

I. Personal and study details

Student's name: **Zelenskyy Mykhaylo** Personal ID number: **434789**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Cybernetics**
Study program: **Cybernetics and Robotics**
Branch of study: **Robotics**

II. Master's thesis details

Master's thesis title in English:

Deep Neural Networks in Embedded Systems

Master's thesis title in Czech:

Hluboké neuronové sítě ve vestavěných systémech

Guidelines:

Explore state-of-the-art solutions and both commercial and non-commercial solutions available on the market for performing object detection and classification using DNN. Summarize the limits and possibility for real-time use in embedded environment.

Improve existing TensorRT converter and design a new approach of conversion of DNN architectures in Tensorflow for utilizing Tensor cores available on Nvidia Jetson AGX Xavier and speeding up inference.

Create a working prototype of on-the-edge object detection software and API for communication with Nvidia Jetson Xavier with awareness of limits on embedded devices.

Evaluate the performance and compare with the existing solutions.

Bibliography / sources:

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning – 2016, MIT Press

[2] Support Resources for Jetson AGX Xavier (dostupné online
<https://developer.nvidia.com/embedded/community/support-resources>)

[3] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. - Focal loss for dense object detection. - 2017 IEEE International Conference on Computer Vision

[4] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. - You only look once: Unified, real-time object detection. - 2016 IEEE Conference on Computer Vision and Pattern Recognition

Name and workplace of master's thesis supervisor:

Ing. Lukáš Hrubý, GoodVision s.r.o., Prague

Name and workplace of second master's thesis supervisor or consultant:

doc. Ing. Tomáš Svoboda, Ph.D., Vision for Robotics and Autonomous Systems, FEE

Date of master's thesis assignment: **08.02.2019** Deadline for master's thesis submission: **24.05.2019**

Assignment valid until: **30.09.2020**

Ing. Lukáš Hrubý
Supervisor's signature

doc. Ing. Tomáš Svoboda, Ph.D.
Head of department's signature

prof. Ing. Pavel Ripka, CSc.
Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Deep Neural Networks in Embedded Systems

Bc. Mykhaylo Zelenskyy

Supervisor: Ing. Lukáš Hrubý
May 2019

Acknowledgements

I wish to thank my supervisor, Ing. Lukáš Hrubý, for his patient guidance, enthusiastic encouragement and useful critiques of this thesis.

I would also like to express my deep gratitude to my family and friends for their support during my whole studying time.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university thesis.

Prague, May 20, 2019

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 20. května 2019

Abstract

This thesis explores possibilities of running deep neural networks on embedded systems for traffic monitoring. The main goal is to design, implement and test a prototype of the system for vehicle and people detection from video records and camera streams using neural networks. In this work, three different CNN architectures are compared, namely RetinaNet, YOLO and SqueezeDet. Software prototype is designed so that it can be deployed on commercial off-the-shelf devices, e.g. Jetson Xavier module from NVIDIA.

Keywords: neural networks, embedded systems, machine learning, edge computing, jetson, retinanet, yolo, squeezeDet

Supervisor: Ing. Lukáš Hrubý

Abstrakt

Tato práce prozkoumává možnosti využití hlubokých neuronových sítí ve vestavěných systémech pro účely monitorování dopravy. Hlavním cílem je navrhnout, implementovat a otestovat prototyp systému využívajícího neuronové sítě pro detekci dopravních prostředků a lidí z kamerových záznamů a streamů. V práci jsou porovnány tři architektury neuronových sítí: RetinaNet, YOLO a SqueezeDet. Prototyp software je navržen tak, aby mohl být nasazen na komerčně dostupná zařízení, např. Jetson Xavier od NVIDIA.

Klíčová slova: neuronové sítě, vestavěné systémy, strojové učení, edge computing, jetson, retinanet, yolo, squeezeDet

Překlad názvu: Hluboké neuronové sítě ve vestavěných systémech

Contents

1 Introduction	1	6 Implementation	35
1.1 Problem overview	1	6.1 Docker	35
1.2 Overview of technologies for traffic survey	2	6.2 Prerequisites installation	36
1.2.1 Manual counts	2	6.3 API	37
1.2.2 Pneumatic tube detector	2	6.4 YOLOv3 detector	38
1.2.3 Sensors for traffic data collection	3	6.4.1 Data provider	38
1.2.4 Video object detection	3	6.4.2 Detector	39
1.3 Solution design	4	6.4.3 Main application	39
1.3.1 The goal of the thesis	4	7 Conclusions	41
1.3.2 Edge computing	5	7.1 Feature work	42
1.3.3 Hardware	5	A Acronyms	43
2 Related works	9	B API definition	45
2.1 Deep neural networks on the edge	9	C CD	47
2.2 Commercial traffic monitoring and vehicle recognition systems	10	D Darknet-19	49
2.3 Non-commercial traffic count and vehicle recognition systems	11	E Darknet-53	51
3 Neural networks	13	F Sample from videos used for evaluation	53
3.1 Architecture	14	G Bibliography	55
3.1.1 Convolutional layer	14		
3.1.2 Non-linearity layer	14		
3.1.3 Pooling layer	15		
3.1.4 Fully connected layer	15		
3.2 Training of CNN	15		
3.2.1 Backpropagation	16		
3.2.2 Overfitting	17		
3.2.3 Batch normalization	17		
4 Network architectures used in the thesis	19		
4.1 ResNet	19		
4.2 RetinaNet	19		
4.3 SqueezeDet	20		
4.4 YOLO	22		
4.4.1 YOLOv1	22		
4.4.2 YOLOv2	24		
4.4.3 YOLOv3	24		
5 Experiments	27		
5.1 Datasets	27		
5.1.1 KITTI dataset	27		
5.1.2 “GV-2018”	28		
5.2 Performance evaluation	29		
5.3 Inference time evaluation	30		
5.3.1 Mixed precision calculation	31		

Figures

1.1 Manual traffic count [1]	2
1.2 Pneumatic tubes detector [2]	3
1.3 Edge computing paradigm [3]	5
1.4 Tensor core operation [4]	7
1.5 Deep Learning accelerator architecture [5]	7
1.6 Vision Accelerator [5]	8
3.1 Neuron cell [6]	13
3.2 Convolutional neural network [7]	14
3.3 Convolutional layer [8]	14
3.4 Convolution computation [8]	15
3.5 Comparison of ReLu and sigmoid non-linearities	16
3.6 Pooling layer [8]	17
4.1 Residual block [9]	20
4.2 ResNet architecture [9]	20
4.3 Feature Pyramid Network [10]	21
4.4 Fire module in SqueezeNet [11]	21
4.5 SqueezeDet comparison with other state-of-art solutions [12]	22
4.6 YOLOv1 architecture [13]	23
4.7 YOLOv2 improvement [14]	24
4.8 YOLOv3 comparison [15]	25
5.1 Image samples from KITTI dataset	28
5.2 Image samples from “GV-2018”	29
5.3 Intersection over Union [16]	30
6.1 Architecture comparison virtual machine vs. container [17]	36
6.2 Application implementation	39
D.1 Darknet-19 [14]	49
E.1 Darknet-53 [15]	51
F.1 Samples from used videos	54

Tables

1.1 NVIDIA Jetson comparison [18]	6
5.1 Characteristics of video for inference time evaluation	31
5.2 PC1 technical specification	31
5.3 PC2 technical specification	32
5.4 Inference time evaluation	32
5.5 Inference time using mixed precision calculation	32
5.6 CNN evaluation	33
6.1 API definition	37

Chapter 1

Introduction

Internet of Things (IoT) and artificial intelligence (AI) is becoming part of our lives making everything smart and intelligent. Integration of IoT and AI in smart cities is one of the promising applications [19]. Machine learning has given the ability to process various tasks without human intervention such as recognizing objects, playing games, diagnosis diseases. Deep learning is one of the major branches in machine learning, and one of the most trending applications of deep learning is traffic object detection, which is the core component of traffic control in smart cities. It makes the urban life convenient and also safer.

1.1 Problem overview

Traffic monitoring is widely used by state and local transportation officials in the planning of road improvements and monitoring of traffic conditions [20]. In private sector traffic monitoring and management can be used for several reasons like identifying the best location for business, analysing how traffic may impact a potential site, scheduling staff hours to peak periods of traffic etc [21].

Traffic data can be collected in many ways, as we will see in following Section 1.2. Some of these methods can be either resource-intensive or time consuming, which leads to infrequent analysis or analysis based on small sample sizes or outdated data. It is possible to predict trends in traffic flow with only small amount of data using different statistical approaches, but it may not be as accurate as sometimes required. When selecting the method for traffic survey, we should take into account different factors:

1. Accuracy – we must be sure that data we collect are sufficiently accurate for our needs. If we want to predict percentage of sales depending on walk-ins, we may need less precise data about pedestrian flow comparing to data that will be used for urban highway planning.
2. Price – some methods are more suitable for a one-time survey, while repeated or continuous survey may require another approach. Therefore, right choice of the method can reduce the cost of the research.

3. Location – in some location, e.g. high-traffic roads or highways, it is almost impossible to use invasive sensors or manual data collecting methods, and cameras and other non-invasive sensors are preferable.

1.2 Overview of technologies for traffic survey

1.2.1 Manual counts

Manual traffic counting are defined as in-person traffic counts, where the counter is physically present at the location of data collection [22] or counts objects from recorded videos of the road. A person usually uses either an electronic held counter or records data using a tally sheet (Fig. 1.1). Manual counts are quite precise with only 1% counting errors and classification errors between 4-5% [23]. Only a small sample of data is taken, and results are extrapolated for the rest of the year or season.



Figure 1.1: Manual traffic count [1]

1.2.2 Pneumatic tube detector

This method uses one or more rubber hoses that are stretched across the road and connected at one end to a data logger, while the other end of the tube is sealed, as shown in Fig. 1.2. When a vehicle tire passes over the tube, sensors send a burst of air pressure along the tube and data are logged. This method can be used to record data from several lanes of traffic and vehicle direction can be determined by recording which tube was crossed first, but if two vehicles cross the tubes at the same time, the direction cannot be determined correctly. One of the advantages of pneumatic tube detector is its low cost and easy deployment. However, its durability is low, it is not

suitable for high flow or high-speed roads, and it is harder to classify some types of vehicles[24].



Figure 1.2: Pneumatic tubes detector[2]

1.2.3 Sensors for traffic data collection

The different sensors can be used for traffic data collection. For example, piezoelectric sensors mounted in a groove cut into road's surface collect data by converting mechanical energy into electrical energy[25]. Basically, it works on the same principle as a pneumatic tube detector.

Another example of sensor used for traffic monitoring is magnetic sensors that detect vehicle by measuring the change in the earth's magnetic field as the vehicle passes over the detector buried in the road[25].

Other sensors as passive or active infrared devices, acoustic detectors, inductive loops are also quite popular in traffic surveys, but their usage is limited.

1.2.4 Video object detection

While manual object counting is labour intensive and pneumatic tube detectors and other sensors do not provide sufficient classification results, systems that can automatically analyse videos of roads become more popular for traffic counting and analysing. Limited computation power used to be one

of the biggest limitation of using video analyses for traffic data management. However, today's technology allows to run advanced AI algorithms for image classification in real-time[26].

Applying algorithm for object detection and classification for object counting has proved to be extremely accurate with accuracy exceeding 99% [27]. These systems are cost-effective as they can provide more complex data, like precise vehicle and pedestrians trajectories, vehicle type etc., comparing to other technologies used in traffic monitoring systems. However, most of the available solutions work only with recorded videos and are not suitable for online traffic count and monitoring, or uses expensive and high energy-consuming servers for calculations, as we will further discuss these systems in Chapter 2.3.

1.3 Solution design

In this section, we will briefly discuss what our goal and use case are, what edge computing means and what hardware we will use.

1.3.1 The goal of the thesis

In this thesis, we would like to explore both commercial and non-commercial solution for object detection and classification using deep neural networks and implement and test a software prototype for real-time traffic count and monitoring utilizing edge computing.

We want to focus on detection of objects that are described in the Standard UK vehicle classification scheme called COBA [28], [29]. That document defines categories for traffic count systems:

1. Car - passengers vehicle with less than 16 seats,
2. Light Goods Vehicle (LGV) - car type delivery vans,
3. Ordinary Goods Vehicle 1 (OGV1) - a rigid vehicle with two or three axles,
4. Ordinary Goods Vehicle 2 (OGV2) - a rigid vehicle with four or more axles,
5. Public Service Vehicle (PSV) - all public service vehicle,
6. Motorcycle (MC) - all types of motorcycles including those with sidecars,
7. Pedal Cycle (PC) - all types of pedal cycles.

For this thesis purposes, we will simplify these classes and we will unite OGV1 and OGV2 into one category "Truck", instead of all PSV we will use buses only, and we will also add another class "Person".

1.3.2 Edge computing

Edge computing is a computing paradigm when data are processed at the edge of the network. Figure 1.3 shows the two-way computing stream in edge computing. We can see that things are not only data consumer, but they also work as data producers here. They can both request service and content from the cloud and perform the computing tasks.

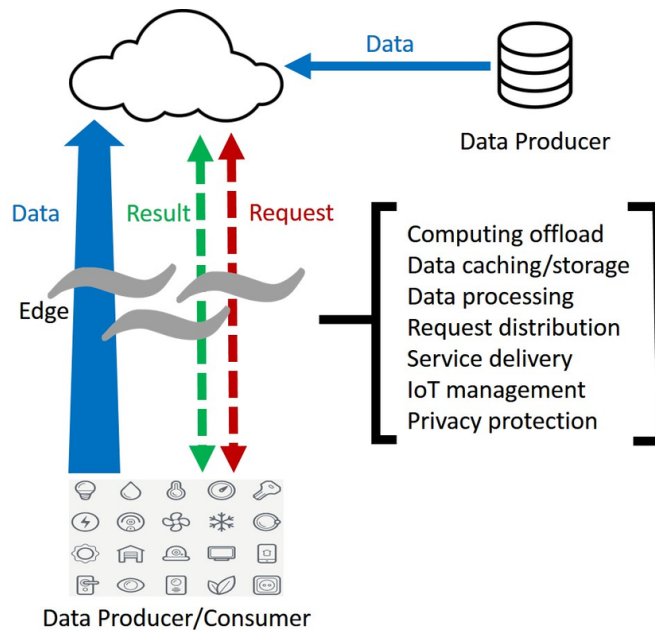


Figure 1.3: Edge computing paradigm [3]

Edge computing is beneficial for many real-time applications such as traffic object detection because it can provide a timely solution without no need to send big amount of visual data to remote server, and allows to process them where they are being collected. Some researches [30] show that platforms built for face recognition application have reduced response time from 900 to 169 ms when computations are moved from cloud to the edge.

1.3.3 Hardware

While edge computing can drastically reduce response time, we should keep in mind that hardware used for edge computing may not be as powerful as one we can use in cloud computing. Some works have proved [31] that such machine learning algorithm as random forests or support vector machine can be run on widely available devices like Raspberry Pi. Another research [32], however, shows that Raspberry Pi does not have enough computation power for deep convolutional neural networks. Because of it, we should find suitable hardware, if we want to use CNN for traffic object detection and recognition.

One of the possible solution for DNN acceleration can be Neural Compute

Stick (NCS) from Intel¹. This type of hardware in the form of USB drive is designed specifically for DNN application and can be used with any other hardware with Ubuntu or Raspbian OS. It supports Caffe and TensorFlow frameworks and contains some pre-trained CNN models, which are already converted into supported format. Experiments [33] show up to $8\times$ speedup on Raspberry Pi 3 with NCS. However, using of NCS is limiting as it is only accelerating unit and cannot be fully tailored to our needs. Converting trained neural networks into right format requires a lot of workarounds, and it is impossible to use architectures not supported by Caffe or TensorFlow.

Nvidia offers a series of embedded modules for edge computing called Nvidia Jetson. These modules are specifically designed for accelerating machine learning applications. First board, Jetson TK1, was presented in 2014 [34]. Jetson TK2 was announced in 2017 [35] and was designed for low power systems like smaller camera drones. The next module called Jetson Xavier introduced in 2018 [36] brings up to $20\times$ acceleration compared to predecessor devices with power efficiency being improved $10\times$. The newest Nvidia Nano was announced in 2019 [37] and is focused on hobbyist robotics thanks to its low price. Comparison between Jetson modules and Inter Neural Compute Stick [38] shows that even the least powerful module, Jetson Nano, three times outperforms NCS. Jetson modules are compared in Table 1.1.

	Jetson TX2 8GB	Jetson AGX Xavier™	Jetson Nano™
GPU	NVIDIA Pascal™ 256 NVIDIA CUDA cores	NVIDIA Volta™ 512 NVIDIA CUDA cores 64 Tensor cores	NVIDIA Maxwell™ 126 NVIDIA CUDA cores
CPU	Dual-core Denver 2 64-bit CPU Quad-core ARM A57 complex	8-core ARM v8.2 64-bit 8MB L2 + 4MB L3	Quad-core ARM Cortex-A57 MPCore processor
Memory	8GB 128-bit LPDDR4	16GB 128-bit LPDDR4x	4GB 64-bit LPDDR4
Storage	32GB eMMC 5.1	32GB eMMC 5.1	16GB eMMC 5.1
Video Encode	2x 4K @ 30 (HEVC)	8x 4K @ 60 (HEVC)	4K @ 30 (H.264/H.265)
Video Decode	2x 4K @ 30 12-bit support	12x 4K @ 30 12-bit support	4K @ 60 (H.264/H.265)
Connectivity	Wi-Fi onboard	Wi-Fi requires external chip Gigabit Ethernet	Wi-Fi requires external chip
Mechanical	400-pin connector	699-pin connector	260-pin edge connector
Camera	12 lanes MIPI CSI-2, D-PHY 1.2 (30 Gbps)	16 lanes MIPI CSI-2, 8 SLVS-EC D-PHY (40 Gbps), C-PHY (109 Gbps)	12 lanes (3x4 or 4x2) MIPI CSI-2, DPHY 1.1 (1.5 Gbps)
Size	87 mm x 50 mm	100 mm x 87 mm	69.6 mm x 45 mm

Table 1.1: NVIDIA Jetson comparison [18]

As we can see, unlike other boards Jetson Xavier is built around NVIDIA Volta™GPU with tensor cores which we will describe later in paragraph 1.3.3.1. Jetson Xavier also uses two engines designed for AI acceleration, namely Nvidia Deep Learning Accelerator² and Vision Accelerator engines. Both of them are described in the paragraph 1.3.3.2.

¹<https://software.intel.com/en-us/neural-compute-stick>

²<http://nvidia.org/>

$$\mathbf{D} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32
FP16
FP16
FP16 or FP32

Figure 1.4: Tensor core operation [4]

1.3.3.1 Tensor cores

Tensor cores are capable of performing one matrix-multiply-and accumulate operation in a 4×4 matrix in one GPU clock cycle [39]. Tensor cores in mixed-precision mode takes input data in half floating-point precision, perform matrix multiplication in half precision and the accumulation in single or half precision, as we can see in Fig. 1.4.

This operation is crucial for most of machine learning applications, especially in deep learning, because, as we will discuss in further chapters, output of each neuron in neural networks are calculated in a similar way.

1.3.3.2 Deep learning and vision accelerators

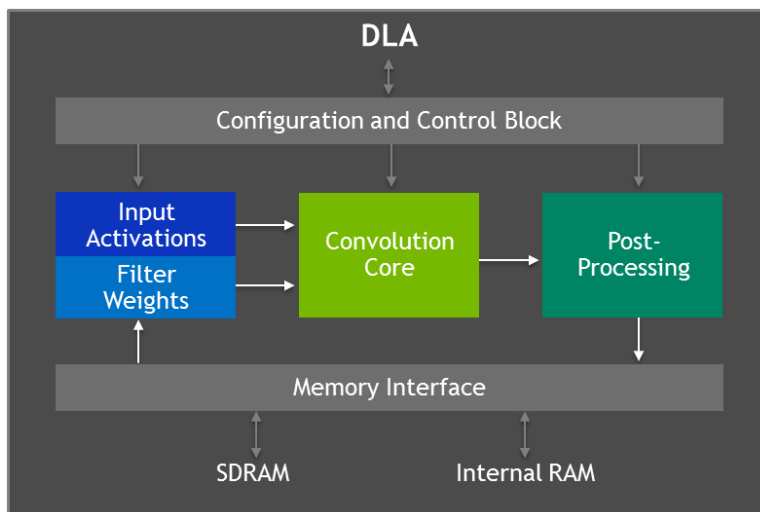


Figure 1.5: Deep Learning accelerator architecture [5]

Deep Learning Accelerator, shown in Fig. 1.5, improves energy efficiency and free up the GPU to run more complex networks and dynamic tasks. The DLA has up to 5 TOPS with INT8 precision or 2.5 TFLOP per second with FP16 precision [40]. It also supports the acceleration of most common CNN layers that are described in Chapter 3

Another engine used in Jetson Xavier is Vision Accelerator, shown in Fig. 1.6. This engine is responsible for the acceleration of algorithm such as optical

flow, point cloud processing, morphological operations, histogramming etc.

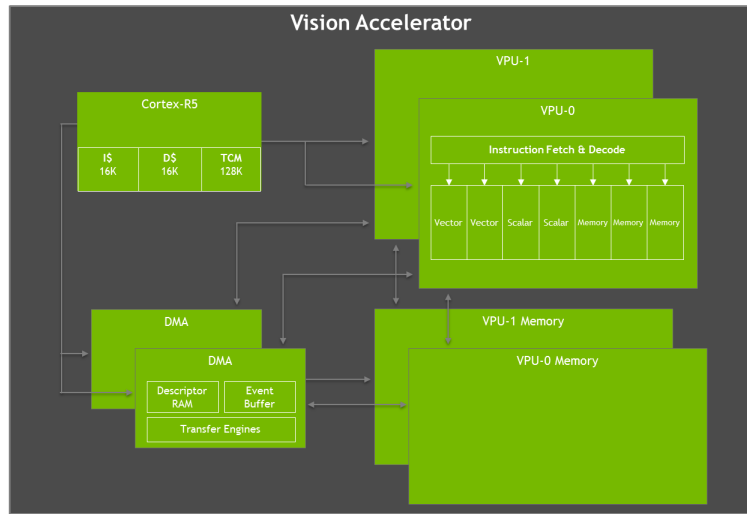


Figure 1.6: Vision Accelerator [5]

Chapter 2

Related works

Application of machine learning in traffic monitoring and management is a trending branch in smart cities development. This topic is broadly discussed in the commercial field and academic communities. Some researches focus on transferring compute-intensive algorithm to the edge, while others explore how to improve existing cloud-based systems.

In this chapter, we will go through related research around deep neural networks on the edge and discuss existing commercial and non-commercial solutions for traffic monitoring.

2.1 Deep neural networks on the edge

It is widely recognized that video processing and object detection is computing intensive to be handled by resource-limited edge devices. Many studies try to solve the problem of migration well-known machine learning algorithms to the edge.

For example, in [41] authors introduce a lightweight Convolutional Neural Network (L-CNN). They implemented a prototype of the L-CNN on a Raspberry Pi 3 and compared it with other algorithms, like Haar Cascades [42], HOG + SVM [43] and SSD GoogleNet [44]. They achieved 1.79 FPS rate with their algorithm, which was slower comparing to Haar classifier. However, it had a smaller false positive rate (6.6% against 26.3%).

This research [45] shows how the careful design of CNN for object detection can lead to real-time performance on embedded devices. They tested DroNet [46] architecture, which was proposed as an efficient neural network for UAV applications, changing the structure of the network to achieve the highest possible performance. Tests showed that on embedded devices like *OdroidXU4* it is possible to achieve up to 10 FPS rate with 80% accuracy if the network architecture is properly changed.

Other researches [47], [48] demonstrate how DNN can be used on older Jetson platforms and how they can be optimized to ensure the highest performance with little or no accuracy loss.

2.2 Commercial traffic monitoring and vehicle recognition systems

There are many commercial solutions for traffic count available on the market.

For example, Miovision, the company from Canada devoted to smart cities, offers semi-automatic system for traffic data processing¹. They ensure 95% accuracy on all traffic data collection studies using AI and manual review, with a minimum of 12% of every hour of provided video being manually reviewed. Despite being accurate, this solution does not produce results in real-time and requires addition system for video data collection. Czech company DataFromSky² offers similar tool: video must be recorded separately, sent to them and output results are manually reviewed.

GoodVision³ offers cloud software platform for fully automated traffic data collection from videos. GoodVision Video Insight provides data from uploaded video within an hour with no human interaction in data collection process. Unlike Miovision or DataFromSky, GoodVision software allows users to analyse their data interactively by drawing lines and zones after video was fully processed. However, this solution again requires video to be recorded in advance and is not able to process videos directly from cameras.

Another company from the Czech Republic, Eyedea Recognition⁴, focuses on developing object detection and object recognition systems based on machine learning and artificial intelligence methods. Their software for make and model recognition analyses vehicle appearance and recognizes its category, make, model and colour. However, this solution has a lot of limitations. First of all, it does not provide vehicle detections, which means another algorithm should be used to do object detections. Accordingly to Eyedea's manual [49], each vehicle must be captured from top frontal or top back view, and its cropped image must be aligned in the way the license plate is horizontally aligned. This solution also lacks recognition of bicycles and pedestrians, which is essential for any traffic management.

Companies like Nationwide Data Collection⁵ or The Traffic Group, Inc.⁶ provide a solution for traffic count using technology we described in Section 1.2, namely manual counting, pneumatic tubes and various sensors. Regardless of being reliable, these systems have lots of disadvantages comparing to systems that utilize machine learning algorithm.

¹<https://miovision.com/datalink/traffic-data-processing/>

²<http://datafromsky.com/>

³<https://goodvisionlive.com/>

⁴<http://www.eyedea.cz/>

⁵<https://nationwidedatacollection.co.uk/>

⁶<https://trafficgroup.com/>

2.3 Non-commercial traffic count and vehicle recognition systems

Traffic count, management and forecast is widely studied by a lot of universities and non-commercial organization around the globe.

In 2005 scientists from the Bruno Kressler Foundation proposed SCOCA – System for Counting and Classifying Automatically vehicles [50]. In their system data are collected from the camera to the local units that are connected through an optical fibre to a central processor. In this application, user can define a region of interest, from where data should be collected. Images are processed as soon as they are received by TRAFFIC DATA EXTRACTOR. The extractor contains two submodules: the first one, called *Detectorandtracker*, analyses the frames in order to locate objects passing through the region of interest; the second one, *ObjectParameterExtractor*, is responsible for object classification and estimation its real-world path. For object detection they use background subtraction, and object tracking is done by tracking the moving map. Classification module runs in parallel to the detector. Authors use model-based classification which relies on a set of three-dimensional models that describes the shape of different groups of vehicles. The 3D model is projected into the 2D scene, and the convex hull of the object is matched against each projection, and the best score is stored. The final classification is done when every object detection is classified. This system has 88.3% classification accuracy, but model-based classification can be time-consuming and requires accurate camera setting, and unlike CNNs it is less adaptable to changes in classification classes. Also, since the final classification result is available after the object leaves the scene, the user does not get all the information about traffic immediately.

Another research [51] proposes a solution for real-time vehicle detection using advanced AdaBoost algorithm. Researchers claims detection rate is 98.41% for day scenes and 95.68% for evening scenes and detector runs at 19 FPS. However, their research is focused on general vehicle detection without classification and for efficient functionality, the algorithm requires a camera to be placed in such a way that vehicle frontal view can be captured. This approach limits the usability of the system for traffic count systems.

Several other studies [52], [53], [54] propose systems for vehicle detection and classification using neural networks. Researchers state the proposed solutions have high classification accuracy. However, they are not suitable for real-time traffic count.

Chapter 3

Neural networks

Artificial neural networks are systems inspired by the human brain [55]. The basic computation unit in the brain is a neuron (see Fig. 3.1), which has input and output. The input is a dendritic tree connected to the outputs of other neurons called axons. Neurons operate in a single direction from the input to the output and their output is binary. Neurons are also basic computation elements of artificial neural networks. Similarly to biological neural networks, it can have several inputs and outputs. Every neuron can be described by function $f(\omega \cdot \mathbf{x} + b)$, where \mathbf{x} is the input, ω denotes weights, b is a bias and f is the activation function. There are several types of artificial neural networks that are commonly used in machine learning. The most popular type used in object detection is a convolutional neural network (CNN), which will be described in the following section.

Neuron

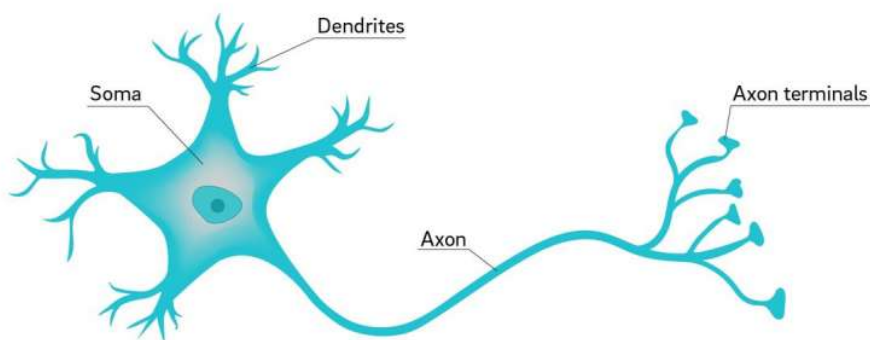


Figure 3.1: Neuron cell [6]

3.1 Architecture

All CNN models have a similar architecture shown in Fig. 3.2. The input of such neural network is an image. CNN consists of a series of convolution and pooling operations followed by fully connected layers. These operations are described in the next paragraphs.

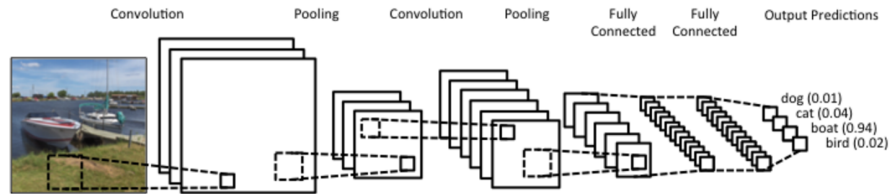


Figure 3.2: Convolutional neural network [7]

3.1.1 Convolutional layer

Convolutional layers consist of neurons placed in a grid of size $N \times M \times C$, where N, M denotes width and height of convolutional filter and C is number of channels in the previous layer (see Fig. 3.3). The filter moves from the left to the right with a certain stride until it completes processing width, then it moves down by the same stride to the beginning of the image and repeats the process until the whole image is traversed. The process computes convolution as shown in Fig. 3.4. Calculated feature map is usually smaller than the input, but it is possible to preserve the same dimensionality by using padding to surround the input with zeros.

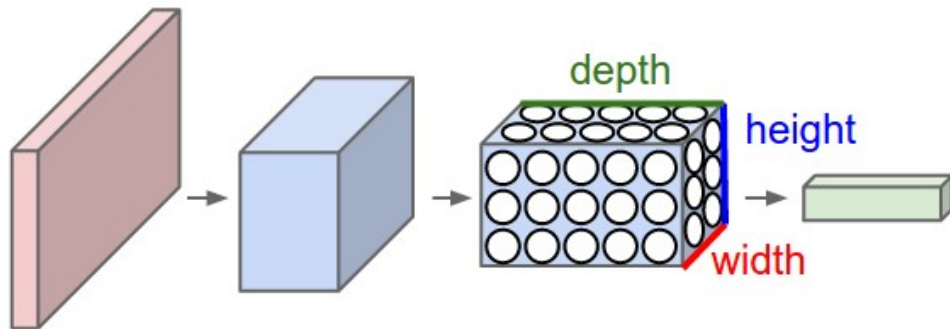


Figure 3.3: Convolutional layer [8]

3.1.2 Non-linearity layer

A non-linearity layer consists of an activation function that takes calculated feature map and creates the activation map as its output. The most common non-linearities used in CNN are sigmoid and ReLu [3.5].

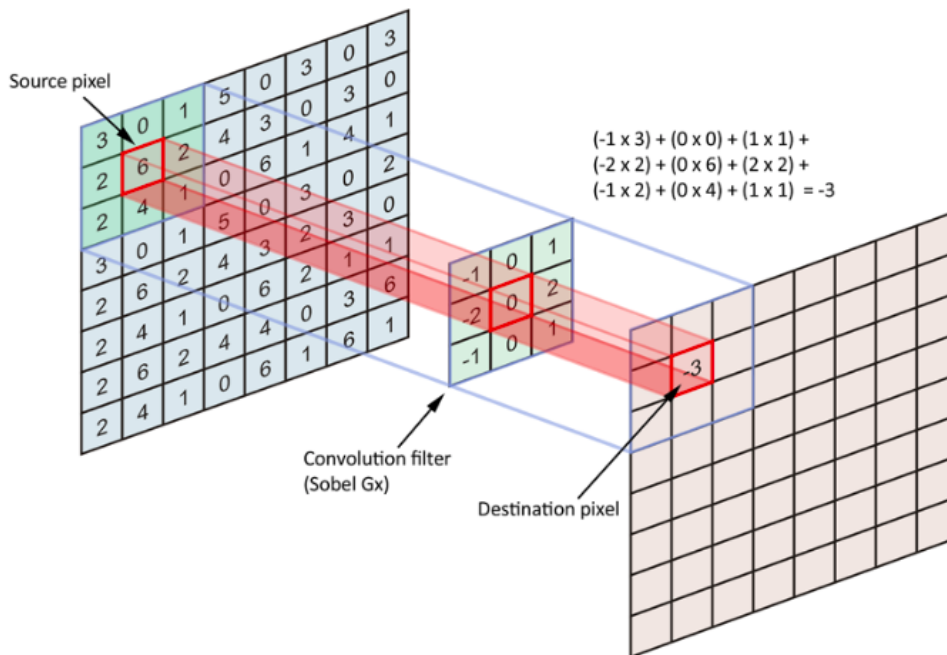


Figure 3.4: Convolution computation [8]

3.1.3 Pooling layer

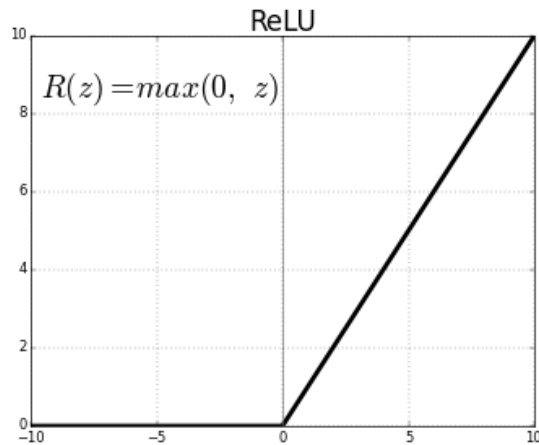
After convolution, pooling layer is used to reduce the dimensionality which enables to reduce the number of parameters. Two most common pooling operations are max and min pooling. It simply slides the input with particular stride and chooses maximal or minimal value in the predefined window (see Fig. 3.6). Pooling may reduce overfitting of the CNN and can reduce the training time [56].

3.1.4 Fully connected layer

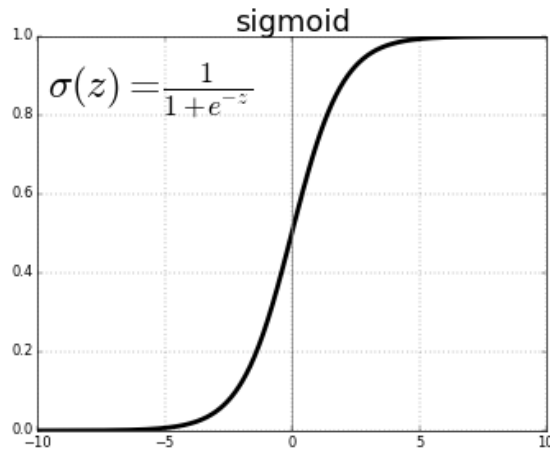
In fully connected layers, each neuron is connected to every neuron in the previous layer just like in feedforward neural networks.

3.2 Training of CNN

Before we can use any neural network, it must be trained to understand how objects we want it to recognize should look like. The weights of filters are randomized, and the filters in lower layers of CNN don't know to look for edges or curves, the filters in higher layers don't know to look for more concrete shapes like wheels, legs, faces. As any supervised learning, CNNs are given a training set of thousands of images with labels to learn features of objects. The learning algorithm is called backpropagation.



(a) : ReLu



(b) : sigmoid

Figure 3.5: Comparison of ReLu and sigmoid non-linearities

■ 3.2.1 Backpropagation

Backpropagation was firstly introduced in [57] in 1986. This process can be separated into four steps: the forward pass, the calculation of loss function, the backward pass and the weight update. During the forward pass, we take a batch of training images and pass it through the network. After the first training forward pass output of the network would probably be randomized, because the network isn't able to look for any type of features, and isn't able to make any reasonable conclusion about training example. This goes to the next step of backpropagation, the calculation of loss function. Each neural network can have its own loss function depending on what its output is, but the most common loss function used for backpropagation is mean squared error

$$L = \frac{1}{2n} \sum_x \|y(x) - a(x)\|^2, \quad (3.1)$$

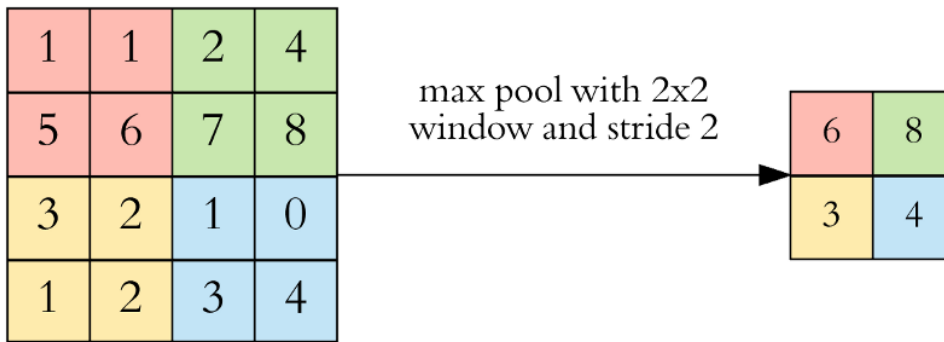


Figure 3.6: Pooling layer [8]

where n denotes the number of training inputs x , $y(x)$ label corresponding to the input x and a is the network's output. By training process, we want to achieve such a result where the predicted label is the same as the training label. To get this, we want to minimize the amount of loss we have, hence we want to find out which weights most directly contribute to the loss of the networks. This leads to the third step, backward pass, where we calculate partial derivations $\frac{\partial L}{\partial \omega}$, $\frac{\partial L}{\partial b}$. Once we compute the derivation, we can update weights and biases by changing them in the opposite direction of the gradient using learning rate η :

$$\omega = \omega - \eta \frac{\partial L}{\partial \omega}, \quad (3.2)$$

$$b = b - \eta \frac{\partial L}{\partial b}. \quad (3.3)$$

η is the parameter that defines how big steps learning process will take to update the weights, thus how fast we want the model to converge. However, the learning rate that is too big could result in jumps that are too large and not precise enough to reach the optimal point.

■ 3.2.2 Overfitting

Overfitting is one of the biggest problems in machine learning. Overfitting means that a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. To solve this problem in neural networks, we can use so-called dropout. During each training step, an individual neuron can be dropped out of the net with probability $1 - p$ or kept with probability p , so that only a reduced network is trained. The removed neurons are then reinserted into the network with unchanged weights. This methods not only decreases overfitting but also improves training speed [58].

■ 3.2.3 Batch normalization

Assume we have a training set of images with cars that has a particular colour. If we try to use the network that was trained on that dataset, it probably

will not work well on cars with another colour. In that case, we might need to retrain the network by trying to align the distribution of cars in different colours. Batch normalization helps with this problem by reducing the amount of covariance shift in hidden layers [59]. It simply normalizes the output of each layer by subtracting the batch mean and dividing by the batch standard deviation. After this change of activation output, the weights in the next layer are no longer optimal. Therefore, batch normalization adds two trainable parameters to each layer and lets gradient descent do the denormalization by changing only these parameters during each activation. Batch normalization also helps with overfitting, because it adds some noise to each layer's activations. It also allows using higher learning rate, because it makes sure no activation goes high or low.

Chapter 4

Network architectures used in the thesis

In previous chapters, we already mentioned some network architectures that are used for object detection in traffic systems. Here we will describe in detail four CNN architectures we used during experiments described in Chapter 5.

4.1 ResNet

Residual networks described in [9] are classification networks with an image as the input and object class and confidence score as the output. In this paper, they introduced shortcut connections that are widely used in modern neural networks. One of the biggest problems with training deep neural networks is vanishing and exploding gradient. During backpropagation, a lot of small or large numbers are multiplied to compute gradients. When the network is deep, multiplying of small numbers will become zero (vanished) and multiplying of large numbers will explode. Normally we expect deeper neural network will have more accurate predictions, but the opposite is true, and this degradation problem is caused by the vanishing gradient. This problem can be solved by adding shortcut connection which adds the input to the output after few weight layers; hence the output is $H(x) = F(x) + x$ (see Fig. 4.1). There are two types of residual connections. The identity shortcuts can be directly used when both input and output have the same dimension, or extra zero padding can be used when dimensions change. In both cases, no extra parameters are needed. Comparing plain and residual network with 34 layers (see Fig. 4.2) Top-1 error drops from 28.54% to 25.03%. On the other hand, if we compare a smaller network with 18 layers, Top-1 error changes from 27.94% to 27.88%, which means shortcut connections perform better in deeper networks. ResNets with a different number of layers are often used as classification networks (backbone) in detectors such as RetinaNet.

4.2 RetinaNet

RetinaNet was proposed by Facebook AI Research¹ and its features are described in [10] and [60]. They proposed using anchor boxes instead of pre-

¹<https://research.fb.com/category/facebook-ai-research/>

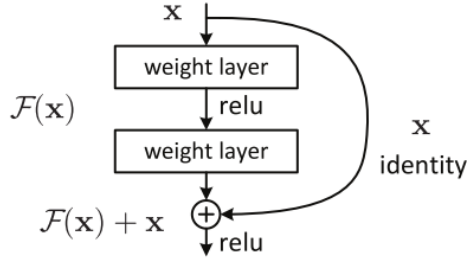


Figure 4.1: Residual block [9]

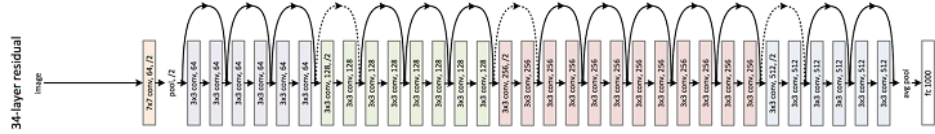


Figure 4.2: ResNet architecture [9]

dicting bounding boxes. Sizes of the anchor boxes are predefined and used in further predictions. Thus, the network does not predict the final size of the object, but instead, it only adjusts the size of the nearest anchor to the size of the object. Also, they suggested a solution for object detection in different scales. Originally a pyramid of the same image at different scales was used to detect the object. However, this solution is time-consuming and has a high memory demand. Instead, a pyramid of features can be used. Although it is not such efficient for accurate object detection as image pyramids, it provides result faster and with less memory consumption. In [10] authors propose Feature Pyramid Network (FPN) which is fast like the described pyramid of features, but more accurate. Its architecture is shown in Fig. 4.3. The other solution, focal loss, solves the class imbalance. Instead of normal cross entropy calculated by

$$C(p, y) = - \sum_i y_i \ln p_i, \quad (4.1)$$

scaled entropy is used using the following equation:

$$C(p, y) = - \sum_i y_i (1 - p_i)^\lambda \ln p_i. \quad (4.2)$$

Here we can see focusing parameter $\lambda \geq 0$ which smoothly adjusts the rate at which easy examples are down-weighted, and thus training is focused on hard negatives. In this thesis, we used Keras implementation of RetinaNet [61] implemented in TensorFlow with ResNet50 as a backbone.

4.3 SqueezeDet

SqueezeDet [12] is a single stage detection pipeline inspired by YOLO, which will be covered later in section 4.4. The main difference between two archi-

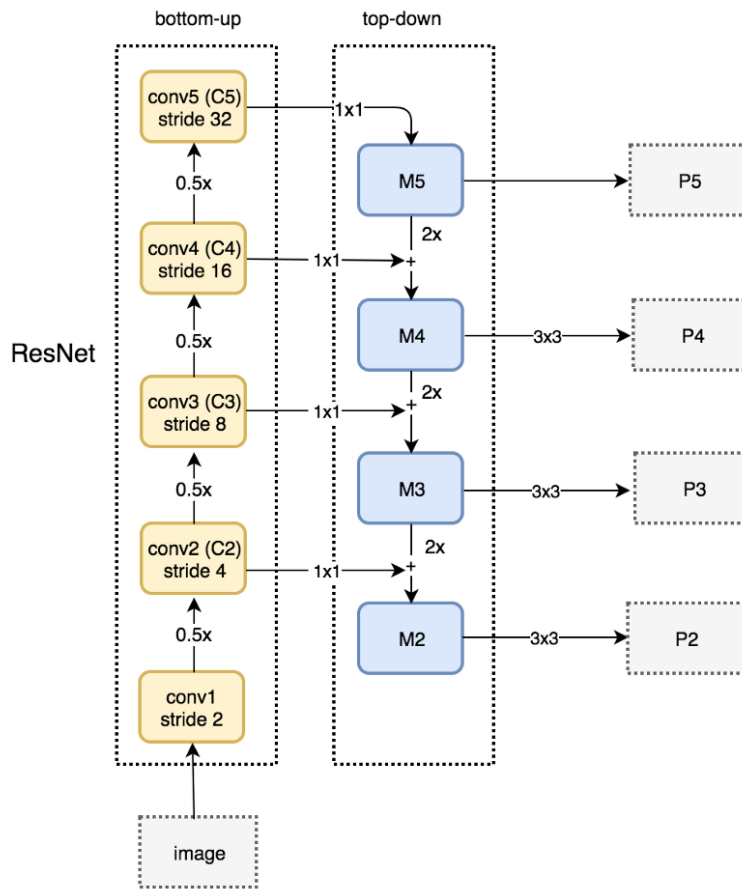


Figure 4.3: Feature Pyramid Network [10]

structures is that SqueezeDet uses SqueezeNet [11] for feature extraction.

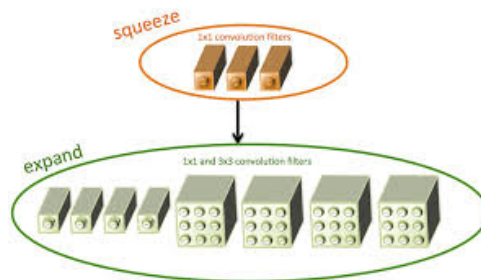


Figure 4.4: Fire module in SqueezeNet [11]

The building brick of SqueezeNet is called fire module [3]. Each fire module contains a squeeze layer and an expand layer. Squeeze layers replace 3×3 filters by 1×1 filters to reduce computation complexity 9 times. Following expand layers contain a number of 1×1 and 3×3 kernels. Squeeze layers reduce depth of calculated feature map, which means the following 3×3 filters in expand layers have to do less computation. Thanks to its architecture,

SqueezeDet can be faster and smaller compared to other state-of-art solutions (see Fig. 4.5), and so can be efficiently used on embedded system.

model	Model Size (MB)	FLOPs $\times 10^9$	Activation Memory Footprint (MB)	Average GPU Power (W)	Inference Speed (FPS)	Energy Efficiency (J/frame)	mAP*
SqueezeDet	7.9	9.7	117.0	80.9	57.2	1.4	76.7
SqueezeDet: scale-up	7.9	22.5	263.3	89.9	31.3	2.9	72.4
SqueezeDet: scale-down	7.9	5.3	65.8	77.8	92.5	0.84	73.2
SqueezeDet: 16 anchors	9.4	11.0	117.4	82.9	51.4	1.6	66.9
SqueezeDet+	26.8	77.2	252.7	128.3	32.1	4.0	80.4
VGG16+ConvDet	57.4	288.4	540.4	153.9	16.6	9.3	79.1
ResNet50+ConvDet	35.1	61.3	369.0	95.4	22.5	4.2	76.1
Faster-RCNN + VGG16 [1]	485	-	-	200.1	1.7	117.7	-
Faster-RCNN + AlexNet [1]	240	-	-	143.1	2.9	49.3	-
YOLO**	753	-	-	187.3	25.8	7.3	-

Figure 4.5: SqueezeDet comparison with other state-of-art solutions [12]

The loss function of SqueezeDet is defined as

$$\begin{aligned}
& \frac{\lambda_{bbox}}{N_{obj}} \sum_{i=1}^W \sum_{j=1}^H \sum_{k=1}^K I_{ijk} \left[\left(\beta x_{ijk} - \beta x_{ijk}^G \right)^2 + \left(\beta y_{ijk} - \beta y_{ijk}^G \right)^2 \right. \\
& \quad \left. + \left(\beta w_{ijk} - \beta w_{ijk}^G \right)^2 + \left(\beta h_{ijk} - \beta h_{ijk}^G \right)^2 \right] \\
& + \sum_{i=1}^W \sum_{j=1}^H \sum_{k=1}^K \frac{\lambda_{conf}^+}{N_{obj}} I_{ijk} \left(\gamma_{ijk} - \gamma_{ijk}^G \right)^2 + \frac{\lambda_{conf}^-}{WHK - N_{obj}} \bar{I}_{ijk} \gamma_{ijk}^2 \\
& + \frac{1}{N_{obj}} \sum_{i=1}^W \sum_{j=1}^H \sum_{k=1}^K \sum_{c=1}^C I_{ijk} l_c^G \log(p_c), \quad (4.3)
\end{aligned}$$

where the first part is the bounding box regression and $(\beta x_{ijk}, \beta y_{ijk}, \beta w_{ijk}, \beta h_{ijk})$ corresponds to the relative coordinate of anchor- k located at grid center- (i, j) . Second part denotes confidence score regression with output γ_{ijk} . The last part is a cross-entropy loss for classification.

We used a tensorflow implementation of SqueezeDet available to download from GitHub [59].

4.4 YOLO

4.4.1 YOLOv1

Another approach for object detection, YOLO architecture, was presented in [13]. A single neural network is used to predict both bounding boxes and class probabilities, hence an image is evaluated only once. The described system divides the input into an $S \times S$ grid, and if the center of an object falls into a grid cell, this cell is responsible for detecting that object. Each cell also predicts B bounding boxes and confidence score for them. Confidence is defined as

$$score = Pr(Object) \cdot IoU_{pred}^{truth}, \quad (4.4)$$

where $Pr(Object)$ is a probability of an object being inside that bounding box and IoU_{pred}^{truth} denotes intersection over union between ground truth and prediction. Each bounding box consists of $(x, y, w, h, score)$, where (x, y) represents the center of the box and (w, h) denotes its width and height. Each grid cells also predicts conditional probability $C = Pr(Class_i|Object)$. The model consists of 24 convolutional layers followed by two fully connected layers as it shows in 4.6.

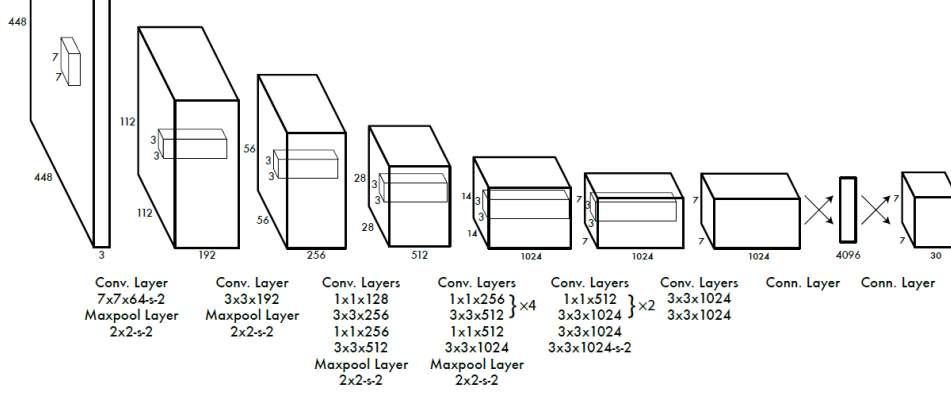


Figure 4.6: YOLOv1 architecture [13]

Training process optimizes the loss function

$$\begin{aligned}
\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{obj} & \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{obj} & \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
+ \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{obj} & (C_i - \hat{C}_i)^2 \\
+ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{noobj} & (C_i - \hat{C}_i)^2 \\
+ \sum_{i=0}^{S^2} \mathbf{1}_{ij}^{obj} \sum_{c \in classes} & (p_i(c) - \hat{p}_i(c))^2, \quad (4.5)
\end{aligned}$$

where

$$\mathbf{1}_{ij}^{obj} = \begin{cases} 1, & \text{if there is an object.} \\ 0, & \text{otherwise,} \end{cases} \quad (4.6)$$

$\mathbf{1}_{ij}^{noobj}$ is inverse function to $\mathbf{1}_{ij}^{obj}$, λ_{coord} and λ_{noobj} are constant to increase the loss from bounding box coordinate prediction and decrease the loss from confidence prediction for boxes that do not contain objects. While YOLOv1 was faster than most of the existing approaches for object detection, it had relatively low 57.9% mAP on the VOC 2012 test set compared to the existing state of the art.

4.4.2 YOLOv2

The improved model YOLOv2 was introduced in [14]. Authors of this state of the art detector refers to it as a better, faster and stronger version of YOLO. For better performance, they added batch normalization and used images with a bigger resolution to train the network. They also removed fully connected layers and used anchor boxes to predict bounding boxes, which lead to a small decrease in mAP from 69.5% to 69.2%, but it also increased a recall from 81% to 88%. We can see how applied changes improved network performance in 4.7.

	YOLO								YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓				
new network?					✓	✓	✓	✓	✓
dimension priors?						✓	✓	✓	✓
location prediction?						✓	✓	✓	✓
passthrough?							✓	✓	✓
multi-scale?								✓	✓
hi-res detector?									✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	78.6

Figure 4.7: YOLOv2 improvement [14]

They also proposed a new classification network called Darknet-19 (see Appendix D) to make YOLO even faster. We can see that Darknet-19 has many 1×1 convolutions to reduce the number of parameters.

4.4.3 YOLOv3

The newest version of YOLO was presented in [15]. Similar to YOLOv2 it predicts bounding boxes using dimension clusters as anchor boxes. The network predicts four coordinates for each bounding box and for training they use a sum of squared error loss. Objectness score for each bounding box is predicted using logistic regression, which should be one if the bounding box prior overlaps a ground truth object by more than any other bounding box prior. They also use three different scales for prediction, which is similar to feature pyramid networks. Deeper extractor called Darknet-53 (see Appendix E) with shortcut connections is used for feature extraction.

Comparing to another state of art solutions, YOLOv3 has similar performance, but it is much faster as shown in Fig. 4.8. Unlike RetinaNet and SqueezeDet, YOLO uses another neural network framework, Darknet¹, written in C and CUDA.

¹<https://pjreddie.com/darknet/>

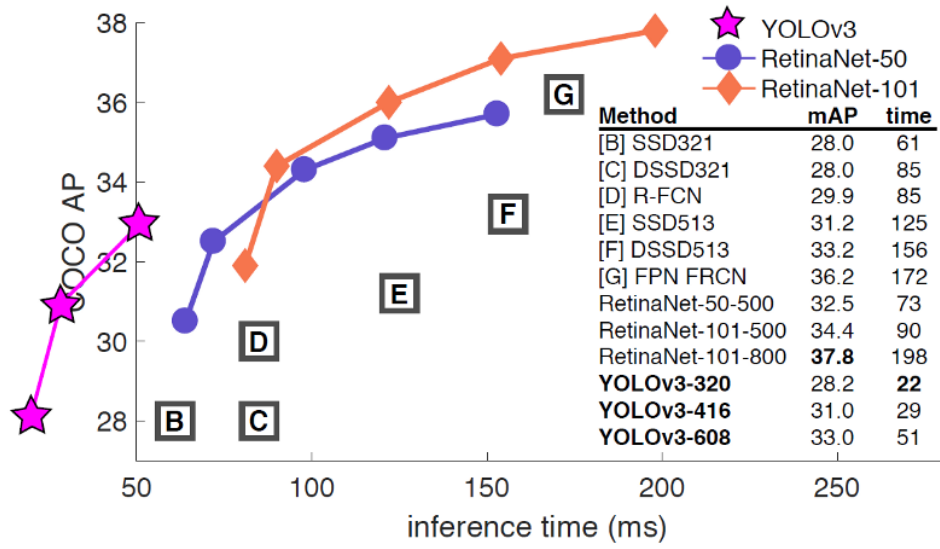


Figure 4.8: YOLOv3 comparison [15]

Chapter 5

Experiments

For experiments, we chose YOLOv3, SqueezeDet and RetinaNet (both with ResNet as a backbone). We trained all three networks on KITTI¹ and internal GoodVision dataset called “GV-2018” that was specifically prepared for our use case. Thanks to the ability of RetinaNet and YOLOv3 to adapt to the size of the input image, we were able to test different image size ratios with no need to retrain these networks. YOLOv3 was evaluated with two different input image resolutions: 608×608 and 418×418 ; for evaluation of RetinaNet, we used three different largest side sizes, 1024, 608 and 418, which means input image will be resized so that its largest side has given size. Unfortunately, SqueezeDet does not adapt its layers to an input image, so we used a network with input image resolution 1242×375 defined by the pre-trained model.

5.1 Datasets

5.1.1 KITTI dataset

KITTI dataset consists of 7481 training images and 7518 test images with a total of 80256 labeled objects. Original dataset has nine different type of objects:

1. “Car”
2. “Van”
3. “Truck”
4. “Pedestrian”
5. “Person_sitting”
6. “Cyclist”
7. “Tram”
8. “Misc”

¹<http://www.cvlibs.net/datasets/kitti/>

9. “DontCare”.

We used only 5 of them for training: “Car”, “Van”, “Truck”, “Pedestrian”, “Person_sitting”, though we merged “Person_sitting” and “Pedestrian” classes into a single “Person” object type. Regardless of the lack of common classes like “Bus”, “Bicycle”, “Motorcycle”, it fits our needs, because all data were gathered by driving around cities, in country areas, and on highways, and so out networks would be trained on real-life traffic data with no redundant information [5.1](#).



Figure 5.1: Image samples from KITTI dataset

5.1.2 “GV-2018”

“GV-2018” was specifically created for object detection and recognition in traffic. It consists of 4917 images with more than 130000 labeled objects. Unlike in KITTI dataset, such classes as “Bicycle”, “Bus”, “Motorcycle” are presented here, which means neural networks trained on this dataset will be more comprehensive and more adequate for our use case. Images for this dataset were gathered from cameras placed at different heights and angles to roads and highways. Samples from the dataset are presented in [5.2](#).

Both datasets are divided into train dataset (data used for training), validation dataset (data used to estimate validation loss during training) and test dataset (data used for evaluation of trained network).



Figure 5.2: Image samples from “GV-2018”

5.2 Performance evaluation

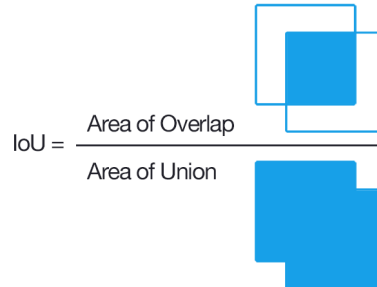
For our evaluation, we calculated average precision for every class in the test set. We had to calculate precision and recall, which are defined as

$$Precision = \frac{TP}{TP + FP}, \quad (5.1)$$

$$Recall = \frac{TP}{TP + FN}, \quad (5.2)$$

where TP is true positives, FP is false positives, and FN is false negatives. It means precision measures how accurate predictions are, while recall refers to the percentage of total relevant results correctly classified by our network. To determine if the prediction is a true positive or a false positive, intersection over union (IoU) has to be measured. As Figure 5.3 shows, IoU is simply the ratio between the area of overlap between the ground truth bounding box and predicted bounding box, and the area encompassed by both ground truth bounding box and predicted bounding box. If IoU is over some predefined threshold, the prediction is considered to be true. Otherwise, it is a false positive. For evaluation, we chose this threshold to be 0.5.

Then we calculate precision/recall curve [62], and average precision is the area under that curve and is calculated for every class independently. Mean average precision is simply average value of average precisions across all classes. Results of the evaluation are presented in table 5.6. YOLOv3



$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

Figure 5.3: Intersection over Union [16]

performs better than both SqueezeDet and RetinaNet at both resolution. Although YOLOv3 trained on KITTI dataset indicates bigger mAP on KITTI test set, it has worse performance on GoodVision test dataset, while YOLOv3 trained on “GV-2018” has almost 10% better mAP. We can notice this trend for all 3 CNNs: while KITTI-trained model has better performance on KITTI test dataset, it fails on GoodVision dataset. If we compare models trained on KITTI and GoodVision datasets using AP of classes presented in KITTI dataset only, the difference will be even bigger.

5.3 Inference time evaluation

For computing on the edge, we should ensure that time delay between receiving input data (stream frame) and providing output data to a user is as small as possible. Time of object detection and classification can be the biggest bottleneck in such systems. Therefore it is necessary to choose such architecture that provides the best result with the least possible inference time. Videos for this evaluation were taken from video databases with free access such as YouTube¹ or Pexels². All videos have different camera view with various object count, video resolution and FPS, as shown in table 5.1. This can affect neural network detector performance. Hence we should also compare various input resolutions of neural networks. Video samples presented in Appendix F.1 shows noticeable difference between videos scenes. Those scenes represent the typical use case for our system.

We compared the time necessary to process one image frame by CNN using Jetson Xavier and other two different PCs with CPU and GPU specifications presented in tables 5.2 and 5.3.

As we can see from table 5.4, PC1 performs best for all three CNNs. On the other hand, PC2 has worse inference time comparing to Jetson Xavier

¹<https://www.youtube.com/>

²<https://www.pexels.com/videos/>

Video file	FPS	Video Resolution
5.4 4K Camera Road in Thailand.mp4	30	1280x720
Cars Driving On Street.mp4	30	1920x1080
Cars On Highway.mp4	25	1920x1080
Cars On The Road.mp4	50	1280x720
City Traffic.mp4	30	1920x1088
Day Traffic Sample Video Dataset.mp4	30	432x240
Pedestrian and Traffic, Human Activity Recognition Video ,DataSet By UET Peshawar.mp4	30	1280x720
Pexels Videos 1601538.mp4	25	1920x1080
Pexels Videos 2577.mp4	30	1920x1088
Pexels Videos 2670.mp4	25	1920x1088
Pexels Videos 3047.mp4	30	1920x1088
Pexels Videos 948404.mp4	24	3840x2178
moderate_traffic.mp4	30	1280x720

Table 5.1: Characteristics of video for inference time evaluation

CPU	Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
GPU	3584-core 11Gb GeForce GTX 1080 Ti @ 1582MHz

Table 5.2: PC1 technical specification

using YOLOv3 and RetinaNet and almost the same while using SqueezeDet. It is worth mentioning that YOLOv3 has similar inference time for both tested resolutions and SqueezeDet has the best inference time among all tested CNNs.

5.3.1 Mixed precision calculation

We have discussed in section [L.3.3](#) that Nvidia Jetson has tensor cores to accelerate matrix calculation using mixed precision matrix multiplication and accumulation. Darknet framework used for YOLOv3 can activate tensor cores directly, because it is written in CUDA. To utilize tensor core using TensorFlow we should use TensorRT platform¹. TensorRT is able to convert TensorFlow CNN graph into the supported format and use it for inference with mixed precision calculation. We achieved a significant decrease in the YOLOv3 inference time, but in RetinaNet the difference between measured inference time using plain TensorFlow and inference time using TensorRT was small, which is also described in table [5.5](#). Insignificant improvement in inference FPS is caused by the fact that TensorRT supports only some layers defined in TensorFlow. We used RetinaNet implemented in Keras and some of its layers are custom and not supported in TensorRT. This problem can be solved by reimplementing RetinaNet in TensorFlow or add support for these layers into TensorRT. However, we have already achieved sufficiently good results with YOLOv3 in both detector precision and its inference time, and we came to a decision that any future work on RetinaNet and its improvement using TensorRT is unnecessary.

¹<https://developer.nvidia.com/tensorrt>

CPU	Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
GPU	640-core 4Gb GeForce GTX 1050 @ 1404MHz

Table 5.3: PC2 technical specification

Model	Machine	Inference time [ms]
YOLOv3 416x416	Jetson Xavier	123
	PC1	35
	PC2	175
YOLOv3 608x608	Jetson Xavier	139
	PC1	39
	PC2	208
SqueezeDet	Jetson Xavier	25
	PC1	24
	PC2	8
RetinaNet 1024	Jetson Xavier	210
	PC1	52
	PC2	228
RetinaNet 608	Jetson Xavier	107
	PC1	28
	PC2	100
RetinaNet 416	Jetson Xavier	74
	PC1	19
	PC2	65

Table 5.4: Inference time evaluation

Model	Original inference time [ms]	Mixed-precision inference time [ms]
YOLOv3 416x416	123	49
YOLOv3 608x608	139	91
RetinaNet 1024	210	186
RetinaNet 608	107	95
RetinaNet 416	74	67

Table 5.5: Inference time using mixed precision calculation

Model	Training dataset	Testing dataset	Bicycle AP [%]	Bus AP [%]	Motorcycle AP [%]	Car AP [%]	Person AP [%]	Truck AP [%]	Van AP [%]	mAP [%]
YOLOv3 608	"GV-2018"	"GV-2018"	62,5	72,4	54,4	89,3	91,2	83,2	80,1	76,2
	KITTI	KITTI	-	-	-	81,4	89,3	80,6	73,5	81,2
YOLOv3 416	"GV-2018"	"GV-2018"	63,6	61,2	40,7	83,4	84,0	72,7	65,2	67,3
	KITTI	KITTI	-	-	-	73,6	80,7	68,3	59,5	70,5
RetinaNet 1024	"GV-2018"	"GV-2018"	46,3	34,8	15,4	52,1	63,5	62,2	43,0	55,2
	KITTI	KITTI	-	-	-	39,5	32,3	53,8	40,6	37,5
RetinaNet 608	"GV-2018"	"GV-2018"	11,0	12,7	0,2	35,2	29,9	51,6	31,1	37,0
	KITTI	KITTI	-	-	-	20,6	23,3	42,2	19,4	26,4
RetinaNet 416	"GV-2018"	"GV-2018"	-	-	-	25,5	18,0	28,3	23,4	17,0
	KITTI	KITTI	-	-	-	23,0	16,6	27,8	20,3	21,9
SqueezeDet	"GV-2018"	"GV-2018"	0,0	5,8	0,6	2,3	16,3	24,6	18,1	15,3
	KITTI	KITTI	-	-	-	19,5	12,9	15,8	16,7	10,2
SqueezeDet	"GV-2018"	"GV-2018"	1,7	8,2	4,1	19,1	12,3	14,6	15,1	15,3
	KITTI	KITTI	-	-	-	13,4	7,8	10,6	11,2	10,8
SqueezeDet	"GV-2018"	"GV-2018"	-	-	-	34,4	39,2	22,9	12,9	17,6
	KITTI	KITTI	-	-	-	30,6	38,5	22,5	10,8	25,6
SqueezeDet	"GV-2018"	"GV-2018"	-	-	-	3,1	2,9	4,7	4,2	3,7
	KITTI	KITTI	-	-	-	-	-	-	-	-

Table 5.6: CNN evaluation

Chapter 6

Implementation

Experiments proved YOLOv3 trained on “GV-2018” dataset to be the most suitable network architecture among all tested CNNs. In this chapter, we will describe how detector with YOLOv3 CNN is implemented and how it communicates with the user. We will also describe what prerequisites are needed for Darknet framework and which environment is used to run our system.

6.1 Docker

Docker¹ containers are used for encapsulation of an application with its dependencies. Like virtual machines, a container holds an isolated instance of an operation system that can be used to run applications. Figure 6.1 shows the architecture differences between VMs and containers. Containers are more lightweight, as they include only the executables and its dependencies and share the same operation system as a host machine [17]. Additionally, several containers can share the same image, while each VM has its own image file [63]. The portability of containers also help with software distribution: once the container is created, it can be used on different machines with no additional settings. They also provide isolation of our application from settings on our machine, which is extremely useful during developing.

NVIDIA offers “nvidia-docker” [64] plugin to enable GPUs inside docker containers. However, this solution does not support Tegra platforms [65] as Jetson Xavier. Fortunately, JetPack, which is discussed in Section 6.2, has Docker support built into the kernel since version 3.2 [66]. However, it still does not support GPU mapping into docker containers, which should be done manually when starting the container. Also some libraries, such as CUDA, can only be installed on Jetson Xavier via JetPack. Hence it is not possible to install them inside a docker container and they should be mapped as well when the container is started. To make this process easier, we use the bash script from Technica Corporation [67] that overwrite process of starting docker container. Since CUDA is available only inside running container, it is not possible to build a docker image with installed prerequisites

¹<https://www.docker.com/>

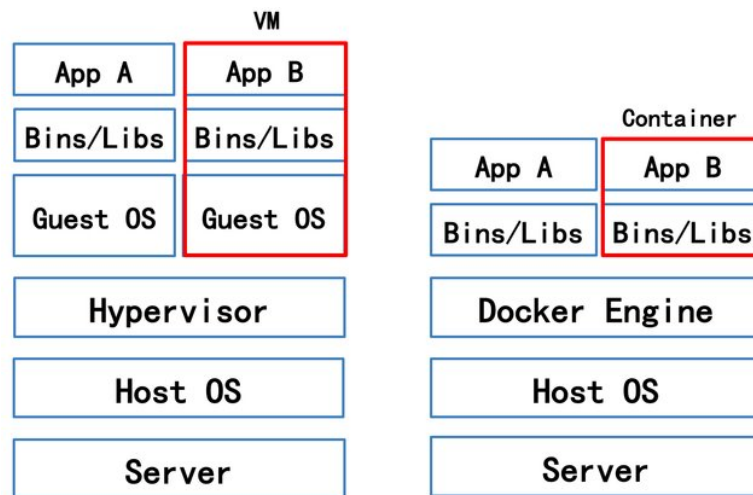


Figure 6.1: Architecture comparison virtual machine vs. container [17]

in a standard way using Dockerfile. Instead, we can install it inside empty running docker container that will be used to create a new docker image.

After the docker image is created, it can be pushed to docker repository and used on any other Jetson Xavier.

6.2 Prerequisites installation

Before we can use Jetson Xavier, we should flash (install base firmware) it first. NVIDIA offers JetPack¹, which is used to flash Jetson with the latest available OS image, which is, in our case Ubuntu 18.04, and installs the libraries needed for building AI applications. JetPack includes the following libraries [68]:

1. TensorRT and cuDNN for increasing performance of deep learning applications,
2. CUDA for GPU calculations,
3. VisionWorks and OpenCV for visual computing.

Since we use docker environment, we do not need to install OpenCV on Jetson directly. It is available online [69] and can be easily installed without JetPack [70] inside docker container.

On the other hand, CUDA 10.0 for Jetson modules is not available for download and can be installed on Jetson via JetPack only, which is also the reason why we have to map CUDA folders into the docker container, as mentioned in Section 6.1.

After Jetson is flashed, we can prepare the docker container. As a base image we used arm64v8/ubuntu:18.04 [71]. We run the container with GPU mapping and CUDA folders mapping using bash script “jetson-docker”

¹<https://developer.nvidia.com/embedded/jetpack>

Path	Operation	Description
/detector	GET	Return detector status
	POST	Start detector
	DELETE	Stop running detector
/detector/detection_image	GET	Return grabbed frame with detections
/detector/detection_stream	GET	Stream frames with drawn detections

Table 6.1: API definition

available on attached CD (see Appendix C). Inside the container, we installed OpenCV 3.2, which is necessary for working with images and video streams, and Protocol Buffer 3.5 [72], a library for serializing structured data, which is used for saving and loading CNN models.

We have said in Chapter 5 that YOLOv3 has proved to be efficient for our use case, hence we have to install Darknet framework to work with YOLO. Darknet is available on GitHub [73]. For our application, we built Darknet with GPU and CUDNN_HALF flags to accelerate inference by using Tensor Cores in Jetson Xavier GPU and with OPENCV flag to enable detecting on videos and video streams.

After all prerequisites were installed, we were able to implement the detector with YOLO architecture.

6.3 API

We implemented a service for communication with Jetson Xavier module and detector in Python3 using Flask framework¹. Endpoints of the service are defined in table 6.1 and their documentation can be found in the Appendix B. Documentation of the API endpoints was created using Swagger. The API runs directly on Jetson Xavier.

The detector can be started via API by sending a POST request to “/detector” endpoint with stream URL and configuration json file attached in form data. The configuration file should contain mandatory parameters for detector and can content optional parameters, which are described in section 6.4. After detector is started, it is possible to get an image with drawn detections by sending GET to request to either “/detector/detection_image” or “/detector/detection_stream” endpoint. The first one returns a single last processed frame, the second one continuously receives frames from the detector and streams them to user with 1 FPS frame rate. Communication between API and detector is ensured via named-pipes, which are a special type of FIFO file that is stored in the local file system. When API needs data from the detector, it sends a request to detector through one pipe, and detector sends encoded data back to API through another pipe. API decrypts received data and sends them to user. Named-pipes content resides in memory rather than being written to disc, which makes this type of communication fast enough

¹<http://flask.pocoo.org/>

even for transmitting big blocks of data, as images can be.

We also wanted to be sure only authorised users would be able to use API, so we secured communication with API with HTTP access authentication. Flask has several standard authentication methods:

1. Basic authentication access

This method requires user to provide a user name and password when making a request. Credentials are encoded in the base64 encoding, which is a process of converting binary data to an ASCII string format by converting that binary data into a 6-bit character representation. Encoded credentials are sent to the server, where they are compared with data stored hashed in database.

2. Digest authentication access

This method is similar to the previous one, but a hash function to the user-name and password is applied before sending them over the network, which makes this method more secure.

3. Token authentication access

This method generates a unique token for logged-in users. The token is then used to access protected pages instead of the credentials, which makes application more secure, because users do not enter their credentials for every page access. Token authentication should be combined with other authentication method to verify that the login information is correct.

For the purpose of our prototype, we used HTTP basic authentication, which can be replaced with another authentication method in future work.

6.4 YOLOv3 detector

Our detector application is divided into three parts: data provider, which grabs images from stream, detector itself, which does inference, and the main application, which handles communication between detector and data provider and sends data to API described in section [6.3](#). Implementation summary is presented in Fig. [6.2](#)

6.4.1 Data provider

This part of the application is responsible for providing data from the video stream. Using OpneCV VideoCapture class, it captures frames from video stream, which can be a video file, RTSP stream or stream from a camera connected directly to Jetson Xavier via USB or NVCSI port.

Data provider runs asynchronously, and stores captured frame into class variable. If the connection is lost, the data provider tries to reconnect, and if

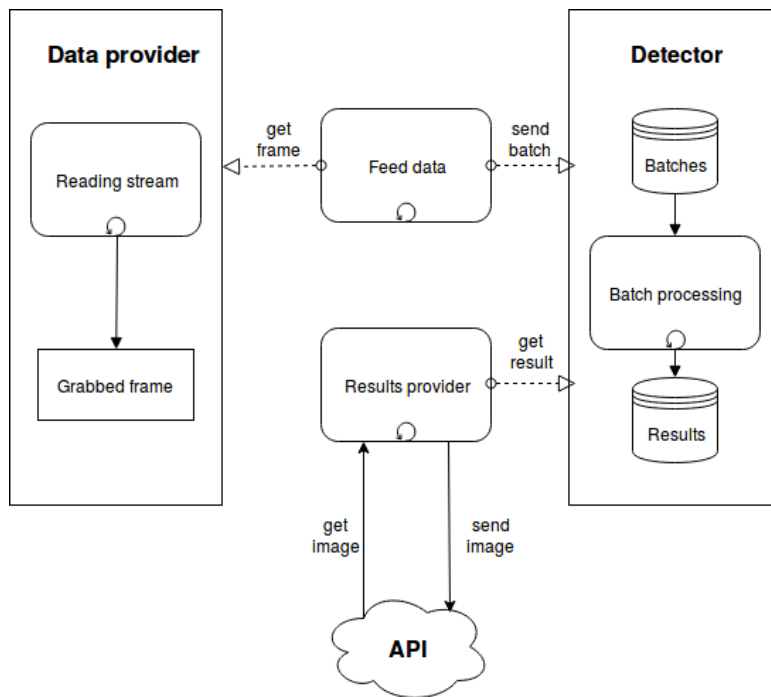


Figure 6.2: Application implementation

even after the predefined number of attempts connection is not restored, it stops.

Mutual exclusion (mutex) is used to avoid race condition when an image is stored to class variable and accessed from other threads.

6.4.2 Detector

The detector class is responsible for inference using YOLOv3 architecture.

The detector should be initialized using method `DarknetInit`, which loads the network's weights, its configuration parameters and label map. After that detector can be started and it will be running asynchronously. Detector thread gets data from detector queue, builds tensor and does inference. Outputs of the inference are bounding boxes with corresponding label and prediction score. Results are pushed into the result queue, which has limited size, and if the result queue is full, the earliest result is overwritten. If detector queue is empty, thread waits for data, and it can be stopped by user only.

The inference is done by functions from Darknet library that are wrapped in Detector class for easier use.

Again mutexes are used to prevent race condition when detector or result queue is accessed by detector thread and main application.


6.4.3 Main application

The main application does two things: it provides communication between data provider and detector and sends data to API when required.

First thread simply gets an image from the data provider and adds it to the detector queue. When the queue is full, the thread waits, and it stops if the data provider finishes getting data from the video stream.

The second thread waits until API asks for an image with detections. When the signal is received via the named pipe, this thread gets the first available result from the detector's result queue, encrypts it to base64 and sends it back to API.

The user starts application from API. The user sends configuration file with optional parameters "frame_skip", "batch_size" and "output_path". The frame skip parameter defines how many frames will be skipped before grabbed frame is stored in data provider, default value is 0. The batch size specifies how many frames are used for single detector run, default value is 1. The output path sets the name of FIFO pipe used for communication between API and detector, default value is "/tmp/output".



Chapter 7

Conclusions

In this work, we compared different methods for traffic count and management. Also, we explored available both commercial and non-commercial solutions for traffic monitoring system and found out how machine learning algorithms, and more precisely deep neural network, can be used in such tools and how edge computing can improve the usability of these systems.

In Chapter [3](#) we described how deep convolutional neural networks works and we discussed popular CNN architecture like ResNet, RetinaNet, YOLO, and SqueezeDet in Chapter [4](#).

Then, in Chapter [5](#), we made a comparison between described networks. We trained RetinaNet, YOLO and SqueezeDet using two datasets, publicly available KITTI and GoodVision internal “GV-2018”. Experiments proved YOLO to be the most suitable network architecture for us. We achieved 81.2% mAP with KITTI dataset and 76.2% mAP with “GV-2018” dataset with 608×608 input resolution, and 70.5% mAP with KITTI and 67.3% mAP with “GV-2018” using 416×416 input resolution. YOLO also proved to be the second fastest tested architecture with SqueezeDet being the fastest. However, SqueezeDet showed poor results during performance evaluation. We tried inference speedup utilizing Tensor Cores in Jetson Xavier GPU and achieved a 20 FPS rate with YOLO, which we consider to be sufficient for real-time traffic monitoring.

Finally, in Chapter [6](#) we proposed a prototype of a traffic monitoring application. This prototype consists of two parts: detector and API. Detector wraps Darknet framework used for YOLO and provides detections and classifications, while API is used to communicate with Jetson Xavier. Through API user can start or stop detector and obtain results from it. The whole application runs in the docker container, which is specifically prepared for using on Jetson Xavier.

As a result, we have a fully functional end-to-end solution for people and vehicles detection using YOLOv3 CNN architecture that runs on Jetson Xavier, embedded system specifically designed for AI inference on the edge.

■ 7.1 Feature work

In future work, we want to add tracking capabilities to make line and zone crossing counts possible. We would also like to extend interaction with users. They should be able to define lines and zones they want to monitor on the fly with no need to restart application. In future releases, we will enable access to data gathered in the past, so users will be able to study the difference between previous and actual traffic situation in the monitored area. For this purposes, we will design database to store information.

In the future, proposed system can be used for various traffic monitoring tasks, as well as for vehicle-to-everything communication and city traffic management, because it will be able to evaluate complex traffic situation in real-time.



Appendix A

Acronyms

AP	Average Precision
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
FIFO	First In First Out
FPS	Frames Per Second
GPU	Graphics Processing Unit
HOG	Histogram of Oriented Gradients
HTTP	Hypertext Transfer Protocol
mAP	Mean Average Precision
RTSP	Real Time Streaming Protocol
SSD	Single-Shot detector
SVM	Support Vector Machine
TFLOP	Trillion Floating-Point Operations Per Second
TOPS	Trillion Operations Per Second
UAV	Unmanned Aerial Vehicle
URL	Uniform Resource Locator

Appendix B

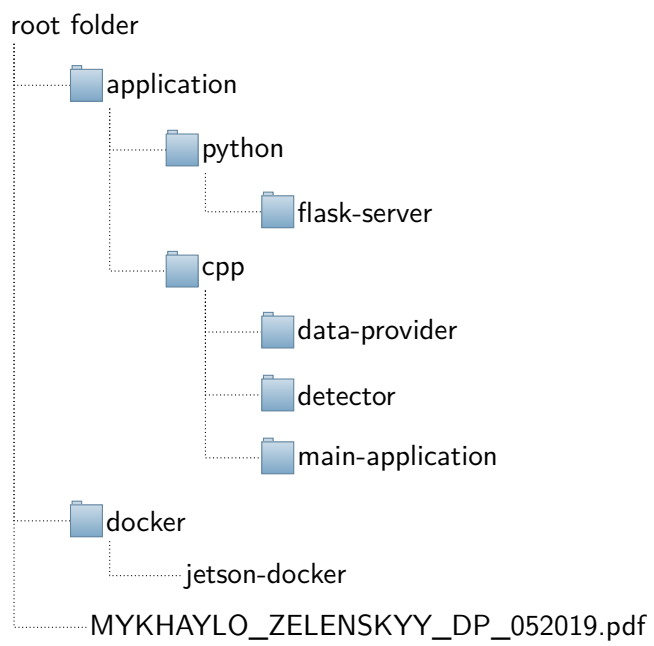
API definition

```
1  swagger: '2.0'
2  info:
3    version: 1.0.0
4    title: Jetson Xavier Object Detector API
5    contact:
6      email: zelenmyk@fel.cvut.cz
7
8  paths:
9    /detector:
10     get:
11       description: Return detector status.
12       produces:
13         - application/json
14       responses:
15         200:
16           description: Detector status.
17     post:
18       consumes:
19         - multipart/form-data
20       parameters:
21         - name: StreamURL
22           in: formData
23           type: string
24           description: URL of~RTSP stream
25         - name: ConfigurationFile
26           in: formData
27           type: file
28           description: Configuration file in~JSON format
29       description: Starts detector.
30       responses:
31         200:
32           description: Detector successfully started.
33         400:
34           description: Detector already running.
```

```
35 delete:
36   description: Stops running detector.
37   responses:
38     200:
39       description: Detector stopped.
40     400:
41       description: Detector not running.
42
43 /detector/detection\_image:
44   get:
45     description: Return latest image with predictions
46     ↪ from~detector
47     produces:
48       - image/png
49       - image/jpg
49     responses:
50       200:
51         description: Image returned.
52       400:
53         description: Detector not running.
54 /detector/detection\_stream:
55   get:
56     description: Stream latest image with predictions
57     ↪ from~detector with 1FPS rate
57     produces:
58       - image/png
59       - image/jpg
60     responses:
61       200:
62         description: Stream started.
63       400:
64         description: Detector not running.
```

Appendix C

CD



Appendix D

Darknet-19

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Figure D.1: Darknet-19 [14]

Appendix E

Darknet-53

	Type	Filters	Size	Output
	Convolutional	32	3×3	256×256
	Convolutional	64	$3 \times 3 / 2$	128×128
1×	Convolutional	32	1×1	
	Convolutional	64	3×3	
	Residual			128×128
	Convolutional	128	$3 \times 3 / 2$	64×64
2×	Convolutional	64	1×1	
	Convolutional	128	3×3	
	Residual			64×64
	Convolutional	256	$3 \times 3 / 2$	32×32
8×	Convolutional	128	1×1	
	Convolutional	256	3×3	
	Residual			32×32
	Convolutional	512	$3 \times 3 / 2$	16×16
8×	Convolutional	256	1×1	
	Convolutional	512	3×3	
	Residual			16×16
	Convolutional	1024	$3 \times 3 / 2$	8×8
4×	Convolutional	512	1×1	
	Convolutional	1024	3×3	
	Residual			8×8
	Avgpool		Global	
	Connected		1000	
	Softmax			

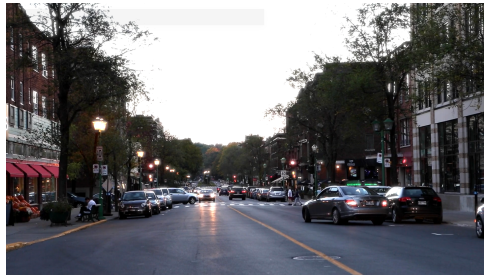
Figure E.1: Darknet-53 [15]

Appendix F

Sample from videos used for evaluation



(a) : 5.4 4K Camera Road in Thailand.mp4



(b) : Cars Driving On Street.mp4



(c) : Cars On Highway.mp4



(d) : Cars On The Road.mp4



(e) : City Traffic.mp4



(f) : Day Traffic Sample Video Dataset.mp4

F. Sample from videos used for evaluation



(g) : Pedestrian and Traffic, Human Activity Recognition Video ,DataSet By UET Peshawar.mp4



(h) : Pexels Videos 1601538.mp4



(i) : Pexels Videos 2577.mp4



(j) : Pexels Videos 2670.mp4



(k) : Pexels Videos 3047.mp4



(l) : Pexels Videos 948404.mp4



(m) : moderate_traffic.mp4

Figure F.1: Samples from used videos

Appendix G

Bibliography

- [1] Innovative Ways to Count Pedestrians and Bicyclists. <http://njbikeped.org/innovative-ways-count-pedestrians-bicyclists/>, Dec 2015. [online; accessed 1-May-2019].
- [2] Tcs for surveys. <https://www.tcsforsurveys.com.au/equipment-hire>. [online; accessed 1-May-2019].
- [3] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.
- [4] Programming tensor cores in CUDA 9. <https://devblogs.nvidia.com/programming-tensor-cores-cuda-9>, Mar 2019. [online; accessed 30-April-2019].
- [5] NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics. <https://devblogs.nvidia.com/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/>, Feb 2019. [online; accessed 30-April-2019].
- [6] Science X staff. Why are neuron axons long and spindly? study shows they’re optimizing signaling efficiency. <https://medicalxpress.com/news/2018-07-neuron-axons-spindly-theyre-optimizing.html>, Jul 2018. [online; accessed 28-April-2019].
- [7] Denny Britz. Understanding convolutional neural networks for NLP. <http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/>, Jan 2016. [online; accessed 25-April-2019].
- [8] Cs231n convolutional neural networks for visual recognition. <http://cs231n.github.io/convolutional-networks>. [online; accessed 30-April-2019].
- [9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

- [10] Tsung-Yi Lin, Piotr Dollar, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
- [11] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, 2016.
- [12] Bichen Wu, Forrest Iandola, Peter H. Jin, and Kurt Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017.
- [13] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [14] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
- [15] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.
- [16] Intersection over union (IoU) for object detection. <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/>, Jun 2018. [online; accessed 1-May-2019].
- [17] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei Zhou. A comparative study of containers and virtual machines in big data environment. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.
- [18] Nvidia embedded systems for next-gen autonomous machines. <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/>. [online; accessed 17-April-2019].
- [19] Sushma Nagaraj, Bhushan Muthiyar, Swetha Ravi, Virginia Menezes, Kalki Kapoor, and Hyeran Jeon. Edge-based street object detection. *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation*, Jun 2018.
- [20] Traffic count program. <http://www.cuppad.org/what-we-do/transportation-planning/traffic-counts/>. [online; accessed 27-April-2019].

- [21] Esri demographics. <https://doc.arcgis.com/en/esri-demographics/data/traffic-counts.htm>. [online; accessed 30-April-2019].
- [22] Olusegun Adebisi. Improving manual counts of turning traffic volumes at road junctions. *Journal of Transportation Engineering*, 113(3):256–267, 1987.
- [23] Pengjun Zheng and Mcdonad Mike. An investigation on the manual traffic count accuracy. *Procedia - Social and Behavioral Sciences*, 43:226–231, 2012.
- [24] Pneumatic tube detector. https://nptel.ac.in/courses/105101008/525_AutoLoop/point2/point.html. [online; accessed 25-April-2019].
- [25] Vehicle sensing: overview of technologies used to count vehicles. <https://www.windmill.co.uk/vehicle-sensing.html>, Mar 2018. [online; accessed 30-April-2019].
- [26] Vivienne Sze, Yu-Hsin Chen, Joel Emer, Amr Suleiman, and Zhengdong Zhang. Hardware for machine learning: Challenges and opportunities. *2017 IEEE Custom Integrated Circuits Conference (CICC)*, 2017.
- [27] Fei Liu, Zhiyuan Zeng, and Rong Jiang. A video-based real-time adaptive vehicle-counting system for urban roads. *Plos One*, 12(11), 2017.
- [28] What is the Standard UK Vehicle Classification Scheme? <http://www.videodatapad.com/faq/standard-uk-vehicle-classification>. [online; accessed 30-April-2019].
- [29] *The COBA manual*, volume 13 of *ECONOMIC ASSESSMENT OF ROAD SCHEMES*.
- [30] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015.
- [31] Mahmut Yazici, Shadi Basurra, and Mohamed Gaber. Edge machine learning: Enabling smart internet of things applications. *Big Data and Cognitive Computing*, 2(3):26, 2018.
- [32] Torbjorn Grande Ostby. *Object Detection and Tracking on a Raspberry Pi using Background Subtraction and Convolutional Neural Networks*. PhD thesis, 2018.
- [33] Opencv, opencv, and movidius ncs on the raspberry pi. <https://www.pyimagesearch.com/2019/04/08/opencv-opencv-and-movidius-ncs-on-the-raspberry-pi/>, Apr 2019. [online; accessed 14-May-2019].

- [34] NVIDIA Newsroom. NVIDIA Unveils First Mobile Supercomputer for Embedded Systems. <https://nvidianews.nvidia.com/news/nvidia-unveils-first-mobile-supercomputer-for-embedded-systems>. [online; accessed 14-May-2019].
- [35] NVIDIA Newsroom. NVIDIA Jetson TX2 Enables AI at the Edge. <https://nvidianews.nvidia.com/news/nvidia-jetson-tx2-enables-ai-at-the-edge>. [online; accessed 14-May-2019].
- [36] Jetson AGX Xavier Module Now Available – A Major Leap Forward for Autonomous Machines. <https://news.developer.nvidia.com/jetson-agx-xavier-module-now-available/>, Dec 2018. [online; accessed 14-May-2019].
- [37] NVIDIA Newsroom. NVIDIA Announces Jetson Nano: \$99 Tiny, Yet Mighty NVIDIA CUDA-X AI Computer That Runs All AI Models. <https://nvidianews.nvidia.com/news/nvidia-announces-jetson-nano-99-tiny-yet-mighty-nvidia-cuda-x-ai-computer-t>. [online; accessed 14-May-2019].
- [38] Alasdair Allan. Introducing the nvidia jetson nano. [online; accessed 14-May-2019].
- [39] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance & precision. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018.
- [40] Nvidia jetson agx xavier delivers 32 teraops for new era of ai in robotics, Feb 2019. [online; accessed 10-May-2019].
- [41] Seyed Yahya Nikouei, Yu Chen, Sejun Song, Ronghua Xu, Baek-Young Choi, and Timothy R. Faughnan. Real-time human detection as an edge service enabled by a lightweight cnn. *2018 IEEE International Conference on Edge Computing (EDGE)*, 2018.
- [42] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*.
- [43] Seung-Hyun Lee, Minsuk Bang, Kyeong-Hoon Jung, and Kang Yi. An efficient selection of hog feature for svm classification of vehicle. *2015 International Symposium on Consumer Electronics (ISCE)*, 2015.
- [44] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *Computer Vision – ECCV 2016 Lecture Notes in Computer Science*, page 21–37, 2016.

- [45] George Plastiras, Maria Terzi, Christos Kyrkou, and Theodoris Theodoridis. Edge intelligence: Challenges and opportunities of near-sensor machine learning applications. *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018.
- [46] Antonio Loquercio, Ana I. Maqueda, Carlos R. Del-Blanco, and Davide Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095, 2018.
- [47] En Li, Zhi Zhou, and Xu Chen. Edge intelligence. *Proceedings of the 2018 Workshop on Mobile Edge Communications - MECOMM18*, 2018.
- [48] Sparsh Mittal. A survey on optimized implementation of deep learning models on the nvidia jetson platform. *Journal of Systems Architecture*, 2019.
- [49] Eyedentify MMR SDK, Technical sheet, Version 2.3.1. http://www.eyedea.cz/download/Eyedentify_MMR_SDK-Technical_sheet.pdf. [online; accessed 14-May-2019].
- [50] Stefano Messelodi, Carla Maria Modena, and Michele Zanin. A computer vision system for the detection and classification of vehicles at urban road intersections. *Pattern Analysis and Applications*, 8(1-2):17–31, 2005.
- [51] Chin-teng Lin, Sheng-Chin Hsu, Kuang-Peng Chou, Linda Siana, and Chien-Ting Yang. Real-time boosted vehicle detection deal with high detection rate using false alarm eliminating method. *ICIC International*, 9:3039–3052, Jul 2013.
- [52] Yiren Zhou, Hossein Nejati, Thanh-Toan Do, Ngai-Man Cheung, and Lynette Cheah. Image-based vehicle analysis using deep neural network: A systematic study. *2016 IEEE International Conference on Digital Signal Processing (DSP)*, 2016.
- [53] Sheeraz Memon, Sania Bhatti, Liaquat Thebo, Mir Talpur, and Mohsin Memon. A video based vehicle detection, counting and classification system. *I.J. Image, Graphics and Signal Processing*, 9:34–41, 2018.
- [54] Anuj Sharma and Tienaah Titus. Automated vehicle recognition with deep convolutional neural networks. *Transportation Research Record: Journal of the Transportation Research Board*, 2645:113–122, 2017.
- [55] Nikolaus Kriegeskorte. Deep neural networks: a new framework for modelling biological vision and brain information processing. 2015.
- [56] Haibing Wu and Xiaodong Gu. Max-pooling dropout for regularization of convolutional neural networks. *Neural Information Processing Lecture Notes in Computer Science*, page 46–54, 2015.

- [57] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.
- [58] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, Jun 2014.
- [59] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ICML*, 2015.
- [60] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017.
- [61] Keras. RetinaNet, GitHub. <https://github.com/fizyr/keras-retinanet>. [online; accessed 14-May-2019].
- [62] Precision-recall curves – what are they and how are they used? <https://acutecaretesting.org/en/articles/precision-recall-curves-what-are-they-and-how-are-they-used>. [online; accessed 1-May-2019].
- [63] Adrian Mouat. *Using Docker*. O’Reilly, 2015.
- [64] NVIDIA. Nvidia-docker, GitHub. <https://github.com/NVIDIA/nvidia-docker>. [online; accessed 14-May-2019].
- [65] Nvidia. Nvidia/nvidia-docker. <https://github.com/NVIDIA/nvidia-docker/wiki/Frequently-Asked-Questions>. [online; accessed 8-May-2019].
- [66] JetPack 3.2 — L4T R28.2 Production Release for Jetson TX1/TX2. <https://devtalk.nvidia.com/default/topic/1030831/jetson-tx2/jetpack-3-2-mdash-l4t-r28-2-production-release-for-jetson-tx1-tx2/>. [online; accessed 14-May-2019].
- [67] Technica-Corporation. TX2-docker, GitHub. <https://github.com/Technica-Corporation/Tegra-Docker/blob/master/bin/tx2-docker>. [online; accessed 14-May-2019].
- [68] Jetpack. <https://developer.nvidia.com/embedded/jetpack>, May 2019. [online; accessed 8-May-2019].
- [69] OpenCV. OpenCV, GitHub. <https://github.com/opencv>. [online; accessed 14-May-2019].
- [70] Opencv with cuda for tegra. https://docs.opencv.org/3.2.0/d6/d15/tutorial_building_tegra_cuda.html. [online; accessed 8-May-2019].

- [71] Ubuntu 18.04 for arm64v8, Docker Hub. <https://hub.docker.com/r/arm64v8/ubuntu/>. [online; accessed 14-May-2019].
- [72] Google. Protocolbuffer v3.5.0, GitHub. <https://github.com/protocolbuffers/protobuf/releases/tag/v3.5.0>. [online; accessed 14-May-2019].
- [73] AlexeyAB. Darknet, GitHub. <https://github.com/AlexeyAB/darknet>. [online; accessed 14-May-2019].