Tampereen ammattikorkeakoulu

Real-Time Object Detection on Edge Devices

Joni Suominen

BACHELOR’S THESIS
January 2022

Degree Programme in ICT Engineering
Embedded Systems and Electronics

ABSTRACT

Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Embedded Systems and Electronics

Joni Suominen:
Real-Time Object Detection on Edge Devices

Bachelor's thesis 46 pages, appendices 0 pages
January 2022

The goal of this thesis was to research the performance of existing edge devices on state-of-art object detection models. Three different devices, with edge computing accelerators from different manufacturers, were chosen, namely NVIDIA Jetson Nano with integrated GPU, Raspberry Pi 4 B with Intel Neural Compute Stick 2, and Axis Q1615-LE Mk III security camera with Google EdgeTPU.

Latency and accuracy tests were performed for all devices, and in addition systems performance measurements, such as power consumption and utilization, were done for the Jetson Nano and Raspberry Pi. The object detection models chosen for the comparison were SSD-MobileNet-V2, YoloX, and EfficientDet, which all represent the latest advances in the field.

In the tests, only Jetson Nano was found capable of running YoloX and EfficientDet models. The fastest performance was achieved by the EdgeTPU, with only 8 ms of processing per image, while Jetson Nano and Neural Compute Stick 2 took 33 and 48 ms per image processed respectively. All the models also remained a good accuracy (> 90 %) post-conversion despite quantization.

In conclusion, all devices proved to be capable of object detection in real-time. As the devices had different form factors, connectivity, and computational units, different use cases can be applied for each device. Further performance improvements could be made by profiling the models to discover potential bottlenecks.

Keywords: edge computing, object detection, deep learning, convolutional neural networks, computer vision

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tieto- ja viestintätekniikka
Sulautetut järjestelmät ja elektroniikka

Joni Suominen:
Reaaliaikainen kohteentunnistus reunalaitteilla

Opinnäytetyö 46 sivua, joista liitteitä 0 sivua
Tammikuu 2022

Työn tavoitteena oli tutkia eri olemassa olevien reunalaskentalaitteiden suorituskykyä huippuluokan kohteentunnistusmalleilla. Kolme laitetta eri valmistajien reunalaskentakiihdyttimillä valittiin vertailuun, joita olivat NVIDIAn Jetson Nano integroidulla näytönohjaimella, Raspberry Pi 4 B Intelin Neural Compute Stick 2:lla ja Axis Q1615-LE Mk III turvakamera Googlen EdgeTPU:lla.

Viive- ja tarkkuustestit suoritettiin jokaiselle laitteelle, jonka lisäksi järjestelmän suorituskykyä, kuten virrankulutusta ja käyttöastetta, mitattiin Jetson Nanolle sekä Raspberry Pi:lle. Vertailuun valittavat kohteentunnistusmallit olivat SSD-MobileNet-V2, YoloX ja EfficientDet, jotka kaikki edustavat alan viimeisintä kehitystä.

Testeissä vain Jetson Nanon todettiin pystyvän ajamaan YoloX- ja EfficientDet-malleja. Nopein suorituskyky saavutettiin käyttämällä EdgeTPU-kiihdytintä, jolla pystyttiin prosessoimaan kuvia jopa 8 ms viiveellä, kun taas Jetson Nano ja Neural Compute Stick 2 vastaavasti vaativat 33 ja 48 ms prosessointia kuvaa kohden. Kvantisoinnista huolimatta, myös tarkkuus pysyi hyvällä tasolla (> 90 %) jokaisella mallilla.

Johtopäätöksenä pystyttiin toteamaan, että kaikki laitteet suoriutuvat reaaliaikaisesta kohteentunnistuksesta erinomaisesti. Koska laitteet erosivat kokoluokaltaan, liitännöiltään sekä laskentayksiköiltään, laitteita voidaan soveltaa eri käyttötarkoituksissa. Parannuksia suorituskykyyn saataisiin profiloimalla malleja, jolloin voitaisiin löytää mahdolliset pullonkaulat.

Asiasanat: reunalaskenta, kohteentunnistus, syväoppiminen, konvoluutioneuroverkot, konenäkö

CONTENTS

SPECIAL VOCABULARY

AI	Artificial Intelligence
ASIC	Application Specific Integrated Circuit
CNN	Convolutional Neural Network
CPU	Central processing unit
DNN	Deep Neural Network
FCOS	Fully Convolutional One-State Object Detector
FP16	Half-precision floating-point format
FP32	Single-precision floating-point format
FPGA	Field-programmable gate array
FPS	Frames per second
GPU	Graphics processing unit
IoU	Intersection over Union
INT8	8-bit signed integer
ML	Machine Learning
mAP	Mean average precision
MSE	Mean squared error
NMS	Non-max suppression
NCS 2	Neural Compute Stick 2
ONNX	Open Neural Network Exchange
OCR	Optical Character Recognition
ReLU	Rectified Linear Unit
RCNN	Region-Based Convolutional Neural Networks
SDK	Software development kit
SSD	Single Shot Detector
SVM	Support Vector Machine
SRAM	Scratchpad memory
VGG	Visual Geometry Group
VPU	Vision processing unit
YOLO	You Only Look Once

1 INTRODUCTION

Object detection has been a widely researched topic for multiple decades, due to its potential in solving numerous different tasks, such as traffic monitoring and medical imaging. In the past decade, state-of-art object detection solutions have moved from traditional hand-engineered solutions to deep neural network-based solutions. Modern object detection models are highly accurate, as they can reach over 99 % accuracy levels, effectively reaching human-level precision while being able to process tens of images per second.

Lately, optimization for edge devices, such as mobile phones and embedded systems, has also been a big trend. Processing data on the device instead of the cloud delivers faster response times and control over privacy to the user while offloading redundant computing from cloud providers. Typical chipsets found in edge devices, however, are not quick enough to process the vast amounts of data collected every second, so devices are equipped with separate edge acceleration hardware to speed up the computing.

This thesis was commissioned by Visy Oy, a leading vendor in AI (Artificial Intelligence) and OCR (Optical Character Recognition) -based solutions for access control and logistics in international ports and terminals. The goal was to find the optimal edge devices and neural network architectures for object detection at the edge. The system performance requirements were as follows:

- The system must perform object detection at a minimum of 5 frames per second
- The system must be real-time, meaning that there should not be many fluctuations in the latency
- The accuracy of the object detection model should remain at a sufficient level (> 90 %) after conversion

NVIDIA’s Jetson Nano, Raspberry 4 B With Intel’s Neural Compute Stick 2 (NCS 2), and AXIS Q1615-LE Mk III network camera were chosen for the comparison. Latency and accuracy were evaluated for each device-model -combination, and in addition, some system-level statistics were recorded from the Jetson Nano and Raspberry Pi, to gain more insight into the performance.

The readers of this thesis are first introduced to the theory behind machine learning and neural networks, to give a basis for understanding modern object detection models. After this, edge accelerators and optimization methods are briefly presented before measurement methods and results are shown. Finally, the results are analyzed and the optimal use case for each device is defined.

2 Theory

2.1 Machine Learning

As the number of devices connected to the internet increases, so does the amount of data available. As it would be difficult for a human to analyze such a large amount of potentially complex data, computer algorithms have been built to extract patterns and make new predictions from this data [1]. These algorithms generally fall under the field of Machine Learning (ML), which is a subset of Artificial Intelligence.

Machine Learning can be seen in practice as a phone might react to specific keywords when spoken or apply filters to your photos when it identifies the scene. These tasks seem very easy to perform for humans, but we would struggle to give computers precise instructions on how to perform them. Thus, it is better to give the computers some data and let them extract the meaningful patterns from it. [2, p. 22]

Machine Learning algorithms can be generally divided into three groups known as supervised learning, unsupervised learning, and reinforcement learning. The most common one is supervised learning, where we try to learn a mapping from some inputs $\boldsymbol{x}$ to outputs $\boldsymbol{y}$ given a labeled training dataset. [3, p. 2] A classic use case of the supervised model could be a regression problem where we try to find housing prices from house sizes in square meters. Supervised algorithms are of interest in this thesis and thus they are discussed more in the following chapters.

In unsupervised, learning we have a dataset as we did in supervised learning, but the labels are missing. This means that we are unable to make predictions such as in supervised learning, but instead, we wish to find underlying structure in the data, so-called latent variables, to then group our inputs together [4, pp. 26-27].

The final form of Machine Learning is reinforcement learning, where an algorithm learns by making actions in an environment. Each action the algorithm makes results in some state, which is then observed, and a corresponding reward is given depending on if the action resulted in a favorable state [5, p. 3]. Recent advancements in computation capability and research on reinforcement learning have shown great success, as programs such as Google's AlphaZero have mastered games like Go on their own [6].

2.2 Neural Networks and Deep Learning

Neural networks are algorithms that were made to mimic the human brain and its neurons. The simplest artificial neuron is called perceptron, which outputs a simple binary value ***y*** based on the dot product of real-valued input vector ***x*** and vector of weights ***w*** summed with bias ***b*** as seen in Figure 1 [7].

$$y = \begin{cases} 1\,, if\ w \cdot x + b \geq 0 \\ 0, \quad\quad otherwise \end{cases} \tag{1}$$

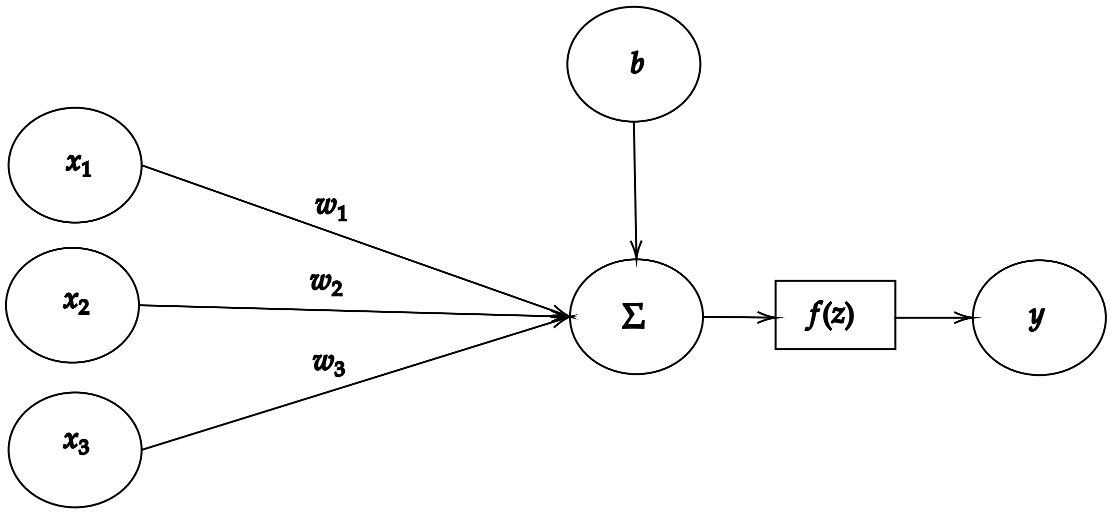

FIGURE 1. Simple perceptron model

This simple perceptron model has since then evolved further, partially from inspiration by neuroscience, engineers, and mathematicians. One way to increase the complexity of the model is to increase the number of neurons parallel in a layer, thus resulting in each neuron being able to extract different features from the input. [1, p. 169]

Another way to process sensory data, such as images, was adopted from neuroscience, where multiple layers of preprocessing occur before the input is displayed as an image by the visual cortex. Based on this it would be wise to add depth to the neural networks that mimic the human brain to process information more deeply. [8, p. 206] This is where the term Deep Neural Network (DNN) comes from. In a DNN, the input layer contains e.g., pixel data from an image, which is then processed by the hidden layers and finally forwarded to an output layer, which then contains higher-level information about the image [7].

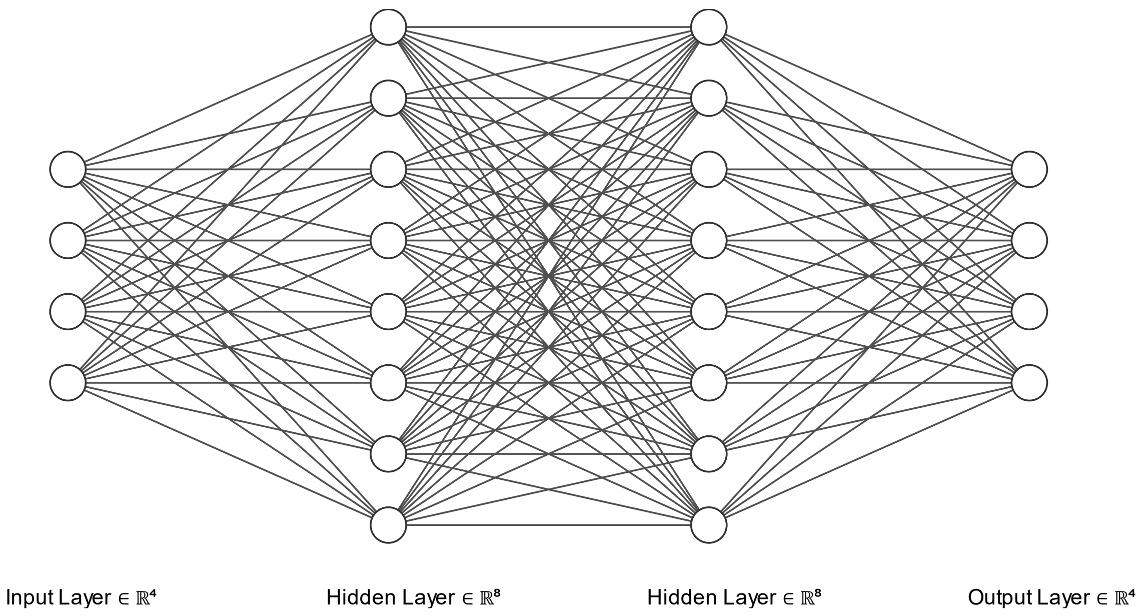

FIGURE 2. Deep Neural Network with 4 layers

2.2.1 Activation functions

Often, we want to solve tasks with non-linear relations or problems where the desired output is a probability, such as the probability of an image containing a cat. To solve these problems, activation functions are introduced to the network. Given a linear input

$$z = w \cdot x + b, \tag{2}$$

we can transform *z* into a non-linear output as follows

$$y = f(z) \tag{3}$$

, where *f(z)* can be any nonlinear activation function. [5, p. 180]

Activation functions in hidden layers have been researched a lot, and at first, the sigmoid and hyperbolic tangent (tanh) were used a lot. These days the Rectified Linear Unit (ReLU) is the most used as it's easy to compute, it does not have a problem with vanishing gradients due to being a piecewise function and in addition, it imitates the human neuron activations closely due to saturation at 0, activating only a subset of neurons at a time [9]. The formula for ReLU is defined as follows

$$f(z) = \max(0, z)\,. \tag{4}$$

At the output layer, the used activation function depends on the task we are trying to solve. For binary classification tasks, we could use the sigmoid activation function mentioned before, which outputs the probability of the input belonging to a class. A special case of sigmoid exists for multi-class classification tasks called SoftMax, which outputs a probability distribution of the classes. [10, pp. 5-8]

2.2.2 Cost function

To analyze how our neural network is performing, we use something called the cost function (or loss function) to measure how big of an error the network is producing in its outputs. Since ideally, we want our network to make perfect decisions, our goal is to minimize the cost. [11, p. 21] In supervised tasks, this would mean that for an input pair (***x*, *y***), where **y** is the ground-truth value, our model would be able to make a prediction ***ŷ***, that is the same as the ground-truth value when the input is ***x***.

One of the most common cost functions is the Mean Squared Error (MSE) cost function, which can be used for regression problems. The cost is usually averaged over *m* training examples and is defined as follows

$$MSE(y, \hat{y}) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2\,, \tag{5}$$

where *m* = number of training examples, y_i = ground-truth value for the ith training example and $\hat{y}_i$ = predicted value. [1, p. 108]

When our output is a vector of conditional probabilities, for example, if our output layer uses a SoftMax activation, it would be desired to penalize distinctly wrong outputs more, such as classifying an image of a cat as a bird. As we know that our input values are in the range of [0, 1], we can use something called cross-entropy loss, which minimizes the negative log-likelihood and is given by

$$L(y, \hat{y}) = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)\,. \tag{6}$$

Due to the logarithm, the function outputs a high error when it incorrectly gives a low probability to a class. [11, p. 110]

2.2.3 Backpropagation and Gradient Descent

As the objective of a machine learning model is to minimize the cost function, we must find a way to adjust the weights and biases to achieve this. A way to solve this is to compute the gradient of the cost functions, which is just a vector of partial derivatives with respect to the weights and biases of the network, informing us how much changing each unit in the network by a tiny amount would change the output. The gradient gives us the magnitude and direction of the steepest ascent, so we can iteratively move towards some local minima of the cost function by taking a step towards the negative direction of the gradient. The size of the step is usually given by the learning rate. Too high of a learning rate might not reach the minima of the cost function, and too small of a learning rate might take too long to find the minima, so the learning rate should be tuned correctly. The network can then be updated by an algorithm called Gradient Descent as follows

$$\theta = \theta - \eta \cdot \nabla C, \tag{7}$$

where θ = the vectorized parameters of the network, η = learning rate and $\nabla C =$ gradient vector of the cost function. [12, pp. 119-124]

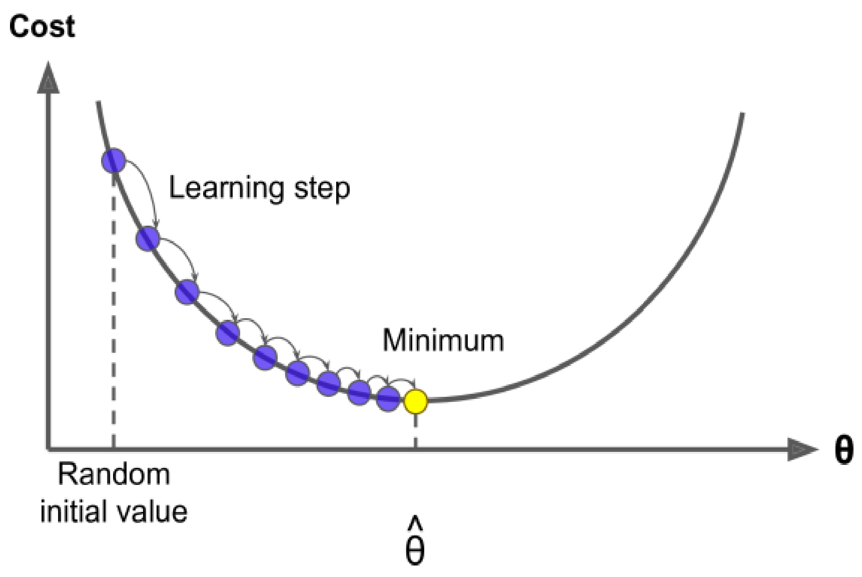

FIGURE 3. Gradient Descent algorithm. Copied from [12, p. 120]

The gradient is calculated using an algorithm called Backpropagation, which computes the partial derivates by moving backward in the neural network. The algorithm is preceded by inputting a training example to the neural network (forward propagating) and computing the cost function, after which the backpropagation algorithm is applied. This last step can be seen as propagating backward in the neural network graph due to the chain rule, which is why it is called Backpropagation. [7] For example, the error of the first weight from the input layer with respect to the cost function can be calculated as follows

$$\frac{\partial C}{\partial w_{11}^2} = \frac{\partial C}{\partial a_1^3}\frac{\partial a_1^3}{\partial z_1^3}\frac{\partial z_1^3}{\partial w_{21}^3}\frac{\partial w_{21}^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{11}^3} \tag{8}$$

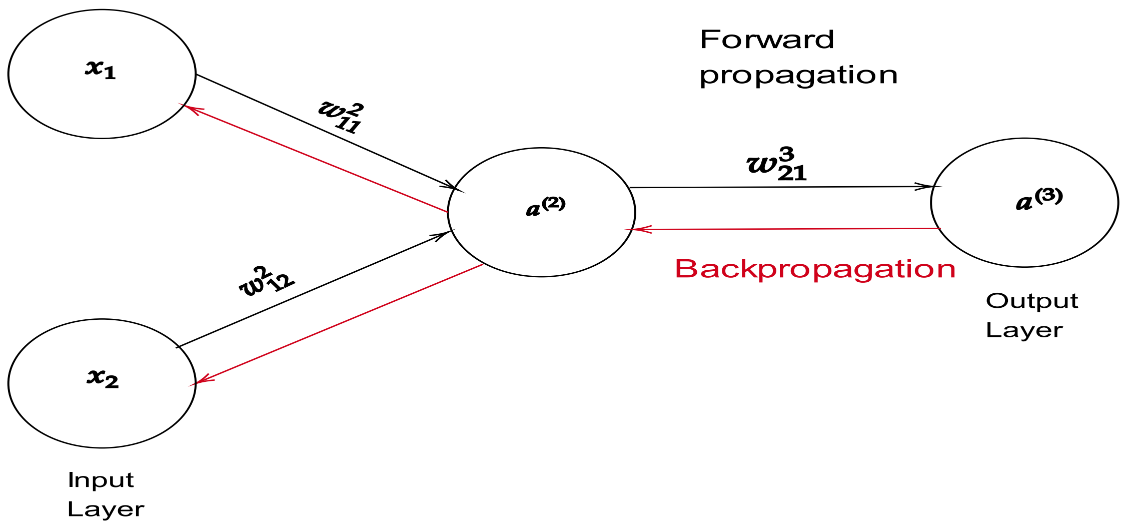

FIGURE 4. Neural network forward propagation and backpropagation

With the regular gradient descent algorithm, the weights are updated after forward propagating every sample and calculating the cost. This however means that it takes a long time to update the network. To speed up the training process, a stochastic gradient descent algorithm is usually applied. Instead of summing over the cost of all the training examples in our dataset, a batch of examples is randomly chosen from the training set and the gradient descent algorithm is applied from the average of the gradients. [7]

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are the state-of-art method for various computer vision tasks these days. They were developed to better extract information from data such as images, where spatial relations are present. CNNs had been researched since the 1960s, but the first landmarks were by Yann LeCun, who first applied backpropagation to convolutional neural networks in 1989 [13] and also released a paper on the LeNet-architecture [14], which laid the foundations for modern deep learning-based computer vision. CNNs later rose in popularity in 2012, after AlexNet won the ImageNet competition and became the state-of-art model for classification. This was largely due to the rise in computing power and utilizing graphics processing units (GPUs) in training. [15]

Compared to regular neural networks, instead of learning weights from each input neuron to each hidden layer neuron, CNNs learn a kernel which we slide and apply through an image in small boxes with a mathematical operation called convolution. This allows us to find patterns in images by not only looking at one pixel at a time but also its surroundings. Convolution is often implemented as cross-correlation in CNNs, which performs the same operation, but with the kernel flipped before sliding it over the input and is mainly used to simplify the mathematical equation. Cross-correlation is then defined as follows

$$(f * g)(i,j) = \sum_{a}\sum_{b} f(a,b)g(i+a,j+b)\,, \tag{9}$$

where *f* = discrete 2d input such as an image, *g* = 2d kernel, and *i* and *j* are the coordinates which convolution will be calculated at. [11, p. 229]

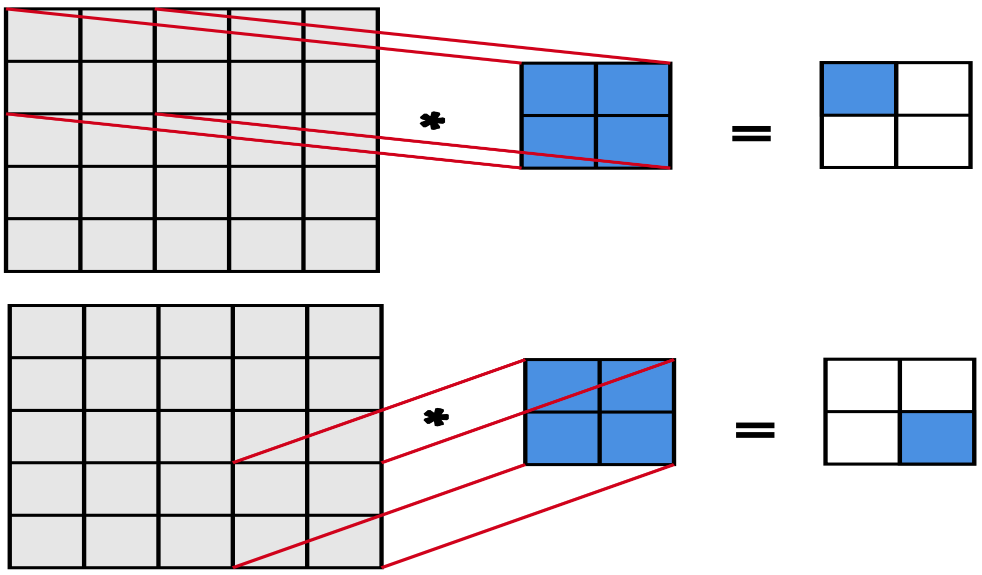

FIGURE 5. Convolutional operation in 2 dimensions

These convolutional operations along with the learned kernels are then used in convolutional layers, where the number of different kernels denotes the depth of the layer. These convolutional layers also help make the model size smaller than a regular dense neural network (DNN), which was seen in Figure 2. As the kernels are shared in the network, we save memory by only having to store weights depending on the size of the kernel instead of the size of the inputs. [1, pp. 335-338]

The different kernels then generate new images from their inputs, which are often called feature maps. Usually, the first convolutional layers of the network find some low-level features of the image, such as edges of the face from a picture of a human face. These are then combined to form larger parts such as an eye and a mouth and then finally an image of the human face with its edges highlighted at the last layer. [16] The feature maps are also passed into activation functions, as was the case with regular neural networks, to introduce non-linearities to the network. An example of a convolutional neural network and its feature maps can be seen in Figure 6.

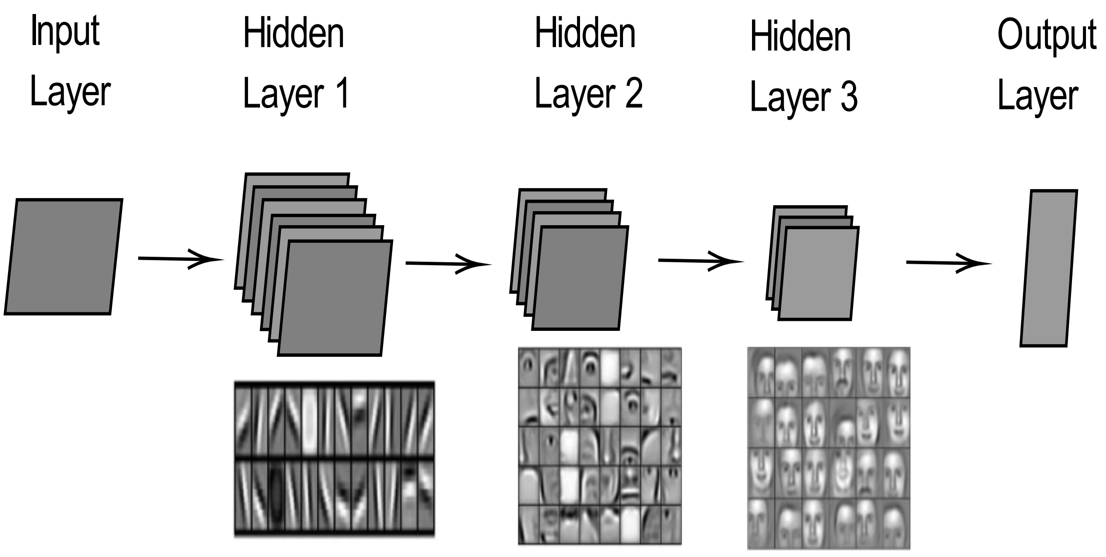

FIGURE 6. Convolutional neural network with hidden layer feature maps visualized. Layer visualizations copied from [16]

2.3.1 Padding and Stride

As we convolve an image with a kernel, our output keeps shrinking relative to the input, as was seen in Figure 5, causing us to gain less information about pixels on the edges after each layer. To avoid this, we pad the input with zeroes so that we can slide the kernel horizontally and vertically as many times as we have pixels in the original input, resulting in an output image that is larger than the original output and containing more information about the edges [1, p. 349].

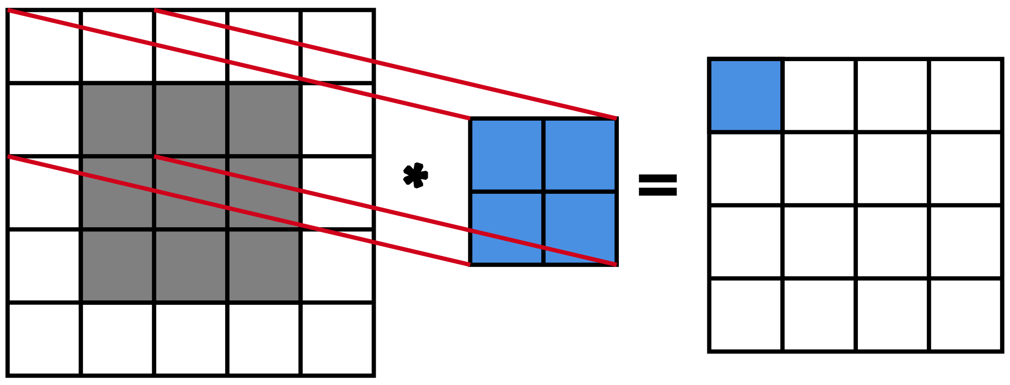

FIGURE 7. Input with padding results in a larger output feature map than in Figure 5

Sometimes it is also desired to reduce the dimensions of the input or speed up the computation, so we downsample by applying stride to the convolution operation. As was seen in Figure 5, we moved the kernel horizontally and vertically by one, which means we had a horizontal and vertical stride of one. To downsample a feature map, we could for example apply a horizontal and a vertical stride of three, as is seen in Figure 8 [11, pp. 237-238].

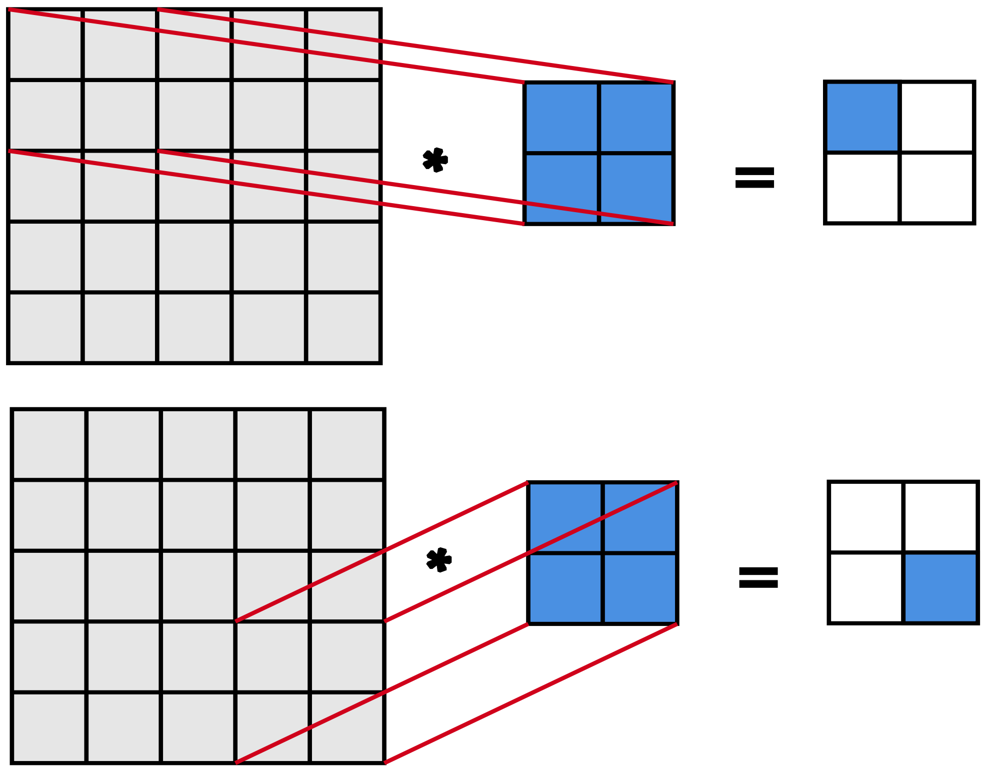

FIGURE 8. Convolutional operation with a horizontal and vertical stride of three

2.3.2 Pooling

Another way to downsample the input image is to apply a pooling layer between the convolutional layers, which has a size and stride like a convolutional layer but computes a function other than a convolution from its inputs, such as an average or a maximum. The most commonly used layer is a max pool layer, where the cell with the biggest value is always selected to the output as the max pooling kernel is slid over the inputs. This is useful, as usually, we try to find edges from the images without feature maps, and sharp features are usually related to high pixel values, thus reducing the input dimension, and finding desired features without having to learn any weights. Another benefit of max pooling is, that it offers some level of translate invariance, meaning that small changes in the location of the pixel can still result in the same output. [12, pp. 442-444] The max pooling operation is shown in Figure 9.

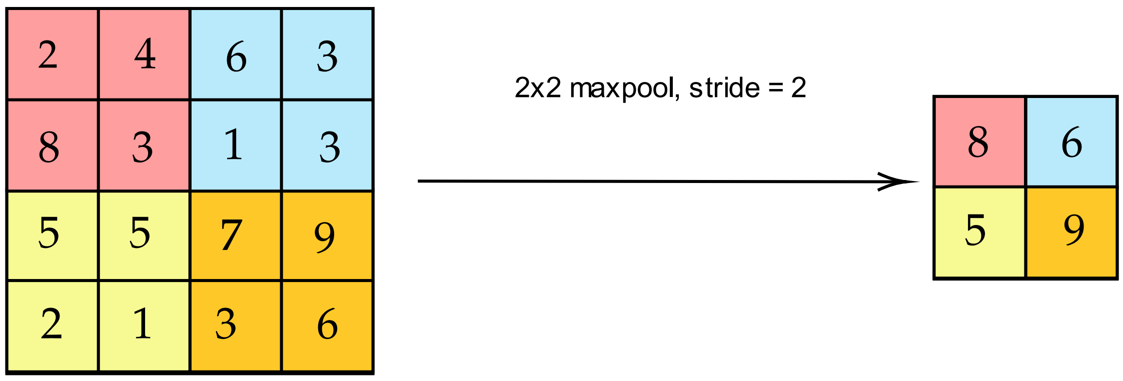

FIGURE 9. 2x2 max pooling with stride = 2

3 Object Detection at Edge

Object detection is a task of locating and classifying an object or multiple objects in a scene. This can be used e.g., for detecting pedestrians and traffic signs in self-driving cars or traffic behavior in intersections. A real-world deployment example of the latter could be to install a security camera near the intersection, which would then stream the images to a remote cloud center to process the data. This has been the de facto standard for the past decade as cloud computing has become increasingly popular, but it now has its concerns due to vast amounts of incoming data and personal privacy. To deal with these issues, edge computing has grown to be a popular solution alongside cloud computing. The data can be pre-processed at an edge, which is a computing node between the sensor (such as a camera) and the cloud, as is shown in Figure 10.

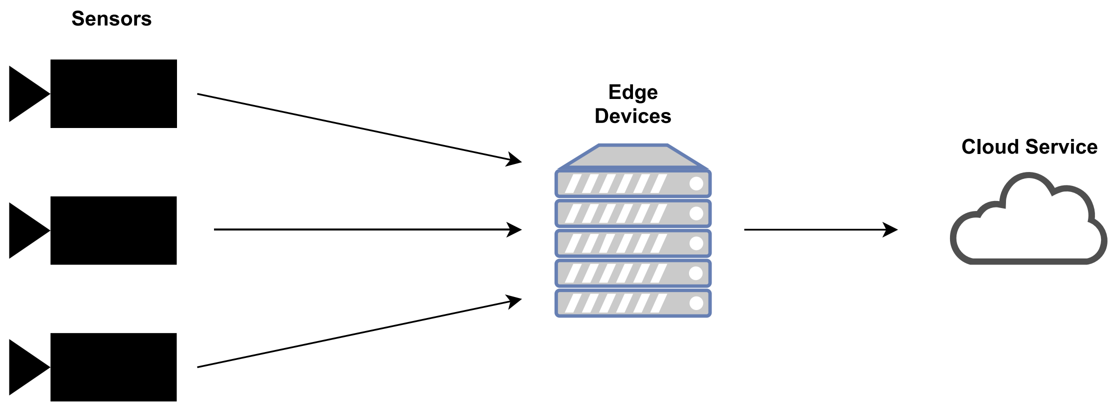

FIGURE 10. Simple edge computing network

The first benefit of this is latency and bandwidth reduction, as we can perform computing ideally in the same network as the devices that operate on this data, reducing the overhead of transferring the data forth and back, which could be crucial in applications such as autonomous cars. The second benefit is being able to process the data before it reaches a third-party cloud environment, like anonymizing faces or hiding unique identifiers from sensitive data. Thirdly we can save storage at large-scale datacenters by filtering data not suitable for postprocessing at the edge. [17] The need for pre-processing and filtering becomes more evident by looking at Cisco's numbers, which reported that M2M devices, such as surveillance cameras and smart meters, would account for 50 % of devices and connections by 2023 and were growing the fastest by far [18].

3.1 Object Detection

In object detection, we want to be able to input an image to an algorithm, which will then return us the number of detections, bounding boxes that tell us the location of the objects within the image, and classes telling us what kind of objects we have detected and the confidences that an object belongs to the class.

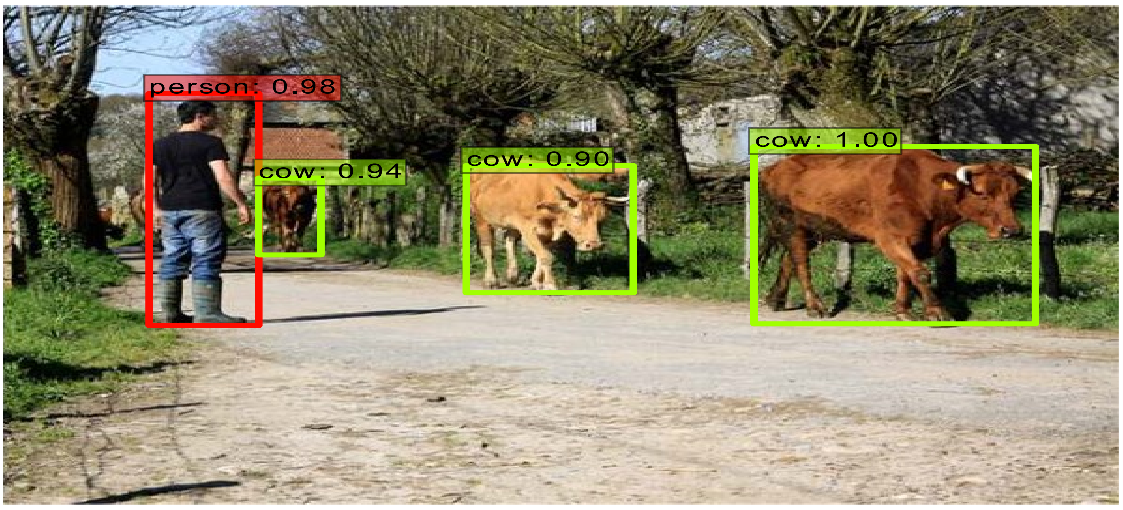

FIGURE 11. The output of a typical object detection model. Copied from [19, p. 13]

Object detection has exploded in the last decade due to the growth of convolutional neural networks but has been available for some detection tasks, such as face detection, for nearly two decades. One of the most popular algorithms was and still is the Viola-Jones algorithm, which could detect faces at 15 FPS back in 2001 [20].

3.1.1 Object Detection Architectures

The modern deep-learning-based object detection models started showing promise in 2014 with the release of the RCNN-model. It however was too slow for the majority of real-life use cases, as it consisted of 3 different stages with both convolutional neural network and support vector machine (SVM) classifiers to train. [21]

Recent object detection models are generally single-shot detectors, meaning we only must pass the input image through one neural network. These architectures usually consist of two parts, a base network, and a classification network, with its final classification layers cut off to generate feature maps at different scales. The classification layers are then replaced by a detector network, which generates appropriate bounding boxes and classifications from these feature maps. Figure 12 shows an SSD model [19] with a VGG-architecture base network.

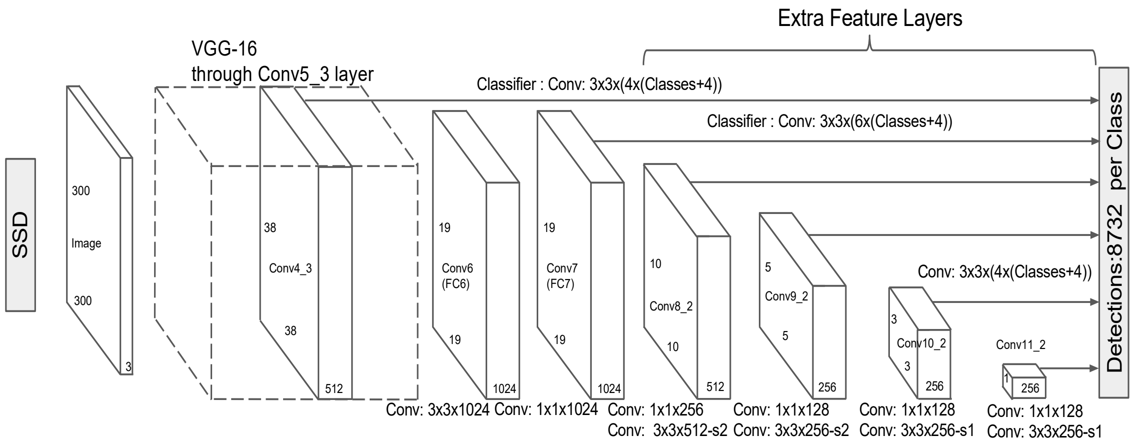

FIGURE 12. SSD Detector with a VGG base network. Copied from [19, p. 4]

The larger feature maps, such as the 38x38 layer in Figure 12, attempt to find small objects by dividing the image into smaller segments, while the smaller feature maps try to locate larger objects. As the feature maps often shrink information about the image into more dense representations, each pixel in a feature map can be seen as representing a larger group of pixels in the original input, and thus forming a grid of cells over the original image. An example of this can be seen in Figure 13, where a 7x7 feature map is laid over the input image.

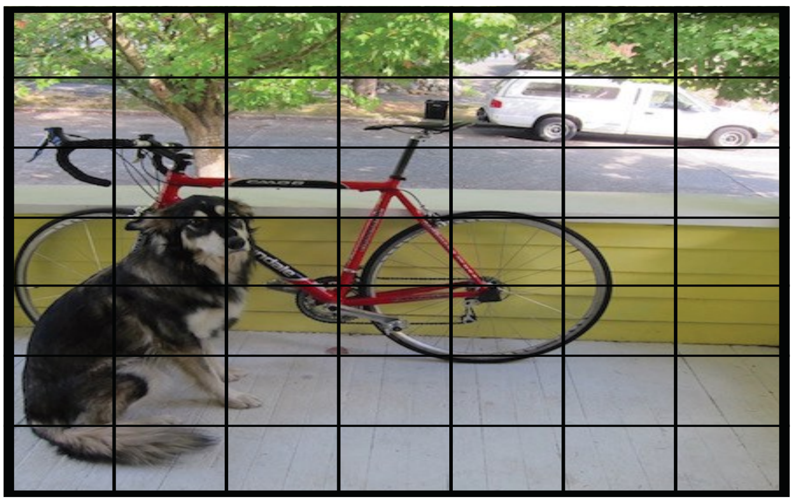

FIGURE 13. 7x7 feature map laid over an input image. Copied from [22]

Modern object detector models can further be divided into two categories, anchor-based, and anchor-free solutions. The anchor-based solutions, such as SSD [19] or EfficientDet [23], try to predict *n* bounding boxes for each cell in a feature map by calculating offsets for given a set of hand-engineered default (or anchor) boxes at different scales. The best bounding boxes are then chosen with an algorithm called non-max suppression (NMS). The anchor-free solutions, such as YoloX [24] or FCOS [25], don't require manually finding optimal anchor boxes for prediction, and only one prediction is selected for an object even at multiple feature maps levels, as overlapping is further reduced by methods such as setting limits for bounding box sizes at each level.

As an object detection model's goal is to predict a correct class and a correct location for each object in the scene, the cost function is usually a sum of the classification error and localization error (regression task). For example, in the Fast R-CNN model, the following loss function was used

$$L_{total} = L_{cls} + L_{loc}, \tag{10}$$

where L_{cls} = SoftMax loss, used for measuring the classification error, and L_{loc} = smooth L1 loss, used for measuring the localization error. [26]

3.1.2 Intersection over union and non-max suppression

In object detection, a prediction is labeled as a true positive, if its bounding box and the ground-truth prediction's bounding box overlap over a certain threshold. This metric is called intersection over union (IoU, Jaccard index) and is calculated by dividing the overlapping area by the union area of the two bounding boxes

$$\mathrm{IoU(A, B)} = \frac{|A \cap B|}{|A \cup B|}, \tag{11}$$

where *A* = area of overlap and *B* = overlapping area. [27]

$$IoU = \frac{|A \cap B|}{|A \cup B|} =$$

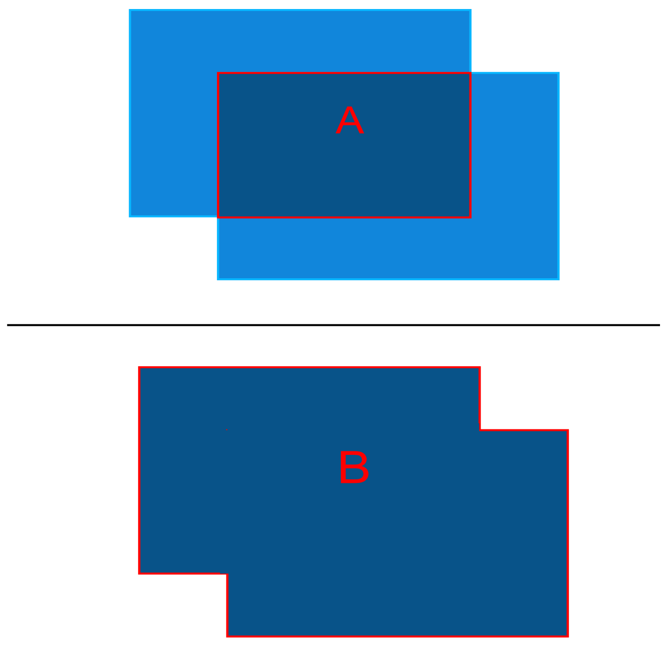

FIGURE 14. Intersection over union

Often the state-of-art models generate up to hundreds of predictions per input image and many of them are overlapping, trying to detect the same object from a different position. Non-max suppression attempts to speed up the processing by making a list of predicted bounding boxes sorted by their confidence, selecting the first one as a reference, and dropping the other bounding boxes in a list if their IoU was over a chosen threshold with the reference bounding box. This process is repeated for all the bounding boxes left in the list, with the ith bounding box chosen as a reference on ith comparison round. [11, pp. 590-593]

3.2 Deep Learning Accelerators

Due to the increasing need to perform tasks involving machine learning algorithms in real-time, especially on devices with limited CPU computing capability, chips optimized for edge inference are required. The current deep learning accelerators can be roughly divided into three categories, ASICs, FPGAs, and GPUs.

Both ASICs and FPGAs are application-specific circuits, which provide consistent low latency and power consumption, but have increased complexity in programming [28, p. 5]. The main differences between the two are, that FPGAs provide flexibility being reprogrammable, while ASICs are more optimized, but single-purpose after production. Current state-of-art deep learning accelerators are ASIC chips such as Google Coral’s EdgeTPU [29] or Intel’s Movidius series VPUs (Vision processing unit) [30].

GPUs are also very popular in deep learning tasks, due to their easy programmability, but the trade-off can generally be seen in power consumption and latency [28, p. 5]. As they usually have lots of cores, they are well suited to parallelizing tasks, such as matrix multiplications and convolutions, which are widely used in neural networks. GPUs can also be utilized in a multitude of ways in a computer vision pipeline, for example in video decoding and encoding in addition to object detection. An example of a Deep Learning Accelerator utilizing GPUs are the NVIDIA Jetson series embedded systems [31], which feature fully-fledged Linux systems for diverse use cases.

3.3 Model Quantization

To run deep learning algorithms on edge, the models have to be compressed into lighter versions. Typically, each hardware vendor, such as NVIDIA, has its own deep learning toolkit which compiles an existing model, trained in e.g., TensorFlow, to another format that better utilizes the hardware in the edge device through means like quantization and hardware-optimized layers.

In the case of neural networks, models are typically trained in 32-bit floats (single-precision, FP32), but when doing inference at the edge, the model takes too much space and inference is too slow or unreliable. By quantizing the neural network’s biases, weights, and activations to a lower bit-width, the model theoretically consumes less memory and runs faster proportional to the reduced memory usage [32]. However, as the model is run on lesser precision, the model accuracy might suffer significantly, which is why the trade-off should be evaluated when deploying new models. Usually, quantization can be done to 16-bit floating points (half-precision, FP16) or 8-bit signed (INT8) without losing too much accuracy.

Many methods exist to quantize a model from e.g., a 32-bit floating-point to an 8-bit floating-point model. Some of the most common algorithms are affine quantization and scale quantization, the latter being a special case of the first due to forcing a zero-point of 0 and thus being symmetric with a range of [-127, 127]. In both cases, a scale value is calculated to transform the input variable to the range of the output variable and then rounded to the closest integer. To calculate the scale factors correctly, calibration is often applied to the network to find the optimal parameters. Calibration requires a small dataset, which will be run through the network to find the distribution of activation values and adjust them accordingly. [33] Figure 15 shows the effect of losing precision when quantizing 32-bit floats to 8-bit integers.

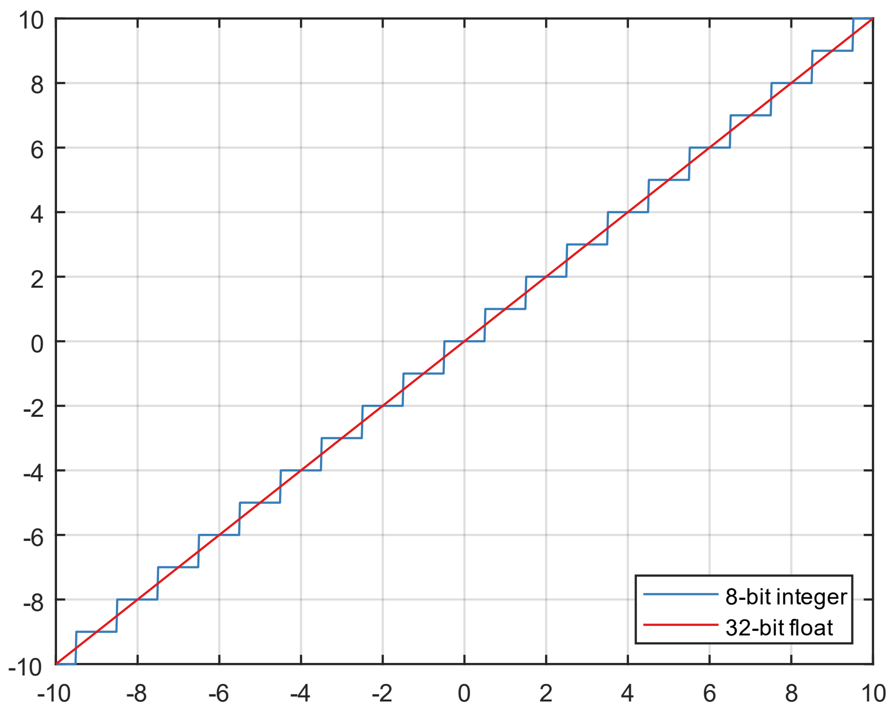

FIGURE 15. Effect of quantizing 32-bit floats to 8-bit integers

4 Test setup

This thesis was done at Visy Oy to research possible edge deployment solutions for their existing deep learning-based solutions. Several different deep learning models were evaluated on three different hardware architectures and a model-hardware combo was deemed as appropriate for real-time object detection if FPS of 5 was achieved and the latency was consistent, and the model maintained a modest accuracy.

4.1 Hardware

Three different devices with edge accelerators from the major hardware vendors Google, Intel, and NVIDIA were chosen based on their pricing, availability, and expected performance based on specifications. The three devices chosen were NVIDIA's Jetson Nano [34], Raspberry Pi 4 B [35] with Intel Neural Compute Stick 2 [36], and Axis Q1615-LE Mk III network camera with a built-in Google Coral EdgeTPU [37]. The key specifications are listed in Table 1.

TABLE 1. Key specifications of hardware used in test scenarios

	NVIDIA Jetson Nano	**Raspberry Pi 4 B + Intel NCS 2**	**Axis Q1615-LE Mk III**
DL Accelerator	NVIDIA 128-core Maxwell GPU	Intel Movidius Myriad VPU	Google Coral EdgeTPU
CPU	ARM A57, Quad-Core @ 1.43 GHz	ARM A72, Quad-Core @ 1.5 GHz	ARTPEC-7
DL Accelerator Memory	4 GB (shared)	500 MB (NCS 2)	8 MB SRAM
Computing Speed	472 GFLOPS	4000 GFLOPS	4000 GFLOPS
SDK	TensorRT	OpenVINO	TensorFlow Lite
Other	Good programma-bility, strong video processing capa-bilities	Works on any host PC	DL acceleration through Axis' ACAP SDK
Price	~110€	~80€	~1250€

Out of the three test devices, the NVIDIA Jetson Nano and Intel NCS 2 share the most in common. The CPU found in Raspberry Pi is a slight upgrade over the one found in Jetson Nano, but the GPU in Jetson Nano far outperforms the Broadcom BCM2711 chip found in Raspberry Pi in computer vision tasks, such as video decoding and encoding. In turn, the Raspberry Pi doesn’t have to share its memory with the VPU, while the Jetson Nano’s CPU and GPU share the same memory, giving the Raspberry Pi an advantage in memory availability. The Axis in turn has its own SoC with limited details available. The EdgeTPU included with the Axis only has 8 MB of SRAM (Scratchpad memory) built-in, but as the models are optimized to small sizes with TensorFlow Lite, in many tasks it is sufficient. If the model requires more memory, the parameter data can be fetched from external memory, but as a downside it introduces latency to the execution.

With a huge price difference between the Axis security camera and the two other devices, one also has to account that a separate camera has to be included within the pricing when deploying external computing units. The Axis camera provides easy deployment, as all the processing units are already encapsulated.

4.2 Model conversion

Due to each device being run on different accelerators and utilizing different SDKs, separate conversion processes had to be employed. The process typically consists of converting the trained model to some open-source intermediate representation.

4.2.1 TensorRT

TensorRT is NVIDIA’s SDK for accelerating deep learning models and tools for inference [38]. It automatically optimizes each part of the neural network to use optimal layers and algorithms for speed and memory usage and allows converting models from single-precision to half-precision or signed 8-bit integers through post-training quantization.

The workflow to migrate existing deep learning models, trained on platforms such as TensorFlow or PyTorch, requires converting the model to an intermediate representation. The suggested platform to use is ONNX (Open Neural Network Exchange) [39], which is an open-source machine learning framework widely used in the industry to optimize and represent machine learning models. This can then be used to convert the model to a serialized TensorRT engine for inference. TensorRT conversion workflow can be seen in Figure 16.

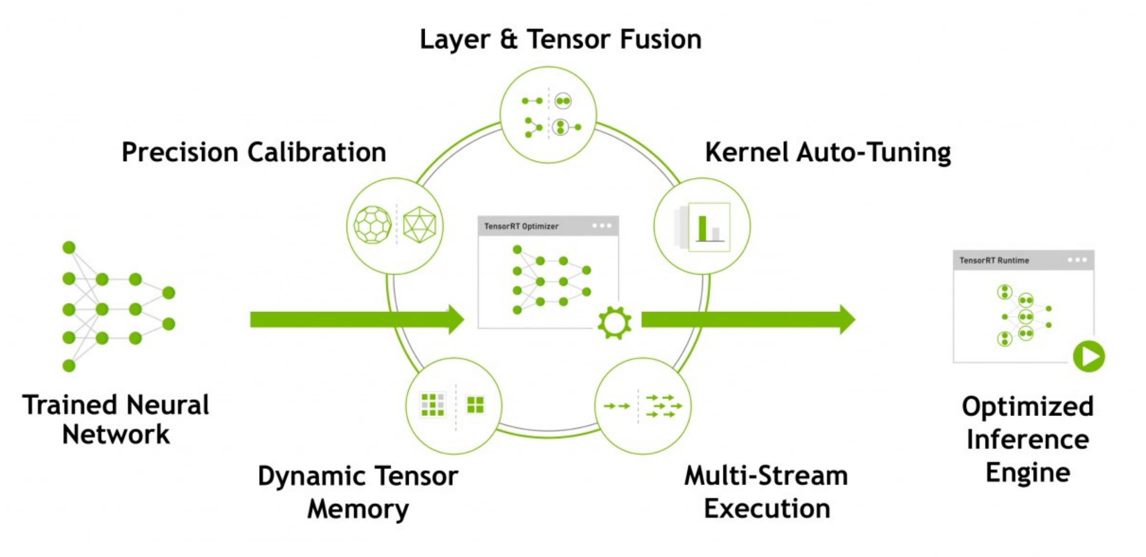

FIGURE 16. TensorRT conversion workflow. Copied from [40]

TensorFlow and NVIDIA also have a joint project called TF-TRT (TensorFlow-TensorRT), which allows converting parts of a TensorFlow model graph to TensorRT nodes [41]. This enables models to run as if they were regular TensorFlow models, but with highly hardware-optimized nodes to make inference faster than regular TensorFlow inference. Pure TensorRT engines are generally recommended, as they optimize all the layers for a specific hardware model.

4.2.2 TensorFlow Lite

TensorFlow Lite is Google’s open-source machine learning toolkit for mobile and IoT devices [42]. It supports on-machine learning and inference optimization for different platforms with limited computing and memory resources. Optimization technologies such as post-training quantization, quantization-aware training, and pruning are also available off-the-shelf. TensorFlow Lite models can be further optimized for accelerators, such as Google's EdgeTPU through Google Coral tools [43]. The workflow for optimizing a TensorFlow model to EdgeTPU can be seen in Figure 17.

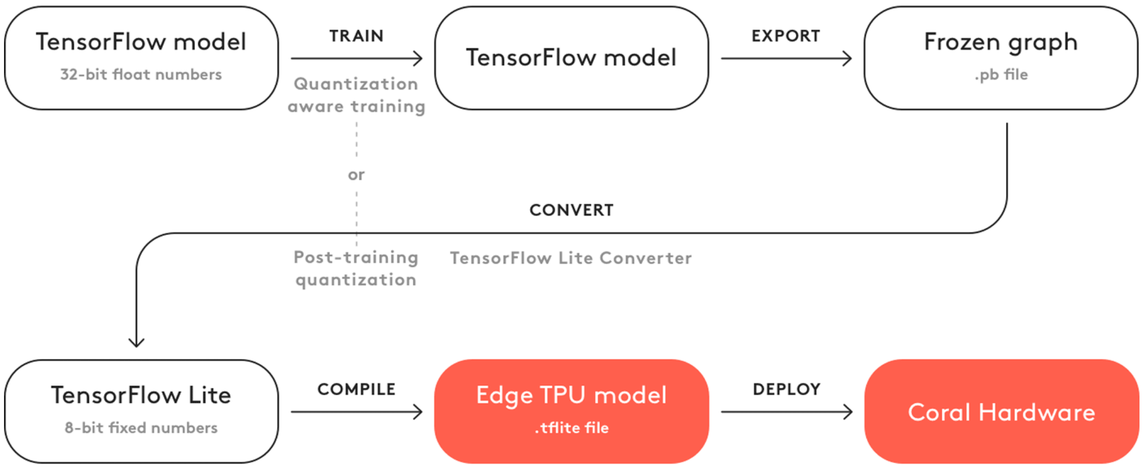

FIGURE 17. EdgeTPU optimization workflow. Copied from [44]

4.2.3 OpenVINO

OpenVINO is Intel's open-source toolkit for optimizing machine learning models to different Intel platforms, such as the Myriad VPU or Intel CPUs [45]. The toolkit automatically transforms and optimizes layers from different machine learning frameworks into Intel platform compatible formats. OpenVINO does not support training but supports a variety of post-training optimization tools, such as quantization and pruning.

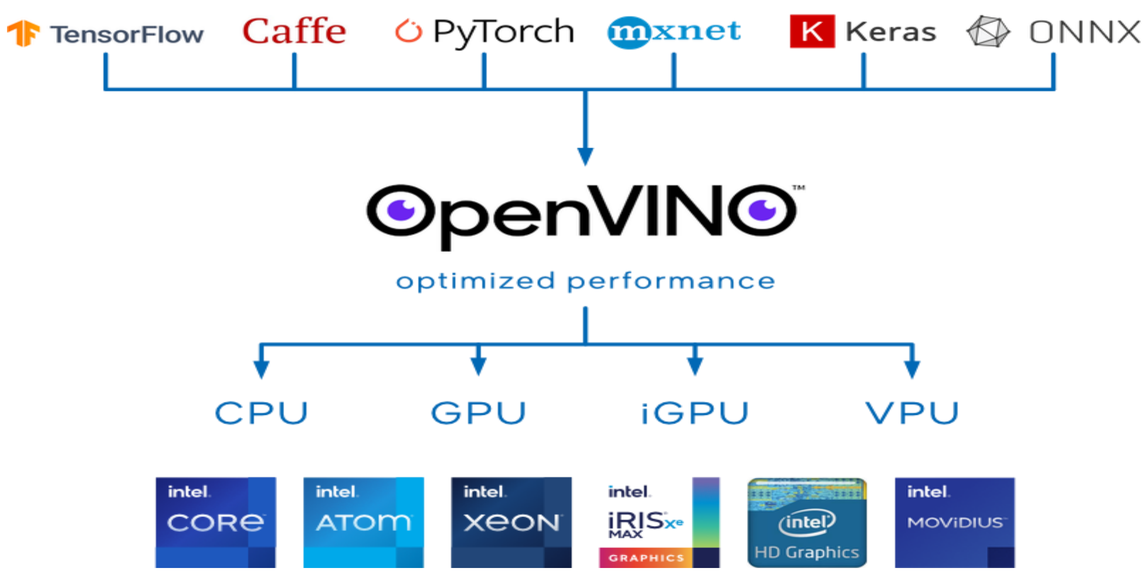

FIGURE 18. OpenVINO workflow. Copied from [45]

4.3 Measurement setup

Two different measurements were conducted, one for accuracy, and one for system stress levels and latency. The accuracy and latency test were performed for all devices, but the system stress levels were not measured from the Axis camera, due to it being a system with less control over as a programmer.

4.3.1 Accuracy evaluation

The accuracy of the models was evaluated using the mAP-metric, which can be calculated as the area under the precision-recall curve. In object detection, a prediction is measured as correct or true positive, if its IoU is over a predetermined threshold with the ground-truth box. If the IoU is less than the threshold, the bounding box is determined as incorrect and a false positive. Otherwise, if there was no object detected or the class was wrong, when there was a ground-truth box, the prediction is labeled as a false negative. Precision measures, what portion of the predictions labeled as positive were actually correct, while recall measures how well we found all the positive samples in the dataset. [12, pp. 92-98] The precision and recall values can then be taken at multiple IoU thresholds and plotted as is seen in Figure 19.

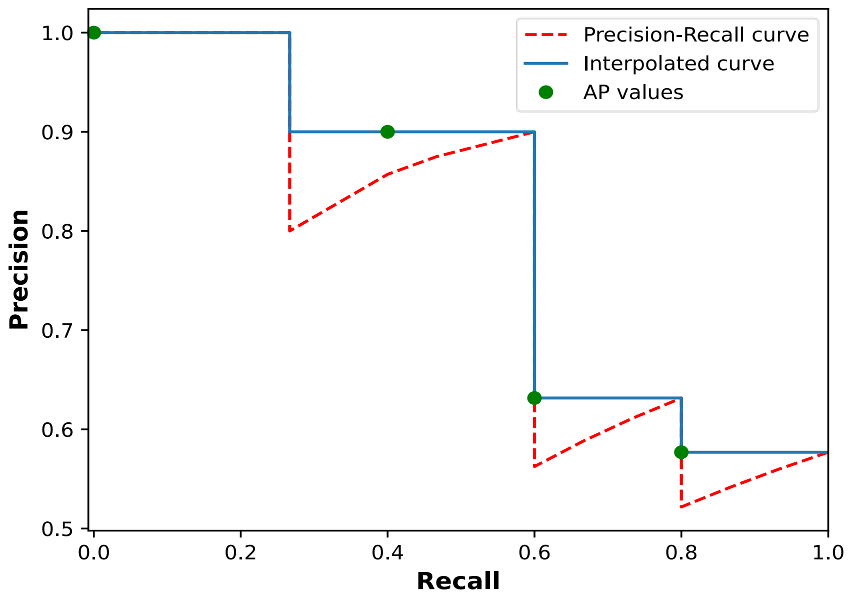

FIGURE 19. 4-point interpolated Precision-Recall curve

Finally, the mean average precision (mAP) can be computed. To ease the computing, the area under the curve is often calculated by taking evenly spaced recall values and interpolating the precision value at each of the taken values into the highest value to the right, as is seen in Figure 19. The equation is given as follows

$$\mathrm{mAP} = \frac{1}{n}\sum_{r=0}^{n} P_{interp}(r)\,, \tag{12}$$

where n = number of samples to interpolate and P_{interp} is a function giving the interpolated value at r. [46, p. 313]

The accuracy in the measurements was computed on a dataset of 997 license plate images, using pycocotools Python API from the authors of the COCO dataset [47]. The program uses 101-point interpolation to compute two mAP metrics. The first one calculates the mAP score at an IoU threshold of 0.5 and is often denoted mAP@0.5. The second one is calculated as an average of mAP scores at IoU thresholds from 0.5 to 0.95, with a spacing of 0.05, and is denoted mAP@0.5:0.95.

4.3.2 Stress test

A stress test was performed on devices to see, how consistent the systems were at performing inference and image processing over 10 minutes with a batch size of one. On the Axis camera, only the latency was monitored. On other devices, the monitored statistics were the latency, CPU/GPU/VPU temperature, CPU/GPU clock frequency and utilization, and power consumption. On Jetson Nano and Raspberry Pi, the connected peripherals were a monitor, keyboard, mouse, and an ethernet cable, but no other software was running.

On Jetson Nano, the Python package jetson-stats [48] was used to monitor all the statistics, which provided an easy-to-use API for NVIDIA's tegrastats [49]. The measurements were performed by inputting images with a batch size of one to the model and latency was measured from the inference engine call to the end of post-processing, such as non-max suppression, if necessary.

In these experiments, OpenCV was used to read images from the disk and process them into appropriate sizes for the neural networks before each inference, but the time spent pre-processing was not taken into consideration in inference speed. This, however, meant that roughly 30 milliseconds of potential inference time were used for processing on the CPU between each inference on the Jetson Nano and Raspberry Pi.

On Jetson Nano, The CPU and GPU utilization were measured as average usage over the polling period (500 ms), while others were instantaneous values also taken every 500 ms. The other statistics, such as power consumption, were queried more as an indicative value rather than the absolute truth, as there was no certainty the peak values would be found without hindering the system with too high polling rates. In theory, though, power consumption of at least 10 W should be expected on the Jetson Nano, as such power supply is required by the manufacturer to run the board at maximum speed.

The same measurement method was utilized on the Raspberry Pi and NCS2, however, the statistics were queried through Raspberry Pi’s built-in vcgencmd, psutil [50], and Intel’s Inference Engine API [51]. These provided the CPU utilization, clock frequencies, and CPU and VPU temperatures, but information such as VPU utilization and system power consumption were missing. The power consumption was measured by powering the Raspberry Pi through the GPIO pins during inference, giving a good indicator of how much power the system consumes in total.

5 Results

The results are divided into two sections, with the first one showing the model performances on each device, such as latency, accuracy, and power consumption, and the second one going further into Jetson Nano and Raspberry Pi by analyzing the system-level performance, such as utilization and temperature. Measurements were conducted at ambient temperature (~ 22 °C), and 10 rounds of warm-up inference were performed as initial inferences usually take significantly longer.

On Jetson Nano, the model performance tests were conducted in FP32 as opposed to FP16, as the differences were noticed to be insignificant. The other devices only supported one quantization level, FP16 on NCS 2 and INT8 on EdgeTPU. Three different object detection architectures were chosen as candidates for comparison, namely EfficientDet-D0, SSD-MobileNet-V2, and YoloX-nano.

5.1 Model performance

SSD-MobileNet-V2 was the only model to successfully run on all the devices, and its performance is shown in Table 2. Input images of size 300 x 300 pixels were inputted to the model with a batch size of one. On Jetson Nano, the model was running as a TF-TRT graph instead of pure TensorRT, due to layer incompatibilities.

TABLE 2. Performance of the SSD-MobileNet-V2 model

Device	**Latency (ms)**	**mAP@ 0.5**	**mAP@ 0,5:0,95**	**Power Consumption (W)**
Jetson Nano	51,9	98,7	40,5	5,04
Raspberry Pi + NCS 2	48,8	98,8	40,7	5,25
Axis + EdgeTPU	8,0	94,0	33,2	-

As can be seen in Table 2, the Axis camera equipped with EdgeTPU achieved the highest speed by over six-fold, while the other two devices measured very similar results in both latency and accuracy. There, however, exists a tradeoff as quantization of the EdgeTPU model dropped accuracy by 4,0 %. In power consumption, similar levels were seen between the Jetson Nano and Raspberry Pi.

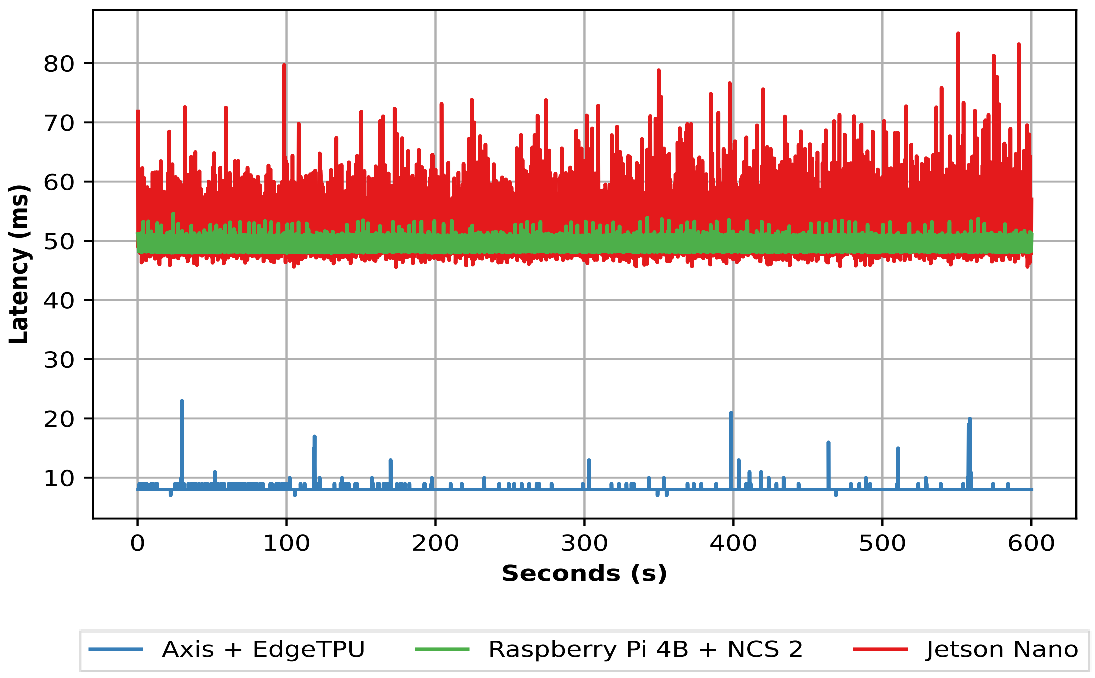

FIGURE 20. Response time graphs on SSD-MobileNet-V2

Comparison of the response times between each device running SSD-MobileNet-V2 was also measured, and they can be seen in Figure 20. It becomes apparent, that the ASIC-based devices have more consistent response times and comply better with the requirements of a real-time system. The few peaks in EdgeTPU response time can likely be attributed to a few of the operations being run on the CPU, which might add inconsistency due to data transferring and conversion. Similarly, as the model was run with the TF-TRT engine on Jetson Nano, inconsistent performance was expected.

The other models, EfficientDet-D0 and YoloX-nano, failed to compile for both NCS 2 and EdgeTPU, but successfully ran on Jetson Nano using pure TensorRT. The model results on Jetson Nano can be seen in Table 3. EfficientDet-D0 was run with an input size of 256 x 256 pixels and YoloX with an input size of 416 x 416 pixels. Non-max suppression was also performed on the YoloX post-inference due to excess bounding boxes.

TABLE 3. FP32 model performance on Jetson Nano

Model	**Latency (ms)**	**mAP@ 0.5**	**mAP@ 0,5:0,95**	**Power Consumption (W)**
EfficientDet-D0	53,2	97,6	51,5	5,75
YoloX-nano	32,6	91,6	27,9	5,4
SSD-MobileNet-V2	51,9	98,7	40,5	5,04

As can be seen from Table 3, the EfficientDet model is slower and consumes more power than both of the counterparts, and at the same time has a lower accuracy than the SSD model seen in Table 2. The YoloX-nano achieves the lowest latency out of all models run on Jetson Nano, but also the lowest accuracy by far, as it's 7,1 % lower than the SSD-MobileNet-V2. The low latency of the YoloX model can possibly be attributed to the anchor-free property, which reduces the overall need for data transferring and helps with the potential bottleneck of sharing memory between GPU and CPU on Jetson Nano.

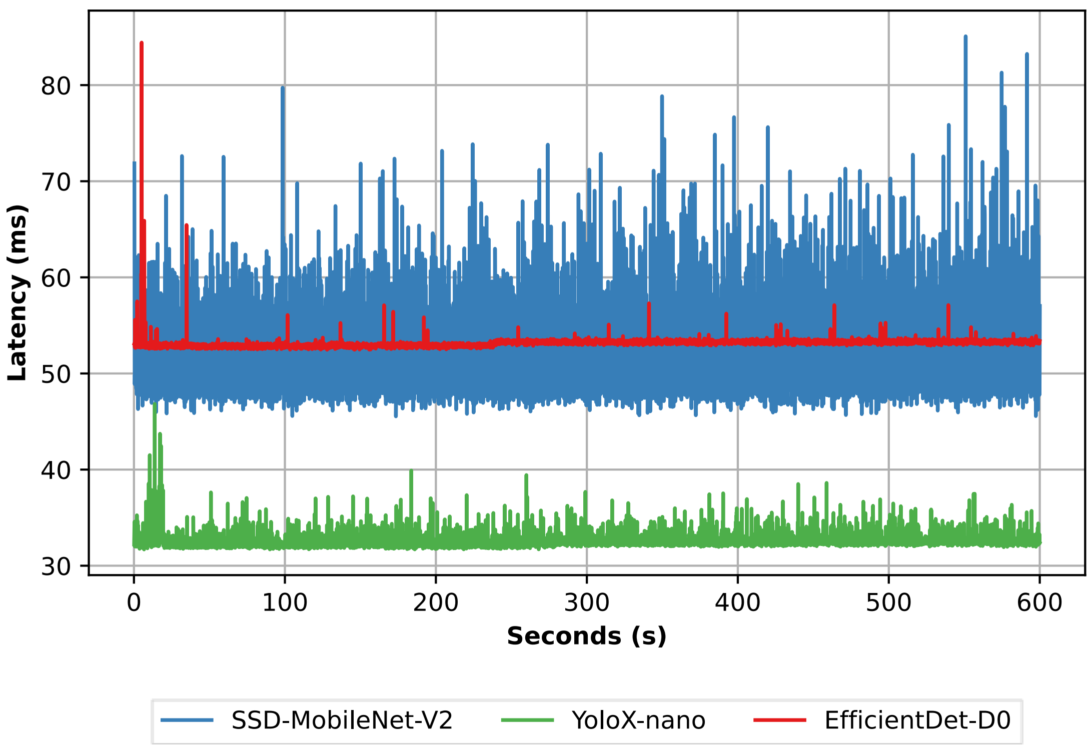

FIGURE 21. Model performances on Jetson Nano

Figure 21 shows the response time graph of all the modes ran on Jetson Nano. YoloX-nano and EfficientDet-D0 show the most promise in latency stability, as both are run using a pure TensorRT engine, which enables peak optimization for the GPU in use.

5.2 System performance

In this section, the system-level information of Jetson Nano and Raspberry Pi 4 B are further analyzed. The tests were conducted using measuring software described in section 4.3.2. Both devices were equipped with a heatsink and also were cooled down between measurements.

5.2.1 Jetson Nano

Jetson Nano, being an all-in-one computer with its GPU as the neural network accelerator, provided easily accessible insight into different parts of its system. As most of the computing is offloaded to the GPU during inference, the utilization levels can be looked at in addition to the latency to get some useful insights. A graph of GPU utilization and average latency for each model at two quantization levels can be seen in Figure 22. Contrary to all expectations, quantization shows no significant performance gain on the Jetson Nano, as the decrease in latency is in the scale of a few milliseconds at best. It also becomes apparent, that the TF-TRT engine might not utilize the GPU as well as pure TensorRT, as the SSD model shows significantly higher CPU usage and lower GPU usage than other models.

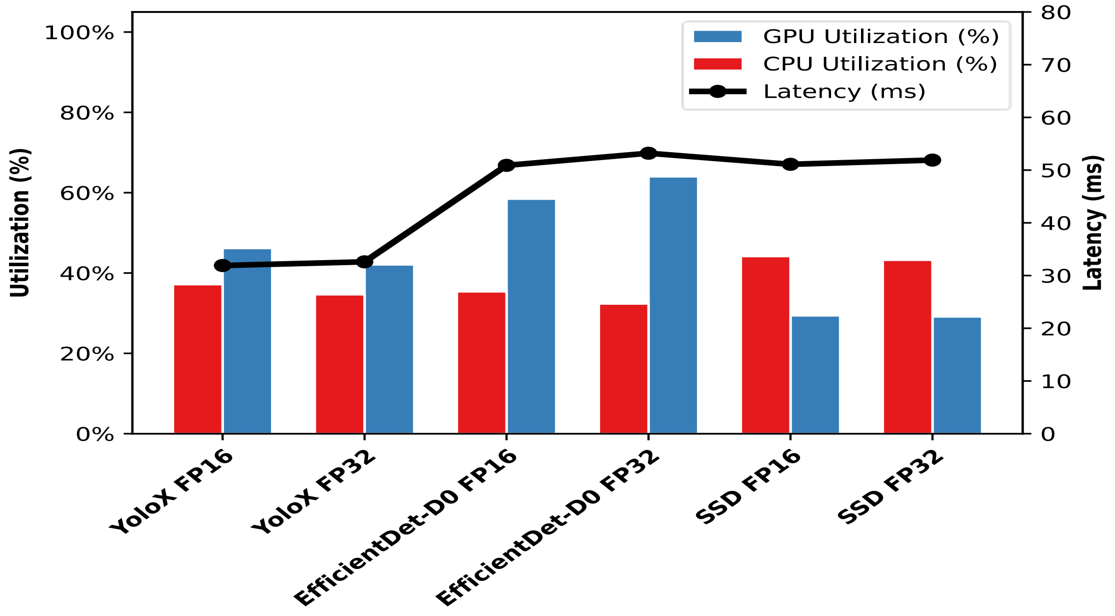

FIGURE 22. Processing unit utilization and latency of different models during stress test

Potential bottlenecks in quantization could be accounted to inefficient usage of the freed-up memory and bandwidth. TensorRT also doesn't quantize the inputs and outputs of the model nor other parts if they result in significant accuracy loss, so it is unlikely to achieve double the performance. Using FP16 could, however, be more beneficial when used with higher batch sizes, as the memory reduction could be used to process more images in parallel, resulting in a higher throughput (images processed per second).

It was also of interest to monitor system information during inference, to get a better view of how the system acts as a function of time, and a graph taken during inference of the FP32 YoloX-nano model can be seen in Figure 23. The dynamic clock frequency aggressively drops down the CPU clock frequency at times, as most of the computing is being done at the GPU. As pre-processing, such as reading and resizing images is done with the CPU between inferences and takes long (~30 ms), the GPU utilization stays on average at relatively low levels and throttling does not occur. If computation was done purely on GPU, some throttling would be expected as the utilization would consistently be at high levels.

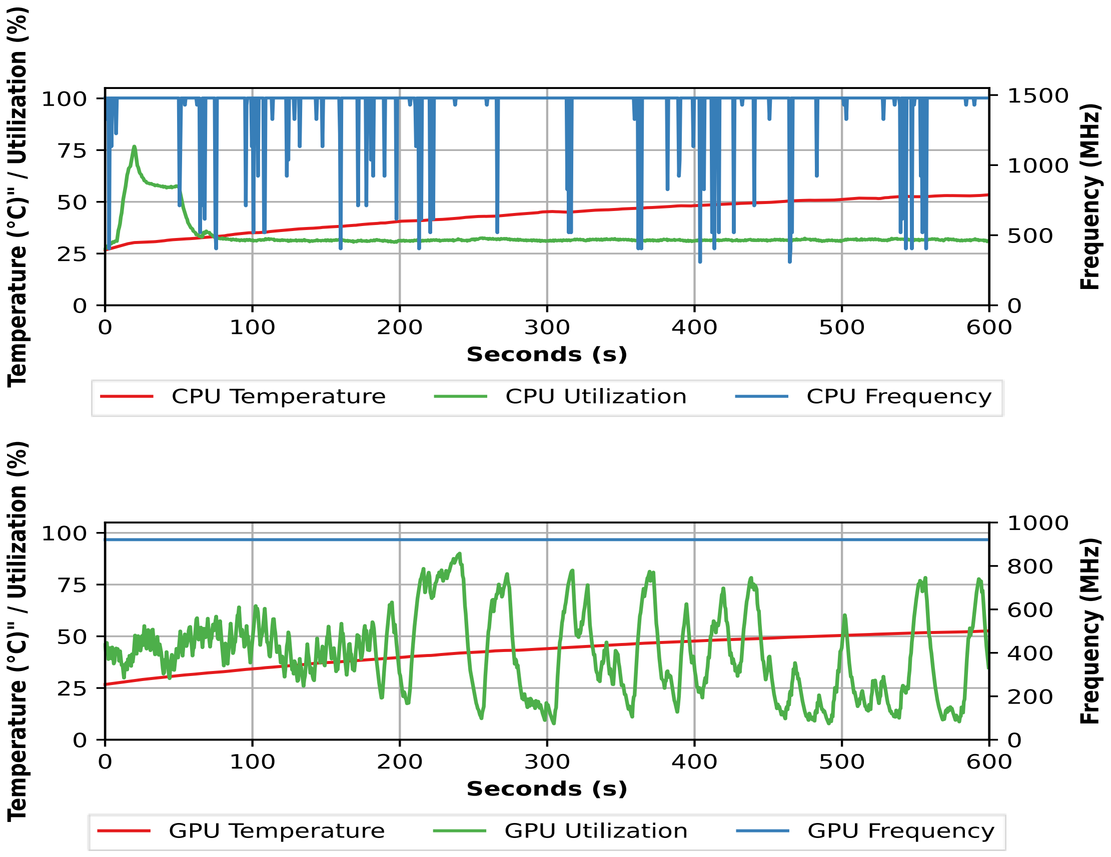

FIGURE 23. Jetson Nano system-level statistics during YoloX FP32 inference

Similar temperature readings were achieved for each model, with an average final temperature of around 53 °C on the GPU and 55 °C on the CPU, and as can be seen from Figure 23, the numbers were constantly increasing on both. Temperature throttling could become an issue after longer times of inference, and thus in production usage, the thermal design of the casing should be thought out.

5.2.2 Raspberry Pi 4 B and Neural Compute Stick 2

The combination of Raspberry Pi 4 B and NCS 2 were also evaluated to gain insight into the total system performance. The communication between the two devices is done through USB, but as the NCS 2 has its own memory chip, all the computing during inference is done inside the device.

System performance data was collected during inference and is seen in Figure 24. As the computing is offloaded entirely to the VPU during inference, the dynamic clock frequency can be seen aggressively dropping the CPU clock speed, and as a result keeping the power consumption at moderate levels.

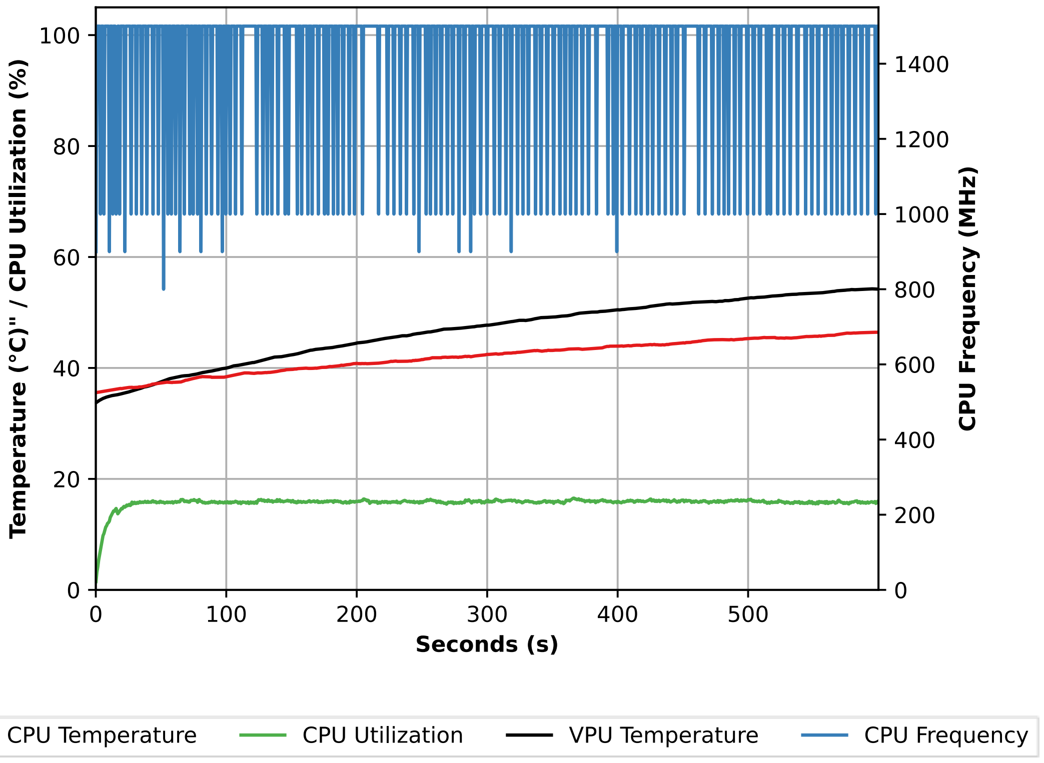

FIGURE 24. Raspberry Pi and NCS 2 system-level statistics during SSD-MobileNet-V2 stress test

Figure 24 also shows the VPU temperature rising at much higher speeds than the CPU, as little computing is performed on the CPU besides loading the images. Since the NCS 2 is an off-board device, when performing inference for longer periods and at different operating temperatures, the unit cannot necessarily be cooled with a separate fan or heatsink like the Raspberry Pi mainboard. As a result, the NCS 2 might start to throttle, resulting in higher inference times.

6 Conclusions

The goal of this thesis was to find how the existing deep learning accelerators perform when doing inference on state-of-art object detection models, and if they are suitable for Visy Oy’s needs. Three different hardware architectures were taken into consideration, and all of them proved to be very well suited for real-life use cases.

The Jetson Nano with a Maxwell-architecture GPU proved out to be the most flexible device out of all, as it was the only device to run all the candidate models. The best performance was achieved with the YoloX-nano model, which provided low and almost non-deterministic latency, but as a trade-off had the lowest accuracy of all the candidates. The two other models had similar average latency, but the SSD-MobileNet-V2 model suffered from high variance, which is undesired for real-time systems. Both FP16 and FP32 precisions were also tested on the platform, but no major performance gain or accuracy loss was seen from quantizing to 16 bits. The probable cause for this is, that the performance tests were conducted with a batch size of 1, which might not fully harness the freed memory from quantization, or the parallelization capabilities of the GPU. Similar results supporting this conclusion were also seen in a Jetson TX1 whitepaper by NVIDIA [52], where the difference in performance between FP16 and FP32 models grew as the batch size was increased. The ideal use case for the Jetson Nano lies in situations, where video processing is a bottleneck, as the GPU can be used for both video processing and object detection.

The Raspberry Pi 4 B and NCS 2 also turned out to be a good combination, even though the NCS 2, unfortunately, has trouble with compiling state-of-art object detection models other than the SSD-MobileNet-V2. Nonetheless, the model proved to work sufficiently, as it had a decent latency, and the response times were consistent since no CPU computing had to be done during inference. The easy installation of the NCS 2 to any platform makes it a good choice for cases, where existing computing units lack the performance for inference.

The Axis camera equipped with an EdgeTPU achieved the highest speed out of all the test configurations, with the SSD-MobileNet-V2 running at 120 FPS, although it also experienced some accuracy loss due to 8-bit quantization. As with the NCS 2, converting models to an ASIC chip proved to be a trouble yet again, and the device also has a more controlled environment by the manufacturer, making application deployment slightly more complicated than for NVIDIA or Intel platforms. The advantages, however, lie in the ease of installation, as no separate computing units or power supplies are required other than a PoE input.

Future work on top of this thesis could be done on comparison of more advanced neural network optimization techniques, such as pruning, clustering, and quantization-aware training, and their effects on edge device performance. Further benchmarking could also be done on the performance of different neural network architectures on a layer-level, to find potential bottlenecks when running inference on edge devices. Finally, the results could be improved by performing accelerator utilization measurements for each model by doing purely inference without any pre- or post-processing.

In conclusion, all the goals of this thesis were achieved as all the devices were found to be suitable for production usage, with varying use cases. Based on the results, prototypes utilizing these devices can now be built and offered as part of Visy Oy’s portfolio.

7 References

[1] I. Goodfellow, Y. Bengio and A. Courville, “Deep Learning”, MIT Press, 2016.

[2] S. Shavel-Shwartz and S. Ben-David, “Understanding Machine Learning - From Theory to Algorithms, 1 ed.”, New York: Cambridge University Press, 2014.

[3] K. P. Murpy, “Machine Learning: A Probabilistic Perspective, London: The MIT Press”, 2012.

[4] G. James, D. Witten, T. Hastie and R. Tibshirani, “An Introduction to Statistical Learning, 2nd ed.”, Springer, 2021.

[5] C. M. Bishop, “Pattern Recognition and Machine Learning”, Cambridge: Springer, 2006.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel and D. Hassabis, “Mastering the game of Go without human knowledge”, Nature, 2017.

[7] M. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015.

[8] L. Deng and D. Yu, “Deep Learning Methods and Applications, 7 ed.”, 2014.

[9] X. Glorot, A. Bordes and Y. Bengio, “Deep Sparse Rectifier Neural Networks”, 2011. Online. Available: https://proceedings.mlr.press/v15/glorot11a.html. Accessed 06.11.2021

[10] C. E. Nwankpa, W. Ijomah, A. Gachagan and S. Marshall, “Activation Functions: Comparison of Trends in Practice and Research for Deep Learning”, 2018.

[11] A. Zhang, Z. C. Lipton, M. Li and A. J. Smola, “Dive into Deep Learning, 0.17.0 ed.”, 2021.

[12] A. Géron, Hands-on Machine Learning with Scikit-Learn, Keras & Tensorflow, O'Reilly Media, 2019.

[13] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard and L. Jackel, “Backpropagation Applied to Handwritten Zip Code Recognition”, 1989.

[14] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-Based Learning Applied to Document Recognition”, 1998.

[15] A. Krizhevsky, I. Sutskever and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, 2012.

[16] H. Lee, R. Grosse, R. Ranganath and A. Ng, “Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks”. Communications of the ACM, vol. 54, no. 10, pp. 95-103, 2011.

[17] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, “Edge Computing: Vision and Challenges”, 2016.

[18] Cisco, “Cisco Annual Internet Report (2018-2023) ”, 9.3.2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Accessed 30.11.2021.

[19] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu and A. C. Berg, “SSD: Single Shot MultiBox Detector”, 2016.

[20] P. Viola and M. Jones, “Rapid Object Detection Using a Boosted Cascade of Simple Features”, 2001.

[21] R. Girshick, J. Donahue, T. Darrell and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation”, 2014.

[22] J. Redmon, S. Divval, R. Girshick ja A. Farhadi, ”You Only Look Once: Unified, Real-Time Object Detection”, 2016.

[23] T. Mingxiang, R. Pang ja L. V. Quoc, ”EfficientDet: Scalable and Efficient Object Detection”, 2020.

[24] Z. Ge, S. Liu, F. Wang, Z. Li and J. Sun, “YOLOX: Exceeding YOLO Series in 2021”, 2021.

[25] Z. Tian, C. Shen, H. Chen ja T. He, ”FCOS: Fully Convolutional One-Stage Object Detection”, 20.8.2019.

[26] R. Girshick, “Fast R-CNN”, 2015.

[27] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid and S. Savarese, “Generalized Intersection over Union: A Metric and A Loss for Bounding Box”, p. 9, n.d.

[28] X. Wang, Y. Han, V. C. Leung, D. Niyato, X. Yan and X. Chen, “Convergence of Edge Computing And Deep Learning: A Comprehensive Study”, p. 36, 2020.

[29] Google Coral, “Products”, n.d. Online. Available: https://coral.ai/products/. Accessed 13.12.2021.

[30] Intel, “Intel® Movidius™ Vision Processing Units (VPUs) ”, n.d. Online. Available: Intel® Movidius™ Vision Processing Units (VPUs). Accessed 13.12.2021.

[31] NVIDIA, “Jetson Modules”, n.d. Online. Available: https://developer.nvidia.com/embedded/jetson-modules. Accessed 13.12.2021.

[32] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney and K. Keutzer, “A Survey of Quantization Methods for Efficient Neural Network Inference”, 2021.

[33] H. Wu, P. Judd, X. Zhang, M. Isaev and P. Micikevicius, “Integer quantization for Deep Learning inference: Principles and empirical evaluation”, 2020.

[34] NVIDIA, “Jetson Nano”, n.d.. Online. Available: https://developer.nvidia.com/embedded/jetson-nano. Accessed 14.12.2021.

[35] Raspberry Pi, ”Raspberry Pi 4 Model B”, n.d.. Online. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. Accessed 17.1.2022.

[36] Intel, “Intel® Neural Compute Stick 2 (Intel® NCS2)”, n.d.. Online. Available: https://www.intel.com/content/www/us/en/developer/tools/neural-compute-stick/overview.html. Accessed 14.12.2021.

[37] Axis, “AXIS Q1615-LE Mk III Network Camera”, n.d.. Online. Available: https://www.axis.com/products/axis-q1615-le-mk-iii. Accessed 14.12.2021.

[38] NVIDIA, “TensorRT”, n.d.. Online. Available: https://developer.nvidia.com/tensorrt. Accessed 16.12.2021.

[39] ONNX, “Open Neural Network Exchange”, n.d.. Online. Available: https://onnx.ai/. Accessed 16.12.2021.

[40] H. Abbasian, Y.-T. Cheng and J. Park, “Speed up TensorFlow Inference on GPUs with TensorRT”, 18.4.2018. Online. Available: https://blog.tensorflow.org/2018/04/speed-up-tensorflow-inference-on-gpus-tensorRT.html. Accessed 11.1.2022.

[41] NVIDIA, “Accelerating Inference In TF-TRT User Guide”, n.d.. Online. Available: https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html. Accessed 16.12.2021.

[42] TensorFlow, “TensorFlow Lite”, n.d.. Online. Available: https://www.tensorflow.org/lite. Accessed 16.12.2021.

[43] Google Coral, “Coral Software”. n.d.. Online. Available: https://coral.ai/software/. Accessed 02.01.2022.

[44] Google Coral, “TensorFlow models on the Edge TPU”, n.d.. Online. Available: https://coral.ai/docs/edgetpu/models-intro/. Accessed 16.12.2021.

[45] Intel, “OpenVINO Software Development Kit”, n.d.. Online. Available: https://docs.openvino.ai/latest/index.html#. Accessed 24.12.2021.

[46] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn and A. Zisserman, THE Pascal Visual Object Classers (VOC) Challenge, Springer Link”. International Journal of Computer Vision, vol. 88, pp. 303-338, 2010.

[47] COCO, “COCO - Common Objects in Context”, n.d.. Online. Available: https://cocodataset.org/#home. Accessed 24.12.2021.

[48] jetson-stats, “jetson-stats”, n.d.. Online. Available: https://rnext.it/jetson_stats/. Accessed 02.01.2022.

[49] NVIDIA, “tegrastats Utility”, n.d.. Online. Available: https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html. Accessed 02.01.2022.

[50] psutil, “psutil documentation”, n.d.. Online. Available: https://psutil.readthedocs.io/en/latest/. Accessed 02.01.2022.

[51] Intel, “Inference Engine Developer Guide”, n.d.. Online. Available: https://docs.openvino.ai/latest/openvino_docs_IE_DG_Deep_Learning_Inference_Engine_DevGuide.html. Accessed 02.01.2022.

[52] NVIDIA, ”GPU-Based Deep Learning Inference: A Performance and Power Analysis”, 2015.