are captured across various surveillance cameras in a city. Annotations include 250 vehicle models as well as having an order of magnitude more identities than the VeRi dataset. However, the viewpoints only include front and rear views for the vehicles. A few example images from the VeRi dataset can be seen in Figure 2.19b.

**PKU-VD** is proposed in [48] and is the newest and largest of the vehicle re-identification datasets. This dataset comprises about two million images and their fine grained labels including vehicle model and color. This dataset is split into two sub-datasets, namely VD1 and VD2 based on cities from which they are captured. The images in VD1 are captured from high resolution cameras while images from VD2 are obtained from surveillance cameras. There are about 71k and 36k identities
in VD1 and VD2, respectively. A few example images from the VeRi dataset can be seen in Figure 2.19c.

2.2.3.2 Architecture and Loss Function

As mentioned previously, most retrieval or re-identification networks are based on traditional image classification network designs. For example, many of these networks have a backbone of a typical network like VGG [26], ResNet [28], MobileNet [49], or one of the other normal image classification networks. The main difference between typical classification networks and a network used for re-identification is the last few layers of the network. Firstly, one of the main challenges with re-identification problems is the large number of classes that need to be identified. In most image recognition tasks there are hundreds or even thousands of objects (in the case of ImageNet) which would need to be identified. However, in the task of re-identification, the objects can easily scale to tens of thousands or even millions of identities. For this reason, a traditional softmax classification layer will not be able to handle this type of data. Furthermore, in the task of retrieval or re-identification we are not necessarily trying to find an exact class, but rather compare the feature that we have extracted to another known object.

In a typical architecture for re-identification, the object is presented to a feature extractor followed by an identification portion which consists of a fully connected layer and an optional normalization layer. Instead of them passing this result to a fully-connected layer to identify the object, the signature is taken from this portion of the network to perform distance comparisons in embedding space. But the question is, without a softmax classification layer, how is the network trained to be able to recognize differences in the objects of different identities? The answer that we have come up with for this question is to utilize manifold learning with triplet loss as well as different sampling techniques to optimize the loss value while training.

Consider a dataset \( X = \{(x_i, y_i)\}_{i=1}^N \) of \( N \) training images \( x_i \in \mathbb{R}^D \) and their corresponding class (identity) labels \( y_i \in \{1 \cdots C\} \). Re-identification approaches aim to learn an embedding \( f(x; \theta) : \mathbb{R}^D \rightarrow \mathbb{R}^F \) to map images in \( \mathbb{R}^D \) onto a feature (embedding) space in \( \mathbb{R}^F \) such that objects of similar identity are metrically close in this feature space.

Let \( D(x_i, x_j) : \mathbb{R}^F \times \mathbb{R}^F \rightarrow \mathbb{R} \) be a metric measuring distance of images \( x_i \) and \( x_j \) in embedding space. For simplicity we drop the input labels and denote \( D(x_i, x_j) \) as \( D_{ij} \). A value of \( y_{ij} = 1 \) is created if both samples \( i \) and \( j \) belong to the same class while \( y_{ij} = 0 \) indicates samples
The triplet loss utilized for this problem was inspired by the seminal work on metric learning for nearest neighbor classification [50] and [51] which proposed a modification for facial retrieval tasks. The idea behind triplet loss is to force all data points that have the same identity (class) to be closer to each other in embedding space than those of different identities. Triplet loss considers not only the object in question (also called the anchor), but also a positive and negative sample (samples from the same identity and different identities, respectively). Equation 2.11 expresses the equation in its full form including distance measures and a summation to achieve the overall triplet loss for the whole set of triplets.

\[
l_{\text{triplet}}(a, p, n) = \sum_{n=1}^{N} \left[ \| f(x^n_a) - f(x^n_p) \|^2_2 - \| f(x^n_a) - f(x^n_n) \|^2_2 + \alpha \right]
\] (2.11)

To simplify the equation, we will use \( D_{ij} \) for the distance measure, where \( ij \) is the pair of images (either anchor and positive or anchor and negative), shown in Equation 2.12.

\[
l_{\text{triplet}}(a, p, n) = [D_{ap} - D_{an} + \alpha]_+
\] (2.12)

Figure 2.20 illustrates the mechanism for which we will be using to modify the network based on triplet loss. The left portion of the figure illustrates how a negative sample could be closer to the anchor before training (in other words, the network would classify these two as the same identity, even though they are different). The right portion illustrates the end goal of the process; eventually all samples of the same identity will be closer in embedding space than those from other identities.
2.2.4 Hardware

We will briefly discuss in this section the hardware that is used for training and inference in the field of deep learning. As deep learning has been growing, so has the market for graphical processing units (GPU). GPUs are processors that contain many cores and can perform large amounts of computation in parallel, lending them extremely well to training and inference of CNN (and ANNs for that matter) networks.

2.2.4.1 Training

As discussed earlier, there is a need throughout the training process to pass images through the network (forward-pass) as well as calculate for each neuron in every layer the corresponding weight update to optimize the loss function. For most deep learning workloads, clusters of machines utilizing GPUs are used to train CNNs. Theoretically, the more GPUs provided for a framework to use, the faster the training will occur (this however has some limitation, which we won’t get into in this document). For the purposes of this research, Clemson’s Palmetto Cluster [52] is used as the heterogeneous computational platform. The cluster contains many nodes with GPUs (including NVIDIA’s K20, K40, P100, and V100 architectures). By utilizing multiple nodes for training at a time, the training times can be cut down to hours instead of days depending on the network architecture and the training set.

2.2.4.2 Inference

Inference is a completely different problem as mostly it is thought of in the realm of deployment. For most deployment applications, the idea is to utilize some sort of embedded or mobile device or platform to execute the network rather than a desktop or server GPU. For the purposes of inference in most cases in this dissertation, the NVIDIA Jetson TX2 and NVIDIA Jetson AGX Xavier will be used as an inference platform. Most models that will be discussed have been optimized for the Jetson platform. One application will also be deployed on a mobile phone as a proof-of-concept, however, as we will see in the results section, the network does run better on an edge device like a Jetson rather than a mobile phone.
2.3 Summary

Computer vision is a large and growing field, even the small portion of it which is deep learning or machine learning based. Through the advent of the CNN, deep learning has been able to breakthrough as a heavy contender for many computer vision applications. Architectural developments have occurred over that past 5 or so years which have allowed developers to utilize deep learning techniques for certain tasks like image classification, object detection, and even more in-depth tasks like re-identification. As applications have difference specifications, different model types have been developed to solve domain specific problems; for example, R-CNN is a very accurate modeling technique for object detection, however not very speed efficient, where SSD and YOLO are both very efficient when it comes to speed but lack slightly when compared in accuracy to R-CNN models. The remainder of this dissertation aims to utilize and eventually build on a number of these architectures for a few industry sponsored projects.
Chapter 3

Research Design and Experiments

In this chapter, we will discuss our research design as well as methods for delivering this research design. To do this we must first discuss each computer vision-based project’s aspects and limitations. Our research design contains 4 separate application domains, each with its own requirements. Each section will discuss the topic (along with any relevant work that has been done), layout a system overview for the design and develop design choices for deep learning techniques that have been utilized. Results corresponding to these experiments for each application domain can be found in Chapter 4.

3.1 Perception for Autonomous Driving

The first application that we will discuss is autonomous driving (AD). Many efforts over the past decade have been sponsored with an end goal of creating a consumer-level autonomous vehicle. Many systems now, in factories, warehouses, and many other environments that claim to be autonomous are actually considered semi-autonomous. There is no actual "controller" or "engineer" that has control of the system at any given point, however, this system is not able to make its own decision based on perception or any other sensory input that it may receive. In the case of autonomous driving, it is much more readily apparent that there is a need for the system to make split second decisions, some of which it has not seen explicitly when being developed (i.e. the developer is not able to algorithmically define every scenario an autonomous vehicle could face when placed on the road). For this reason, systems that can help perform reasoning built into the vehicle
are a necessity.

There are three paradigms for vision-based autonomous driving: mediated perception, behavior reflex, and direct perception. The first paradigm, mediated perception parses each scene into a structured data element and then tools like decision trees and other decision systems (some neural network or deep learning-based) are used to create the system. Huval et al. [53] modified the Overfeat [54] architecture to predict a mask first and then use regression to calculate the final location of the object. Provodin et al. [55] show that a pre-trained CNN on ImageNet has the ability to extract good enough features for autonomous driving while dealing with the problem of missing data. Zhang et al. [56] use a simple Markov random field model to perform instance-level segmentation; this allows not only for the system to understand the types of obstacles but also enumerate and separate them.

The second paradigm of autonomous driving is designated behavioral reflex (also known as end-to-end learning). This paradigm of techniques simply takes a set of input sensor values (most of the time an image) and directly produces driving controls (including steering, acceleration, braking, etc.). Early in 1988, Pomerleau [57] produced one of the first neural network approaches for autonomous driving which was trained on simulated data and was able to achieve 90% direction prediction accuracy for following the road. Following this, many works including Muller et al. [58], Hadsell et al. [59], and Koutnik et al. [60] have chosen this paradigm for learning output control for an autonomous system. In 2016, Bojarski et al. [61] at NVIDIA developed a CNN that was trained end-to-end resulting in a 98% autonomy rate. The authors showed that by simply using steering angles and images from upwards of 100 hours of human driving, the CNN has gained the ability to learn useful features such as the road boundaries and obstacles. This project later morphed into what is now called BB8 [62].

The third and most prominent paradigm for this dissertation is direct perception. This paradigm uses input sensors like cameras (producing images), radar/lidar (producing depth information) and GPS as a vision system (perception system) for an autonomous system. Compared to the behavioral reflex paradigm, the outputs from the "perception" module are fed directly into a different subsystem which performs planning and decision making rather than it being one entire system. There are benefits and drawbacks to both of these systems. As the behavioral reflex systems tend to be smaller (containing less modules) they tend to be much quicker; however, the direct perception paradigm systems have the advantage of being able to craft a more representative
Figure 3.1: Autonomous Driving Architecture

depiction of the environment. Direct perception also allows engineers and users to view exactly what
the system is seeing and why it may be acting in a certain manner.

In previous (and future) collaboration with Clemson University International Center for
Automotive Research (CUICAR), we have developed a system that can help aide a decision-making
system placed into an autonomous vehicle based on the direct perception paradigm described above.
As they develop the physical automobile itself, we have been tasked to develop a system which can
retrieve input from sensors (in this case, cameras) and provide as much useful information about the
environment as possible. By doing this, we are allowing the planning and drive-time systems to act
in a more precise manner avoiding obstacles and maneuvering successfully towards a destination. In
the following sections we will discuss the entire system design proposed for the autonomous vehicle
as well as the perception module itself. We will then delve deeper into the network design module
and discuss design choices for deep learning network creation and training.

3.1.1 System Design

As discussed above, autonomous driving systems can be design as very complex systems.
These diverse architectures contain different components, all with the same goal: retrieve some sort
of information from the environment and provide control to the vehicle. To have the ability to
understand each piece of the system that we design, we have chosen a "direct perception" model
which allows us to see the output of each portion of the pipeline before it is sent to the control
center. Figure 3.1 illustrates a high-level overview of the pipeline that we have constructed.
On the far left, we can see the array of sensor that could be present in an autonomous vehicle. For purposes of this design, we will be focusing solely on camera based inputs (mainly standard RGB cameras as well as stereo vision cameras). Second from the left is the "perception module" which will be the focus of this section. The last step before sending the information to the vehicle mechanics for vehicle control is planning. This step consists of both motion planning (aspects like speed control and direction control) and route planning (which takes a look ahead and plans the best path to take to get to a certain objective). However, for the purposes of this dissertation we will be focusing mainly on the perception module itself and giving the modules down the pipeline as much information as possible to move the vehicle effectively.

There are many design decisions that go into designing something as complex as a module for an autonomous vehicle. For instance, not only do you want the system to take in as much information as possible and be able to perform with a high degree of accuracy, but there is an inherent "real-time" nature to any autonomous system task. Without each module in the system working in real-time, there is no way the control system will be able to avoid obstacles or even stay on the road for that matter. All of these considerations must come into play when designing a system that must be as robust as an autonomous vehicle. Firstly, we must know the system that we are targeting for our design. In this case, we have utilized NVIDIA’s Drive PX2 platform, a platform enabling autonomous driving developers to build and deploy self-driving vehicles that are functionally safe and can also be certified by international safety standards. Figure 3.2 illustrates the Drive PX2 SOC architecture while Table 3.1 gives important specification information about the SOC device.

A few specifications to notice are the 12 simultaneous camera inputs and the dual GPUs (accompanied by numerous CPUs with the ability to produce 20 FP16 TFLOPS. Before discussing the design of the perception system, it must be noted that the entire autonomous driving system will be placed on this SOC. For that reason, the perception module will be allotted the use of one single Parker GPU and accompanying CPUs while the planning and control modules will be allotted to the other. This places a design constraint on our system in that we are only allowed to use 256 CUDA cores for the computation; therefore, our models must be embedded systems friendly and be as small as possible.

**Perception System Design** As described above, the perception system will be focused on extracting as much information as possible from the environment to pass onto the control and motion...
Figure 3.2: NVIDIA Drive PX2 Architecture

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Architecture</td>
<td>Pascal (16nm)</td>
</tr>
<tr>
<td>CPU</td>
<td>2x Tegra X2 (Parker) + 2x Pascal GPU</td>
</tr>
<tr>
<td>GPU</td>
<td>4x Denver + 8x Cortex A57</td>
</tr>
<tr>
<td>Accelerator</td>
<td>2x Parker GPU (each 256 CUDA cores)</td>
</tr>
<tr>
<td>Onboard Memory</td>
<td>16GB LPDDR4</td>
</tr>
<tr>
<td>Storage</td>
<td>128GB eMMC</td>
</tr>
<tr>
<td>Performance</td>
<td>20 FP16 TFLOPS</td>
</tr>
<tr>
<td>Power Rating</td>
<td>250W</td>
</tr>
<tr>
<td>IO</td>
<td>12 GMSL Camera + HDMI + GMSL Display + Vehicle Harness</td>
</tr>
</tbody>
</table>

Table 3.1: NVIDIA Drive PX2 Specifications
planning systems. For this experiment, we will use as input standard RGB cameras as well as stereo vision cameras (from which we can estimate depth information). Figure 3.3 illustrates the flow of the data; sensors to image processing to DNN to detections and depth estimation. As specifications for the project described, some interesting information that should be given to the planning module include localization of objects of interest such as cars and pedestrians, depth information for those objects that are on the road, as well as classification of any type of road signs that should govern the way the vehicle performs planning and motion. For example, not only should the vehicle know when there is a vehicle in front of it, but it should also understand that it is approaching a stop light or a stop sign. This will give enough valuable information to the planning module for it to make an informed decision and maneuver its way around the test track.

In order to perform detection on objects that will be relevant, we must first choose the correct dataset. There is always the choice of creating a dataset from scratch that exactly matches the problem you are trying to solve; however, with the amount of data that deep learning networks need to see to learn anything useful, labeling these datasets is a very time consuming task. For this reason, we have chosen to use a few already designed datasets that fit our purposes. Firstly, we have chosen the KITTI dataset (described in 2.2.2.1) which contains objects relevant for autonomous driving such as vehicles and pedestrians. This dataset not only contains images with detection labels for each, but also stereo images with labels allowing us to test our system on not only traditional RGB images but also obtain depth information from a stereo camera as well. Secondly, we would like to recognize traffic signs including yield signs, stop signs, etc. that will be present on the test track. LISA is a public dataset (also mentioned in 2.2.2.1) that contains 47 different US traffic sign types ranging from 6x6 pixels to 168x168 pixels. This dataset will allow us to not only recognize a traffic sign but also recognize them at different distances away and infer that distance, which is also valuable information for the planning module. Figure 3.4 illustrates sample images taken from the
Figure 3.4: Example Images from the KITTI dataset

Figure 3.5: Example Images from the LISA dataset
KITTI dataset and Figure 3.5 illustrates a few sample images from the LISA traffic sign dataset. In the next section we will discuss how these datasets were used with specific architectures in an attempt to create a perception module to be used for autonomous driving.

3.1.2 Network Design

Two main design considerations were taken into account when designing the deep learning portions of this module. First, we would like our design to be able to run on a single Tegra/GPU pair on the NVIDIA Drive PX2 in real-time while the other is using the information to create a decision plan. Secondly, we want to be as accurate as possible when running inference with our model. Typically, these two parameters are polar opposites (meaning we can either create a network with a high accuracy but runs very slowly or we can create a a network that is very fast but lacks in the accuracy comparison). For comparison, we have chosen 2 architectures, Faster-RCNN [2] and NVIDIA’s DetectNet [10]. As described in Section 2.2.2.2, Faster-RCNN is a two-stage network which computes proposals and then runs classification and regression on these proposed boxes. DetectNet on the other hand is a single-shot approach similar to that of SSD and YOLO; however there are extra data augmentation layers before the feature extractor and then custom clustering layers after predictions have been made in an attempt to minimize overlapping boxes with the same object. The original DetectNet architecture is show in Figure 3.6 and the inference architecture (removing the loss terms) is shown in Figure 3.7.

Due to the necessity of having a real-time system, the DetectNet architecture was chosen rather than the Faster-RCNN architecture. Results, however, of the Faster-RCNN model can be found in Chapter 4 for breadth. To utilize both RGB and stereo images as input, we first constructed a slightly different architecture incorporating the depth image as input. By adding this new input,
we also needed to construct a new loss function which could incorporate the depth information. Figure 3.8 illustrates the addition of the new input as well as the addition loss term.

Traditionally, DetectNet uses a linear combination of two separate loss functions to produce its final loss value: \( \text{coverage}_{\text{loss}} \) and \( \text{bbox}_{\text{loss}} \). Coverage loss (\( \text{coverage}_{\text{loss}} \)) is the sum of squares differences between the true and predicted object coverage across all grid squares in an image and can be written as:

\[
\frac{1}{2N} \sum_{i=1}^{N} |\text{coverage}_i^{t} - \text{coverage}_i^{p}|^2
\]

Bounding box loss (\( \text{bbox}_{\text{loss}} \)) is the average L1 loss for the true bounding box coordinates relative to the predicted coordinates for the object covered in each grid square.

\[
\frac{1}{2N} \sum_{i=1}^{N} [(x_1^t - x_1^p) + (y_1^t - y_1^p) + (x_2^t - x_2^p) + (y_2^t - y_2^p)]
\]

In addition to these loss functions, we have added a depth loss (noted \( \text{L3 Loss} \) in Figure 3.8). This function will simply take the depth value for a specific grid cell and compare it with the depth value given by the stereo camera. A formulation of the \( \text{depth}_{\text{loss}} \) term can be described in the
following equation:
\[
\frac{1}{2N} \sum_{i=1}^{N} |depth_i^d - depth_i^p|^2
\] (3.3)

Each of these loss values is combined as a linear combination and utilized as the overall error for the training process. Each portion of the individual loss function could be weighted differently in terms of how each should contribute to the total loss value, however, this is left for future work. By weighting the terms differently, we can force the optimizer to learn different thresholds for acceptability (i.e., if one term can have an error of 0.1 like the detection accuracy while another can only have an error or 0.01 like the depth). In Chapter 4, we will show inference times for both the regular DetectNet architecture as well as the architecture with the depth added and a brief comparison of DetectNet and Faster-RCNN as further justification as to why we have chosen to use the DetectNet model for this particular case. We will also illustrate the inference capabilities of each of these networks. Tetreault [63] has shown through the application of semantic segmentation that fusing multimodal data (such as RGB and depth images) can lead to performance increases as the network is able to learn different representations of the data from multiple data sources.

## 3.2 Drone-based Object Detection

Drone-based detection and surveillance systems have become ubiquitous as autonomous systems have been on the rise. Many companies and individuals are utilizing drone technology to not only to surveil warehouse and other important areas for security but also for automation of tasks such as inventory, package location, package delivery, and warehouse management. For this reason, there are many competitions and initiatives to further develop technologies for drone-based activities. Before we discuss the challenge for which this work is a part of, first we will discuss a little bit of the what has been done previously in the field of drone detection-type systems.

Cario et al. [64] provide a thorough review of the applications for which deep learning has been applied to Unmanned Aerial Vehicles (UAV) including perception, planning, and motion control or the UAV. Zhou et al. [65] propose an efficient road detection and tracking framework for a UAV based on a graph-cut detection approach as well as homography-based tracking giving the ability to follow any detected road or landmark it finds. Radovic et al. [66] describe a procedure and parameter selection for training CNNs on a set of aerial images, specifically in the transporta-
<table>
<thead>
<tr>
<th>Model</th>
<th>Speed (ms)</th>
<th>Approx. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD Mobilenet</td>
<td>30</td>
<td>33</td>
</tr>
<tr>
<td>SSD Inception</td>
<td>42</td>
<td>23.8</td>
</tr>
<tr>
<td>Faster RCNN Inception</td>
<td>58</td>
<td>17.2</td>
</tr>
<tr>
<td>Faster RCNN ResNet-50</td>
<td>89</td>
<td>11.2</td>
</tr>
<tr>
<td>Faster RCNN ResNet-101</td>
<td>106</td>
<td>9.4</td>
</tr>
</tbody>
</table>

Table 3.2: Titan X Inference Times

...tion field utilizing a YOLO-based network design. Jung et al. [67] outline an approach for drone perception for autonomous drone racing that is able to estimate the center of each gate it is tasked to maneuver through. Lastly, Smolyanskiy et al. [68] from NVIDIA have proposed TrailNet, a DNN-based solution with the ability to autonomously maneuver through an outdoor environment while performing low-level obstacle detection and avoidance.  

**DAC HDC Challenge** The DAC HDC Challenge is a system design contest which tasks developers to implement neural network (or machine learning) based solutions to object detection for drones. Each team is provided a dataset containing 98 classes including (but not limited to) cars, boats, trucks, drones, and people. Each team must then develop a machine learning inference engine with the ability to take a new image with a similar object and localize it on the image. One major stipulation of this project, however, is that each design has to run on a Jetson TX2 in "real-time" (defined as 20 fps). Table 3.2 provides the baseline execution times for each network type we have discussed running on an NVIDIA GeForce GTX Titan X, a much more powerful GPU than the Tegra X2 found on the Jetson.

However, the large problem with most of these inference times is that, even on superior hardware like the Titan X, the FPS value does not meet the real-time requirement set by the DAC committee. This means that it will definitely not be up-to-standard when running on a smaller embedded device like the NVIDIA Jetson platform. For this reason, we have designed a new network with the ability to execute inference on the Jetson TX2 while also maintaining a comparable level of accuracy to other models previously mentioned.

### 3.2.1 System Design

As described before, the DAC requires that all participants utilize the Jetson TX2 development kit for all designs. Table 3.3 illustrates a few of the important specifications for the project.

Notice that some of the specs hint that this SOC is half of the Drive PX2; this is, in fact, the case be-
cause both SOCs are built with the same hardware, the Jetson simply has much less computational power as it is only one GPU and group of CPUs with much less on-board memory.

The pipeline for this project was provided to all participants; however, the main purpose of the challenge was to develop a machine learning or deep learning solution to plug into the pipeline for perception. The entire pipeline can be seen in Figure 3.9. In the next section we will discuss the design of the network as well as our design decisions that were able to give us a "real-time" performance metric.

However, before we discuss the network architecture, we will first illustrate the problem with a few samples from the dataset. Figure 3.10 shows a small sample of images taken directly from the training set provided by the DAC committee. The first thing to notice about the dataset is that the objects we are attempting to identify are quite small compared to the overall image size. Some of the objects depicted in these samples take up areas less than 20x20 pixels. This aspect of the dataset has lead to the selection of a specific network architecture. A second point to notice about

---

<table>
<thead>
<tr>
<th>GPU Architecture</th>
<th>Pascal (16nm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Tegra X2 (Parker) + 1x Pascal GPU</td>
</tr>
<tr>
<td>GPU</td>
<td>2x Denver + 4x Cortex A57</td>
</tr>
<tr>
<td>Accelerator</td>
<td>Parker GPU with 256 CUDA cores</td>
</tr>
<tr>
<td>Onboard Memory</td>
<td>8GB LPDDR4</td>
</tr>
<tr>
<td>Storage</td>
<td>32GB eMMC</td>
</tr>
</tbody>
</table>

Table 3.3: NVIDIA Jetson TX2 Specifications

---

Figure 3.9: DAC Pipeline
the dataset is that it is much different than the KITTI or LISA datasets previously described; there are many different classes with the same type of object. For example, within the dataset there are 98 classes (of which there are ∼15 different person classes and ∼25 different car classes). For this particular challenge, we are only tasked with localization of objects rather than detection (making the problem slightly more manageable); however to train a localization network on this data, the network must in turn learn a representation for each class (essentially training a detector). Taking these dataset characteristics and specifications for the challenge into consideration, we will discuss the network architecture design in the next section.

3.2.2 Network Design

As previously discussed and shown in Table 3.2, due to the hardware performance limitations of the Jetson TX2 compared to the GTX Titan X, we will modify an existing network architecture in an attempt to generate a faster inference time with similar accuracy. First, we have chosen to use an SSD-based architecture instead of an RCNN variant due to its more efficient speed of inference. Our first design for this competition utilizes a standard SSD architecture model (shown in Figure 2.16). The way the pipeline is set up, batches of images are passed to the inference module which returns a set of bounding boxes for the objects it locates in an image. As we are utilizing an SSD-based architecture, we are only able to process 3/5 of every batch of images and still maintain a close enough FPS value to be considered real-time. Processing 3/5 of the images, however, is not conducive to a good system as we are in essence missing 40% of all images (incurring an accuracy value of 0 for every image).

For this reason, we have chosen to modify the network slightly in order to give us a much more real-time solution while being able to process all images. We are using a variant of the SSD architecture denoted Embedded-SSD (eSSD) with a MobileNet [49] feature extractor. We change the size of the input to be 256x256, which means that each of our images will be resized before training or inference can occur. We also reduce the number of feature maps used in the final box prediction creating a smaller network able to infer much quicker. A depiction of the eSSD architecture is shown in Figure 3.11.

Notice in Figure 3.11 that we have removed one of the additional feature extraction layers. While this will make the network perform quicker inference, we have also lost an entire feature map for each grid square in the image compared with the SSD architecture. However, for our particular
dataset, most objects are relatively small; therefore, we can get away with having fewer feature maps as a large number of the objects found in each grid cell will be a close match to one of the new feature maps.

To gain as much performance benefit as possible, we have also chosen to overclock the Tegra X2 GPU (using a provided script with NVIDIA Jetpack [69] software) to maximize throughput during inference. In Chapter 4 we will provide some of the result detection images as well as differences in inference time for each of the models with and without overclocking. We will also explore the effect of batching the images for inference rather than feeding them one-by-one into the network.

3.3 Embedded/Mobile Diagnostics

Manufacturing plants have traditionally been known as elements of industry that stick to well known processing and robotics techniques. However, in recent years, many manufacturing companies have looked to computer vision (in particular, deep learning-based computer vision) to aide associates on the factory floors with mundane or repetitive tasks. For example, as parts move down an assembly line, deep learning systems can aide associates in pointing out small flaws in parts they will be assembling as well as incorrect parts that may have been delivered. Before going into detail about the applications we have developed, we will first visit a few pieces of work that have been completed or researched within the field of manufacturing using deep learning.

Wuest et al. [70] discusses a general overview of available machine learning techniques that are currently present in the field of manufacturing as well as laying out potential benefits and successful applications of these machine learning techniques. Wang et al. [71] presents a comprehensive
survey of commonly used deep learning algorithms and techniques as well as discussing how these techniques create a "smart" manufacturing process. They also delve into future trends and challenges associated with implementing deep learning solution in a manufacturing setting. In collaboration with BMW [72], we discuss the libraries, tools, and infrastructures needed to develop a deep learning pipeline for automotive manufacturing. We also evaluate the effectiveness of a trained deep learning classifier in a real-world manufacturing setting.

In this section we will be discussing two manufacturing based deep learning applications that were developed in collaboration with BMW. The first is a logistics project involving detection of barcode labels and the second involves trailer yard management inside of the manufacturing facility.

### 3.3.1 Deep Logistics

One of the most important portions of maintaining a smooth manufacturing process is keeping the assembly line fed with the correct materials at all times. This process starts with making sure the correct materials have been delivered to the plant and are then distributed to the correct locations at the right time. The first application that we will investigate is a deep learning solution for detecting and reading barcode labels on all boxes received at the plant. Traditionally, an associate (or multiple associates) will make their way throughout the storage warehouse scanning each barcode manually; it is clear that this a very time consuming and mundane task for a skilled associate. By supplementing the associates’ work with an AI system, we allow the associate to be more productive in the same amount of time. Consequently, the associate is able to perform other tasks with the time that was saved.

**System Design** When developing a system to aide an associate, one of the major considerations is ease of use. We have developed a deep learning system that can be deployed on not only embedded devices but also on mobile devices like an iPhone. It is desirable for this deep learning detection network to run in (semi)real-time which will allow the associate to simply point a camera at interesting boxes in the warehouse and automatically extract useful information.

First, we will discuss the entire pipeline (shown in Figure 3.13) with which an associate will use this application for label detection in a warehouse environment. The original dataset we are given consists of only about 30 images entirely of the barcode labels that we would like to detect. This itself is not conducive to training a detector as it not only needs to learn representations of the objects for which is is looking but also must learn the ability to locate these objects. To combat this
problem, we have decided to create a synthetic dataset for training. Creating a synthetic dataset will not only allow us to tackle the detection problem rather than classification, but will also allow us to control the types of images that the network sees during training.

We have decided to use the Sun Dataset [73] as a way to randomize the background on which the labels will be placed. The dataset is created by randomly selecting an image from the Sun dataset, randomly choosing a number between 1 and 10 (reflecting the number of barcode labels that will be placed on the image), and then systematically placing these labels on the image. Figure 3.12 illustrates a few of the images that were created using this technique, that were in turn utilized for training. By utilizing the Sun dataset with random backgrounds we are able to ensure that the trained network learns to locate barcodes on numerous different backgrounds and orientations. We are employing the concept of transfer learning as we train on synthetically generated data and test on real-images with barcode labels.

Following dataset creation, we then use the Darknet framework (YOLO-based [4] network, described in the next section) to train using the synthetic dataset. Once we have a trained network, we then convert the weights (parameters of the network) to a format easily read by a library called Forge [74]. Developed by Matthijs Hollemans, Forge is a collection of helper code that aides in the construction deep neural networks for use with Apple’s MPSCNN framework. Once we have a converted model, we can implement a custom network by using the Forge framework code that will execute on an iPhone device. In the next section we will discuss the DNN design that was chosen for implementation.

Network Design As discussed in Chapter 2, YOLO models have come to fruition as one of the fastest detector models available. For this reason, we have chosen a YOLO-based model to train and evaluate on our target device, the iPhone 7 Plus. Although we are targeting the iPhone, we will provide results for other hardware for inference using the trained YOLO model. After careful consideration and the stipulation on the project to make the model run on the mobile device as close to "real-time" as possible, we have decided to go with a Tiny-YOLO model. Figure 3.14 illustrates the layers that are present in a traditional YOLO architecture while Figure 3.15 shows how the model has been shrunk to create a much more embedded-device friendly network.

Further discussion about execution time and accuracy values for inference as well results illustrating the detected labels before extraction can be found in Chapter 4.
Figure 3.12: Barcode Label Synthetic Dataset Samples
3.3.2 Deep Yard Management

As manufacturing material and vehicle parts delivery is another contributing factor to the success of an automotive manufacturer, it is important to keep track of all delivery vehicles and their locations in a trailer yard. Typically, it is the task of an associate to locate each truck and provide its parking location. Similar to the label detection task described above, an AI solution can greatly improve the efficiency of this task. A camera can be mounted to a vehicle such as a golf cart or even a drone which can be controlled through the trailer yard. By utilizing a deep learning model for detection, associates will no longer need to manually locate each trailer ID and associate it with its given parking lot number.

System Design As described for the barcode label detection problem, we have developed a pipeline that can be used for not only training a detector to locate trailers and their IDs but also their location in the trailer yard (i.e. parking lot number). A depiction of the entire pipeline is described in Figure 3.16. We will first train a typical detector to learn 2 classes (trailer IDs and parking lot numbers). Figure 3.17 depicts a representative sample of the images that are used during the training process for locating trailer ID and parking lot number. For purposes of the experiments and the need for "real-time" execution, we have chosen an SSD model similar to that of the DAC challenge discussed in Section 3.2.2. We will show results for multiple models in Chapter 4. Once we have detected
<table>
<thead>
<tr>
<th>layer</th>
<th>filters</th>
<th>size</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 conv</td>
<td>32</td>
<td>3 x 3 / 1</td>
<td>416 x 416 x 3</td>
<td>416 x 416 x 32</td>
</tr>
<tr>
<td>1 max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>416 x 416 x 32</td>
<td>208 x 208 x 32</td>
</tr>
<tr>
<td>2 conv</td>
<td>64</td>
<td>3 x 3 / 1</td>
<td>208 x 208 x 32</td>
<td>208 x 208 x 64</td>
</tr>
<tr>
<td>3 max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>208 x 208 x 64</td>
<td>104 x 104 x 64</td>
</tr>
<tr>
<td>4 conv</td>
<td>128</td>
<td>3 x 3 / 1</td>
<td>104 x 104 x 64</td>
<td>104 x 104 x 128</td>
</tr>
<tr>
<td>5 conv</td>
<td>64</td>
<td>1 x 1 / 1</td>
<td>104 x 104 x 128</td>
<td>104 x 104 x 64</td>
</tr>
<tr>
<td>6 conv</td>
<td>128</td>
<td>3 x 3 / 1</td>
<td>104 x 104 x 64</td>
<td>104 x 104 x 128</td>
</tr>
<tr>
<td>7 max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>104 x 104 x 128</td>
<td>52 x 52 x 128</td>
</tr>
<tr>
<td>8 conv</td>
<td>256</td>
<td>3 x 3 / 1</td>
<td>52 x 52 x 128</td>
<td>52 x 52 x 256</td>
</tr>
<tr>
<td>9 conv</td>
<td>128</td>
<td>1 x 1 / 1</td>
<td>52 x 52 x 256</td>
<td>52 x 52 x 128</td>
</tr>
<tr>
<td>10 conv</td>
<td>256</td>
<td>3 x 3 / 1</td>
<td>52 x 52 x 128</td>
<td>52 x 52 x 256</td>
</tr>
<tr>
<td>11 max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>52 x 52 x 256</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>12 conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>26 x 26 x 256</td>
<td>26 x 26 x 512</td>
</tr>
<tr>
<td>13 conv</td>
<td>256</td>
<td>1 x 1 / 1</td>
<td>26 x 26 x 512</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>14 conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>26 x 26 x 256</td>
<td>26 x 26 x 512</td>
</tr>
<tr>
<td>15 conv</td>
<td>256</td>
<td>1 x 1 / 1</td>
<td>26 x 26 x 512</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>16 conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>26 x 26 x 256</td>
<td>26 x 26 x 512</td>
</tr>
<tr>
<td>17 max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>26 x 26 x 512</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>18 conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>19 conv</td>
<td>512</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>20 conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>21 conv</td>
<td>512</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>22 conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>23 conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>24 conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>25 route</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>26 conv</td>
<td>64</td>
<td>1 x 1 / 1</td>
<td>26 x 26 x 512</td>
<td>26 x 26 x 64</td>
</tr>
<tr>
<td>27 reorg</td>
<td></td>
<td>/ 2</td>
<td>26 x 26 x 64</td>
<td>13 x 13 x 256</td>
</tr>
<tr>
<td>28 route</td>
<td>27</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29 conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 1280</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>30 conv</td>
<td>425</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 425</td>
</tr>
</tbody>
</table>

Figure 3.14: YOLO Architecture
<table>
<thead>
<tr>
<th>layer</th>
<th>filters</th>
<th>size</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>conv</td>
<td>16 x 3 x 3 / 1</td>
<td>416 x 416 x 3</td>
<td>416 x 416 x 16</td>
</tr>
<tr>
<td>1</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>416 x 416 x 16</td>
<td>208 x 208 x 16</td>
</tr>
<tr>
<td>2</td>
<td>conv</td>
<td>32 x 3 x 3 / 1</td>
<td>208 x 208 x 16</td>
<td>208 x 208 x 32</td>
</tr>
<tr>
<td>3</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>208 x 208 x 32</td>
<td>104 x 104 x 32</td>
</tr>
<tr>
<td>4</td>
<td>conv</td>
<td>64 x 3 x 3 / 1</td>
<td>104 x 104 x 32</td>
<td>104 x 104 x 64</td>
</tr>
<tr>
<td>5</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>104 x 104 x 64</td>
<td>52 x 52 x 64</td>
</tr>
<tr>
<td>6</td>
<td>conv</td>
<td>128 x 3 x 3 / 1</td>
<td>52 x 52 x 64</td>
<td>52 x 52 x 128</td>
</tr>
<tr>
<td>7</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>52 x 52 x 128</td>
<td>26 x 26 x 128</td>
</tr>
<tr>
<td>8</td>
<td>conv</td>
<td>256 x 3 x 3 / 1</td>
<td>26 x 26 x 128</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>9</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>26 x 26 x 256</td>
<td>13 x 13 x 256</td>
</tr>
<tr>
<td>10</td>
<td>conv</td>
<td>512 x 3 x 3 / 1</td>
<td>13 x 13 x 256</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>11</td>
<td>max</td>
<td>2 x 2 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>12</td>
<td>conv</td>
<td>1024 x 3 x 3 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>13</td>
<td>conv</td>
<td>512 x 3 x 3 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>14</td>
<td>conv</td>
<td>425 x 1 x 1 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 425</td>
</tr>
</tbody>
</table>

Figure 3.15: Tiny-YOLO Architecture

Figure 3.16: Trailer ID/Parking Lot Detection Pipeline
Figure 3.17: Trailer ID/Parking ID Dataset Samples
trailer IDs and parking lot numbers, we will then use a system that can perform optical character recognition (OCR) with each detection to determine the correlated trailer ID and parking lot number. As this is an ongoing project at the time of this writing, we will show results in Chapter 4 for the SSD model with the ability to localize the trailer IDs and parking numbers, but will not show at this point the extraction of the text within each detected bounding box.

3.4 Vehicle Re-Identification

The final application that we will discuss is the problem of vehicle re-identification. As mentioned in previous sections, there has been countless works that attempt to create a re-identification system specifically for person or face re-identification. However, the problem of vehicle re-identification has not been studied quite as extensively for a number of reasons; some of which include lack of availability of labeled data and a necessity to be viewpoint agnostic. In both person and face re-identification, the front and back of a person can be discerned to be the same based on features like the color and type of clothing worn by the person as well as the hair color (and of course other factors). The same goes for the left and right sides of a person or a face; in most cases the left and right sides of a person are mirror images of each other, or close enough that they can be thought of as such. In vehicle re-identification, the problem is that the front and back of cars looks completely different, therefore making the problem of distinguishing the same car from different viewpoints much more difficult.

One of the first approaches to learning visual cues and relationships between objects using Siamese CNNs was done by [75]. A Siamese CNN is constructed and trained in a manner by which an embedding space is created where similar objects or examples have similar embeddings and different objects have much different embeddings. Face verification was mentioned before as one of the potential use cases for manifold learning and re-identification; as such, [76] illustrated the use of contrastive loss with a Siamese network architecture to learn embeddings for face verification. Along the same lines, a more recent work, [51], utilizes triplet loss instead of contrastive loss to learn face embeddings. The major difference between the two approaches is the number of samples with which the loss value is computed; in the case of triplet, there are three samples simultaneously utilized to compute the loss value while with constrastive loss, there are only two. While contrastive loss may be more computationally efficient in terms of training the network (i.e. fewer samples for each
"batch", many works [77, 78, 31, 79, 80, 81] have shown that utilizing triplets actually substantially increases the accuracy of the network as it is able to learn a better embedding for each of the objects.

A second method that could potentially be used for obtaining an embedding for an object is by utilizing a traditional softmax layer as shown in [82, 38]. In these methods, a fully-connected (signature/embedding) layer is added prior to the softmax layer and each identity is considered to be its own category; i.e. the number of categories or classes is equivalent to the number of identities in the dataset. Once the network is constructed in this manner, it is trained using traditional cross-entropy loss. The softmax layer is then removed to reveal a last layer which will produce an embedding for the given object. However, since the network was trained for classification and not for creating an embedding, the performance usually suffers in comparison to methods that include some sort of embedding loss (siamese, triplet, etc.) when training. As the problem in question is not only to identify objects that are in the dataset and discriminate against them but also to provide embedding for unseen objects as well, this method is not suitable for our purposes. Also, as datasets continue to grow and the number of identities grows exponentially, using cross-entropy loss will become impractical. As a result of these limitations, there are a few works that try to unify the cross-entropy loss and some sort of metric loss terms [34, 83, 84].

Many other approaches that contain much more elaborate methods for re-identification have also been studied. [85, 86, 87, 88, 89, 90] provide a solution for fine-grained vehicle classification; this is a close-knit problem with re-identification, however re-identification is finer-grained problem as it must be able to not only extract differences in different make and model but also differences in the same make and model (i.e. dents, scratches, etc.) that could tell them apart from one another. [9] describes a method of fusing hand-crafted features with information about color, type, texture, etc. with features extracted from a CNN. [78] introduced a method that utilizes a classification loss as well as ranking terms for fine and course-grained object descriptions. In [79], triplet loss is used for embedding creation; however, the authors include clustering methods along with the deep learning techniques instead of using solely DL methods. Aside from this achieving state-of-the-art results as well as good variations between identities, the complexity of the computation is much more than we are looking for as an end solution. [91] uses a bi-directional LSTM to create synthetic embeddings to estimate unknown views and is trained with a combination of generative adversarial, siamese, and reconstruction loss. Again, this method leads to a very complex solution. [92] uses trajectory information and a combination of LSTM and Siamese CNN networks to create a re-identification
representation of an object.

Each of the methods described above either involve handcrafting some sort of feature set, utilizing very large complicated structured networks, or using time-series data to perform re-identification. We argue however, that with simple triplet loss in combination with techniques for optimally selecting candidates for each batch that a regular CNN architecture can match or out-perform other methods. This work was part of internship material created at NVIDIA and due to NDA and patent protection, most of the information about network architecture and methods are not able to be shared at the time of this writing. The next section will discuss in very brief detail the portions of the project that can be discussed so as not to infringe on an NDA or patent.

3.4.1 System & Network Design

The application with which the aforementioned triplet loss techniques have been deployed are in a smart garage. The idea behind the whole project is to create a system in which there are multiple cameras (in this case hundreds) in a garage or smart city scenario with a background CNN-based system with the ability to "identify" vehicles within the view of any camera from any viewpoint. This is where vehicle re-identification comes into play. Each camera in the system is fed through a detector (frame-by-frame) producing bounding boxes where vehicles (or other objects) are located. These bounding boxes are then used as input to an (optional) tracker which is able to perform some in-camera tracking. One limitation of a tracker across a multiple camera system is that the cameras must contain an area of overlap for the tracking to work properly; this is the problem that re-identification alleviates.

The research in this portion of the dissertation focuses mainly on the re-identification portion and assumes that the bounding box that is used as input has been determined from a detector and (possibly) from a tracker as well. The architecture includes a feature extractor (can be ResNet, MobileNet, etc.) followed by a fully connected layer (called the embedding layer) and an optional normalization layer. The feature extractor can be trained using a classification cross-entropy loss with a small subset of the data as a pre-trained model. Then the softmax layer can be removed and the embedding and normalization layer can be added and trained (with transfer learning) using metric learning (triplet loss).
3.4.2 Batch Selection

As mentioned previously, it is becoming more and more obvious that selection of a batch when utilizing metric learning approaches like triple and contrastive loss is one of the most important aspects of the training process. Typical loss functions used for re-identification (triplet, siamese, etc.) are heavily computationally prohibitive for any practical sized dataset as they rely on exponentially more data than the dataset contains (i.e. $O^3$ data for triplet loss as we need anchor, positive, and negative samples). For this reason, to train effectively in a reasonable amount of time, effective batching strategies must be created. If typical batch creation (i.e. random batches) is utilized, there are many batches where trivial data points are utilized or there are not triplets in a batch causing lack of convergence in the optimization algorithm. There are a few techniques that have been proposed, specifically for face re-identification batch creation that are utilized in this research in the context of vehicle re-identification. Figure 3.18 illustrates a few of these techniques which will be explained below and a more informative blog post about negative mining by Olivier Moindrot [93] describes the pitfalls of each.

Probably the most popular sampling approach to locating the "best" samples for use with metric learning batch selection is a method called hard data mining. This technique was created for tasks in computer vision such as object detection where the harder the object was to detect the
more times the sample was utilized in the training process. Hard negative mining involves choosing the hardest samples from the validation set during training which theoretically helps the model learn more effectively and converge to a solution quicker. However, most real-world datasets always have some set of outliers and the hard negative approach tends to pick these as suitable samples for training, which would cause the model to classify normal samples poorly.

As discovered by many who have tried, hard negative mining has many pitfalls. For this reason, [51] proposed a technique called semihard sampling, which makes an attempt at mining moderately hard triplets that are neither hard nor easy. This will help produce meaningful gradients when performing optimization. The downfall of semihard sampling is that the samples must be generated offline causing a very large upfront overhead as well as utilizing only the CPU for computation. [80] introduced a method for constructing these samples on a GPU by randomly sampling $P$ identities from the training set $X$ and then randomly selecting $K$ images for each of the identities, resulting in $PK$ images per batch. Along with this contribution, the authors also contributed two new batching techniques; namely, batch hard and batch all [94]. The authors open-source implementations of these batching techniques also provides a technique called batch sample [94] which are all shown to greatly improve the state-of-the-art in person re-identification.

### 3.5 Summary

In this chapter, we have discussed our research design for each of the four applications we have developed. First we discussed autonomous driving and the perception system that was developed utilizing deep learning for detection of automobiles, pedestrians, etc. Next, we discussed a drone-based detection system design challenge, where we designed a smaller model with the ability to run in real-time on an edge-computing (EC) device like a Jetson TX2 or Jetson AGX Xavier. Following, we discussed a deep learning approach to manufacturing, mentioning both deep learning for logistics in the form of barcode detection and recognition as well as trailer yard management. Finally, we explored the techniques utilized for re-identification and provided an understanding of how the current techniques for person/face re-identification are utilized in this work for the application to vehicle re-identification. We were not able to delve deeper into our methods due to NDA and patent issues, but open source options have been discussed. Results and other methods will be published shortly in [95]. In Chapter 4 we will discuss our results for each of these applications.
and discuss trade-offs for the architectures that were chosen for development.
Chapter 4

Results

In this chapter, we present results for each of the applications discussed in Chapter 3. The focus of the results is on the accuracy that we are able to achieve with each network as well as its ability to execute inference on an edge computing (EC) device in real-time, whether that be an embedded device like the Jetson or a mobile phone like the iPhone 7 Plus. We will also provide example output from each of the networks as a way of visualizing the correctness of the model.

When discussing accuracy, there are many different numerical quantities that can be used to measure performance; some of these include Mean Average Precision (mAP), Intersection over Union (IoU), and Recall for a detection problem and Ranking for a re-identification problem. For purposes of the experiments in this dissertation, we will be focusing on mAP and IoU for detection problems and ranking for re-identification. mAP is considered to be the average of all samples of ratio of detections which are predicted correctly (true positives) compared to all positive detections (a summation of true positives and false positives). A pictorial representation of IoU can be seen in Figure 4.1. Notice that the larger overlap of the predicted bounding box and the ground truth bounding box, the larger the IoU.

Ranking can be thought of in terms of matching the same object in distance space. For instance, objects that are of the same identity (class) should be closer together in embedding space than those of different objects. Therefore, the ranking for these objects of the same class should be lower (i.e. Rank-1 is the closest, Rank-2 is second closest, etc.). Figure 4.2 illustrates ranking in terms of vehicle identification. Given an image from the testing set (i.e. a probe image) find the distance between it and each of the rest of the testing set (i.e. the gallery of images). The closer
the distance is to the probe image, the lower the ranking; this signifies that these object may be of the same identity or are very closely related in embedding space (possibly the same make and model vehicle).

4.1 Perception for Autonomous Driving

In this section, we will evaluate and discuss the pipeline that has been developed for CUICAR as well as some of the results from creation of a perception module. First, we will discuss the task of detection utilizing multiple architecture types. As discussed in Section 3.1.2, we will mainly focus on the DetectNet architecture for this round of experiments; however, we will also provide visual and speed results for other architectures that were investigated including Faster-RCNN and SSD.

Our first attempt at creating an object detector for an autonomous driving perception system utilized the Faster-RCNN model. In our first set of experiments, we investigate feature extractors for the Faster-RCNN model Zeiler and Fergus(ZF) [96] which contains 5 shareable layers
of convolution and the Simonyan and Zisserman model (VGG) [97] which contains 13 shareable convolution layers. Figure 4.3 illustrates our first attempt at creating a detector with the Faster-RCNN model for perception in autonomous driving. Notice that we were simultaneously able to detect both pedestrians and cars as well as make an attempt to calculate their centroid. The information at the bottom of Figure 4.3 illustrates the centroids of each of the detected objects. Instead of depth information, this was an attempt at providing useful information for the perception subsystem to pass to a planning system. The results shown illustrate a model trained using the Pascal VOC dataset (discussed in Section 2.2.2.1).

Table 4.1 shows inference times for the Faster-RCNN models trained on the Pascal VOC dataset. Notice, that even when using a desktop GPU like an NVIDIA K20, we still are not able
to achieve "real-time" inference. Also, notice that due to memory limitations on the Jetson TX1, we are not able to run the VGG16 model. To contrast with a newer piece of hardware, the Jetson AGX Xavier has a much more capable GPU (including Tensor cores and more memory) which has the ability to compute a much larger operation. This architecture was not made available by NVIDIA until after the conclusion of this project so the results discussed below were all for previous architectures.

After performing many tests with the Faster-RCNN networks we next decided to work with a different network architecture boasting a higher frame rate in an attempt to make the perception system much quicker. For the next set of tests, we utilized the DetectNet architecture (discussed in Section 3.1.2) trained on the KITTI dataset. However, one of the downsides of using the DetectNet architecture is its inherent single-class design. It is very difficult to train DetectNet on a multi-class dataset, which is a large problem for a task like autonomous driving where many classes need to be located at once. Nevertheless, as a proof of concept, we utilized the DetectNet architecture with the addition of a multi-modal input (stereo vision) in an attempt to provide more information to the planning module. Figures 4.4 and 4.5 illustrate how we were able to utilize the depth information from a stereo vision camera to estimate the depth of each of the detected vehicles. Since we are designing networks that can work on edge computing (EC) devices, we need to test inference speeds on our edge computing devices. Again, we will test on the a Jetson TX1 where we achieve 128ms/7.8fps for the original DetectNet architecture and 168ms/5.9fps for the DetectNet-depth architecture.

Most recently, we have utilized the Tensorflow Object Detection API [98] as a medium for testing recently developed networks like SSD with an Inception or MobileNet feature extractor. For
inference of KITTI images, we are able to achieve 12fps/15fps for the Inception and MobileNet-based architectures on the Jetson TX2, respectively. Figure 4.6 illustrates accuracy values for each class (car and pedestrian) as well as an overall accuracy value for each model that we have tested including Faster-RCNN, SSD, and Embedded SSD (developed for the DAC Challenge and discussed in Section 3.2.2). Notice that while the Faster-RCNN outperforms all other models in terms of accuracy, all SSD models are much faster due to their single-shot architectures. For this reason, we are more likely to choose one of the faster SSD-based models with 10% degradation in accuracy if we are able to achieve a better frame rate on an edge computing device.

Also, in Figure 4.6 we can see that the accuracy on cars and pedestrians on the same model are vastly different. This can be explained by the number of examples for each class in the training set. Geiger et al. [11] develops the KITTI dataset as well as provides statistics about the dataset (shown in Figure 4.7). There are far fewer pedestrians in the dataset (and per image) than vehicles and therefore the models are trained with a biased representation of the data. This could be alleviated by training each class with a bias to learn each class more evenly. Figures 4.8 and 4.9 illustrate inference performance using the Embedded SSD model. One important feature to point out in these figures is that, although the detection performance is relatively decent, we are still missing a few objects (especially for pedestrians); this is expected since we are using a much smaller model for detection.
Figure 4.6: Jetson TX2 Inference Times on KITTI Images

Figure 4.7: KITTI Statistics [11]

Figure 4.8: Inference on KITTI Cars
4.2 Drone-based Object Tracking

In this section, we will evaluate and discuss the pipeline that was developed for the DAC system design challenge as well as provide some results on the performance of each network design. As there were multiple submissions for this challenge, we will be comparing results and talking about the optimizations that were made to create a more "real-time" system (which is 20 fps on a Jetson TX2 by the standards of this competition). First of all, this challenge is much different than the previous application. In the previous detection system, we are attempting to identify multiple objects (however, there are far fewer objects in the datasets we are using). In the DAC dataset, we are given 98 classes; therefore we would like to be able to identify as many of these 98 classes as possible with the highest mAP per class as possible.

Figure 4.10 illustrates mAP values for each of the submissions that have been done for the DAC challenges. One of the first things to notice is that for every network configuration, we have run 3 sets of test. Initially the code provided by the contest committee batches 5 images together, but only passes 3 to the GPU for computation; in essence this approach does not computing bounding boxes on 40% of the images. However, when all 5 images are passed to the GPU for computation, the frame rate decreased (as expected) due to the increased amount of computation. For the first submission, we tested both SSD and the Embedded SSD (described in Section 3.2) models which are shown in Figure 4.10 above the delimiter v0.2.0. Notice that none of these models exceed 20
fps, which is the requirement for the competition.

After the first submission, many teams created solutions that did not exceed the 20 fps requirements leading the DAC committee to create an addendum to the score calculation so all teams could be scored; the original equation for calculating a team’s score is given by Equation 4.1 while the modified equation is given by Equation 4.2.

\[ TS = R_{IoU} \times (1 + ES) \]  \hspace{1cm} (4.1)

\[ TS = \min(1, \frac{FR}{20}) \times (R_{IoU}) \times (1 + ES) \]  \hspace{1cm} (4.2)

In the above equations, \( TS \) is the total score for a team, \( R_{IoU} \) is the average IoU for a team, \( ES \) is the energy consumption during inference for a team, and \( FR \) is the true frame rate during inference for a given team. Each team’s score is now scaled by their true frame rate during inference, which still gives credit to those teams who are able to achieve frame rates greater than 20.

After evaluation of the first set of models, it was clear that without using a compression or quantization software for the model, we would need to optimize the Embedded SSD model in order to have a chance to compete with other teams. The first optimization (denoted \( v0.3.0 \)) that was completed was splitting the model such that part of the inference graph utilized the CPU while the other portion was completed on the GPU. After evaluating performance at different nodes in the graph it seemed that there was a significant portion of the time being taken by post-processing steps.
(post-CNN steps). After moving this computation from the GPU onto the CPU, we were able see significant speedups and were able to achieve a "real-time" frame rate. However, keep in mind that we are still performing inference on only 3/5 images in each batch. This in effect is only allowing our total mAP for the test set to maximize at 60%. We were able to move up in the rankings based on this optimization; however we were still in 10th position out of 60 teams.

For the most recent submission (v0.4.0), we were able to optimize the code further by utilizing two techniques (one deep learning-based and the other hardware-based). Firstly, instead of feeding each image into the network at once, we were able to use batches of images directly with the graph providing a slight speedup. As we defined the input shape to the graph as \([\text{None}, W, H]\), we were able to pass in multiple images and achieve faster inference times. Secondly, we were able to overclock the Tegra X2 GPU on the Jetson TX2 by maximizing the governor speeds. By performing both of these optimizations, we were able to achieve higher than 30 fps while executing inference with 3/5 images. As can be seen in Figure 4.10 (far right), we are able to run an entire image batch through our network with a frame rate greater than the "real-time" requirement. At the time of this writing we are awaiting the results for this penultimate submission; however, we have calculated our overall IoU and based our energy score on previous results and have estimated that our model should easily place within the top 5.

For the remainder of the competition, we utilized the Embedded SSD architecture for our model design and focused on optimization of this model. Figure 4.11 illustrates the evaluation performance of the model we have tested. The red line illustrates the average mAP for the entire dataset, which is about 91.5%. Note that we have tested on an evaluation set that was a portion of the original training set, which explains the high mAP value compared to the values that will be presented with the private test set. Also, notice that there are a good number of classes that exceed this number, meaning that we are achieving very good results on most classes. There are a few classes with extremely low mAP, in particular car3 and truck2. These are the smallest items in the dataset, sometimes appearing as small as 4x4 pixels. For a CNN that is working on a 640x360 dimensional image, a 4x4 object is extremely difficult to detect.

A few results from the competition are shown in Figures 4.12 and 4.13. Each image demonstrates a different class that the model is able to localize and identify. Notice that most of the objects in the images are quite small compared to the image size.
Figure 4.11: Evaluation Results of eSSD Architecture

Figure 4.12: DAC Challenge Output 1
4.3 Embedded & Mobile Diagnostics

In this section we will evaluate and discuss the models that were trained for both tasks related to diagnostics and management in collaboration with BMW. As described in Chapter 3, there were two aspects to this project: first we will be detecting barcode labels to alleviate the need for manual location by associates and the second is combining trailer IDs and parking lot numbers with each other for trailer location in factory trailer yards.

4.3.1 Label Detection

For barcode label detection, we have previously discussed the use of a Tiny-YOLO model (in Section 3.3.1). Here we will present evaluation results from that model on multiple hardware configurations as well as visualize some of the results.

Figure 4.14 depicts execution of the trained Tiny-YOLO model on several hardware platforms. As expected, the more powerful the GPU that is available for inference, the faster the network will run. However, what is interesting about the performance numbers here is that there may be no need to run inference on a backend system with a desktop grade GPU. The numbers in this figure show that an embedded device like a Jetson TX2 is easily able to achieve $\sim 25$ fps as well as mobile devices like the iPhone 7 Plus having the ability to run in $\sim 13$ fps. For this reason, to deploy a system of this magnitude, it may only be necessary for an associate to carry a mobile device running the app or have a device carrying an embedded device to perform the computation. This will make the system much more mobile and convenient in a factory setting (without having to be connected to a backend server containing more powerful GPUs).

A few example inference outputs are shown in Figure 4.15 while an example output of a test
Figure 4.14: Barcode Label Detection Performance

Figure 4.15: Synthetic Inference

Figure 4.16: Real Warehouse Barcode Detection
case involving barcodes on a box as it would be in a warehouse is shown in Figure 4.16. Lastly, a screen capture of the developed iPhone application running on an iPhone 7 is shown in Figure 4.17 with a frame rate approaching 13 fps.

One final piece of information to notice about each one of these results is the slight imperfections on the detections. This is due to the small model size that we have used in order to run a real-time system on a mobile device. By using a more powerful desktop GPU (or even an embedded device with a GPU), we should be able to train a deeper model to detect the barcodes slightly better. However, for this application, it is simple to extend each bounding box slightly in order to make sure that we have each label completely covered before cropping and passing to the barcode reading software.
4.3.2 Trailer Yard Management

The next project for which we will discuss results for in this dissertation is trailer yard management. The pipeline in Figure 3.16 demonstrates a 2 part process: detection and optical character recognition. The first portion of this pipeline will be reviewed here as the second portion is still in development at the time of this dissertation by another colleague in the FCTL group at Clemson University.

Again, as we have done with the label detection network, we tested on multiple pieces of hardware in an attempt to understand how well it can perform and if edge computing provides enough computational power. Figure 4.18 shows the inference performance on the Jetson TX2 as well as a few desktop GPUs (including the new Tesla V100). It’s easy to see from this figure that inference will benefit greatly by executing on a backend server with a high power GPU. The main reason for this performance difference (compared to previous SSD models which were able to run in real-time on the Jetson) is the size of the input image. Rather than working with images that are 640x360, we are instead working with images that are 2K or 4K in size. This causes multiple problems with inference including slow inference times due to image size as well as scaling issues when locating small objects within the image.

Figure 4.19 presents a few output images from the current detector illustrating 3 detections that were created by the SSD model and then passed to the OCR system for recognition. It can be seen that the detector works relatively well in locating the Trailer IDs on the corners of each trailer. Currently the OCR is being performed by cropping out each detection and then using a technique.
called Tesseract OCR [99] published by Google. In the near future, this part of the pipeline will be replaced with a CNN that is able to recognize individual digits and these will be combined for identifying the Trailer ID. This work is being completed by a colleague in the FCTL group at Clemson University.

4.4 Vehicle Re-Identification

In this section, we will briefly discuss the some of the evaluation mechanisms that were used for the vehicle re-identification project (without releasing information from NDA and patent). As we are not allowed to discuss any numerical results, we will focus on only visual results for this section. Suffice it to say that the numbers that will be published with our technique rival state-of-the-art performance with a much simpler training mechanism.

4.4.1 Visualization Results

Even though we are not displaying any of the numerical results in this manuscript, we will still provide some visual representations of typical results that one might expect and be able to interpret after a vehicle re-identification system has been trained. Visually inspecting ones results is often the first way to determine limitations of the system; this case is no different. The visualization technique that we will highlight in this section allows for a probe image to be taken from the dataset, the system then determines which of the gallery images it is closest to in the embedding space, and then ranks them based on the closest distance. An example of a few of the probe-gallery ranking
image pairs can be seen in Figure 4.20.

The left column in this visualization shows the probe images that were taken from the query set (i.e. taken from the VeRi dataset). Each of the corresponding right columns illustrates the resulting gallery images that were an output from the re-identification system as the top-ranked embeddings. One of the most important take-aways from this figure is that under each of the images, the corresponding camera from which that object was located is displayed. For this particular visualization, we have chosen to only consider gallery images that are detected in a different camera than that of the probe image. This technique is used for two reasons: first, if the gallery image is taken from the same camera it could possibly have a similar viewpoint causing it to look almost identical (tracking should be able to handle this problem without re-identification), and second, different cameras could have different lighting conditions or shadows associated with their locations which could cause problems with the system and we would like to visualize those problems.

The gallery images are ranked in terms of their distance in embedding space away from the probe image. This means that the first image (far left) in the right column is the closest in the gallery set and the last image (far right) is the 10th ranked embedding closest to the probe image. This does not mean that it should be a completely different vehicle however. As mentioned in the background section, the VeRi dataset has thousands of images and each identity in that dataset has 10s or hundreds of images corresponding to the same vehicle. That means that theoretically, all ranks up to the number of members in that identify class should be the same identity. In practice though this is not the case due to inaccuracies in the system itself (i.e. the system is not perfect).

In the visualization, each of the green boxes that is shown means that the gallery image it has chosen to rank in the top-10 is actually of the same identity while red boxes indicate a different identity. In some cases, as in row 6 (the pickup truck with something in the back), it is obvious to the human eye that the images chosen incorrectly are actually a different identity. It can easily be seen that the same things that are in the back of the truck in the probe image are not the same as in the incorrect gallery images. However, there are other cases (such as rows 3 and 8) where the incorrect selections from the gallery are not as easily distinguishable from the probe image. Upon first glance, it may seem that the incorrect selections are in fact the same identity; however, when paying closer attention it can be seen that they are in fact 2 different vehicles. In the case of row 3, these are actually 2 different taxis and the only difference is their ID number and in row 6, the incorrect selection does seem to have a slightly different rear end than the others and the probe
image. These very small differences illustrate how difficult of a problem vehicle re-identification really is, however, our network was still able to perform well across different viewpoints.

4.5 TensorRT Optimizations for Classification and Detection

The applications described in 3 and their corresponding results discussed previously have one thing in common: the end goal of the project is to be able to run the network on the edge. This could mean multiple different things depending on the application requirements, hardware availability, etc. For some applications like drone surveillance or autonomous driving it is a necessity to have some compute capability on the object itself, whether that is an embedded CPU or something like a mobile processor. For other applications like the aforementioned trailer yard management project, it may not be necessary to have any compute capability on the edge (aside from a means of capturing videos for later processing). These tradeoffs will determine whether or not an edge-computing device is needed. In this section we will discuss a solution utilizing a software called TensorRT [12] that will allow for conversion of a neural network to be able to run faster on edge computing devices.

TensorRT is a platform developed by NVIDIA to perform high-performance and low-bit width deep learning inference. It contains an inference optimizer as well as a runtime environment with the ability to transform a deep learning model (classification, detection, etc.) into an optimized model for serving inference. The process is relatively simple to utilize TensorRT. After a model has been trained, simply use the frozen graph (in this case we are discussing TensorFlow graphs) and use the runtime and optimizer to create a TensorRT optimized graph. This graph most likely will contain a much smaller number of nodes as well as occupy a much smaller footprint on the device; this is perfect for utilizing edge-computing devices. Further, as these networks have been minimized, the inference times are also much lower. A few of the optimizations that are present in the toolkit include precision calibration, layer and tensor fusion, kernel auto-tuning, and dynamic tensor memory and are depicted in Figure 4.21 which is taken from the TensorRT blog post.

Two of the most important features that we will discuss here are layer and tensor fusion and precision calibration. Firstly, layer and tensor fusion allows for the combination of layers as well as getting rid of unused layers and tensors throughout the graph. Any layer or tensor that has an unused output are completely eliminated to avoid unnecessary computation. Then combinations of layers, as large as possible, are fused to create a single larger layer; for example, most combinations
Figure 4.20: Visual Results for Vehicle Re-Identification with VeRi Dataset. Green are Correct Retrievals while Red are Incorrect Retrievals (All from Different Cameras)
of convolution, bias, ReLU are fused together into a single layer. This process is known as vertical fusing. There is also a concept of horizontal fusing, which is done after vertical fusing and takes layers that perform the same operation on the same input tensor and fuse the operations together into a single layer. Secondly, TensorRT has the ability to convert models into different precision modes including FP32, FP16, INT8, and (newly announced) INT4. For the purposes of this experiment we will stick to converting a model to FP16 and extrapolate the performance for INT8 and lower precision values. The precision is converted to smaller levels by using a concept called quantization, where all of the FP32 values are converted based on a threshold value to FP16 values, truncating a lot of unnecessary values either on the higher or lower end of the range. This allows for a much faster inference on hardware such as the Volta or Jetson AGX Xavier as the hardware allows for mixed-precision operations.

Instead of going through each application and only accelerating the networks that were used, we have elected to choose a couple more and accelerate them all with TensorRT and show the inference performance. Firstly, since re-identification networks are very closely related to classification networks in the sense that their backbones are the same, we have optimized a few different types of architectures with TensorRT and the results are shown in Table 4.2. One important thing to notice is the range of speedup values that come out of the TensorRT optimizer. For example, when optimizing a MobileNet model or a ResNet model, only by using the TensorRT optimizer are we able to get a 4x or more speedup. With other architectures, the optimizer is incapable of fusing layers and removing extraneous layers, therefore not drastically changing the size or inference time.
of the model. With this insight, a developer is now able to choose a model that will fit into their application as well as utilize larger models that before would not have been possible to use on a resource constrained device.

When it comes to detection, we would like to optimize networks the same way in order to be able to deploy larger models on an edge device which, for instance, could be installed on a drone or on an autonomous security device. We use the same optimization techniques as mentioned previously and utilize the resulting models to run inference. The inference time results as well as the speedup values are shown in Table 4.3. At first glance, it can easily be seen that the optimization was able to create much more efficient networks for classification than it was for detection. However, this does not mean that the optimizations it performed are not useful. If we take the SSD Inception network, for example, we start out with a network that can run inference in about 55ms, which corresponds to about 18fps. For most real-time applications, 20 or 30fps is the threshold for real-time application. In this case, without any optimizations we would need to use a different smaller network. After TensorRT optimizations we were able to take the inference time (at the same accuracy level) down to about 29ms, which is about 35fps.

As we have discussed, for many applications that need to run on the edge, TensorRT is the ideal solution to run inference on larger models and bring out the necessary extra little bit of performance. For detection tasks like autonomous driving, drone surveillance, and logistics, larger, more accurate models can be utilized and still run in real-time. For re-identification, since we need a detection model and a classification model, we have the ability to optimize both and still achieve state-of-the-art performance with both networks while running in real-time (or near real-time). TensorRT is not able to optimize all network architectures, but is a useful tool for those that it can optimize if the application can be built with that architecture.

<table>
<thead>
<tr>
<th>Network</th>
<th>Original TF</th>
<th>TF-TRT (FP32)</th>
<th>TF-TRT (FP16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNet V1 (128)</td>
<td>2.372</td>
<td>0.425 [5.58x]</td>
<td>0.413 [5.74x]</td>
</tr>
<tr>
<td>MobileNet V1 (224)</td>
<td>4.279</td>
<td>0.883 [4.84x]</td>
<td>0.776 [5.51x]</td>
</tr>
<tr>
<td>Inception V2</td>
<td>5.408</td>
<td>1.984 [2.73x]</td>
<td>1.728 [3.13x]</td>
</tr>
<tr>
<td>Inception V3</td>
<td>7.827</td>
<td>3.117 [2.51x]</td>
<td>2.807 [2.79x]</td>
</tr>
<tr>
<td>Resnet50</td>
<td>9.819</td>
<td>2.679 [3.67x]</td>
<td>2.386 [4.12x]</td>
</tr>
</tbody>
</table>

Table 4.2: TensorRT Classification Model Optimizations on Jetson AGX Xavier (time in ms)
Table 4.3: TensorRT Detection Model Optimizations on Jetson AGX Xavier (time in ms)

<table>
<thead>
<tr>
<th>Network</th>
<th>Original TF</th>
<th>TF-TRT (FP32)</th>
<th>TF-TRT (FP16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD MobileNet V1</td>
<td>27.2</td>
<td>22.4 [1.21x]</td>
<td>20.8 [1.31x]</td>
</tr>
<tr>
<td>SSD MobileNet V2</td>
<td>43.6</td>
<td>27.1 [1.61x]</td>
<td>24.4 [1.78x]</td>
</tr>
<tr>
<td>SSD Inception V2</td>
<td>54.9</td>
<td>34.9 [1.57x]</td>
<td>28.7 [1.91x]</td>
</tr>
<tr>
<td>SSD Resnet50</td>
<td>242.6</td>
<td>142.9 [1.70x]</td>
<td>77.4 [3.13x]</td>
</tr>
</tbody>
</table>

4.6 Summary

In summary, we have shown our results for each of the four applications: perception for autonomous driving, drone-based detection, warehouse management and logistics, and vehicle re-identification. For autonomous driving we were able to demonstrate our success in object detection for important objects like vehicles and pedestrians while incorporating other important information like depth, that will aide in the planning of motion. We were also able to show our progress in completing an inference engine for a drone-based detection system as well as the use of a new architecture called Embedded-SSD. We were also able to extend our work with object detectors and work with both warehouse management (in the form of barcode label detection) and trailer yard management. Finally we were able to develop a vehicle re-identification system with the ability to provide an embedding for any detected vehicle and then compare it with a gallery of images to extract its identity. Finally, we were able to accelerate a few of the networks with NVIDIA’s TensorRT with small explanations of how they can be deployed on edge-devices with this software. All of the applications discussed in this dissertation were developed with the end goal of a base project, operating at the edge, that is a building block for a complete system. In Chapter 5 we will highlight our conclusions from this research and give future work that could be performed with each of these applications to either utilize newer hardware or further optimize the networks past the scope of this dissertation.
Chapter 5

Conclusions & Future Work

Our research seeks to address a few typical problems in developing deep learning pipelines for computer vision tasks. By developing applications in multiple domains, we are able to conclude that the metrics and pipeline components that we have created are able to provide a starting point for any computer vision application. Detection specifically with some metric learning have been the focused topics that could utilize deep learning. Although there are many different areas of research occurring for object detection in deep learning, it is necessary to have a method for inserting a trained network into a pipeline for easy testing and evaluation. This dissertation attempts to design a general framework that can be used for most object detection tasks in deep learning-based computer vision as well as evaluate specific architectures on edge-computing devices. We will also build some conclusive arguments and rules based on the observations that we have seen during application development and deployment.

Chapter 2 serves as both a background discussion as well as a literature survey discussing machine learning and deep learning in general, followed by more specific architectures and techniques for object detection. Deep learning detection models such as RCNN, SSD, and YOLO all provide a necessary step forward in the field; however, they all have their shortcomings. While they are all able to extract features to alleviate the need for feature engineering, they still must be optimized for a particular application. The network designer must also make sure to choose the correct network architecture for the task at hand (for example, choosing a real-time architecture or one where the accuracy is very high).

Chapter 3 takes a deep dive into the four applications that govern the development of this
dissertation. Each application is derived from a different domain of problems; however, all applications fall under the umbrella of computer vision-based deep learning. Our research on autonomous driving originates from the fact that we would like not only a cost effective solution but also a straight-forward and expandable solution. For this reason, we have chosen not to use an end-to-end training approach, but rather split the pipeline into multiple stages: sense, perceive, plan, act. We have focused solely on the perception piece of this pipeline and were able to show the ability of deep learning to not only localize objects in an environment but also provide more information (like depth) about the detected objects. Our research on drone-based vision has shown the use of an embedded detection architecture that is able to run in real-time on embedded devices while still performing quite well when looking at evaluation metrics like mAP and IoU. Our evaluation performed on logistics and management tasks for manufacturing facilities illustrates the usefulness of deep learning in this domain as well as its effectiveness on a cheaper edge-computing device. Our short discussion of vehicle re-identification shows a slightly different application for computer vision, but still solidifies the use of edge-computing to solve some problems and the need for optimized networks to be able to run at the edge.

Chapter 4 demonstrates some of the results for autonomous driving perception, drone-based vision, embedded/mobile diagnostics, and re-identification. Our pipeline for autonomous driving perception not only has the ability to localize objects but also provide more information to the planning module for it to make a more informed decision about driving path. For drone-based detection, we are able to show the ability of a modified deep learning network to localize very small objects within an image with a high degree of accuracy. We are also able to show detection for diagnostic reasons in manufacturing warehouses including barcode labels and trailer IDs, which helps manufacturing facilities run more smoothly. Lastly we showed the ability to help a surveillance system with the ability to "identify" objects in multiple camera environments.

5.1 Contributions & Conclusions

The work presented in this dissertation has investigated design strategies for development of deep-learning based computer vision applications as well as enablement of the extraction of important features from the design process useful in the deployment process on edge devices. The development and deployment of the aforementioned applications has lead to the creation of a set of general
principles for creating deep-learning based computer vision applications for edge computing devices. The principles range from deep learning architecture design to hardware optimizations, each of which needs to be tuned for each application or use case.

1. All computer vision applications utilize some sort of sensors as input. For example, autonomous driving or drone surveillance all use RGB cameras (or some other form of camera) as input. By designing a deep learning network and training it properly, it will have the ability to learn from this modality of data and produce a state-of-the-art results. Further, for applications which rely on more sensor data (i.e., RGB cameras, stereo cameras, LIDAR, RADAR, etc. in the case of autonomous driving), sensors can be combined at different portions during the network creation to create a feature. This feature can then be used in the control of the system as a whole.

2. The design decision about whether or not to deploy a system fully on the edge or not is very important and should be made by examining the requirements of the system. For example, for a surveillance system where actions should be taken immediately if an anomaly is detected, edge computing is a necessary solution. However, for a logistics application similar to the one described above, real-time may not be a necessity; therefore, the data could be streamed to a larger more accurate network in the cloud or in a data center.

3. Deep learning network design can enable deployment on embedded or edge computing devices by limiting the depth, width, and thickness of the network. The trade-off between speed and memory is discussed in the results section; the larger the network, (typically) the more accurate the network will be. However, it is not always necessary to have a flawless network; sometimes it is much more important to have a real-time network with a slightly sub-state-of-the-art accuracy performance. Aside from simply creating smaller networks, there are also other means of creating more speed-efficient neural networks, namely converting NxN convolutions into Nx1 and 1xN convolutions or utilizing 1x1 convolutions or depth-wise separable convolutions as done in MobileNet.

4. During training there are multiple techniques that can be utilized for creating a smaller network. The first is traditional pruning where weight (or more in general, entire filters) are removed from the network if they do not produce a large change in the output of the network.
Other techniques for producing smaller optimized models utilize knowledge distillation to teach a large \textit{(teacher)} network all the necessary information and then \textit{distill} that knowledge into a smaller \textit{student} network.

5. Models that have been created through a training process have the ability to be optimized as well for edge-computing. There are many techniques including quantization, layer-fusion, and hardware kernel optimizations that take a pre-trained model and modify it for inference on an edge computing device. Typically these optimizations only hinder the accuracy performance of the network by tenths of percents while creating a much smaller network in terms of layers and parameters.

In short, some of the design decisions that were used in this dissertation lead to the creation of this list of principles. For any application that needs to be deployed on the edge, the deep learning network should be as small as possible while still providing the necessary accuracy to produce the desired result. Pruning and distillation techniques used during training along with quantization and other techniques utilized on a pre-trained model before inference provide a framework for going from application to design to development to optimization to deployment. Architectural changes are a must when designing a network, but other optimization techniques could be an easier starting point for immediate benefits.

5.1.1 Contributions

1. Development of deep learning models for autonomous driving perception including object detection and depth prediction.

2. Development of object detection models for drone-based vision tasks such as surveillance.


4. Deployment of deep learning detection models on embedded and mobile devices.

5. Discussion of vehicle re-identification and some of the work that has helped further research and development for improved surveillance applications.

6. Development of the above set of general principles for creating and deploying a deep-learning based computer vision application on an edge computing device.
5.2 Future Work

As part of this work, we would like to provide a few research directions that could be further explored. Firstly, the autonomous driving project showed the ability to not only create a deep learning solution for objects like vehicles, pedestrians, and signs but also showed the ability to use multi-modal data; however, autonomous driving is still a thriving field. Many research institutions that are studying autonomous driving have said that at least 8 or 10 deep neural networks will be needed to provide a fully autonomous driving solution. An extension to this work could be to create these different neural networks with different abilities and utilize some sort of single hardware solution to run them all. TensorRT could be utilized heavily as described before to be able to run all of these networks in real-time.

Another set of future work that could be pursued following the research done in this dissertation involves further optimization of algorithms for edge-computing. In this dissertation, solutions like utilizing smaller architectures with modified filters was used along with toolkits built for inference optimization. It would be very useful to look into other techniques that could further compress the network architecture while maintaining the accuracy. This could open up many doors in the realm of using larger architectures for embedded/mobile devices for inference or other deep learning tasks. For example, if we were able to create a network that could run in real-time directly on a camera in an airport, we would have the ability to run analytics directly on the edge and only save information to the backend server when an anomaly occurred. This could drastically decrease data center footprints by simply creating more optimized networks with the ability to run on smaller, lower power hardware.

Lastly, multi-branch networks could assist in many of the tasks described in this dissertation. Firstly with autonomous driving, a single backbone network (or feature extractor) could be used with a branching layer where there are many heads (including detection, segmentation, etc.). The idea behind a multi-branch network is that the same set of features learned for detection can be used for the task of segmentation since they are all trying to identify the same types of objects. Multi-branch networks also have the ability to provide more information to upstream systems.
Bibliography


[61] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. CoRR, abs/1604.07316, 2016.


