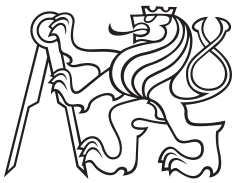
Master's Thesis



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
April 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, dubna 9, 2019

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 9. April 2019

Abstract

Keywords:

Supervisor: Ing. Lukáš Hrubý

Abstrakt

Klíčová slova:

Překlad názvu: Hluboké neuronové sítě ve vestavěných systémech

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Nvidia Jetson

Nvidia Jetson is a series of embedded modules producing by Nvidia designed for accelerating machine learning applications. First board, Jetson TK1, was presented in 2014. Jetson TK2 was announced in 2017 and was designed for low power systems like smaller camera drones. The next module called Jetson Xavier introduced in 2018 brings up to 20× acceleration compared to predecessor devises with power efficiency being improved 10×. The newest Nvidia Nano was announced in 2019 and is focused on hobbyist robotics thanks to its low price. The last three modules are compared in 1.1.

	Jetson Nano™	Jetson TX2			Jetson AGX Xavier™
		TX2 (4GB)	TX2 4GB	TX2i	
GPU	NVIDIA Maxwell™ architecture with 128 NVIDIA CUDA® cores	NVIDIA Pascal™ architecture with 256 NVIDIA CUDA cores			NVIDIA Volta™ architecture with 512 NVIDIA CUDA cores and 64 Tensor cores
CPU	Quad-core ARM® Cortex®-A57 MPCore processor	Dual-core Denver 2 64-bit CPU and quad-core ARM A57 complex			8-core ARM v8.2 64-bit CPU, 8 MB L2 + 4 MB L3
Memory	4 GB 64-bit LPDDR4	8 GB 128-bit LPDDR4	4 GB 128-bit LPDDR4	8 GB 128-bit LPDDR4	16 GB 128-bit LPDDR4x
Storage	16 GB eMMC 5.1	32 GB eMMC 5.1	16 GB eMMC 5.1	32 GB eMMC 5.1	32 GB eMMC 5.1
Video Encode	4K @ 30 (H.264/H.265)	2x 4K @ 30 (HEVC)			8x 4K @ 60 (HEVC)
Video Decode	4K @ 60 (H.264/H.265)	2x 4K @ 30 12-bit support			12x 4K @ 30 12-bit support
Connectivity	Wi-Fi requires external chip	Wi-Fi onboard	Wi-Fi requires external chip		Wi-Fi requires external chip
	Gigabit Ethernet				
Camera	12 lanes (3x4 or 4x2) MIPI CSI-2, DPHY 1.1 (1.5 Gbps)	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)
Size	69.6 mm x 45 mm	87 mm x 50 mm			100 mm x87 mm
Mechanical	260-pin edge connector	400-pin connector			699-pin connector

Figure 1.1: Nvidia Jetson comparison
https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/

As we can see, unlike other boards Jetson Xavier is build around NVIDIA Volta™GPU with tensor cores. These tensor cores accelerate large matrix operations and can perform mixed-precision matrix multiplication and accumulate calculation in a single operation. Each core provides a 4×4×4 matrix processing array which performs the operation $D = A \cdot B + C$ shown in 1.2, where A, B, C, D are 4×4 matrices. 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.

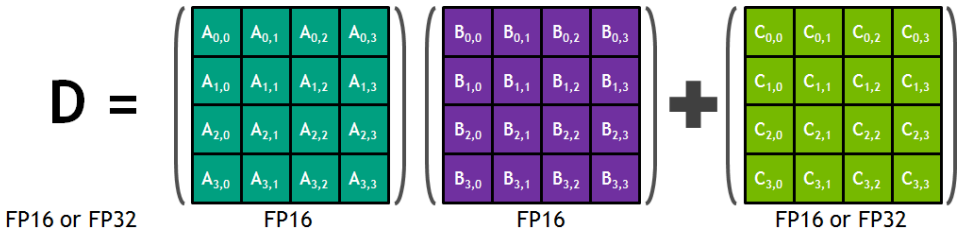


Figure 1.2: Tensor core operation
https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/

Another advantage of Jetson Xavier is using of Nvidia Deep Learning Accelerator engines http://nvdla.org/. These engines improve energy efficiency and free up the GPU to run more complex networks and implemented dynamic tasks and has up to 5 trillion operations per second (TOPS) INT8 or 2.5 TFLOPS FP16 performance with a power consumption of only 0.5-1.5W. They also supports accelerating of CNN convolution, deconvolution, activation functions, min/max/mean pooling, local response normalization, and fully-connected layers.

Chapter 2

Related works

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.

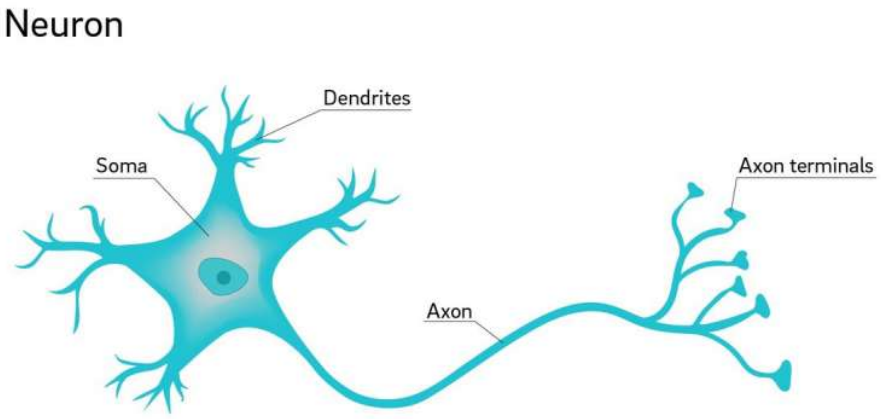


Figure 3.1: Neuron cell[?]

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.

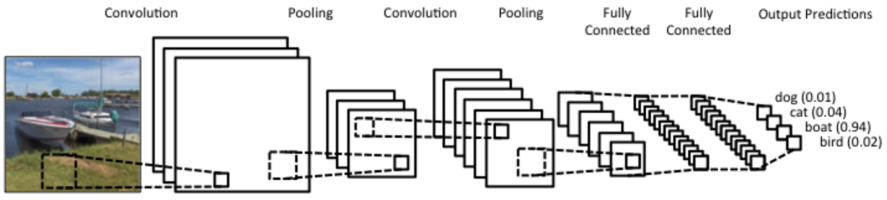


Figure 3.2: Convolutional neural network[?]

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.

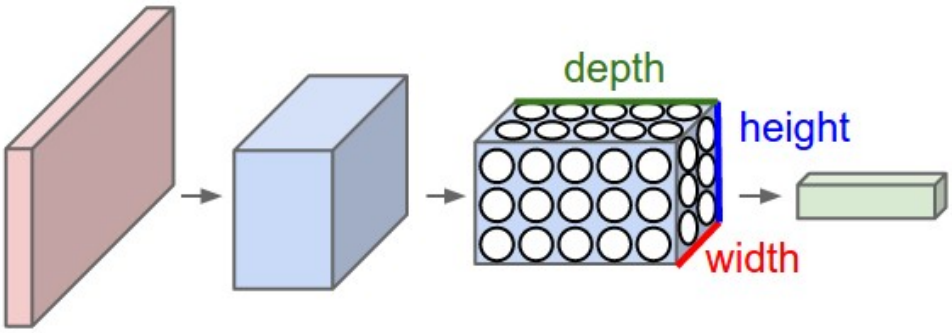


Figure 3.3: Convolutional layer[?]

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.

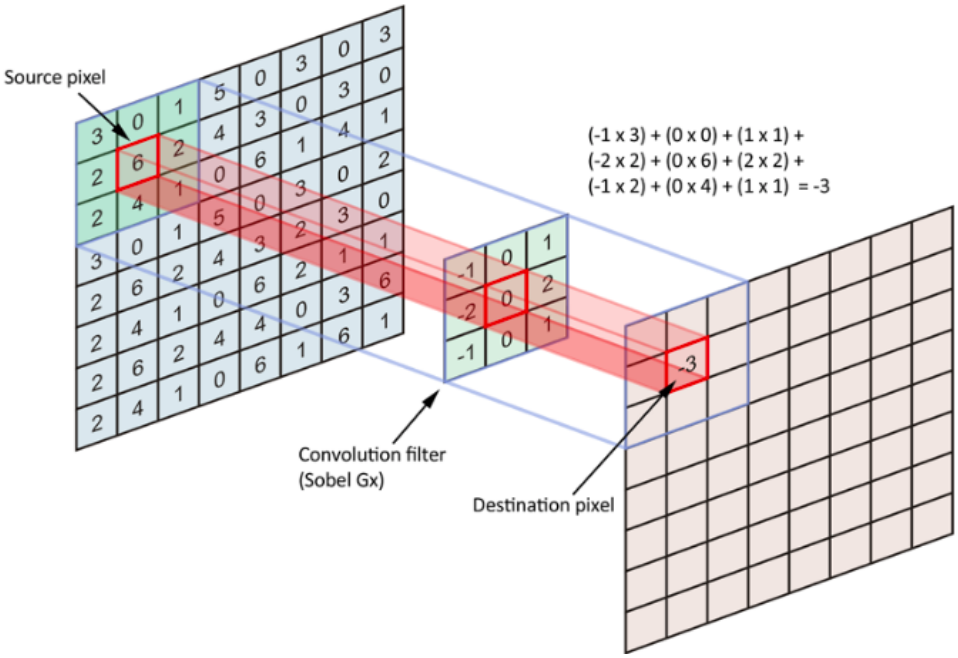


Figure 3.4: Convolution computation[?]

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

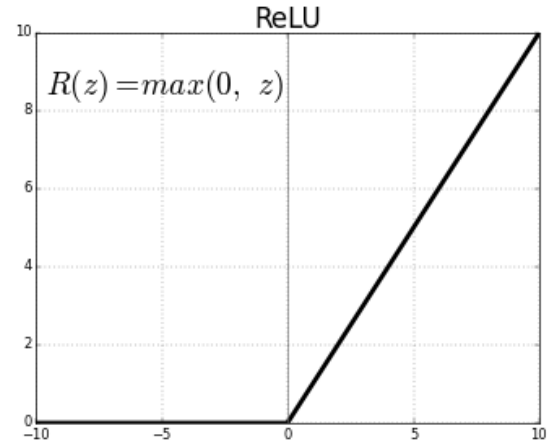
In fully connected layers, each neuron is connected to every neuron in the previous layer just like in feedforward neural networks.

3.2 Training of CNN

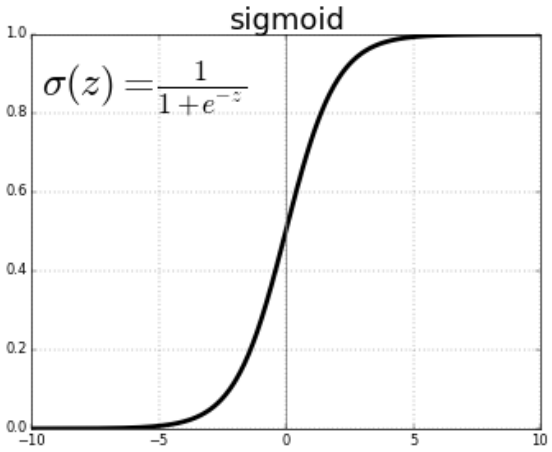
In this section, we will discuss how CNN are trained.

3.2.1 Backpropagation

For training CNN we should prepare training dataset as CNN belongs to supervised learning techniques. During training process, the network performs a forward pass for each example in training dataset. Computed output is compared to corresponding ground truth, then loss is calculated and the error is backpropagated to change parameters of the network. The goal is to find such weights and biases for every neuron that the output is as close as



(a) : ReLu



(b) : sigmoid

Figure 3.5: Comparation of ReLu and sigmoid non-linearities

possible to ground truth. That process is called backpropagation and was described in [**?**].

Each neural network can have its own loss function depending on what its output is. The most common loss function used for backpropagation is

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

where n denotes number of training inputs x, $y(x)$ label corresponding to the input x and a is network's output.

To update weight and biases we use gradient descent:

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

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

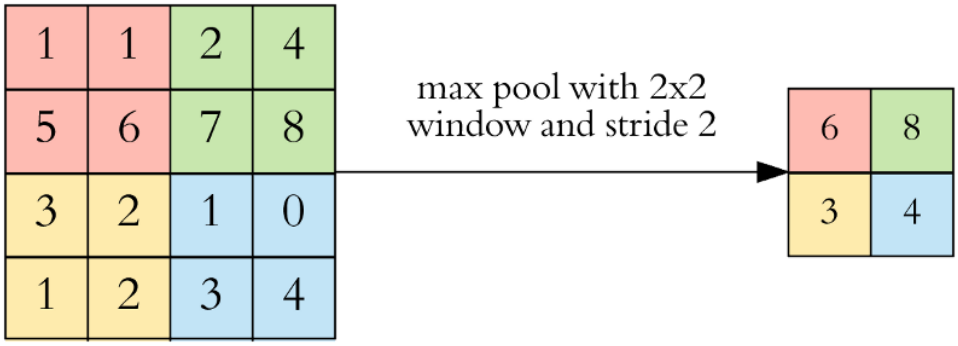


Figure 3.6: Pooling layer[?]

3.2.2 Overfitting

Overfitting is one of the biggest problem 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, 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.

Chapter 4

Used networks

4.1 ResNet

Residual networks described in [?] are classification networks with image as the input and object class and confidence score as the output. In this paper they introduced shorcut connections that are widely used in modern neural networks. One of the biggest problem 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 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 not 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.

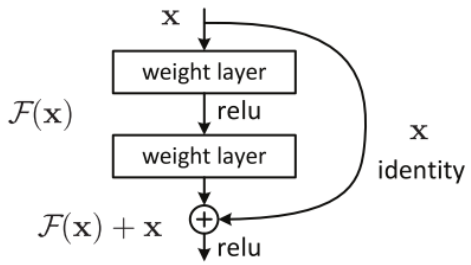


Figure 4.1: Residual block

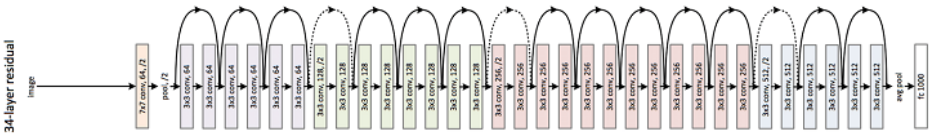


Figure 4.2: ResNet architecture

4.2 RetinaNet

https://medium.com/@14prakash/the-intuition-behind-retinanet-eb636755607d

RetinaNet was proposed by Facebook AI Research and its features are described in [?] and [?].

They proposed using anchor boxes instead of predicting 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 object. However, this solution is time consuming and has 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 [?] authors propose Feature Pyramid Network (FPN) which is fast like 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.

4.3 YOLO

4.3.1 YOLO v1

A new approach for object detection, YOLO architecture, was presented in [?]. A single neural network is used to predict both bounding boxes and class probabilities, hence an images is evaluated only once. 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

$$score = Pr(Object) \cdot IoU_{pred}^{truth}, \tag{4.3}$$

where $Pr(Object)$ is 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 4.4.

Training process optimizes loss function

$$\begin{aligned}
&\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i + \hat{y}_i)^2 \right] \\
&\quad + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} + \sqrt{\hat{h}_i} \right)^2 \right] \\
&\quad + \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{obj} \left(C_i - \hat{C}_i \right)^2 \\
&\quad + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{noobj} \left(C_i - \hat{C}_i \right)^2 \\
&\quad + \sum_{i=0}^{S^2} \mathbf{1}_{ij}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2
\end{aligned} \tag{4.4}$$

,

where

$$\mathbf{1}_{ij}^{obj} = \begin{cases} 1, & \text{if there is an object.} \\ 0, & \text{otherwise,} \end{cases} \tag{4.5}$$

$\mathbf{1}_{ij}^{noobj}$ 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 existing approaches for object detection, it had relatively low 57.9% mAP on the VOC 2012 test set comparing to existing state of art.

4.3.2 YOLO v2

New version of YOLO was introduced in [**?**]. Authors of this state of art detector refers to it as a better, faster and stronger version of YOLO. For a

better performance they added batch normalization and used images with 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 4.5.

They also proposed new classification network called Darknet-194.6 to make YOLO even faster. We can see that Darknet-19 has many 1×1 convolutions to reduce the number of parameters.

4.3.3 YOLO v3

The newest version of YOLO was presented in [**?**]. Similar to YOLOv2 it predicts bounding boxes using dimension clusters as anchor boxes. The network predicts 4 coordinates for each bounding box and for training they use sum of squared error loss. Objectness score for each bounding box is predicted using logistic regression, which should be 1 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. Features are now extracted with deeper extractor called Darknet-53 (see 4.7) with shortcut connections.

Comparing to other state of art solutions, YOLO v3 has similar performance, but it is much faster as it is seen in 4.8

4.4 SqueezeDet

SqueezeDet (https://arxiv.org/pdf/1612.01051.pdf) is a single stage detection pipeline inspired by YOLO. The main difference between two architectures is that SqueezeDet uses SqueezeNet (https://arxiv.org/pdf/1602.07360.pdf) for feature extraction.

The building brick of SqueezeNet is called fire module 4.9. 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 4.10), and so can be efficiently used on embedded system.

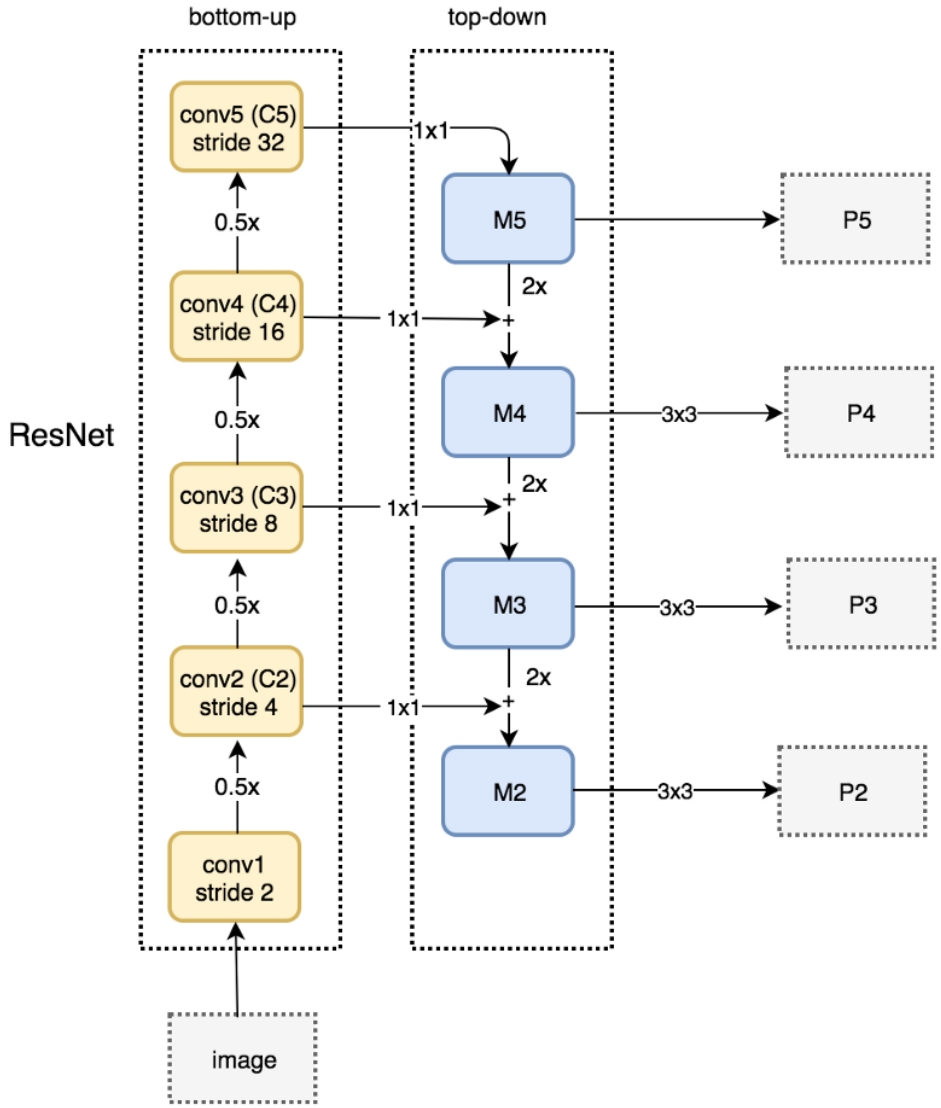


Figure 4.3: Feature Pyramid Network

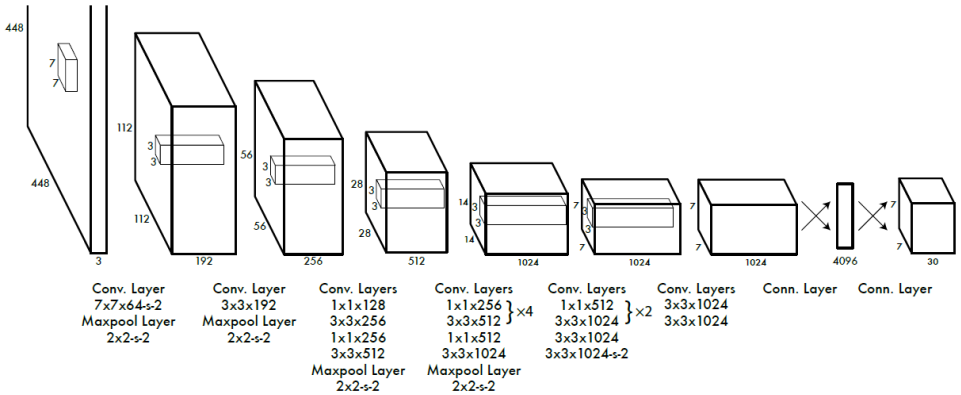


Figure 4.4: YOLO v1 architecture

	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**

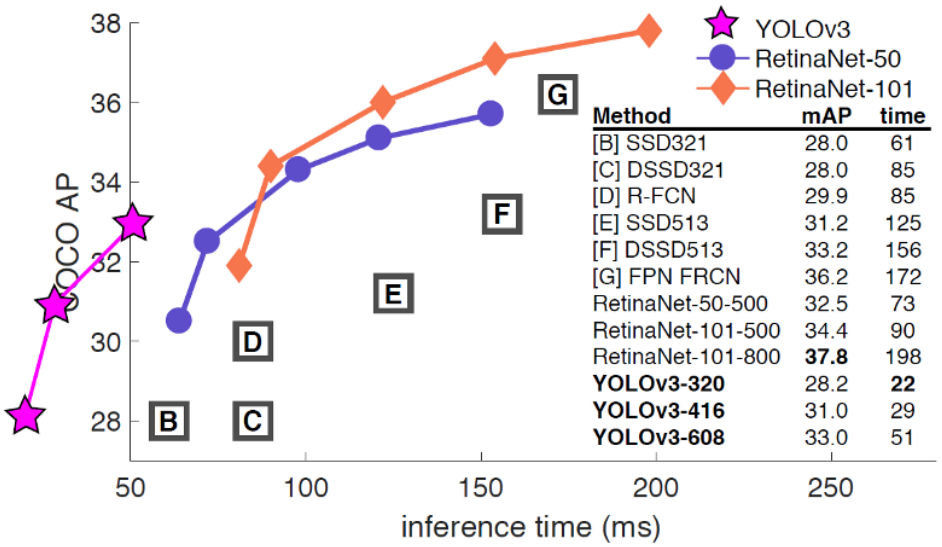
Figure 4.5: YOLO v2 improvement

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 4.6: Darknet-19

	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 4.7: Darknet-53



Method	mAP	time
[B] SSD321	28.0	61
[C] DSSD321	28.0	85
[D] R-FCN	29.9	85
[E] SSD513	31.2	125
[F] DSSD513	33.2	156
[G] FPN FRCN	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	**37.8**	198
YOLOv3-320	28.2	**22**
YOLOv3-416	31.0	29
YOLOv3-608	33.0	51

Figure 4.8: YOLO v3 comparation

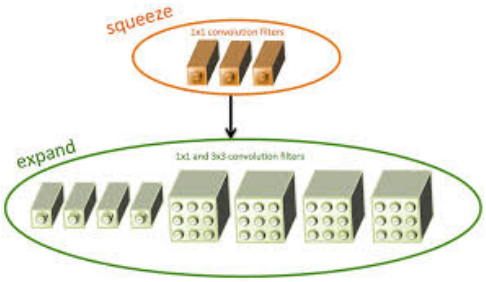


Figure 4.9: Fire module in SqueezeNet

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 4.10: SqueezeDet comparison with other state-of-art solutions

Chapter 5

Experiments

As soon as ResNet is just a classification network without bounding boxes proposal, we decided to evaluate performance of networks for both detection and classification, hence, RetinaNet, YOLOv3.

5.1 Inference time evaluation

For computing on the edge, we should ensure as small as possible time delay between receiving input data, in this case stream frame, and providing output data to a user. 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. We compared necessary time to process one image frame on Jetson AGX Xavier with other 2 different machines, their CPU and GPU specifications are mentioned in tables 5.1 and 5.2. Videos for this evaluation were taken from video databases with free access, like YouTube or Pexels. All videos have different camera view with various object count, video resolution and FPS, which can affect neural network detector performance. Video samples are presented on 5.0, where obvious difference between videos can be noticed. As it is shown in tables **??** and **??**, PC1 performs best for both YOLO and RetinaNet, while PC2 has the worst performance. Jetson AVX Xavier has 2.5 times worse performance with YOLO and 4 times worse with RetinaNet comparing to PC1. The difference in performance of YOLO and RetinaNet on Jetson AVX Xavier is caused by frameworks used for computation. Darknet used with YOLO is optimized for utilizing of Tensor Cores in Jetson, while TensorFlow is not able to use Tensor Cores for faster matrix calculation. This problem can be solved by accelerating inference in TensorFlow with TensorRT. Unfortunately, in case of RetinaNet there are layers like *Exit*, *Switch*, *LoopCond*, *LogicalAnd*, *Enter* etc. in graph that are not fully supported in TensorRT, which means TensorRT optimizes only known layers and the resulting network does not gain any speed improvement. The possible solution for this problem could be writing custom layer converter for a network graph.

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

Table 5.1: PC1 technical specification

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

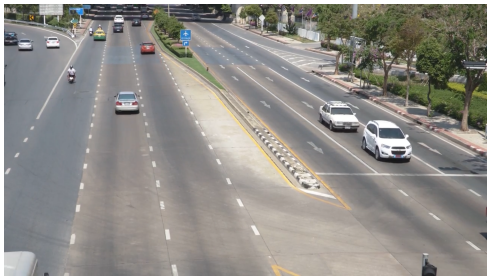
Table 5.2: PC2 technical specification

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

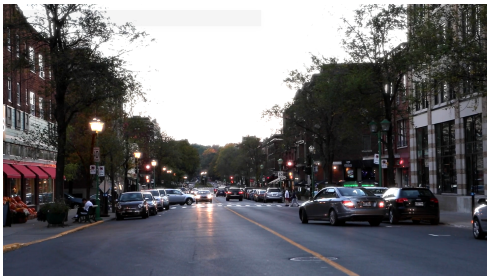
Table 5.3: Characteristics of video for inference time evaluation

Model	Machine	Inference time [ms]
YOLOv3 416x416	Jetson Xavier	123
	PC1	35
	PC2	175
YOLOv3 608x608	Jetson Xavier	139
	PC1	39
	PC2	208
RetinaNet	Jetson Xavier	200
	PC1	52
	PC2	228
squeezeDet	Jetson Xavier	25
	PC1	8
	PC2	24

Table 5.4: Inference time evaluation



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

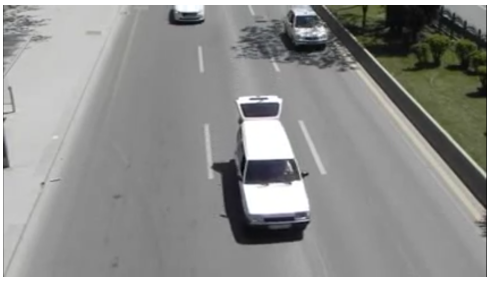


(b) : Cars Driving On Street.mp4

(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 5.0: Samples of used videos