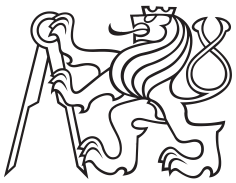
Master's Thesis



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

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography. Prague, května 3, 2019

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu. V Praze, 3. May 2019

Abstract

Keywords:

Supervisor: Ing. Lukáš Hrubý

Abstrakt

Klíčová slova:

Překlad názvu: Hluboké neuronové sítě ve vestavěných systémech

Contents

Figures

Tables

Chapter 1

Introduction

Internet of Things (IoT) and artificial intelligence (AI) is becoming part of out lives making everything smart and intelligent. Integration of IoT and AI in smart cities is one of the promising application[17]. Machine learning has given the ability to process various tasks withou human intervention such as recognizing objects, playing games, diagnosis diseases. Deep learning is one of the major branches in of machine learning, and one of the most trending application of deep learning is traffic object detection, which is the core component of traffic control in smart cities. It not only makes the urban life convenient, but also safer.

1.1 Problem overview

Traffic counts are widely used by state and local transportation official in the planning of load improvements and monitoring of traffic conditions [18]. In private sector traffic counts data 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 and so on [19].

There are several technologies how traffic data can be collected, each one having its own advantages and disadvantages.

1.2 Overview of technologies for traffic measurement

1.2.1 Manual counts

Manual traffic counts are defined as in-person traffic counts, where the counter is physically present at the location of data collection[20] 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%[21]. 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 counts[1]

1.2.2 Pneumatic tube detector

This methods 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 can not 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 kind of vehicles[22].

1.2.3 Sensors for traffic data collection

Different sensor can be used for traffic counts. For example, piezoelectric sensors mounted in a groove cut into road's surface collect data by converting mechanical energy into electrical energy[23]. Basically it works on the same principle as pneumatic tube detector.

Another example of sensor used for traffic counts is magnetic sensors that detect vehicle by measuring the change in the earth's magnetic field as the vehicle pass over the detector buried in the road[23].

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

Figure 1.2: Pneumatic tubes detector[2]

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. Applying algorithm for object detection and classification for object counting has proved to be extremely accurate with accuracy exceeding 99%[24]. These systems are cost-effective as they can count in many directories at once with only one camera needed for several lines or exits at a junction. However, most of available solutions work only with recorded videos and are not suitable for online traffic count and monitoring, or uses remote servers for any calculations, and 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 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 propose a software solution for real-time traffic count and monitoring utilizing edge computing.

We want to focuse on detection of objects that are described in the Standard UK vehicle classification scheme called COBA [25][26]. 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) - rigid vehicle with two or three axles,
4. Ordinary Goods Vehicle 2 (OGV2) - rigid vehicle with four or more axles,
5. Public Service Vehicle (PSV) - all public service vehicle,
6. Motorcycle (MC) - all types of motorcycles inclding 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.

Edge computing is beneficial for many real-time applications such as traffic object detection because it can provide a timely solution without needing to communicate with remote server. Some researches [27] show that platforms built for face recognitions application has 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 [28] that such machine learning algorithm as random forests or support vector machine can be run on widely available devices as Raspberry Pi. Another research[29], however, shows that Raspberry Pi does not have enough computation power

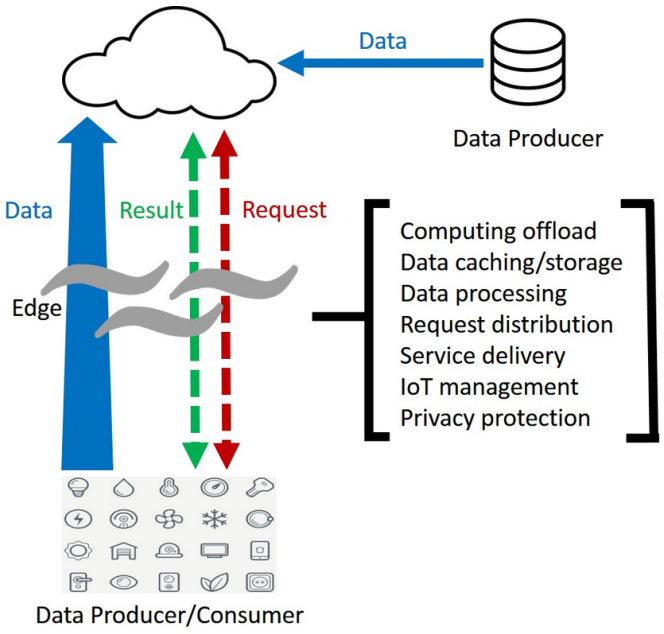


Figure 1.3: Edge computing paradigm[3]

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.

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[1] . Jetson TK2 was announced in 2017[2] and was designed for low power systems like smaller camera drones. The next module called Jetson Xavier introduced in 2018[3] brings up to 20× acceleration compared to predecessor devises with power efficiency being improved 10×. The newest Nvidia Nano was announced in 2019[4] and is focused on hobbyist robotics thanks to its low price. The last three modules are compared in Table 1.1.

As we can see, unlike other boards Jetson Xavier is build 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 Accelerator [5] and Vision Accelerator engines. Both of them are described in paragraph 1.3.3.2.

[1]https://nvidianews.nvidia.com/news/nvidia-unveils-first-mobile-supercomputer-for-embedded-systems

[2]https://nvidianews.nvidia.com/news/nvidia-jetson-tx2-enables-ai-at-the-edge

[3]https://news.developer.nvidia.com/jetson-agx-xavier-module-now-available/

[4]https://nvidianews.nvidia.com/news/nvidia-announces-jetson-nano-99-tiny-yet-mighty-nvidia-cuda-x-ai-computer-that-runs-all-ai-models

[5]http://nvdla.org/

	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	Wi-Fi requires external chip
	Gigabit Ethernet		
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 x87 mm	69.6 mm x 45 mm

Table 1.1: NVIDIA Jetson comparison[16]

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[30]. Tensor cores in mixed-precision mode take 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 similar way.

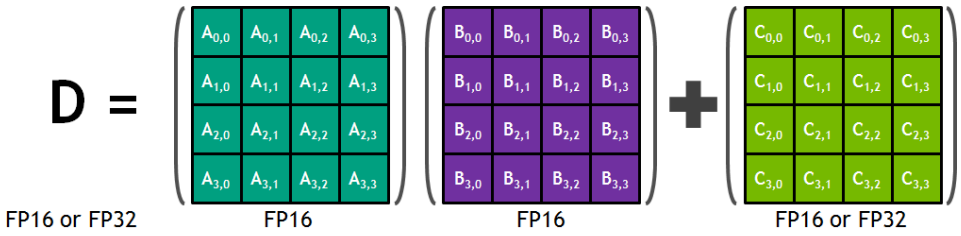


Figure 1.4: Tensor core operation[4]

1.3.3.2 Deep learning and vision accelerators

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 trillion operations per second (TOPS) with INT8 precision or 2.5 trillion floating-point operations per second (TFLOP) with FP16 precision[?]. It also support 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 responsable for acceleration of algorithm such as optical

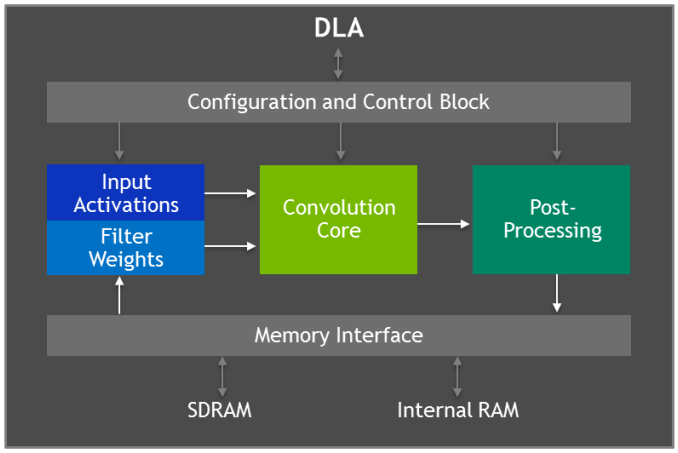


Figure 1.5: Deep Learning accelerator architecture[5]

flow, point cloud processing, morphological operations, histogramming and so on.

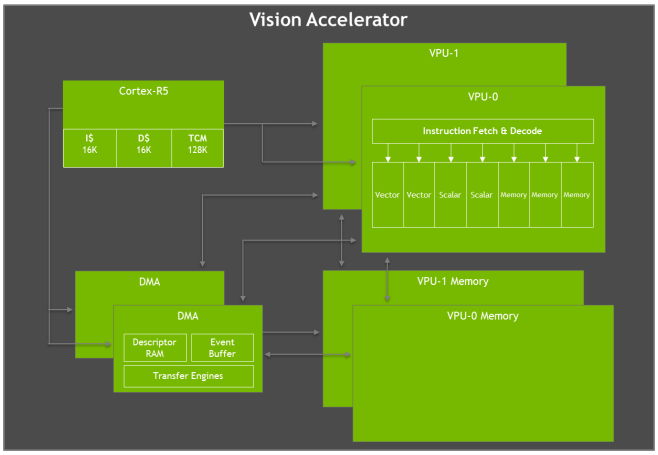


Figure 1.6: Vision Accelerator[5]

Chapter 2

Related works

2.1 CNN on the edge

2.2 Commercial traffic count systems

2.3 Non-commercial traffic count systems

Chapter 3

Neural networks

Artificial neural networks are systems inspired by a brain. The basic computation unit in a brain is a neuron (see 3.1), which has input and output. The input is a dendritic tree connected to the outputs of other neurons called axons. Neurons operates 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 convolutional neural network (CNN), which will be described in the following section.

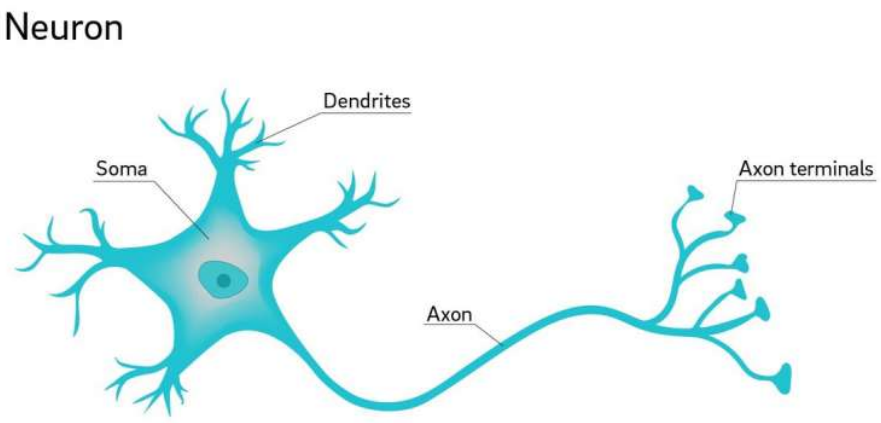


Figure 3.1: Neuron cell[6]

3.1 Architecture

All CNN models has a similar architecture as it is shown in 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.

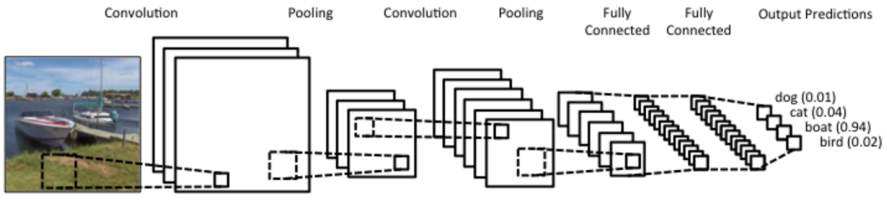


Figure 3.2: Convolutional neural network[7]

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 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 till the whole image is traversed. The process computes convolution as it is shown in 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.

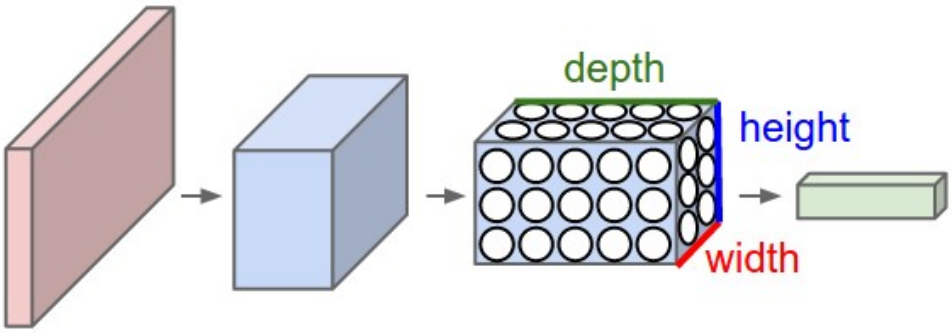


Figure 3.3: Convolutional layer[8]

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.

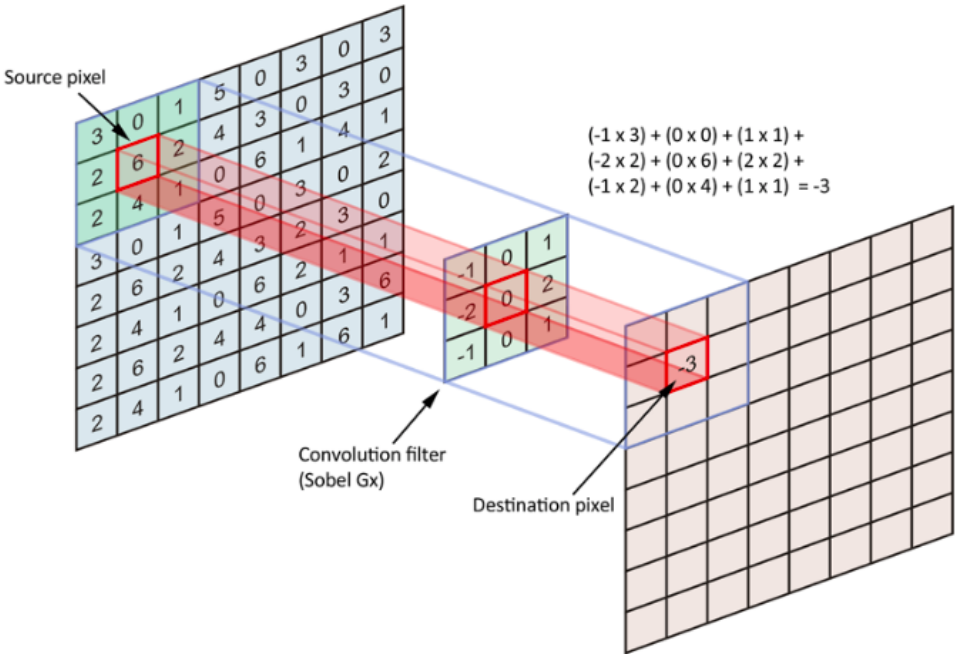


Figure 3.4: Convolution computation[8]

3.1.3 Pooling layer

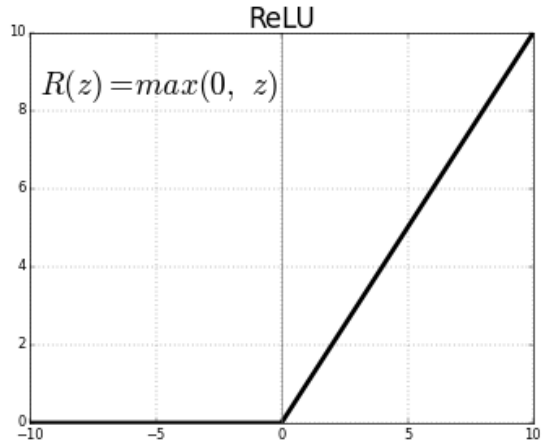
After convolution, pooling layer is used to reduce the dimensionality which enables to reduce number of parameters. Two most common pooling operation are max and min pooling. It simply slides the input with particular stride and choose maximal or minimal value in predefined window (see 3.6). Pooling helps to not overfit CNN and can reduce the training time.

3.1.4 Fully connected layer

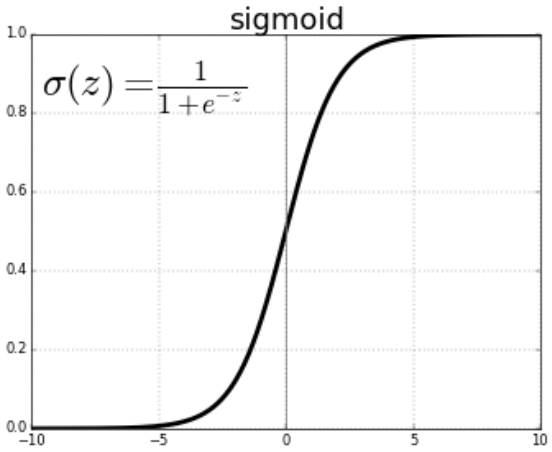
In fully connected layers, each neuron is connected to every neuron in the previous layer just like in feedforward neural networks.

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 weight 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. Learning algorithm is called backpropagation.



(a) : ReLu



(b) : sigmoid

Figure 3.5: Comparation of ReLu and sigmoid non-linearities

3.2.1 Backpropagation

Backpropagation was firstly introduced in[31] in 1986. This process can be separated into 4 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 first training forward pass output of the network would probably be randomized, because the network isn't able to look for any kind of features, thus 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

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

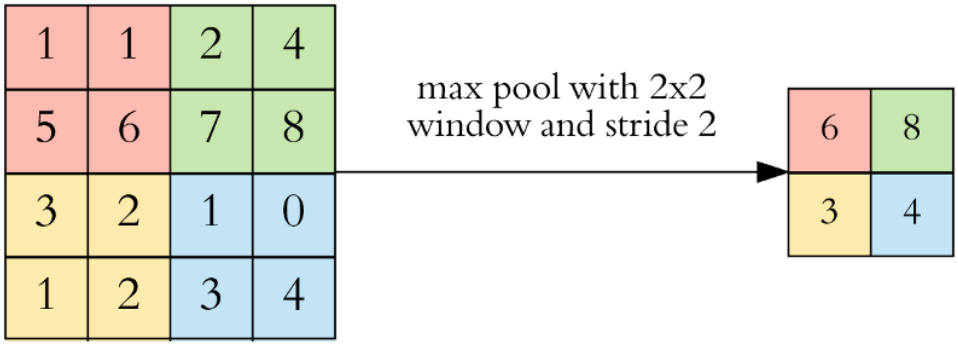


Figure 3.6: Pooling layer[8]

where n denotes number of training inputs x, $y(x)$ label corresponding to the input x and a is 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:

$$\omega = \omega - \eta \frac{\partial L}{\partial \omega} \tag{3.2}$$

$$b = b - \eta \frac{\partial L}{\partial b} \tag{3.3}$$

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. Some techniques help to decrease training speed and performance.

3.2.2 Dropout

Overfitting is one of the biggest problems in machine learning. Overfitting means that a model performs well on a training dataset, but fails on test 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.

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[32]. 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 this chapter we will talk about different CNN architectures that will be used in this work.

4.1 ResNet

Residual networks described in [33] 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 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 4.2) Top-1 error drops from 28.54% to 25.03%. On the other hand, if we compare 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 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 Research[2] and its features are described in [34] and [35]. They proposed using anchor boxes instead of predicting bounding boxes. Sizes of the anchor boxes are predefined and used

[2] https://research.fb.com/category/facebook-ai-research/

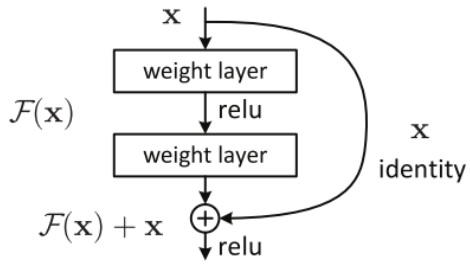


Figure 4.1: Residual block

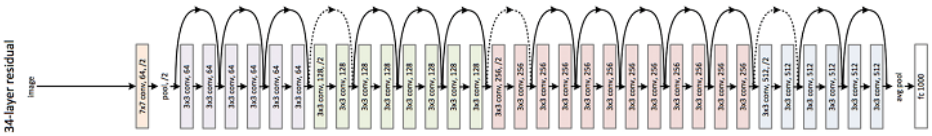


Figure 4.2: ResNet architecture

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 [34] authors propose Feature Pyramid Network (FPN) which is fast like the described pyramid of features, but more accurate. Its architecture is seen in 4.3. The other solution, focal loss, solves class imbalance. Instead of normal cross entropy calculated by

$$C(p, y) = -\sum_i y_i \ln p_i, \tag{4.1}$$

scaled entropy is used using following equation:

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

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[1] implemented in *TensorFlow* with ResNet50 as a backbone.

4.3 SqueezeDet

SqueezeDet[10] is a single stage detection pipeline inspired by YOLO, which will be covered later in section 4.4. The main difference between two archi-

[1] https://github.com/fizyr/keras-retinanet

tectures is that SqueezeDet uses SqueezeNet[9] for feature extraction.

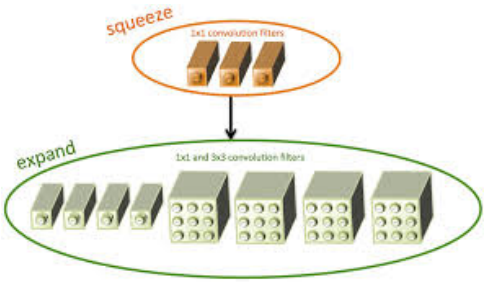


Figure 4.4: Fire module in SqueezeNet[9]

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 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 fewer computation. Thanks to its architecture, SqueezeDet can be faster and smaller comparing 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[10]

The loss function of SqueezeDet is defined as

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

where 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 cross-entropy loss for classification.

We used a tensorflow implementation of SqueezeDet available to download from GitHub[1].

4.4 YOLO

4.4.1 YOLO v1

A new approach for object detection, YOLO architecture, was presented in [11]. 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 a $S \times S$ 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

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

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 2 fully connected layers as it is shows in 4.6.

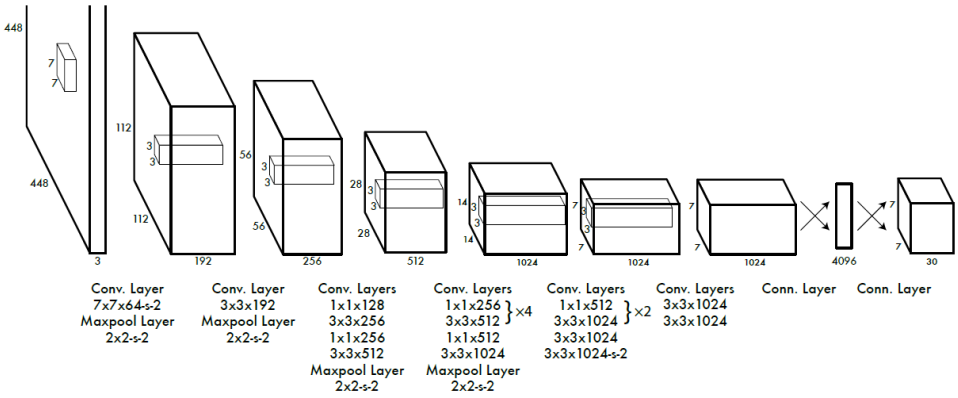


Figure 4.6: YOLO v1 architecture[11]

[1] https://github.com/BichenWuUCB/squeezeDet

Training process optimizes loss function

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

where

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

$\mathbf{1}_{ij}^{noobj}$ 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 does not contain objects. While YOLO v1 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 YOLO v2

A new version of YOLO was introduced in [12]. Authors of this state of the art detector refer 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: YOLO v2 improvement[12]

They also proposed a new classification network called Darknet-19 (see Appendix A) 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 YOLO v3

The newest version of YOLO was presented in [13]. 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 3 different scales for prediction, which is similar to feature pyramid networks. Deeper extractor called Darknet-53 (see Appendix B) with shortcut connections is used for feature extraction.

Comparing to other state of art solutions, YOLO v3 has similar performance, but it is much faster as it is seen in 4.8. Unlike RetinaNet and SqueezeDet, YOLO uses another neural network framework, Darknet[1], written in C and CUDA.

[1] https://pjreddie.com/darknet/

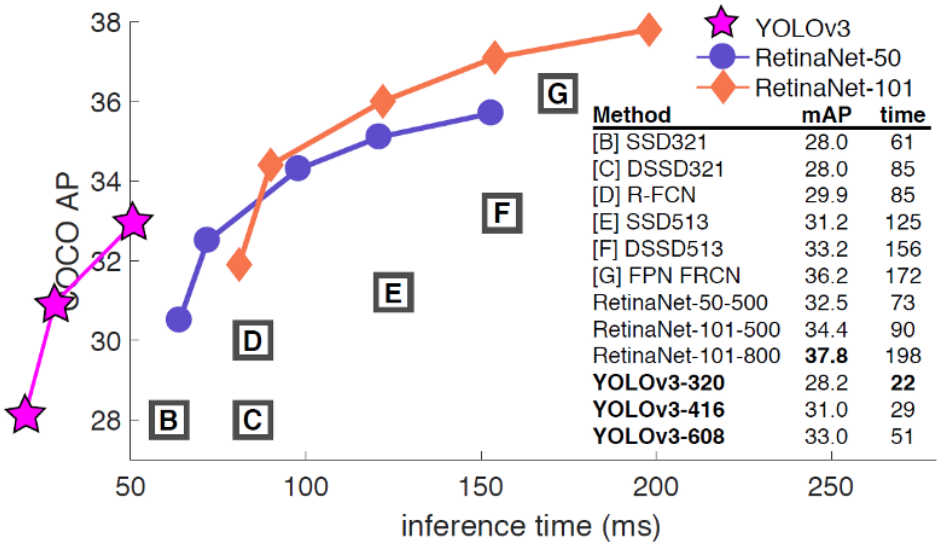


Figure 4.8: YOLO v3 comparation[13]

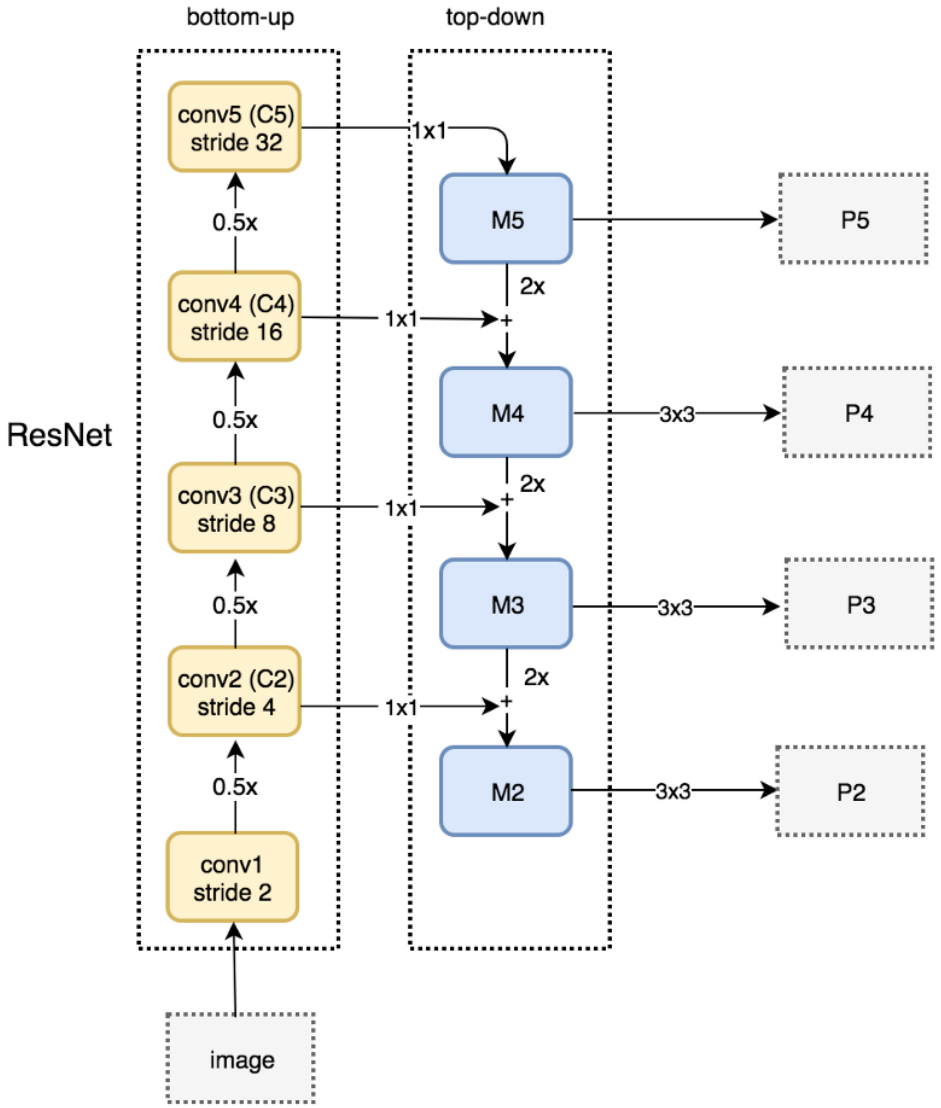


Figure 4.3: Feature Pyramid Network

Chapter 5

Experiments

For experiments, we chose SqueezeDet, YOLOv3, and RetinaNet. We trained all three networks on KITTI[2] 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. 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 datasets has 9 different type of objects:

1. "Car"
2. "Van"
3. "Truck"
4. "Pedestrian"
5. "Person_sitting"
6. "Cyclist"
7. "Tram"
8. "Misc"
9. "DontCare".

[2] http://www.cvlibs.net/datasets/kitti/

We used only 5 of them for training: “Car”, “Van”, “Truck”, “Pedestrian”, “Person_sitting”, though we merged “Person_sitting” and “Pedestrian” classes into single “Person” object type. Regardless of 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 complex 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.

5.2 Performance evaluation

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

$$Precision = \frac{TP}{TP + FP}, \tag{5.1}$$

Figure 5.2: Image samples from "GV-2018"

$$Recall = \frac{TP}{TP + FN}, \tag{5.2}$$

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[36], and average precision is 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 5.1. YOLOv3 performs better than both SqueezeDet and RetinaNet at both resolution. Although YOLOv3 trained on KITTI dataset indicates bigger mAP on KITTI evaluation set, it has worse performance on GoodVision evaluation set, while YOLOv3 trained on

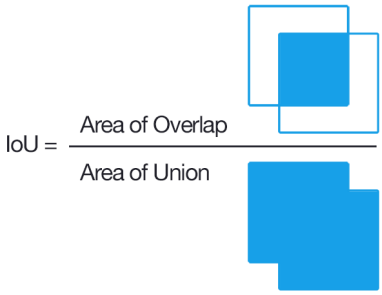


Figure 5.3: Intersection over Union[14]

GoodVision dataset has almost 10% better mAP. We can notice this trend for all 3 CNNs: while KITTI-trained model has better performance on KITTI test set, it fails on GoodVision set. If we compare models trained on KITTI and GoodVision dataset 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, in this case 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 YouTube[1] or Pexels[2]. All videos have different camera view with various object count, video resolution and FPS, as shown in table 5.2. This can affect neural network detector performance, hence we should also compare various input resolutions of neural networks. Video samples presented in Appendix C.1 shows obvious difference between videos scenes. Those scenes represents typical use case for our system.

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 5.2: Characteristics of video for inference time evaluation

[1] https://www.youtube.com/
[2] https://www.pexels.com/videos/

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 5.3 and 5.4.

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

Table 5.3: PC1 technical specification

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

Table 5.4: PC2 technical specification

As we can see from table 5.5, PC1 performs best for all three CNNs. On the other hand, PC2 has worse inference time comparing to Jetson Xavier 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.

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 5.5: Inference time evaluation

5.3.1 Mixed precision calculation

We have discussed in section 1.3.3 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 core directly, because it is written in CUDA. To utilize tensor core using TensorFlow we should use TensorRT platform[1]. TensorRT is able to convert TensorFlow CNN graph into supported format and use it for inference with mixed precision calculation. We achieved 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 5.6.

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 5.6: Inference time using mixed precision calculation

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 layer 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

Model	Training dataset	Testing dataset	Bicycle AP [%]	Bus AP [%]	Motorcycle AP [%]	Car AP [%]	Person AP [%]	Truck AP [%]	Van AP [%]	mAP [%]
YOLOv3 608	GoodVision	GoodVision	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
		GoodVision	–	–	–	58,6	72,3	73,2	67,0	67,8
YOLOv3 416	GoodVision	GoodVision	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
		GoodVision	–	–	–	52,1	63,5	62,2	43,0	55,2
RetinaNet 1024	GoodVision	GoodVision	46,3	34,8	15,4	39,5	32,3	53,8	40,6	37,5
	KITTI	KITTI	–	–	–	35,2	29,9	51,6	31,1	37,0
		GoodVision	–	–	–	20,6	23,3	42,2	19,4	26,4
RetinaNet 608	GoodVision	GoodVision	11,0	12,7	0,2	25,5	18,0	28,3	23,4	17,0
	KITTI	KITTI	–	–	–	23,0	16,6	27,8	20,3	21,9
		GoodVision	–	–	–	2,3	16,3	24,6	18,1	15,3
RetinaNet 416	GoodVision	GoodVision	0,0	5,8	0,6	19,5	12,9	15,8	16,7	10,2
	KITTI	KITTI	–	–	–	19,1	12,3	14,6	15,1	15,3
		GoodVision	–	–	–	13,4	7,8	10,6	11,2	10,8
squeezeDet	GoodVision	GoodVision	1,7	8,2	4,1	34,4	39,2	22,9	12,9	17,6
	KITTI	KITTI	–	–	–	30,6	38,5	22,5	10,8	25,6
		GoodVision	–	–	–	3,1	2,9	4,7	4,2	3,7

Table 5.1: CNN evaluation

Chapter 6
Implementation

6.1 Docker

Docker[1] containers are used for an 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[15]. Additionally, several containers can share the same image, while each VM has its own image file[37]. The portability of containers also help with software distribution: once container is created, it can be used on different machines with no additional settings. They also provides isolation of out application from settings on our machine, which is extremely useful during developing.

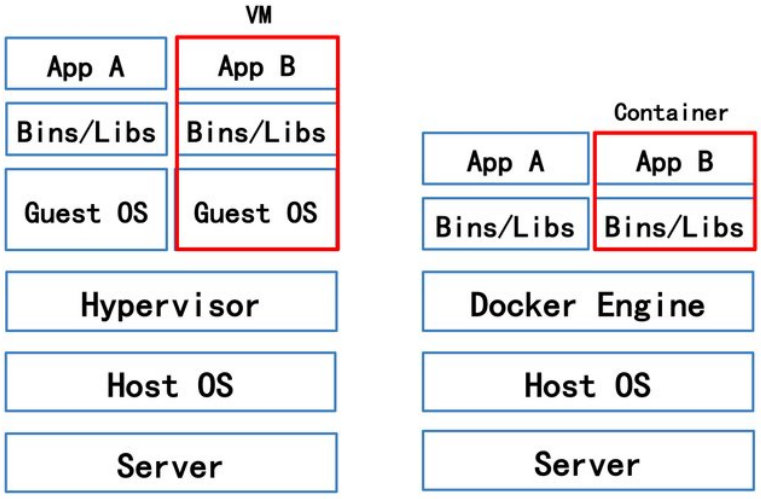


Figure 6.1: Architecture comparison virtual machine vs. container[15]

NVIDIA offers its own docker called "nvidia-docker"[2] to enable GPUs

[1] https://www.docker.com/
[2] https://github.com/NVIDIA/nvidia-docker

inside docker containers. However, this solution does not support Tegra platforms (https://github.com/NVIDIA/nvidia-docker/wiki/Frequently-Asked-Questions) as Jetson Xavier. Fortunately, JetPack, which is discussed in Section 6.2, has Docker support built into kernel since version 3.2[1]. However, it still does not support GPU mapping into docker containers, which should be done manually when starting 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 docker container and they should be mapped as well when container is started. To make this process easier, we use bash script that overwrite process of starting docker container. Since CUDA is available only inside running container, it is not possible to build docker image with installed prerequisites 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

instalace software, jetpack, darknet, cuda, etc

6.3 YOLOv3 detector

popsat implementaci detektoru

6.4 API

We implemented a service for communication with Jetson Xavier module and detector. Endpoints of the service are defined in table and their documentation can be found in the Appendix

API was implemented in Python3 using Flask framework[2].

[1] https://devtalk.nvidia.com/default/topic/1030831/jetson-tx2/jetpack-3-2-mdash-l4t-r28-2-production-release-for-jetson-tx1-tx2/

[2] http://flask.pocoo.org/

Chapter 7
Conclusions

zhodnoceni vysledku prace

7.1 Feature work

co by se mohlo udelat, jak by se prace mohla rozsirit

Appendix A

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 A.1: Darknet-19

Appendix B

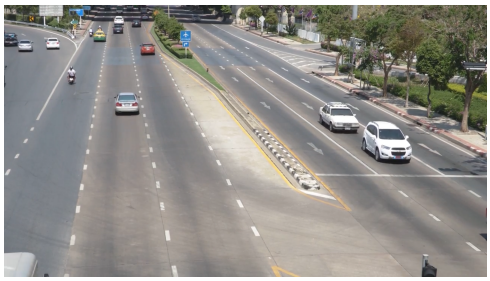
Darknet-53

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

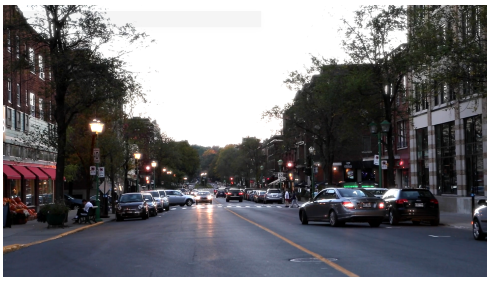
Figure B.1: Darknet-53

Appendix C

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

(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 C.1: Samples from used videos

Appendix D

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] 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.

[10] 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.

[11] 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.

[12] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[13] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[14] 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].

[15] 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.

[16] Nvidia embedded systems for next-gen autonomous machines. `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/`. [online; accessed 17-April-2019].

[17] Sushma Nagaraj, Bhushan Muthiyan, 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.

[18] Traffic count program. `http://www.cuppad.org/what-we-do/transportation-planning/traffic-counts/`. [online; accessed 27-April-2019].

[19] Esri demographics. `https://doc.arcgis.com/en/esri-demographics/data/traffic-counts.htm`. [online; accessed 30-April-2019].

[20] Olusegun Adebisi. Improving manual counts of turning traffic volumes at road junctions. *Journal of Transportation Engineering*, 113(3):256–267, 1987.

[21] Pengjun Zheng and Mcdonad Mike. An investigation on the manual traffic count accuracy. *Procedia - Social and Behavioral Sciences*, 43:226–231, 2012.

[22] Pneumatic tube detector. `https://nptel.ac.in/courses/105101008/525_AutoLoop/point2/point.html`. [online; accessed 25-April-2019].

[23] Vehicle sensing: overview of techologies used to count vehicles. `https://www.windmill.co.uk/vehicle-sensing.html`, Mar 2018. [online; accessed 30-April-2019].

[24] Fei Liu, Zhiyuan Zeng, and Rong Jiang. A video-based real-time adaptive vehicle-counting system for urban roads. *Plos One*, 12(11), 2017.

[25] What is the Standard UK Vehicle Classification Scheme? `http://www.videodatapad.com/faq/standard-uk-vehicle-classification`. [online; accessed 30-April-2019].

[26] *The COBA manual*, volume 13 of *ECONOMIC ASSESSMENT OF ROAD SCHEMES*.

[27] 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.

[28] 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.

[29] Torbjorn Grande Ostby. *Object Detection and Tracking on a Raspberry Pi using Background Subtraction and Convolutional Neural Networks*. PhD thesis, 2018.

[30] 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.

[31] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[32] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ICML*, 2015.

[33] 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.

[34] 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.

[35] 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.

[36] 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; accesed 1-May-2019].

[37] Adrian Mouat. *Using Docker*. O'Reilly, 2015.