Masaryk University
Faculty of Informatics

Fast object detection on mobile platforms using neural networks

Bachelor's Thesis

Tomáš Repák

Brno, Spring 2021

Masaryk University
Faculty of Informatics

Fast object detection on mobile platforms using neural networks

Bachelor's Thesis

Tomáš Repák

Brno, Spring 2021

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Tomáš Repák

Advisor: doc. RNDr. Tomáš Brázdil, Ph.D.

Acknowledgements

I want to thank my family for enduring the noise that the evaluation board produced day and night over the month when I was running the evaluation. I would also like to thank my advisor, doc. RNDr. Tomáš Brázdil, Ph.D., for asking detailed questions that forced me to gain a better understanding of the model.

Abstract

The most common neural network architectures used for object detection on mobile platforms are SSD (Single Shot Detector) and YOLO (You Only Look Once). Both have their advantages and their drawbacks in terms of precision, performance or memory usage. The goal is to compare current state-of-the-art neural networks such as EfficientDet against either of the two, provide an implementation, an analysis and potentially propose modifications. Models are assumed to be trained on the publicly available COCO dataset.

Keywords

deep learning, machine learning, neural networks, NXP Semiconductors, object detection

Contents

Introduction **1**

1 Object detection **3**
1.1 Neural networks for object detection 5
1.1.1 Convolutional neural networks 5
1.1.2 Depthwise separable convolution 6
1.2 Basic structure of a neural network detection model . . 8
1.2.1 Region proposals 8
1.2.2 Backbone 9
1.2.3 Neck 9
1.2.4 Head 9
1.2.5 Bounding Boxes 10
1.2.6 Anchor Boxes 10
1.3 Object detection on mobile platforms 12
1.3.1 Limitations of mobile devices with regards to object detection 12
1.3.2 Optimization techniques 14

2 EfficientDet **17**
2.1 Architecture 17
2.1.1 Backbone 17
2.1.2 Neck 19
2.1.3 Head 23
2.2 Activity 23
2.3 Training 24
2.4 EfficientDet-lite 26

3 SSD MobileNetV2 **29**
3.1 Architecture 29
3.1.1 Backbone 29
3.1.2 Head 30
3.2 Activity 30
3.3 Training 32

4 EfficientDet versus SSD MobileNetV2 **33**
4.1 Backbones 33

4.2 Necks . . . 33
4.3 Heads . . . 33
4.4 General overview . . . 34

5 Implementation and environment **35**
5.1 Implementation . . . 35
5.2 Environment . . . 35
5.2.1 Target device . . . 35
5.2.2 Host . . . 35
5.3 Tools . . . 35
5.4 Models . . . 36
5.4.1 EfficientDet . . . 36
5.4.2 EfficientDet-lite . . . 37
5.4.3 SSD MobileNetV2 . . . 37

6 Evaluation **39**
6.1 Evaluation Metrics . . . 39
6.1.1 Latency . . . 39
6.1.2 Memory usage . . . 39
6.1.3 Mean Average Precision (mAP) . . . 40
6.1.4 Average Recall . . . 42
6.1.5 Number of detections . . . 43
6.2 Results . . . 43
6.2.1 Latency . . . 43
6.2.2 Memory usage . . . 44
6.2.3 Mean Average Precision . . . 45
6.2.4 Average Recall . . . 46
6.2.5 Number of detections . . . 46
6.2.6 Precision per-class analysis . . . 47
6.2.7 Recall per-class analysis . . . 48
6.3 Summary . . . 51

7 Conclusion and Future work **53**

Bibliography **55**

Index **63**

A Attached source code **65**

A.1 automl 65
A.2 thesis 65

List of Tables

2.1 *EfficientDet architecture scaling* 18

2.2 *EfficientDet-lite architecture scaling* 27

3.1 *MobileNetV2 bottleneck depth-separable convolution. h, w represent the height and width of the input, k the number of input channels, k' the number of output channels, t the expansion factor and s stride. Taken from* [41] 30

6.1 *Latency results of compared models in miliseconds. Average inference time is measured across 20288 test images* 44

6.2 *Memory usage results of compared models. All values are in kilobytes. TFlite file size denotes how much space does* `.tflite` *file take, Model size denotes the memory consumption of loading the execution graph and Tensor memory denotes the amount of memory requested to be allocated by* `TfLiteInterpreter`. 45

6.3 *Mean Average Precision results of compared models on* `COCO test-dev2017 dataset`. *The mAP is the AP averaged over all IOU thresholds and categories, AP_{50} is the AP averaged over all categories at 0.5 IOU threshold. Similarly, AP_{75} is the AP averaged over all categories at a 0.75 IOU threshold. AP_{small}, AP_{med}, AP_{large} are the average precisions on small, medium and large objects.* 46

6.4 *Average Recall results of compared models on* `COCO test-dev2017` *dataset. AR max values represent average recalls when the maximum number of detections per image is limited to a specific value* $(1, 10, 100)$. *AR small, med, large represent average recall values across small, medium, and large-sized objects.* 47

6.5 *Average number of detections with score thresholds of 0.25, 0.40, 0.50, 0.75 across all* `COCO test-dev2017` *dataset* 48

6.6 *Top three classes and their corresponding AP scores across the dataset.* 49

6.7 *Worst three classes and their corresponding AP scores across the dataset.* 49

6.8 Top three classes and their corresponding AR scores across the dataset. All AR values are the COCO evaluation AR_{max100} values. 50
6.9 Worst three classes and their corresponding AR scores across the dataset. All AR values are the COCO evaluation AR_{max100} values. 51

List of Figures

1.1 *Example object detection with EfficientDet. Taken from [9]* 4
1.2 *An illustration of 3x3x3 convolution. h, w, d represent the height, width and depth of the input. Taken from [20].* 6
1.3 *Max pooling and average pooling illustration. Taken from [21].* 7
1.4 *A comparison of standard convolutional kernels (a) and Depthwise separable convolution kernels (b, c). Taken from [2].* 8
1.5 *A typical structure of a single-shot, neural network based object detection model. Reference image taken from [23].* 9
1.6 *Illustration of anchor boxes and their aspect ratios. Figures b) and c) represent different anchor boxes matched to ground-truth annotation in Figure a). Highlighted anchor box in c) has aspect ratio of 2, square anchor box highlighted in b) has aspect ratio of 1 and wider highlighted anchor box in b) has aspect ratio of $\frac{1}{2}$. Taken from [4].* 11
2.1 *Comparison of Feature Pyramid Networks. Taken from [26]. P_3 to P_7, in general P_i, represent an output from backbone's i-th layer and serve as an input to the i-th Bi-FPN level.* 21
2.2 *Focal loss with γ parameter. Setting γ to 0 transforms the focal loss to classical Cross Entropy. Taken from [47]* 25
3.1 *MobileNetV2 bottleneck depth-separable convolution with residual connection in Netron [50] visualization tool.* 31
6.1 *Illustration of various Intersection Over Union values. Taken from [54]* 40
6.2 *An illustration of how ground-truth bounding box (green) versus predicted bounding box (red) might look like. (a) To quantify the quality of the prediction, IOU value between the two bounding boxes is computed. Reference image taken from [56].* 42

Introduction

With the evolution of Convolutional Neural Networks (CNN), object detection has become a vast area of research in computer vision and machine learning. Detecting objects in an image or video stream is one of the most prominent parts in the area of self-driving cars, augmented reality, or military industry. There is intensive research ongoing in object detection, and several models have been invented to tackle this area.

To increase performance, models have grown large in size and complexity. Models such as YOLO [1] contain several million parameters to achieve their accuracy. Such complexity has become an issue when deploying these models to mobile platforms with limited resources, such as memory or computational power. These platforms often provide only a tiny fraction of the resources that are needed to run state-of-the-art models. Several optimized models were introduced for usage on mobile platforms, such as MobileNet [2] or YOLO-LITE [3]. Among these models, two major approaches are used – Single Shot MultiBox Detector (SSD) [4], and You Only Look Once (YOLO) [1]. Recently, *EfficientDet* [5] has been published. Authors of EfficientDet extend their previous research on EfficientNets [6] and report better performance and efficient scalability while preserving state-of-the-art accuracy on COCO [7, 8] dataset.

Analysis of EfficientDet architecture and performance in terms of speed and memory on a mobile device is conducted in this work. Since EfficientDet is a fairly new model, this work provides an insight into its performance in a mobile environment. It shall also provide necessary information to decide whether to use EfficientDet in a given application on a mobile device or whether it is better to opt for a different model. The implementation part of this work can also serve as an example of running EfficientDet on a mobile device. This work is created in cooperation with NXP Semiconductors industrial partner, which defined the assignment and provided the necessary equipment.

1 Object detection

The topic of object detection is concerned with detecting objects in an image or video input. The most iconic objects are dogs, cats, people, and cars, although objects can be anything we train the model to detect. It is, however, not enough to decide whether there is an object present in the input or not. The model is also supposed to express where the object is (localization) and what category the object belongs to (classification). Figure 1.1 illustrates an example of a detection output.

According to Sahu and Felzenszwalb [10], "the goal of object detection is to detect all instances of objects from a known class, such as people, cars or faces in an image.". In this work, we will follow this definition and describe the process of detecting objects in an input image using neural networks.

Object detection belongs to the area of computer vision, which comprises many other problems. Other problems from this area are image segmentation or image classification. According to Sahu et al. [11], the goal of image segmentation is to simplify or break down the representation of an image into meaningful and easily analysable parts. Image classification is concerned with assigning a category (class label) to an input image. Both of mentioned areas are closely related to object detection. Knowledge from both problems needs to be combined in order to tackle object detection. If we segment an image into individual parts (which might be objects) and then use classification to classify these individual parts, then we are not only able to detect objects but also predict a category they belong to. Although it sounds straightforward, technical limitations are an issue, especially on mobile platforms.

Convolutional Neural Networks have proven themselves to be very efficient in image classification and other computer vision tasks. Therefore, they are the prominent approach used in object detection. There exist other, mostly historical algorithms, such as Histogram of Oriented Gradients [12] or Scale-Invariant feature transform [13]. This work is focused on neural networks.

Nowadays, two main approaches exist – two-step detection and one-step detection. Two-step detection refers to an approach where we first identify an area of an image that might be an object that we

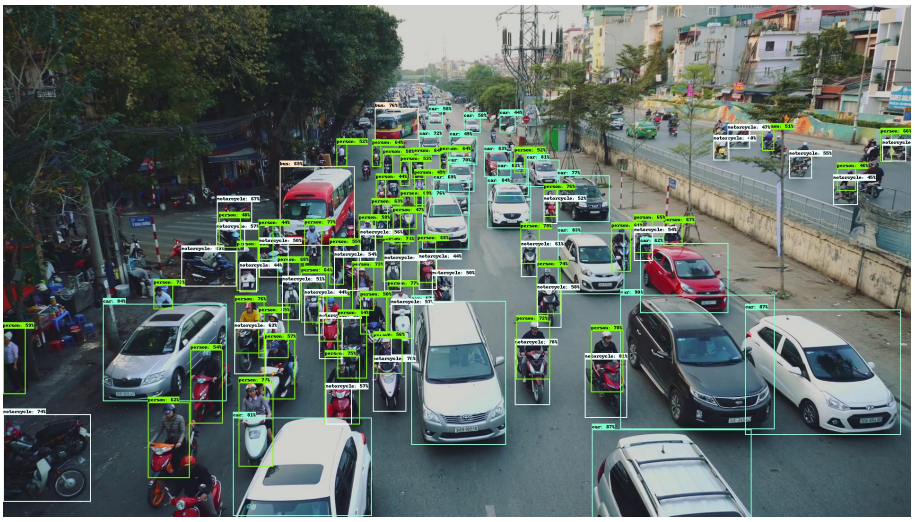


Figure 1.1: Example object detection with EfficientDet. Taken from [9]

want to detect. This area is often called *region proposal*. There may be up to several thousands of region proposals generated from a single input image [14]. A classification network is run on region proposals to predict a class the region belongs to. Examples of this approach are Regional Convolutional Neural Networks (RCNN) [14] and its modifications, such as Fast RCNN [15], Faster RCNN [16] and Mask RCNN [17]. On the other hand, one-step detection tries to eliminate the need to generate region proposals and produce an output in a single pass. Examples of one-step detectors are YOLO and SSD. According to [18], the two-step approach usually performs better in terms of accuracy, whereas the one-step approach is faster and more efficient in terms of memory.

1.1 Neural networks for object detection

Most of the current state-of-the-art object detectors are based on neural networks. In a two-step approach, a classification neural network is run on region proposals, and in one-step detection, the whole model is a neural network. The reason behind the choice of neural networks is that convolutional neural networks have shown remarkable results in the area of computer vision. Since AlexNet [19] won the ImageNet Large Scale Visual Recognition Challenge, many efforts have been put into the research of convolutional neural networks.

1.1.1 Convolutional neural networks

Convolutional neural networks are a class of neural networks commonly employed in computer vision tasks, such as image classification, face recognition, or object detection. At the heart of a convolutional neural network is a convolution operation.

A convolution takes a feature map as its input and applies a *convolutional kernel* (filter) to the feature map. There are usually several convolutional kernels per layer. The size of the convolutional kernel is smaller than the size of the input feature map so that the kernel can be applied at various locations of the feature map. At each location, a weighted sum of the feature map values and convolutional kernel weights is computed, and an activation function is applied. An example of a three-dimensional convolution is illustrated in Figure 1.2.

The reason why convolutional neural networks achieved such great success is that the convolution operation operates on a small area, called *receptive field*. Through the process of learning, convolutional kernels can, for example, learn to detect edges or curves in the input image. A network can then learn a linear combination of these edges and curves to correctly classify the input image.

Another key concept in convolutional neural networks is pooling layers. Pooling layers usually follow the convolutional layers in the architecture and serve to reduce the feature map size by preserving only the most essential information from the feature map. Pooling layers also apply convolution to the input, but in a different manner than the original convolution. There are several types of pooling, such as max

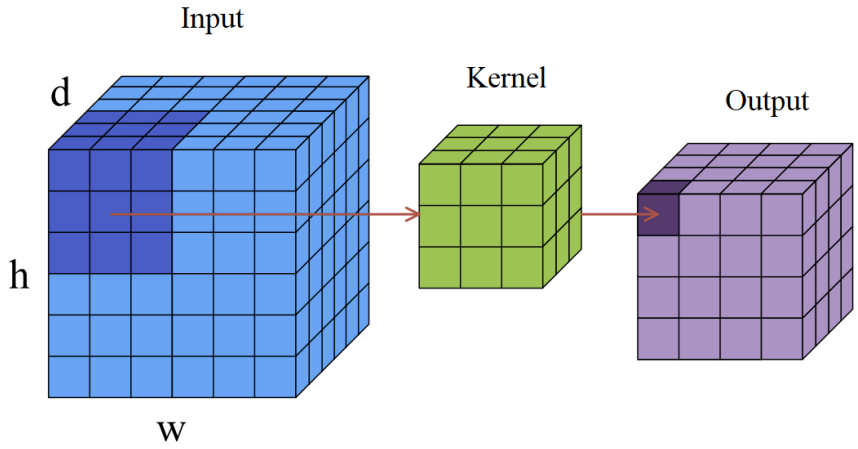


Figure 1.2: An illustration of 3x3x3 convolution. h, w, d represent the height, width and depth of the input. Taken from [20].

pooling or average pooling. Both methods look at their receptive field in their input and extract the maximum or average value from the receptive field. An example of those methods can be seen in Figure 1.3. Using kernels of larger size, therefore, results in a greater reduction of the layer's output size.

In convolutional neural network architectures, such as AlexNet [19], dense layers are attached on top of the final convolutional or pooling layer to produce the final predictions.

1.1.2 Depthwise separable convolution

As convolutional neural network models grew larger and more complex, researchers started to re-think the parts of a typical convolutional network. Some improvements have been made, and Depthwise separable convolution has been introduced. Compared to a standard convolution, depthwise separable convolution is faster and more efficient. Depthwise separable convolution is nowadays commonly used in many state-of-the-art convolutional neural network models, such as [5, 22].

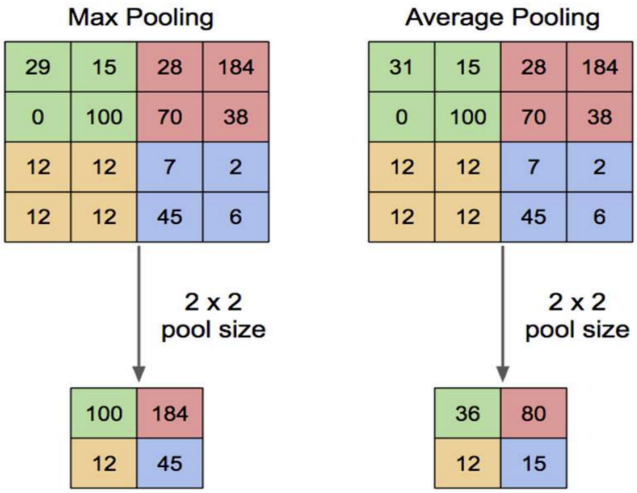
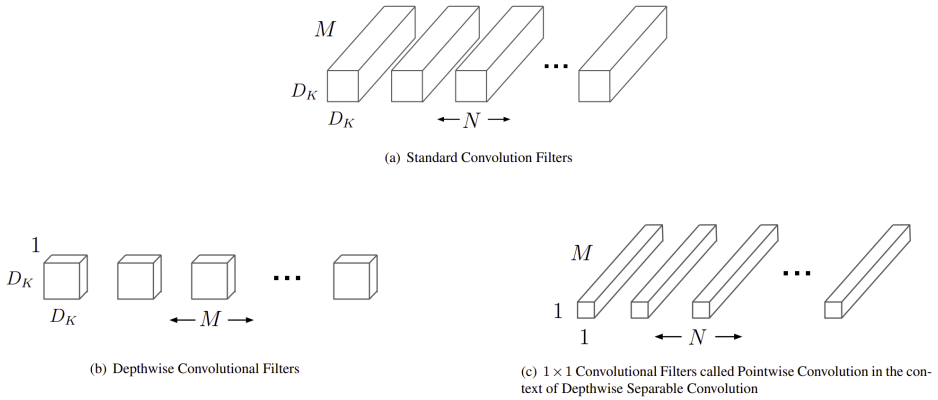


Figure 1.3: Max pooling and average pooling illustration. Taken from [21].

The idea of depthwise separable convolution is first to apply 3x3 convolution to each input channel and then linearly combine the results with 1x1 pointwise convolution. An illustration of the kernels is depicted in Figure 1.4. As discussed in [2], along with mathematical reasoning, the computational cost of depthwise separable convolution compared to the standard convolution is

$$\frac{1}{N} + \frac{1}{D_k^2} \tag{1.1}$$

where N is the number of output channels, and D_k is the width and height of the convolutional kernel. Authors have shown that depthwise separable convolution almost retains its standard counterpart accuracy.



(a) Standard Convolution Filters

(b) Depthwise Convolutional Filters

(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 1.4: A comparison of standard convolutional kernels (a) and Depthwise separable convolution kernels (b, c). Taken from [2].

1.2 Basic structure of a neural network detection model

A single detection model is usually a complex structure consisting of several parts. This is convenient in transfer learning because these parts can be treated as modules, which can be modified, removed, or swapped. An example of a typical neural network-based object detector is illustrated in Figure 1.5.

1.2.1 Region proposals

In two-step object detection, the first step is to identify areas in the image, which might be objects that we want to detect. These areas are referred to as region proposals. Determining region proposals is a subject of research. A widely used algorithm is the Selective Search [24], which can yield region proposals of various scales. Another method for generating region proposals is the Region Proposal Network (RPN) [16]. According to Ren et al. [16], "the observation is that the convolutional feature maps used by region-based detectors, like Fast R-CNN, can also be used for generating region proposals."

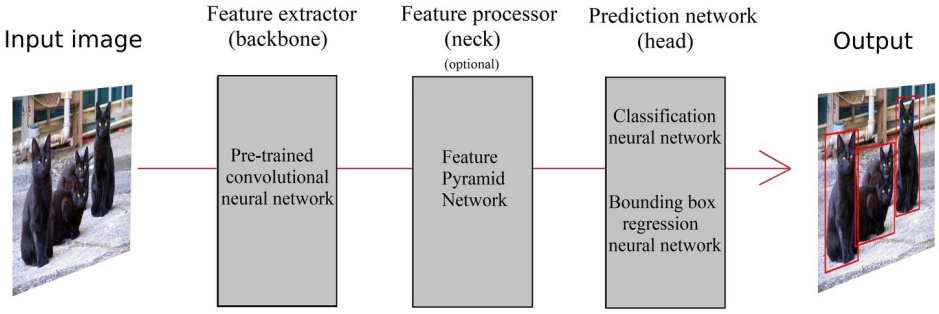


Figure 1.5: A typical structure of a single-shot, neural network based object detection model. Reference image taken from [23].

1.2.2 Backbone

Since most object detectors are complex networks, it is complicated and time-consuming to train them from scratch. To minimize the effort, pre-trained convolutional neural networks are used to extract features of input data, which are then used by other parts of the model. These networks are referred to as feature extractors or backbones. Donahue et al. [25] show that features extracted by convolutional neural networks[1] generalize well to tasks similar to image classification. Object detection is considered to be a similar task.

1.2.3 Neck

Neck is an optional part between the backbone and the head. It is a part where additional processing of extracted features, such as feature fusion, takes place. Output from the neck is passed to one or more heads for the final stage. An example of a neck is Feature Pyramid Network (FPN) [26].

1.2.4 Head

Head refers to a part of a model, which produces the final output. A classification neural network or Support Vector Machine are examples

1. AlexNet [19] has been selected as a representative model.

of a head. If we consider AlexNet [19], dense layers are the head. There can be several heads at once. In the EfficientDet model, there are two heads in parallel – one for predicting bounding boxes and a second one for predicting classes. Each of them produces an output, which is merged and post-processed to produce a final output.

1.2.5 Bounding Boxes

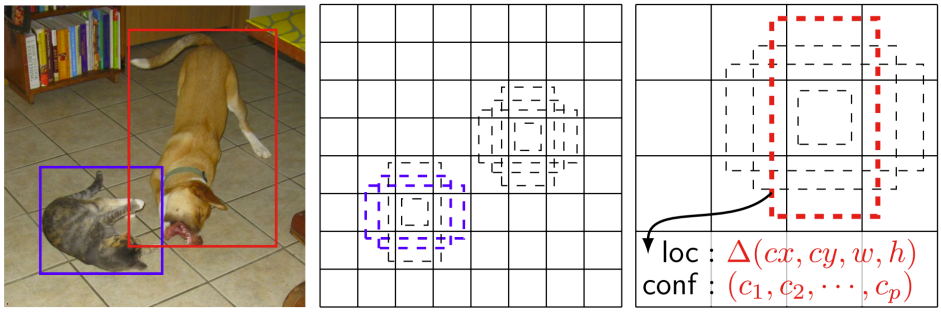
When we want to detect an object, we need to localize it. Therefore, we need information where in the image the object is. We accomplish this task by encoding the output of a model into a *bounding box* structure.

A bounding box is an artificial rectangle surrounding the object. A bounding box is defined using coordinates in the image, though the exact implementations may differ. One way to define a bounding box is by using the center point of the box, combined with the width and height of the box. Another way is to define the boundaries of the box by minimum and maximum x and y coordinates. Localization encoded in bounding boxes is a suitable approach due to the fact that it can be expressed as a regression problem [14] and therefore can be trained.

1.2.6 Anchor Boxes

Anchor boxes are a key mechanism used in object detection, namely in one-step detection. Anchor boxes were first introduced in Faster-RCNN [16] and were employed with great success in state-of-the-art networks such as [4, 5].

Anchor boxes can be thought of as bounding boxes that are assumed to be present throughout the dataset. For example, we can estimate from the training dataset how an average ground-truth bounding box for a given class looks and compute the anchor box responsible for detecting that specific class. While this is a valid approach, it is usually not used in practice. A common approach is to define a set of *aspect ratios* that define the ratio between height and width of the anchor box. Each aspect ratio defines a single anchor box. Commonly used aspect ratios are $\{\frac{1}{2}, 1, 2\}$, with additional ones being added if necessary. As discussed in [4], more anchor boxes of different aspect ratios produce better results.



a) Image with GT boxes b) 8×8 feature map c) 4×4 feature map

Figure 1.6: Illustration of anchor boxes and their aspect ratios. Figures b) and c) represent different anchor boxes matched to ground-truth annotation in Figure a). Highlighted anchor box in c) has aspect ratio of 2, square anchor box highlighted in b) has aspect ratio of 1 and wider highlighted anchor box in b) has aspect ratio of $\frac{1}{2}$. Taken from [4].

When using anchor boxes for prediction, features from the backbone or neck are used. These features are tiled into a grid, as seen in Figure 1.6. Suppose there are K predefined arbitrary aspect ratios. For each tile in the feature map, K anchor boxes are created, with their corresponding aspect ratio, and predictions for all K anchor boxes centered in the middle of the tile are produced. Refer to Figure 1.6, where highlighted detections in Figure 1.6 a) correspond to anchor boxes generated in Figure 1.6 b) and Figure 1.6 c). To summarize, if we use only a single feature map, tile it into an $N \times N$ grid, and use K aspect ratios, we obtain a total of $N \times N \times K$ outputs. To detect objects of different sizes, anchor box predictions are usually made on several feature maps of different resolutions, as illustrated in Figure 1.6 b) and c).

1.3 Object detection on mobile platforms

Although technology evolves rapidly in magnitudes, small devices such as mobile phones or embedded chips still present a very limited environment for the execution of large machine learning models. There is a tremendous difference between running a model on a Google server with several processors optimized for machine learning and an embedded chip. Most of the time, we want similar performance on these small, limited devices. Through research, people invented ways of at least partially accomplishing this task.

Mobile platforms also offer `on edge` processing, which means the data is not sent to a server for evaluation. Instead, the data is evaluated directly on the device.

1.3.1 Limitations of mobile devices with regards to object detection

From a hardware point of view, a mobile device is just like a usual computer, only much smaller. Almost every hardware component is squeezed into as small parts as possible, which reduces their size and performance. Some components, such as graphical output, may even be omitted entirely. In mobile phones, tablets, or some boards, there is no stable access to electrical power, and the system is powered via battery. Using computationally expensive operations significantly reduces the battery operating time. Many optimizations have been invented and are necessary in order to deploy machine learning models to mobile devices effectively.

Real-Time execution
Object detection may be required in applications such as self-driving cars with pedestrian detection or augmented reality [27]. Both of them need to process images in real-time. Self-driving cars need to be able to react quickly to a given situation. Augmented reality needs to respond quickly as well. Otherwise, the user will not use the application. We require real-time processing in both scenarios, though applications in the area of self-driving cars need to consider other factors as well, such as safety and ethics. In real-time applications, we usually aim for a processing speed of at least 24 Frames Per Second (FPS) and higher,

though the exact required numbers may differ in different use cases. Nevertheless, such a requirement needs fast hardware and algorithms that are able to provide such speed on a mobile platform.

Memory
State-of-the-art neural networks are huge models that may require memory storage capacities in hundreds of megabytes, which is often too much to request in a mobile environment, due to size or price requirements. Several techniques how to reduce model size have been invented, such as Deep Compression [28], which leverages probably the most used optimization techniques in a mobile environment – Quantization [29, 30] and Pruning [31] – along with other techniques, including weight sharing. Results show that even large state-of-the-art neural networks can be shrunk into a tiny fraction of the original model size while preserving its accuracy and dramatically improving inference times.

Another issue to keep in mind when dealing with mobile devices is power consumption. Mobile devices may not be connected to a power source and can be powered via battery. Using power expensive operations, such as DRAM memory access [28, 32] may lead to a very quick depletion of battery power, which is highly undesired in mobile devices. Fitting necessary data to faster on-chip SRAM memory is recommended.

Latency
The usual CPUs in mobile devices are based on ARM architecture. In contrary to desktop computers, mobile devices' CPUs are much slower since they run on a lower frequency and are more power-efficient. Because of their reduced working frequency, inference times of machine learning models are usually higher on mobile devices. This presents an important issue when deploying machine learning models to mobile device applications where inference speed plays an important role. A usual way to boost the performance is to optimize the model or leverage hardware accelerators.

1.3.2 Optimization techniques

Along with general neural network optimization techniques, such as Pruning or Weight sharing, there are a few notable optimization techniques, which are frequently adopted in a mobile environment.

Hardware Accelerators
Mobile environment may offer hardware accelerators designed specifically for neural networks. Taking advantage of these accelerators may result in a great performance gain. The most popular hardware accelerators are Graphics Processing Units (GPU). GPU accelerators are used for many purposes, which include activities such as gaming, cryptocurrency mining, and training or executing machine learning models. In machine learning and artificial intelligence, other used accelerators are Neural Processing Units (NPU) [33], or Tensor Processing Units (TPU) [34]. In computer vision and real-time applications, Digital Signal Processors (DSP), which immediately process signals from sensors, are also used.

In a mobile environment, the most used hardware accelerators are mobile GPUs, NPUs and DSPs. For example, in newer iPhones, we can find Apple's Neural Engine [35, 36], which is a type of an NPU. On the evaluation board, there is a GPU present.

Quantization
In most computer architectures, numerical data types such as integers or floats are usually represented in 32 bits. For integers in the context of computer vision or object detection, most of the time, 32 bits are redundant and consume more space and resources than necessary. Similar reasoning can be applied to decimal and floating-point numbers.

Quantization is an optimization method that tries to reduce redundancy by converting parameters in the model into less resource-consuming ones. The process of converting parameters sacrifices model accuracy for efficiency. There are several conversions available. In a mobile environment, we usually opt for 8-bit integer quantization or 16-bit float quantization. The choice which conversion to choose may vary. For example, if we aim for maximum memory efficiency, the 8-bit

integer quantization is the preferred approach [2]. There might be a hardware accelerator present in the system, which is only compatible with 16-bit floating-point numbers (GPU) or 8-bit integers (NPU). Each of them requires a different type of quantization.

There are two general types of quantization – quantization aware training and post-training quantization. Quantization aware training is a method employed in training the model. Parameters are simulated in low-precision mode, which improves the resulting accuracy of a quantized model because the model learns how to work with low-precision parameters. On the other hand, post-training quantization is applied after training. Parameters are then statically transformed into low-precision data types. Since there is no additional tuning, post-training quantization usually suffers some accuracy drop compared to non-quantized model or quantization aware training. Nevertheless, both methods can dramatically reduce the model size.

2. Research has shown that even 4-bit integers may work [37].

2 EfficientDet

EfficientDet [5] is a family of one-step object detection models published in November 2019. It is a complex model, which builds on top of other related research. Authors claim good scalability and efficiency, making it a good candidate for a model which can be used in a mobile environment. There are currently nine versions of the model, EfficientDet-D0 being the smallest one, scaling up to EfficientDet-D7 and EfficientDet-D7x, increasing in size and complexity.

2.1 Architecture

The architecture of EfficientDet is optimized for efficiency and scalability. Therefore, different versions of EfficientDet have different architecture. The architecture is determined by scaling coefficient ϕ, from which the architecture is derived. The scaling coefficient affects many different aspects of the network – depth, width, and resolution. Figure 2.1 shows the concrete numbers. Authors have previously explored this scaling coefficient in EfficientNets and have shown that such scaling brings positive results. Therefore, the same mechanism has also been employed in EfficientDets.

2.1.1 Backbone

Based on previous research, EfficientDet extends on efficiency and scalability of EfficientNet [6]. Han and Le [6] proposed a scaling method for neural networks which scales network width, depth, and resolution[1] in a way that preserves top performance, despite the model being up to 8 times smaller than other state-of-the-art models. This method has been incorporated in their EfficientNet models.

EfficientNet incorporates *Mobile Inverted Residual Bottlenecks* as building blocks, which were first introduced in [41], and optimize memory requirements as well as computational cost. To optimize even more, *Squeeze and Excitation* (SE) [42] layers are added to the network.

1. More details about scaling these architectural parameters can be found in [38, 39, 40].

Table 2.1: EfficientDet architecture scaling

Model	Input size	BiFPN layers	ClassNet and BoxNet layers
D0 ($\phi = 0$)	512	3	3
D1 ($\phi = 1$)	640	4	3
D2 ($\phi = 2$)	768	5	3
D3 ($\phi = 3$)	896	6	4
D4 ($\phi = 4$)	1024	7	4
D5 ($\phi = 5$)	1280	7	4
D6 ($\phi = 6$)	1280	8	5
D7 ($\phi = 7$)	1536	8	5

Squeeze and Excitation layers are used to capture channel-wise information in a better way than the standard convolution. Standard convolution fuses the information in local receptive fields across several channels. Squeeze and Excitation networks try to assign weights to each channel, strengthening the important and more informative ones while discarding the non-important ones. As authors of Squeeze and Excitations networks claim, incorporating SE layers in a model increases the overall computational cost by a very negligible amount while improving the network's ability to generalize. Furthermore, EfficientNet architecture was determined by Neural Architecture Search (NAS) [43], making the architecture more optimal and efficient than hand-crafted.

Like EfficientDet, EfficientNet comes in several versions ranging from EfficientNet-B0 to EfficientNet-B7. Due to all these optimizations and matching scaling versions, EfficientNet was chosen as the backbone for EfficientDet, providing a flexible and efficient backbone.

2.1.2 Neck

Feature Pyramid Network

The neck of EfficientDet once again builds on top of other structures. In order to describe the neck of EfficientDet, Feature Pyramid Networks (FPN) [26] need to be explained first. Feature Pyramid Networks were invented because of the scale variation problem. According to Lin et al. [26], it is still necessary to tackle multiscale problems with pyramid representations, even though deep convolutional neural networks are in general robust to scale variation.

In object detection, we may want to detect a dog. But we do not know whether the dog takes a lot of space in the image or is just lying around in the background. One way to tackle this problem is to use anchor boxes on different scales and try to classify the image in these boxes. Another way to solve this problem is to take feature maps from backbone layers and try to make a prediction using each one of the selected feature maps. This method is a predecessor of Feature Pyramid Network, which combines features from the last backbone layers (high-level features) with features from lower backbone layers (low-level features). Feature Pyramid Network method builds feature maps in a bottom-up pathway and top-down pathway with lateral connections. The bottom-up pathway is simply a feed-forward pass of the backbone. The top-down pathway takes the feature map from a higher layer, upsamples it, and adds it together with a feature map from the layer below. See Figure 2.1 for illustration. This process allows us to utilize low-level features from the backbone, which contribute to detecting small objects in the image.

When features get merged together, the process is called *feature fusion*. A concrete example of feature fusion is the following. Let P_{last} be an output feature map of the backbone's last convolutional layer. Only an arbitrary convolution operation is applied to this feature map, as there are no higher layers to fuse with. Let P_{last-1} be an output feature map of the backbone's second-to-last layer. This feature map is fused with P_{last} in the following way:

$$P^{out}_{last-1} = Conv(P_{last-1} + Resize(P_{last})) \tag{2.1}$$

In general,

$$P^{out}_{layer} = Conv(P_{layer} + Resize(P_{layer+1})) \tag{2.2}$$

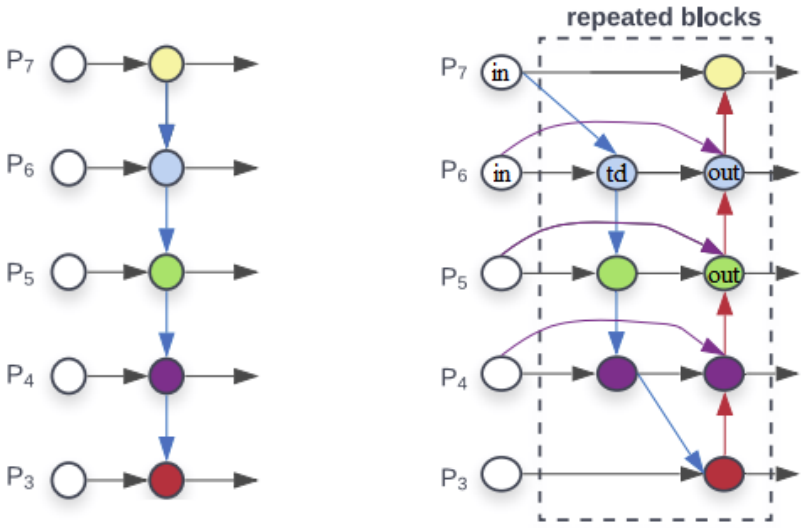
Resize operation is usually upsampling the feature map because features from higher layers usually have lower resolution than lower layers, and we need to match them. After fusion, the resulting feature maps contain both high-level and low-level features, which helps to achieve better overall accuracy. *Conv* operation is an arbitrary convolution applied to process these newly fused features. EfficientDet uses depthwise separable convolution.

Weighted Bi-Directional Feature Pyramid Network
Weighted Bi-Directional Feature Pyramid Networks build on top of FPN. Authors of EfficientDet stated in their work that when fusing features from different layers, contributions from layers are not equal. Some features are more important than others. Authors have decided to let the network learn this importance by itself by assigning weights to feature maps. Building on top of Equation 2.2, weighted feature fusion is computed like this:

$$P_{layer}^{out} = Conv(w_1 \cdot P_{layer} + w_2 \cdot Resize(P_{layer+1})) \tag{2.3}$$

Where w_1, w_2 are the weights. Authors of EfficientDet point out that these weights are unbounded and may cause training instability. For this reason, normalizing the weights to range $[0, 1]$ is considered a good approach. Authors have explored Softmax normalization and found out that Softmax implementation is significantly slower on GPU and therefore have resorted to their own method called *Fast normalized fusion*. Fast normalized fusion normalizes the weights by the sum of weights incoming to the node, with a small constant ϵ added to ensure that no numerical instability happens. Extending on Equation 2.3, normalized feature fusion equation looks like this:

$$P_{layer}^{out} = Conv(\frac{w_1 \cdot P_{layer} + w_2 \cdot Resize(P_{layer+1})}{w_1 + w_2 + \epsilon}) \tag{2.4}$$



(a) Feature Pyramid Network

(b) Weighted Bi-directional Feature Pyramid Network

Figure 2.1: Comparison of Feature Pyramid Networks. Taken from [26]. P_3 to P_7, in general P_i, represent an output from backbone's i-th layer and serve as an input to the i-th Bi-FPN level.

Lastly, bi-directional connections need to be introduced. Authors have based their decisions on NAS-FPN [44], and Path Aggregation Network [45] and have modified these architectures in a way that improves efficiency. Refer to Figure 2.1b for an illustration of BiFPN. Following the notation from the original work, an example feature fusion at level 6 is computed as follows (Cells that are used in the computation are highlighted in Figure 2.1b for better illustration.). First, a top-down fusion P_6^{td} has to be computed.

$$P_6^{td} = Conv(\frac{w_1 \cdot P_6^{in} + w_2 \cdot Resize(P_7^{in})}{w_1 + w_2 + \epsilon}) \tag{2.5}$$

Afterwards, a bottom-up feature fusion is computed to produce the output from the current level [5].

$$P_6^{out} = Conv(\frac{w_1' \cdot P_6^{in} + w_2' \cdot P_6^{td} + w_3' \cdot Resize(P_5^{out})}{w_1' + w_2' + w_3' + \epsilon}) \tag{2.6}$$

Note that the exact formulas differ across most of the levels due to the complicated structure of BiFPN. For example, comparing the top-down pathway computation for level 6 (See Equation 2.5) and level 5, we obtain the following. Weights are independent of previous equations.

$$P_5^{td} = Conv(\frac{w_1' \cdot P_5^{in} + w_2' \cdot Resize(P_6^{td})}{w_1' + w_2' + \epsilon}) \tag{2.7}$$

Notice the difference in *Resize* function argument. While level 6 fuses with the direct input of level 7 – P_7^{in}, level 5 fuses with the top-down pathway of level 6 – P_6^{td}.

Figure 2.1b illustrates a single BiFPN layer. There are several BiFPN layers in an EfficientDet model, allowing high-level feature fusion. The exact number of layers is determined by the aforementioned scaling coefficient ϕ. Refer to Table 2.1 for concrete numbers. Each layer produces five outputs from different levels that correspond to 5 inputs obtained from the backbone network. These outputs are fed into ClassNet and BoxNet for final predictions.

2.1.3 Head

EfficientDet for object detection[2] has two heads – BoxNet and ClassNet. Both networks are convolutional networks that use depthwise separable convolution to maximize efficiency. BoxNet and ClassNet always have the same amount of layers, and the exact number is again determined by EfficientDet's scaling factor ϕ. BoxNet is a network for bounding box regression, while ClassNet is concerned with predicting the classes and scores.

2.2 Activity

EfficientDet takes an image as an input, resized to the model's version size (See Figure 2.1), and converted to RGB format. Values are expected to be integers in the $[0, 255]$ range. The preprocessing is handled by the model itself and includes mean subtraction and standard deviation normalization. Preprocessed image is then fed to the backbone network.

Swish [46] activation function is used throughout the whole model, including EfficientNet backbone, BiFPN layers, BoxNet and ClassNet. According to [46], Swish activation function works better for deep neural network architectures than other commonly used activation functions. Swish has the following formula[3]:

$$f(x) = x \cdot sigmoid(x) \tag{2.8}$$

Where

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

Outputs from backbone layers 3 to 7 are used as an input for the BiFPN layer, where these features are fused together (See Equations 2.5 and 2.6 for an example fusion on level 6.). After the features are fused in several layers (See Table 2.1), they are fed into ClassNet and BoxNet. For each predefined anchor box, ClassNet produces logits from which

2. EfficientDet configuration can be modified, enabling e.g., image segmentation, which attaches another head to the network.

3. In the original paper, Swish was defined as $f(x) = x \cdot sigmoid(\beta x)$, where β was a constant or a trainable parameter. EfficientDet uses Tensorflow implementation, which sets the β parameter to 1.

classes and their corresponding scores are determined. An *argmax* operation is applied to logits to determine class labels, and *sigmoid* operation is applied to logits to obtain scores[4]. BoxNet, on the other hand, outputs offsets for predefined anchor boxes. These offsets have to be post-processed and mapped to actual bounding boxes.

After both outputs are merged, a Non-Max Suppression operation is called to ensure only the most valid results are kept. Non-Max Suppression discards all predictions that have a confidence score lower than a given threshold or have an IOU value (See Section 6.1) higher than a given threshold with another prediction whose confidence score is higher. The final output from a model is a tensor of fixed shape $[1, max\_detections, 7]$, where *max_detections* is a hyperparameter (100 by default), with each detection being encoded in an array with seven floating-point elements as follows:

$$[\, image\_id,\ ymin,\ xmin,\ ymax,\ xmax,\ score,\ class\,] \tag{2.10}$$

Where *image_id* is an id of an image (0 if only a single image is supplied to the network), $ymin, xmin, ymax, xmax$ are the coordinates of the predicted bounding box. These are not normalized to $[0, 1]$ range, as opposed to other models, and are the exact coordinates in the input image. *score* represents a confidence score for a given prediction, and finally, *class* is an integer in $[0, num\_classes)$ interval, where *num_classes* is a hyperparameter depending on the dataset, representing the number of classes in that dataset.

2.3 Training

All EfficientDet versions have been trained using the Stochastic Gradient Descent algorithm. A momentum of 0.9 along with weight decay 4e-5 has been used. The learning rate is linearly increased from 0 to 0.16 in the first training epoch and then annealed down using the cosine decay rule [5]. The loss function for a single image looks as follows:

$$L_{total} = \frac{L_{Class}}{N} + w_{Loc} \cdot \frac{L_{Loc}}{4N} \tag{2.11}$$

4. Sigmoid is commonly used in multi-label classification problems because it treats each of the values independently. Softmax would not produce independent results.

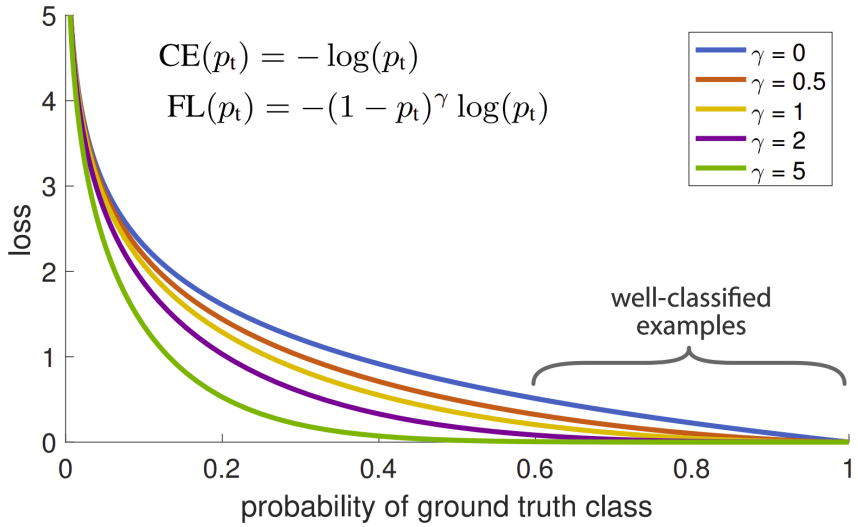


Figure 2.2: Focal loss with γ parameter. Setting γ to 0 transforms the focal loss to classical Cross Entropy. Taken from [47]

Where N is the number of anchor boxes that were matched to ground-truth boxes in the image. L_{Class} is classification loss in the form of focal loss. Let X be a vector of the network's predictions with real numbers in $[0, 1]$ range. Let Y be a one-hot encoded target vector. Denote by X_i the i-th element of vector X, representing the network's confidence that the output belongs to class i, and by Y_i the i-th element of vector Y. Over all possible i, define a vector P as follows. P_i represents an element on i-th index in vector P.

$$P_i = \begin{cases} X_i & \text{if } Y_i = 1 \\ 1 - X_i & \text{otherwise} \end{cases} \tag{2.12}$$

The formula of a focal loss is then

$$FL(P) = -\alpha(1-P)^{\gamma} \cdot log(P) \tag{2.13}$$

EfficientDet uses $\alpha = 0.25$ and $\gamma = 1.5$. Focal loss is a loss function that penalizes hard examples more than the easy ones. When training object detection models, the training set usually contains much more negative examples ("background" or "no detection" classes) than the

positive ones, resulting in a class imbalance. The focal loss puts the emphasis on hard examples and reduces the loss for well-classified examples. Well-classified predictions are considered predictions with a score greater than 0.5 [47]. Increasing the γ parameter decreases the loss for well-classified examples, see Figure 2.2. The second parameter α is a balancing parameter for classes.

L_{Loc} is a localization loss expressed by Huber loss [48]. Let X be a vector of network's outputs and Y a vector of target outputs. Define a vector V as $V = Y - X$. For each element x in vector V, Huber loss is computed as

$$HL(x) = \begin{cases} 0.5 \cdot x^2 & \text{if } |x| \leq d \\ 0.5 \cdot d^2 + d \cdot (|x| - d) & \text{if } |x| > d \end{cases} \tag{2.14}$$

where d is *delta* parameter, which is set to 0.1. w_{Loc} is localization error weight, that is by default set to 50.

2.4 EfficientDet-lite

At the time of writing this thesis, pre-trained checkpoints of additional six EfficientDet versions, called *EfficientDet-lite* were released. Since the authors suggest that these models are even more optimized for mobile devices, an analysis of those models is performed as well.

EfficientDet-lite models have a smaller input image size (See Table 2.2) and do not use Swish activation function, in contrary to the original version. Instead, they use ReLU6 activation function, whose formula is the following:

$$ReLU6(x) = min(max(0, x), 6) \tag{2.15}$$

Comparing Swish and ReLU6, ReLU6 can be pronounced as faster and more efficient because it only uses simple mathematical operations *min* and *max*. Swish, on the other hand, relies on a *sigmoid* operation (See Equation 2.9), which involves powers and fractions, which are computationally more expensive. Reducing the input size also reduces the computational needs for the model.

Table 2.2: EfficientDet-lite architecture scaling

Model	Input size	BiFPN layers	ClassNet and BoxNet layers
lite0	320	3	3
lite1	384	4	3
lite2	448	5	3
lite3	512	6	4
lite3x	640	6	4
lite4	640	7	4

3 SSD MobileNetV2

MobileNets are a family of object detection models, firstly published in 2017 [2]. MobileNets were designed to target resource-constrained mobile and embedded environment. This design led to a model called MobileNet with very fast inference times and reasonable accuracy. The major efficiency improvement was reached by re-thinking an idea of a general convolution operation and introducing depthwise separable convolution.

One year later, MobileNetV2 was introduced [41]. MobileNetV2 came with additional optimizations – *Inverted residuals* and *linear bottlenecks*. Evaluating MobileNetV2 showed further improvements in both accuracy and inference time compared to its predecessor, MobileNet. MobileNetV2 was also evaluated for object detection task with the SSDLite module. The difference between standard SSD and SSDLite modules is that SSDLite uses depthwise separable convolutions, as opposed to SSD, which uses standard convolutions.

Another year later, MobileNetV3 emerged, improving even further on MobileNetV2 results and targeting specifically mobile CPUs with two models – MobileNetV3-Large and MobileNetV3-Small. Performance gains stem from several optimizations, such as Neural Architecture Search [43], or replacing a computationally expensive sigmoid function by hard sigmoid, which is then used in hard-swish activation function [49].

For purposes of this work, MobileNetV2 with SSD module was chosen to be compared against EfficientDet due to its popularity in the object detection field. All further details presented in this work stand for SSD MobileNetV2.

3.1 Architecture

3.1.1 Backbone

MobileNetV2, which serves as a backbone for the SSD MobileNetV2 model, leverages an idea of embedding *manifolds of interest* into low-dimensional subspaces. This idea is embodied in the form of *linear bottlenecks* used in the model. *Inverted residuals* are also an invention

Table 3.1: MobileNetV2 bottleneck depth-separable convolution. h, w represent the height and width of the input, k the number of input channels, k' the number of output channels, t the expansion factor and s stride. Taken from [41]

Input	Operator	Output
$h \times w \times k$	1x1 Conv2D, ReLU6	$h \times w \times (tk)$
$h \times w \times (tk)$	depthwise 3x3, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times (tk)$	linear 1x1 Conv2d	$\frac{h}{s} \times \frac{w}{s} \times (k')$

that is introduced in this model. The authors of MobileNetV2 claim that these inventions improve performance.

As discussed in [41], the basic building block for MobileNetV2 is the *bottleneck depth-separable convolution with residuals*. This module is a 1x1 convolution to higher channel dimension, determined by *expansion factor t*, followed by depthwise 3x3 convolution and 1x1 linear convolution. The decision for linear convolution stems from mathematical reasoning about manifolds of interest. A summary can be viewed in Table 3.1. See Figure 3.1 for the overall module structure in Netron [50] visualization tool. Even though there are more convolution operations, the design of the network allows using input and output dimensions, which in the end results in a performance gain [41].

3.1.2 Head

As there is no neck present in the SSD MobileNetV2 model, features from several backbone layers are fed straight to the head. The head of SSD MobileNetV2 object detector is a standard SSD module [4], which outputs predictions for a predefined set of anchor boxes.

3.2 Activity

MobileNetV2 takes an input image of size 320x320. The image is expected to be in RGB format. Preprocessing includes normalizing pixel values to $[-1, 1]$ range by multiplying the image by $(2.0/255.0)$ and subtracting value 1 from each resulting pixel value.

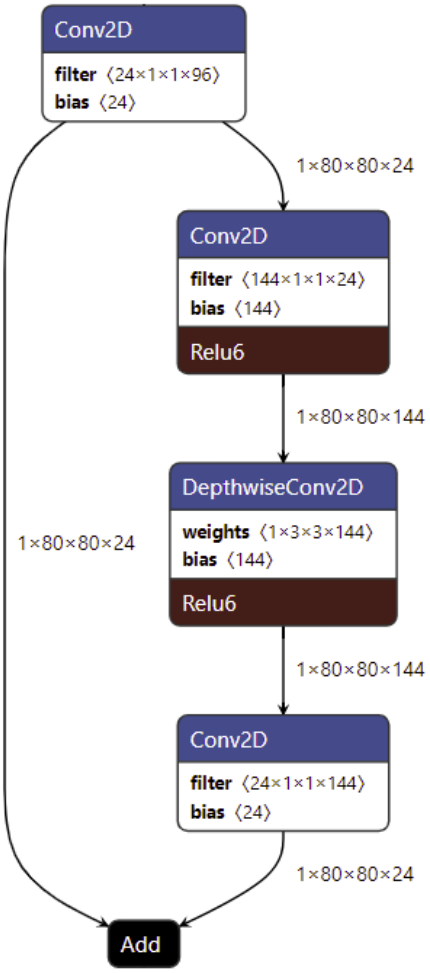


Figure 3.1: MobileNetV2 bottleneck depth-separable convolution with residual connection in Netron [50] visualization tool.

MobileNetV2 uses ReLU6 (See Equation 2.15) as its activation function.

Similar to EfficientDet, outputs from different backbone layers are supplied as an input to the SSD module for final predictions. SSD applies a single convolution to the input feature maps. After all predictions have been computed, post-processing is applied to produce final outputs. Post-processing is done by a Non-Max Suppression operation.

3.3 Training

MobileNetV2 was trained using the Stochastic Gradient Descent algorithm with RMSProp optimizer. Both weight decay and momentum were set to 0.9. An initial learning rate of 0.045 is multiplied by 0.98 after every epoch [41].

Since the SSD module is present in the model, the loss function is a weighted sum of the localization loss and the confidence loss. Because the precise definition is complex and notation-heavy, and, as opposed to EfficientDet, is available in the original paper, refer to [4] for the detailed description.

4 EfficientDet versus SSD MobileNetV2

MobileNetV2 [41] with SSD module is nowadays widely used in object detection on mobile platforms. It is one of the biggest competitors in this field, and therefore the analysis will be focused on this model. Analysis and comparison of both EfficientDet and SSD MobileNetV2 performance shall be conducted.

4.1 Backbones

Starting from the backbones, EfficientDet uses EfficientNet, whereas SSD MobileNetV2 employs MobileNetV2. Depending on which version of EfficientDet we use, input sizes range from 512x512 up to 1536x1536 (See Figure 2.1). MobileNetV2, on the other hand, requires an input of only 320x320. Both EfficientNet and MobileNetV2 use linear bottleneck depth-separable convolution with residuals, although EfficientNet utilizes other optimizations on top of these blocks (see Section 2.1.1). These optimizations present only a small relative computational overhead.

4.2 Necks

Neck is present only in EfficientDet. It is the BiFPN structure that fuses features from different backbone layers in order to gain a better semantical representation of the input image. There are several layers of BiFPN, depending once again on the specific version of EfficientDet that was chosen. Outputs from all five levels are fed into ClassNet and BoxNet. BiFPN is part of the network where authors of EfficientDet sacrifice latency times for accuracy. Fusing features costs some computational power but offers non-negligible accuracy gains – About four mAP points on COCO validation set when using EfficientDet-D3 [5].

4.3 Heads

EfficientDet's and MobileNetV2's heads work on the same principle and yet differ in some details, which may be crucial for performance

or accuracy. Both EfficientDet and MobileNetV2 output predictions for a predefined set of anchor boxes. EfficientDet uses 9 anchor boxes per feature map location (3 scales, 3 aspect ratios), whereas SSD MobileNetV2 uses 6 anchor boxes per feature map location (5 aspect ratios + 1 anchor box with aspect ratio 1 and extra scale[1]). Another difference is in the resolutions on which the heads operate. While EfficientDet-D0[2] heads operate on feature map resolutions ranging from 64x64 to 4x4 in a top-down manner, MobileNetV2 takes feature maps of resolution from 20x20 to 1x1. This results in a significant computational cost difference and also influences the postprocessing stage because EfficientDet's postprocessing needs to post-process several tens of thousands of outputs. Meanwhile, MobileNetV2 only post-processes outputs in thousands. Furthermore, EfficientDet's anchor box predictors (ClassNet, BoxNet) have several layers, as opposed to MobileNetV2's SSD module. To equalize things a bit, EfficientDets use depthwise separable convolutions to speed up the feed-forward pass. SSD module in MobileNetV2 uses standard convolutions. [3]

4.4 General overview

In general, it is important to mention that EfficientDet is meant to be efficient and scalable, but it is not primarily aimed at mobile and embedded devices. On the other hand, MobileNetV2 improves the design of MobileNet, which is specifically targeted at a constrained environment. There are several differences, with one being a key difference – EfficientDet generally uses a more complex structure and higher resolution features than MobileNetV2.

1. Scale S_k for this anchor box in SSD layer k is computed as $S_k = \sqrt{S_k S_{k+1}}$, as proposed in [4].
2. Higher EfficientDet versions operate on even higher resolution feature maps. For example, EfficientDet-D2 operates on resolutions ranging from 96x96 to 6x6.
3. Authors of MobileNetV2 have explored a new SSD module with MobileNetV2, called SSDLite, which changes standard convolutions to depthwise separable convolutions [4]. This optimization is not present in the model that is described and used for evaluation.

5 Implementation and environment

5.1 Implementation

Information about the source code is attached in the appendix of the work.

5.2 Environment

5.2.1 Target device

All measured experiments are conducted on i.MX 8QuadMax board, running YOCTO Linux 5.4 Zeus operation system. There are four cores of ARM Cortex-A53, along with two cores of ARM Cortex-A72. Arm Neon acceleration is used on this board[1]. For memory, 64-bit LPDDR4 memory at 1600Mhz is running on the board. Two Vivante Corporation GPUs GC7000XSVX are present [52], which can leverage Neural Networks API (NNAPI) interface and serve as TensorflowLite delegates for further acceleration. All measurement is conducted on CPU only, and no other hardware accelerators were used. The execution graph was run using four threads. Measuring application is written in C++, with C++17 standard set.

5.2.2 Host

The application was cross-compiled on Ubuntu Linux 19.04 (Disco Dingo) virtual machine. TensorflowLite 2.4.1 shared library along with Python 3.7 were used throughout the whole work. Shell scripts converting the original models from [5] use python 3.7 to convert all models.

5.3 Tools

Since the whole analysis runs in a mobile environment, the TensorflowLite library is used due to its compatibility with mobile devices.

1. Tensorflow Lite CPU kernels are heavily optimized for Arm Neon [51]

Tensorflow allows the conversion of models so that they can be executed on mobile devices. In order to use machine learning models with the TensorflowLite library, models must be converted to `.tflite` format. TensorflowLite converter used via Python API is used to convert all models. Execution of converted models is enabled via `TfLiteInterpreter` object, which provides an API for working with the model. TensorflowLite shared library (`libtensorflowlite.so`) is used and linked against in the application. TensorflowLite library was built using `Bazel 3.1.0` and `GCC 9.2.0`. Bazel is a build tool developed and maintained by Google and is a native build tool for Tensorflow libraries. The application is built using `GNU Make 4.2.1` build tool. Cross-compilation settings for the target board are set, and the application is cross-compiled using these settings. For image processing, the OpenCV library is used. Memory usage is measured via Linux `proc` tool, `/proc/<pid>/smaps_rollup` to be exact. Rollup version accumulates the result of `smaps` for better readability. Latency times are measured via standard C++ `chrono` library.

5.4 Models

5.4.1 EfficientDet

In order to obtain `.tflite` version of EfficientDet model, guidelines in official EfficientDet repository [9] were followed. As observed in an initial evaluation, the model checkpoint that is provided in the repository does not reach the reported performance unless hyperparameters are matched with those used during the official evaluation. This includes setting `min_score_thresh` from 0.5 to 0. This hyperparameter determines the minimum score threshold for Non-max Suppression operation in postprocessing. Another hyperparameter that is different from the official evaluation is `iou_thresh` which once again determines the intersection over union threshold in Non-Max Suppression. Using settings of these hyperparameters that is provided by checkpoints results in some detections being discarded due to these thresholds, resulting in an overall loss of mAP score.

For EfficientDet versions higher than D0 it was necessary to enable `TF_SELECT_OPS` in order to successfully convert the model. `TF_SELECT_OPS` enables TFLite Interpreter to pull Tensorflow core to execute opera-

tions that are by default not supported by TensorflowLite runtime. EfficientDet versions D0 and D2 are converted and analyzed.

Input images are loaded using the OpenCV library. Preprocessing includes changing BGR representation to RGB and resizing the image to match the EfficientDet version's input size.

5.4.2 EfficientDet-lite

From EfficientDet-lite models, versions lite0, lite2, and lite3 have been selected. The choice of lite0 and lite2 was made because versions D0 and D2 of the original model are evaluated. Versions lite0 and lite3 were chosen because of their input image size. Lite0 requires an input of size 320x320, which is the same as SSD MobileNetV2. Due to this fact, a comparison of these two models can be made more accurately. Lite3 requires an input of size 512x512, which is the same as the baseline EfficientDet-D0 model. Therefore, an evaluation of how much progress and efficiency has been achieved can be done.

Regarding the conversion, parameters, and image loading, everything remains the same as for the aforementioned EfficientDet models.

5.4.3 SSD MobileNetV2

Pre-trained SSD MobileNetV2 from [53] was used. The model was converted using TensorFlow Object Detection API guidelines. Converting an object detection model using this API requires the user to create an intermediate model, which can be afterwards converted to `.tflite` format. This model uses dynamic tensors, which have to be of a known size before conversion. Converting the model without the intermediate steps results in a model which has an input tensor of shape $(1, 1, 1, 3)$ where no image can be provided. MobileNet of input size 320x320 was converted. 320x320 image resolution is the baseline, and it is native SSD MobileNetV2 resolution. Similar to the EfficientDet model, some of the hyperparameters needed to be adjusted. Once again `score_threshold` hyperparameter was set to 0 and `iou_threshold` was set to 0.5 to ensure the same configuration as EfficientDet model.

Input images are again loaded with the OpenCV library. Preprocessing includes changing BGR representation to RGB, together with

normalizing the image to values $[-1, 1]$ by multiplying the image by $(2.0/255.0)$ and subtracting value 1 from each pixel.

6 Evaluation

Models are evaluated on `COCO test-dev2017` object detection dataset with bounding box annotations. Annotations are not public, and to evaluate the model, results are submitted to an evaluation server[1]. This server requires the output to be in a specific format as described in [8]. Both EfficientDet and Mobilenet were trained on `COCO train2017` dataset, therefore evaluating on `test-dev2017` is a fair option.

6.1 Evaluation Metrics

In the analysis, several metrics are taken into account. Metrics have been chosen with respect to a mobile environment and the metric's reputation in object detection.

6.1.1 Latency

In real-time object detection, latency is a crucial metric. There are two types of latencies. One type of latency measures the complete execution of the model, from image loading and preprocessing to reading the output tensors of the model. The second type of latency often referred to as inference latency, measures the latency of only feed-forward pass of the network. Input preprocessing and reading output tensors are excluded from the measurement. In the experiment, only inference latency is evaluated.

6.1.2 Memory usage

Mobile devices are very constrained in terms of memory. Although modern mobile devices, such as mobile phones and tablets, can provide several gigabytes of memory space, it is usually not enough for large models. There are techniques, such as quantization or pruning, which can help reduce the size of a model to a more convenient amount. Memory consumption of a model can be a deciding factor when deciding which model to use. Therefore, it makes it an important metric to take into account.

1. https://competitions.codalab.org/competitions/20794

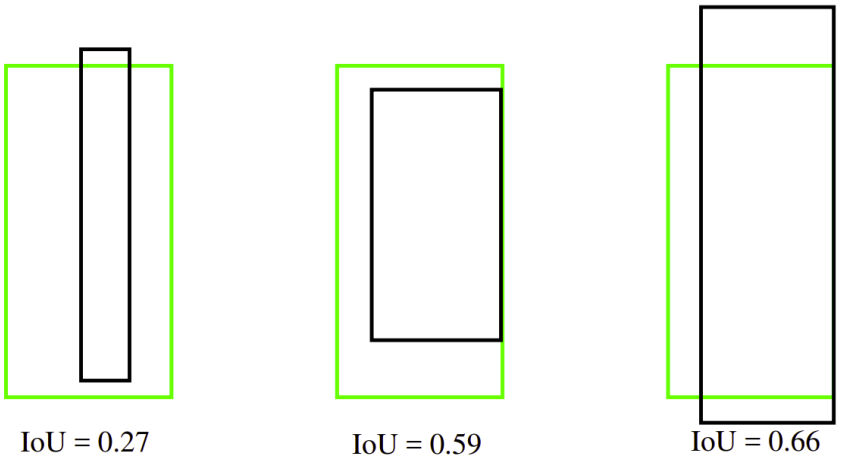


Figure 6.1: Illustration of various Intersection Over Union values. Taken from [54]

6.1.3 Mean Average Precision (mAP)

Mean average precision (mAP) is a common evaluation metric of object detection models. Most of the papers on object detection evaluate their models and approaches on either COCO or PASCAL VOC datasets by reporting mean average precision. Mean average precision is a useful metric to quantify the quality of an object detection model. COCO dataset uses automatical evaluation on a remote server which produces an evaluation result of submitted predictions. The following explanation holds for the COCO dataset.

To describe mean average precision, first, we need to define basic precision. Suppose we evaluate the model on an annotated dataset and produce the predictions. One has to decide whether for a given object in a given image the prediction is correct (True Positive) or not (False Positive).

In object detection, usually, the decision is based on an *Intersection Over Union*[2] *threshold* (IOU threshold). Consider the example in Figure 6.2a. The green bounding box represents a ground-truth bounding box, and the red bounding box represents the model's prediction. An average human could state that the prediction is not perfect. In

2. Also known as the Jaccard index.

order to quantify the quality of the prediction, we use Intersection Over Union value that, given two bounding boxes $b1, b1$ expresses the overall overlap of the boxes. IOU is defined as

$$IOU(b1, b2) = \frac{Area(b1 \cap b2)}{Area(b1 \cup b2)} \tag{6.1}$$

Where *Area* is a function that computes the two-dimensional area represented by the intersection or union of boxes. See Figure 6.1 for an illustration of different IOU values. IOU produces real values in $[0, 1]$ range, where 1 represents perfect overlap and 0 means that there is no intersection between $b1$ and $b2$.

In a single image, given a prediction P, with predicted class label P_c and predicted bounding box P_{bbox}, we set an IOU threshold t (usually 0.5) and determine whether the prediction is correct or not as follows. The prediction is correct (True Positive) if there exists a ground-truth bounding box GT_{bbox} and ground-truth class label GT_c, such that

$$IOU(P_{bbox}, GT_{bbox}) \geq t \tag{6.2}$$

and

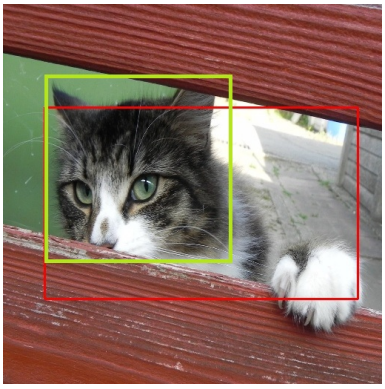
$$P_c = GT_c \tag{6.3}$$

If there is no ground-truth bounding box with an IOU value greater or equal to t, or the predicted class label does not match, the prediction is considered a wrong one (False Positive) [55].

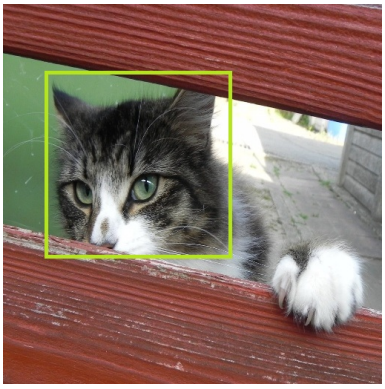
Suppose there are N images in the dataset with class C. Denote by TP the number of True Positive predictions across these N images, and FP the number of False Positives predictions across these N images. Precision for class C is then defined as

$$Precision = \frac{TP}{TP + FP} \tag{6.4}$$

Once we obtain this classification, whether detections are True Positives or False Positives, we can evaluate the precision of all classes at different values of the IOU threshold, yielding a list of precisions. COCO evaluation employs this technique and produces results evaluated at different thresholds. Since there are multiple object categories in the dataset, precision at each IOU threshold is calculated separately



(a) Prediction where IOU needs to be calculated



(b) An example of False Negative detection (no bounding box predicted)

Figure 6.2: An illustration of how ground-truth bounding box (green) versus predicted bounding box (red) might look like. (a) To quantify the quality of the prediction, IOU value between the two bounding boxes is computed. Reference image taken from [56].

for each class, meaning there are `num_classes` precision values at each IOU threshold level. All `num_classes` predictions at a given level are then averaged to produce the Average Precision value at that level. The final mean average precision is obtained by averaging precisions from all IOU threshold levels. This is the metric reported by the research paper and is also the primary metric for the COCO dataset.

In addition to the primary metric, COCO evaluation also provides the exact values of average precision at IOU threshold of 0.5 and 0.75, along with average precision values for objects of different scales (See [8] for details).

6.1.4 Average Recall

Another commonly reported metric of object detection models is Average Recall (AR). Similarly to mean average precision, we first need to define basic recall. Suppose we evaluate a model on a testing image, in which there is only one ground-truth bounding box for a single class, and our model predicts that there is no object in the image (See Figure 6.2b). Such detection is considered to be a False Negative because

there is no detection, even though there should be one. Following the notation from Subsection 6.1.3, Denote by FN the number of False Negative detections across N images containing class C. The recall is then defined as

$$Recall = \frac{TP}{TP + FN} \tag{6.5}$$

COCO evaluation again computes the recall separately for all classes across different IOU thresholds. In the case of recall, some metrics evaluate average recall with max_Dets parameter, which limits the maximum number of detections per image. Similar to average precision, COCO evaluation also provides average recall values for objects of different scales.

6.1.5 Number of detections

For some applications, such as self-driving cars, it might be necessary to detect a pedestrian or a car even with a lower confidence score than the usual threshold of 0.4 or 0.5 because an immediate action may be required. These applications must follow strict safety rules, and there are still open discussions and negotiations being held in this area. In fact, NXP Semiconductors suggested that this metric is taken into account. Because of this requirement, number of detections across different threshold values (0.25, 0.40, 0.50, 0.75) are evaluated.

6.2 Results

6.2.1 Latency

Latency was measured by taking the time difference between start and end time of inference – `TfLiteInterpreter::Invoke()` call. Loading model, allocating tensors, setting input values, and reading output values is excluded from this measurement. Only the actual time of a single feed-forward pass is measured. As the EfficientDet model handles input preprocessing and output postprocessing directly in the model, the final inference times are measured with these steps included. SSD MobileNetV2, on the other hand, expects an already pre-processed image on its input but still uses postprocessing to produce final outputs. This is a slight misalignment, as EfficientDet requires to

Table 6.1: Latency results of compared models in miliseconds. Average inference time is measured across 20288 test images

Model	Average inference time (ms)
SSD MobileNetV2	158.7
EfficientDet-lite0	398.3
EfficientDet-lite2	985.2
EfficientDet-lite3	1581.4
EfficientDet-D0	2983.1
EfficientDet-D2	11768.5

do prior computation before an actual feed-forward pass. Nevertheless, it has a negligible effect on the general results. Latency results are depicted in Table 6.1. The latency across all different models varies a lot. SSD MobileNetV2 was the fastest, with an average inference time of 158ms (6.33 FPS). The second fastest model, EfficientDet-lite0, with an average inference time of 398ms (2.51 FPS), is about 2.5x slower than SSD MobileNetV2. Compared to EfficientDet-D0, with an average inference time of 2983ms (0.33 FPS), the lite version performs much better in terms of latency. Another interesting comparison is between EfficientDet-lite3 and EfficientDet-D0 since they both take an input of size 512x512. EfficientDet-lite3, however, is roughly 2x faster than its standard counterpart.

6.2.2 Memory usage

Memory usage was measured using Linux `/proc` tool at four stages in the application.

1) After initialization of the application, before the model is loaded
2) After the model is loaded
3) After the model's tensors have been allocated
4) Before each inference

Taking the difference between 1) and 2) yields the model definition's memory usage, the difference between 2) and 3) gives us the memory needed for the model's tensors. Note that these tensors are being

Table 6.2: Memory usage results of compared models. All values are in kilobytes. TFlite file size denotes how much space does `.tflite` file take, Model size denotes the memory consumption of loading the execution graph and Tensor memory denotes the amount of memory requested to be allocated by `TfLiteInterpreter`.

Model	TFLite file size	Model size	Tensor memory
SSD MobileNetV2	23683	5320	84
EfficientDet-lite0	13368	6048	108
EfficientDet-lite2	22267	9824	104
EfficientDet-lite3	36524	12780	104
EfficientDet-D0	16390	7624	200
EfficientDet-D2	35040	16616	428

reused throughout the computation. Therefore, the amount of needed memory is not very demanding. Measuring the memory before each inference may provide an insight into how the memory requirements change over time. From the experiment, an increasing tendency is observed. Results can be viewed in Table 6.2.

6.2.3 Mean Average Precision

The inference was run on all test images from `COCO test-dev2017` dataset, and all outputs were logged for further processing. After the processing, results were submitted to the COCO evaluation server for remote evaluation. Results obtained were as follows in Table 6.3. Observed results were all lower than the performance reported in the models' papers. EfficientDet-D0 achieved a 33.1 mAP score, while the reported performance in the paper was 34.6 mAP score. EfficientDet-D2 reached a 41.3 mAP score, as opposed to 43.9 mAP reported in the paper. According to [53], SSD MobileNetV2 should reach 20.2 mAP score on `COCO test-dev2017`, but in my experiments, a result of only 11.7 mAP score was achieved. The difference is most likely caused by different hyperparameter settings. Paper results of EfficientDet could be reproduced by using the authors' evaluation script. I found out that this evaluation script used different hyperparameter settings than

Table 6.3: Mean Average Precision results of compared models on `COCO test-dev2017 dataset`. The mAP is the AP averaged over all IOU thresholds and categories, AP_{50} is the AP averaged over all categories at 0.5 IOU threshold. Similarly, AP_{75} is the AP averaged over all categories at a 0.75 IOU threshold. AP_{small}, AP_{med}, AP_{large} are the average precisions on small, medium and large objects.

Model	mAP	AP_{50}	AP_{75}	AP_{small}	AP_{med}	AP_{large}
SSD MobileNetV2	0.117	0.273	0.078	0.010	0.098	0.232
EfficientDet-lite0	0.266	0.435	0.277	0.075	0.286	0.422
EfficientDet-lite2	0.346	0.527	0.365	0.147	0.372	0.497
EfficientDet-lite3	0.380	0.558	0.403	0.175	0.418	0.534
EfficientDet-D0	0.331	0.513	0.350	0.131	0.366	0.483
EfficientDet-D2	0.413	0.602	0.442	0.221	0.445	0.553

the provided pre-trained checkpoints, yielding different evaluation results.

6.2.4 Average Recall

Similar process as in evaluating mean average precision was applied. Results were submitted to COCO evaluation server. Results can be found in Table 6.4.

6.2.5 Number of detections

In most of the state-of-the-art models, usual thresholds of 0.4 and 0.5 are used to filter out detections and non-detections. While this threshold is useful in applications where we require good detecting results, there may be applications in which it is critical to detect an object, even if its confidence score is low compared to other detections. I have therefore extracted detections with a set of thresholds and compared the average number of detections across `COCO test-dev2017` dataset. The results can be found in Table 6.5. The results suggest that SSD MobileNetV2 produces more detections of lower scores, whereas EfficientDet tends to be more certain with its detections. In

Table 6.4: Average Recall results of compared models on `COCO test-dev2017` dataset. AR max values represent average recalls when the maximum number of detections per image is limited to a specific value (1, 10, 100). AR small, med, large represent average recall values across small, medium, and large-sized objects.

Model	AR max1	AR max10	AR max100	AR small	AR med	AR large
SSD MobileNetV2	0.133	0.166	0.166	0.014	0.132	0.340
EfficientDet-lite0	0.233	0.348	0.373	0.118	0.414	0.567
EfficientDet-lite2	0.284	0.436	0.465	0.226	0.509	0.641
EfficientDet-lite3	0.305	0.471	0.501	0.257	0.556	0.676
EfficientDet-D0	0.275	0.418	0.447	0.196	0.499	0.626
EfficientDet-D2	0.323	0.503	0.537	0.320	0.579	0.690

fact, EfficientDet-D0 has, on average, about 30% more detections at a score threshold of 0.75. EfficientDet-D2 almost doubles the number of detections at a 0.75 score threshold compared to SSD MobileNetV2. Similar tendencies have been observed on EfficientDet-lite models as well.

6.2.6 Precision per-class analysis

In addition to the baseline metrics, an inspection of the model's performance across all classes was performed as well. Exported `csv` files can be found in `results` directory.

Inspecting the results, the most well-detected and worst-detected classes with their respective AP scores were the following. See Table 6.6 and Table 6.7.

From a general point of view, the most successfully detected objects were animals (bear, giraffe, elephant) and objects related to means of transport or traffic (train, bus, stop-sign). The fact that the networks were able to perform well on vehicles and traffic-related objects is a crucial result, especially for a company like NXP Semiconductors.

On the other hand, the hardest object to detect was a hair drier, which was the hardest object for detection for all evaluated models.

Table 6.5: Average number of detections with score thresholds of 0.25, 0.40, 0.50, 0.75 across all `COCO test-dev2017` dataset

Model	0.25	0.40	0.50	0.75
SSD MobileNetV2	25.71	8.96	4.60	0.67
EfficientDet-lite0	5.28	2.84	2.05	0.62
EfficientDet-lite2	6.02	3.45	2.55	0.90
EfficientDet-lite3	6.29	3.69	2.82	1.17
EfficientDet-D0	5.87	3.39	2.54	0.96
EfficientDet-D2	6.59	4.05	3.08	1.24

Other objects that were hard to detect were a handbag, toaster, spoon, or book. I think that the successfully detected objects have one thing in common. They are usually large objects in real life (train, giraffe, elephant). Additionally, in the case of well-detected animals, most of them have a property that is specific just for them. Elephants have a trunk, and giraffes are of yellow color with unusual body shape (long legs, tall neck). The experiment suggests that large objects and objects with a specific property are easier for the networks to detect. On the other hand, objects that were hard to detect were mostly objects of smaller size. A hair drier, spoon, handbag, or book are very small objects in real life compared to an elephant or a train.

In my opinion, the reason why small objects were harder to detect could be because of the input image size. Models with input sizes ranging from 320x320 to 512x512 had big trouble detecting a hair drier. On the other hand, EfficientDet-D2, which operates on an input image size of 768x768, experienced about four times better average precision score for hair drier class. The improvement was not so dramatic with other classes.

6.2.7 Recall per-class analysis

To complement precision tables from previous section, the best and worst recall values for specific classes are also provided. See Table 6.8 and Table 6.9.

Table 6.6: Top three classes and their corresponding AP scores across the dataset.

Model	Top 1 class	Top 1 score (AP)	Top 2 class	Top 2 score (AP)	Top 3 class	Top 3 score (AP)
SSD MobileNetV2	train	0.335	giraffe	0.319	bus	0.29
EfficientDet-lite0	train	0.579	stop-sign	0.571	giraffe	0.567
EfficientDet-lite2	giraffe	0.652	bear	0.645	elephant	0.645
EfficientDet-lite3	giraffe	0.693	bear	0.688	elephant	0.681
EfficientDet-D0	bear	0.643	elephant	0.638	giraffe	0.636
EfficientDet-D2	bear	0.718	giraffe	0.707	elephant	0.697

Table 6.7: Worst three classes and their corresponding AP scores across the dataset.

Model	Worst 1 class	Worst 1 score (AP)	Worst 2 class	Worst 2 score (AP)	Worst 3 class	Worst 3 score (AP)
SSD MobileNetV2	hair drier	0.004	toaster	0.006	handbag	0.008
EfficientDet-lite0	hair drier	0	handbag	0.03	toaster	0.032
EfficientDet-lite2	hair drier	0.011	toaster	0.066	spoon	0.075
EfficientDet-lite3	hair drier	0.028	book	0.095	handbag	0.104
EfficientDet-D0	hair drier	0.025	handbag	0.069	book	0.075
EfficientDet-D2	hair drier	0.106	book	0.123	spoon	0.134

Table 6.8: Top three classes and their corresponding AR scores across the dataset. All AR values are the COCO evaluation AR_{max100} values.

Model	Top 1 class	Top 1 score (AR)	Top 2 class	Top 2 score (AR)	Top 3 class	Top 3 score (AR)
SSD MobileNetV2	train	0.422	giraffe	0.393	cat	0.385
EfficientDet-lite0	toilet	0.671	train	0.660	giraffe	0.650
EfficientDet-lite2	toilet	0.722	giraffe	0.719	train	0.718
EfficientDet-lite3	giraffe	0.756	toilet	0.751	elephant	0.745
EfficientDet-D0	giraffe	0.711	toilet	0.710	elephant	0.707
EfficientDet-D2	giraffe	0.769	bear	0.764	elephant	0.758

The top results are similar to the results of per-class average precision. Among the best classes in terms of recall, there are two new classes that were not present in per-class average precision analysis - toilet and cat.

The class with the worst average recall values in all EfficientDet models is again a hair drier. A significant difference between EfficientDet-lite0 and other EfficientDet models can be observed in recall values of a hair drier class. A noticeable class in terms of bad recall values is a spoon. Spoon placed among the worst 3 classes in terms of recall values in all evaluated models.

Table 6.9: Worst three classes and their corresponding AR scores across the dataset. All AR values are the COCO evaluation AR_{max100} values.

Model	Worst 1 class	Worst 1 score (AR)	Worst 2 class	Worst 2 score (AR)	Worst 3 class	Worst 3 score (AR)
SSD MobileNetV2	toaster	0.012	handbag	0.015	spoon	0.018
EfficientDet-lite0	hair drier	0.005	spoon	0.108	toaster	0.108
EfficientDet-lite2	hair drier	0.137	spoon	0.187	knife	0.209
EfficientDet-lite3	hair drier	0.146	spoon	0.224	knife	0.244
EfficientDet-D0	hair drier	0.124	spoon	0.173	knife	0.189
EfficientDet-D2	hair drier	0.185	spoon	0.278	knife	0.301

6.3 Summary

From the observed results, we can notice that standard versions of EfficientDet tend to be more accurate and confident in their predictions and require a bit more memory than MobileNetV2 but come with much greater inference times. In a mobile environment, inference times of about 3 seconds for EfficientDet-D0 and 12 seconds for EfficientDet-D2 are unacceptable in real-time applications.

Since EfficientDet is a fairly new model, it was not possible to execute the model on mobile GPU using NNAPI due to unsupported operations that EfficientDet uses. Attempts to run the model using NNAPI resulted in the unsupported operations being transferred to CPU for computation, which requires extra synchronization, and as a result, inference times were slower than pure CPU execution, mostly because pure CPU execution can leverage the advantage of ARM Neon acceleration. Quantization of the model was also unsuccessful, as the confidence scores and class labels of predictions were corrupted[3]. If an implementation of unsupported operations is added to NNAPI and the model is correctly quantized, it might be possible to efficiently use EfficientDet in mobile applications, although most likely not in real-time applications.

3. Apart from confidence scores and a bit off class labels, after a manual inspection, the bounding box predictions seemed to correctly localize some objects.

It is important to stress out the fact that EfficientDet was designed to be efficient, although it is not targeted at a mobile environment[4].

EfficientDet-lite versions of EfficientDet, on the other hand, performed very well and were much closer to SSD MobileNetV2 performance in terms of speed. Although the fastest version, EfficientDet-lite0 was still about 2.5x times slower than SSD MobileNetV2, it showed an impressive 7.5x speed-up compared to the standard EfficientDet-D0 version while retaining a reasonable mAP score. Despite several optimizations, due to the nature of EfficientDet architecture, EfficientDet-lite versions are still more computationally demanding than SSD MobileNetV2.

Overall, EfficientDet-lite versions of EfficientDet are much more suitable for use in mobile and embedded environment than the standard versions. In terms of inference speed, SSD MobileNetV2 is still the faster one.

A notable result from the experiment, especially for NXP, is that the networks perform well on large real-life objects and generally objects that are related to means of transport or traffic. These results suggest that neural networks are a viable approach, and it might be rewarding to explore object detection with neural networks even more.

4. In EfficientNet paper [6], it is explicitly stated that authors did not target any specific device and preferred optimizing the number of floating-point operations per second (FLOPS), rather than latency.

7 Conclusion and Future work

In this work, a comparison of state-of-the-art EfficientDet models and SSD MobileNetV2 was conducted. Latency, memory consumption, number of detections, and overall mAP score were measured and analyzed. Results show that EfficientDet in its standard version is at the moment not a viable model for applications that require real-time image processing. EfficientDets offer higher accuracy at the cost of higher inference times than its counterpart, SSD MobileNetV2.

On the other hand, EfficientDet-lite models showed interesting and promising results. If enablement of EfficientDet-lite models using hardware accelerators is achieved, these models could make their entrance into mobile environment usage.

At the time of writing this work, the authors of EfficientDet proposed a new family of convolutional neural networks, called EfficientNetV2 [57]. In future work, experimenting with EfficientNetV2 in the context of object detection, namely using EfficientNetV2 as a backbone, is expected. After support of currently unsupported operations is provided, or inconsistencies when quantizing EfficientDet are resolved, further experiments with EfficientDet can be carried out.

This work was realized in cooperation with NXP Semiconductors. The company showed interested in EfficientDet and its application possibilities in mobile and embedded environment. Results of this work shall provide insight for further decisions about employing EfficientDet models in their products. The implementation part of this work can serve as an example of how to execute EfficientDet models using the TensorflowLite library on a mobile device.

Bibliography

1. REDMON, J.; DIVVALA, S.; GIRSHICK, R.; FARHADI, A. You Only Look Once: Unified, Real-Time Object Detection. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. Available from DOI: 10.1109/CVPR.2016.91.
2. HOWARD, Andrew G.; ZHU, Menglong; CHEN, Bo; KALENICHENKO, Dmitry; WANG, Weijun; WEYAND, Tobias; ANDREETTO, Marco; ADAM, Hartwig. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv e-prints*. 2017. Available from arXiv: 1704.04861 [cs.CV].
3. HUANG, R.; PEDOEEM, J.; CHEN, C. YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers. In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 2503–2510. Available from DOI: 10.1109/BigData.2018.8621865.
4. LIU, Wei; ANGUELOV, Dragomir; ERHAN, Dumitru; SZEGEDY, Christian; REED, Scott; FU, Cheng-Yang; BERG, Alexander C. SSD: Single Shot MultiBox Detector. In: LEIBE, Bastian; MATAS, Jiri; SEBE, Nicu; WELLING, Max (eds.). *Computer Vision – ECCV 2016*. Cham: Springer International Publishing, 2016, pp. 21–37. ISBN 978-3-319-46448-0. Available from DOI: 10.1007/978-3-319-46448-0_2.
5. TAN, M.; PANG, R.; LE, Q. V. EfficientDet: Scalable and Efficient Object Detection. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 10778–10787. Available from DOI: 10.1109/CVPR42600.2020.01079.
6. TAN, Mingxing; LE, Quoc. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In: CHAUDHURI, Kamalika; SALAKHUTDINOV, Ruslan (eds.). *Proceedings of the 36th International Conference on Machine Learning*. PMLR, 2019, vol. 97, pp. 6105–6114. Proceedings of Machine Learning Research. Available also from: http://proceedings.mlr.press/v97/tan19a.html.

7. LIN, Tsung-Yi; MAIRE, Michael; BELONGIE, Serge; HAYS, James; PERONA, Pietro; RAMANAN, Deva; DOLLÁR, Piotr; ZITNICK, C. Lawrence; PAJDLA, Tomas; SCHIELE, Bernt; TUYTELAARS, Tinne. Microsoft COCO: Common Objects in Context. In: *Computer Vision – ECCV 2014*. Cham: Springer International Publishing, 2014, pp. 740–755. ISBN 978-3-319-10602-1. Available from DOI: 10.1007/978-3-319-10602-1_48.

8. LIN, Tsung-Yi; PATTERSON, Genevieve; R. RONCHI, Matteo; CUI, Yin; MAIRE, Michael; BELONGIE, Serge; BOURDEV, Lubomir; GIRSHICK, Ross; HAYS, James; PERONA, Pietro; RAMANAN, Deva; DOLLÁR, Piotr; ZITNICK, Larry. *COCO - Common Objects in Context*. 2015. [Online]. Available also from: https://cocodataset.org. Accessed: 2021-03-27.

9. TAN, M.; PANG, R.; LE, Q. V. *EfficientDet*. 2019 [Github repository]. Available also from: https://github.com/google/automl/tree/master/efficientdet. Accessed: 2021-05-24.

10. AMIT, Yali; FELZENSZWALB, Pedro. Object Detection. In: 2014, pp. 537–542. ISBN 978-0-387-30771-8. Available from DOI: 10.1007/978-0-387-31439-6_660.

11. SAHU, S.; SARMA, H.; JYOTI BORA, D. Image Segmentation and its Different Techniques: An In-Depth Analysis. In: *2018 International Conference on Research in Intelligent and Computing in Engineering (RICE)*. 2018, pp. 1–7. Available from DOI: 10.1109/RICE.2018.8509038.

12. DALAL, N.; TRIGGS, B. Histograms of oriented gradients for human detection. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. 2005, vol. 1, 886–893 vol. 1. Available from DOI: 10.1109/CVPR.2005.177.

13. LINDEBERG, T. Scale Invariant Feature Transform. *Scholarpedia*. 2012, vol. 7, no. 5, p. 10491. Available from DOI: 10.4249/scholarpedia.10491. revision #153939.

14. GIRSHICK, R.; DONAHUE, J.; DARRELL, T.; MALIK, J. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 580–587. Available from DOI: 10.1109/CVPR.2014.81.

15. GIRSHICK, R. Fast R-CNN. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1440–1448. Available from DOI: 10.1109/ICCV.2015.169.

16. REN, S.; HE, K.; GIRSHICK, R.; SUN, J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2017, vol. 39, no. 6, pp. 1137–1149. Available from DOI: 10.1109/TPAMI.2016.2577031.

17. HE, Kaiming; GKIOXARI, Georgia; DOLLÁR, Piotr; GIRSHICK, Ross. Mask R-CNN. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2980–2988. Available from DOI: 10.1109/ICCV.2017.322.

18. CHAHAL, Karanbir Singh; DEY, Kuntal. A Survey of Modern Object Detection Literature using Deep Learning. *arXiv e-prints*. 2018. Available from arXiv: 1808.07256 [cs.CV].

19. KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks. *Communications of the ACM*. 2017, vol. 60, no. 6, pp. 84–90. ISSN 0001-0782. Available from DOI: 10.1145/3065386.

20. DONG, Hongwei; ZHANG, Lamei; ZOU, and. PolSAR Image Classification with Lightweight 3D Convolutional Networks. *Remote Sensing*. 2020, vol. 12, p. 396. Available from DOI: 10.3390/rs12030396.

21. YANI, Muhamad; IRAWAN, S; S.T., M.T. Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail. *Journal of Physics: Conference Series*. 2019, vol. 1201, p. 012052. Available from DOI: 10.1088/1742-6596/1201/1/012052.

22. MAO, Qi-Chao; SUN, Hong-Mei; LIU, Yan-Bo; JIA, Rui-Sheng. Mini-YOLOv3: Real-Time Object Detector for Embedded Applications. *IEEE Access*. 2019, vol. 7, pp. 133529–133538. Available from DOI: 10.1109/ACCESS.2019.2941547.

23. GAL, Shira. *Free black cats Stock Photo*. [N.d.]. Available also from: https://www.freeimages.com/photo/black-cats-1187325. Accessed: 2021-05-20.

24. UIJLINGS, Jasper; SANDE, K.; GEVERS, T.; SMEULDERS, A.W.M. Selective Search for Object Recognition. *International Journal of Computer Vision*. 2013, vol. 104, pp. 154–171. Available from DOI: 10.1007/s11263-013-0620-5.

25. DONAHUE, Jeff; JIA, Yangqing; VINYALS, Oriol; HOFFMAN, Judy; ZHANG, Ning; TZENG, Eric; DARRELL, Trevor. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. In: XING, Eric P.; JEBARA, Tony (eds.). *Proceedings of the 31st International Conference on Machine Learning*. Bejing, China: PMLR, 2014, vol. 32, pp. 647–655. Proceedings of Machine Learning Research, no. 1.

26. LIN, T.; DOLLÁR, P.; GIRSHICK, R.; HE, K.; HARIHARAN, B.; BELONGIE, S. Feature Pyramid Networks for Object Detection. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 936–944. Available from DOI: 10.1109/CVPR.2017.106.

27. LI, X.; TIAN, Y.; ZHANG, F.; QUAN, S.; XU, Y. Object Detection in the Context of Mobile Augmented Reality. In: *2020 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. 2020, pp. 156–163. Available from DOI: 10.1109/ISMAR50242.2020.00037.

28. HAN, Song; MAO, Huizi; DALLY, William J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv e-prints*. 2015. Available from arXiv: 1510.00149 [cs.CV].

29. WU, Jiaxiang; LENG, Cong; WANG, Yuhang; HU, Qinghao; CHENG, Jian. Quantized Convolutional Neural Networks for Mobile Devices. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

30. JACOB, Benoit; KLIGYS, Skirmantas; CHEN, Bo; ZHU, Menglong; TANG, Matthew; HOWARD, Andrew; ADAM, Hartwig; KALENICHENKO, Dmitry. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.

31. HAN, Song; POOL, Jeff; TRAN, John; DALLY, William J. Learning both Weights and Connections for Efficient Neural Networks. *arXiv e-prints*. 2015. Available from arXiv: 1506.02626 [cs.NE].

32. WANG, Ying; LI, Huawei; LI, Xiaowei. Re-architecting the on-chip memory sub-system of machine-learning accelerator for embedded devices. In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016, pp. 1–6. Available from DOI: 10.1145/2966986.2967068.

33. FIACK, L.; RODRIGUEZ, L.; MIRAMOND, B. Hardware design of a neural processing unit for bio-inspired computing. In: *2015 IEEE 13th International New Circuits and Systems Conference (NEWCAS)*. 2015, pp. 1–4. Available from DOI: 10.1109/NEWCAS.2015.7181997.

34. JOUPPI, Norman P.; YOUNG, Cliff; PATIL, Nishant; PATTERSON, David; AGRAWAL, Gaurav; BAJWA, Raminder; BATES, Sarah; BHATIA, Suresh; BODEN, Nan; BORCHERS, Al; BOYLE, Rick; CANTIN, Pierre-luc; CHAO, Clifford; CLARK, Chris; CORIELL, Jeremy; DALEY, Mike; DAU, Matt; DEAN, Jeffrey; GELB, Ben; GHAEMMAGHAMI, Tara Vazir; GOTTIPATI, Rajendra; GULLAND, William; HAGMANN, Robert; HO, C. Richard; HOGBERG, Doug; HU, John; HUNDT, Robert; HURT, Dan; IBARZ, Julian; JAFFEY, Aaron; JAWORSKI, Alek; KAPLAN, Alexander; KHAITAN, Harshit; KILLEBREW, Daniel; KOCH, Andy; KUMAR, Naveen; LACY, Steve; LAUDON, James; LAW, James; LE, Diemthu; LEARY, Chris; LIU, Zhuyuan; LUCKE, Kyle; LUNDIN, Alan; MACKEAN, Gordon; MAGGIORE, Adriana; MAHONY, Maire; MILLER, Kieran; NAGARAJAN, Rahul; NARAYANASWAMI, Ravi; NI, Ray; NIX, Kathy; NORRIE, Thomas; OMERNICK, Mark; PENUKONDA, Narayana; PHELPS, Andy; ROSS, Jonathan; ROSS, Matt; SALEK, Amir; SAMADIANI, Emad; SEVERN, Chris; SIZIKOV, Gregory; SNELHAM, Matthew; SOUTER, Jed; STEINBERG, Dan; SWING, Andy; TAN, Mercedes; THORSON, Gregory; TIAN, Bo; TOMA, Horia; TUTTLE, Erick; VASUDEVAN, Vijay; WALTER, Richard; WANG, Walter; WILCOX, Eric; YOON, Doe Hyun. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Computer Architecture News*. 2017, vol. 45,

no. 2, pp. 1–12. ISSN 0163-5964. Available from DOI: 10.1145/3140659.3080246.

35. APPLE INC. *The future is here: iPhone X*. 2017 [Press Release]. Available also from: https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/. Accessed: 2021-05-20.

36. APPLE INC. *Apple unveils all-new iPad Air with A14 Bionic, Apple's most advanced chip*. 2020 [Press Release]. Available also from: https://www.apple.com/newsroom/2020/09/apple-unveils-all-new-ipad-air-with-a14-bionic-apples-most-advanced-chip/. Accessed: 2021-05-20.

37. CHOUKROUN, Yoni; KRAVCHIK, Eli; YANG, Fan; KISILEV, Pavel. Low-bit Quantization of Neural Networks for Efficient Inference. *arXiv e-prints*. 2019. Available from arXiv: 1902.06822 [cs.LG].

38. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. Deep Residual Learning for Image Recognition. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

39. ZAGORUYKO, Sergey; KOMODAKIS, Nikos. Wide Residual Networks. *arXiv e-prints*. 2016. Available from arXiv: 1605.07146 [cs.CV].

40. HUANG, Yanping; CHENG, Youlong; BAPNA, Ankur; FIRAT, Orhan; CHEN, Mia Xu; CHEN, Dehao; LEE, HyoukJoong; NGIAM, Jiquan; LE, Quoc V.; WU, Yonghui; CHEN, Zhifeng. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv e-prints*. 2018. Available from arXiv: 1811.06965 [cs.CV].

41. SANDLER, Mark; HOWARD, Andrew; ZHU, Menglong; ZHMOGINOV, Andrey; CHEN, Liang-Chieh. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520. Available from DOI: 10.1109/CVPR.2018.00474.

42. HU, Jie; SHEN, Li; SUN, Gang. Squeeze-and-Excitation Networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.

43. ZOPH, Barret; LE, Quoc V. Neural Architecture Search with Reinforcement Learning. *arXiv e-prints*. 2016. Available from arXiv: 1611.01578 [cs.LG].

44. GHIASI, Golnaz; LIN, Tsung-Yi; LE, Quoc V. NAS-FPN: Learning Scalable Feature Pyramid Architecture for Object Detection. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

45. LIU, Shu; QI, Lu; QIN, Haifang; SHI, Jianping; JIA, Jiaya. Path Aggregation Network for Instance Segmentation. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.

46. RAMACHANDRAN, Prajit; ZOPH, Barret; LE, Quoc V. Searching for Activation Functions. *arXiv e-prints*. 2017. Available from arXiv: 1710.05941 [cs.NE].

47. LIN, Tsung-Yi; GOYAL, Priya; GIRSHICK, Ross; HE, Kaiming; DOLLAR, Piotr. Focal Loss for Dense Object Detection. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2017.

48. HUBER, Peter J. Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*. 1964, vol. 35, no. 1, pp. 73–101. Available from DOI: 10.1214/aoms/1177703732.

49. AVENASH, R; VISWANATH, P. Semantic Segmentation of Satellite Images using a Modified CNN with Hard-Swish Activation Function. In: *VISIGRAPP (4: VISAPP)*. 2019, pp. 413–420.

50. ROEDER, Lutz. *Netron: Visualizer for neural network, deep learning and machine learning models*. [N.d.]. Available also from: https://www.lutzroeder.com/ai.

51. TENSORFLOW. *Accelerating TensorFlow Lite with XNNPACK Integration*. 2020 [Online]. Available also from: https://blog.tensorflow.org/2020/07/accelerating-tensorflow-lite-xnnpack-integration.html. Accessed: 2021-05-24.

52. NXP SEMICONDUCTORS. *i.MX 8QuadMax Automotive and Infotainment Applications Processors*. Eindhoven, Netherlands, 2020.

53. TENSORFLOW. *TensorFlow 2 Detection Model Zoo*. 2021 [Github repository]. Available also from: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md. Accessed: 2021-05-06.

54. REZATOFIGHI, Hamid; TSOI, Nathan; GWAK, JunYoung; SADEGHIAN, Amir; REID, Ian; SAVARESE, Silvio. Generalized Intersection Over Union: A Metric and a Loss for Bounding Box Regression. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

55. ZENG, Nick. *An Introduction to Evaluation Metrics for Object Detection*. 2018 [Online]. Available also from: https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection/. Accessed: 2021-05-23.

56. GARDNER-HOBBS, Julian. *Free Emma Puss Stock Photo*. [N.d.]. Available also from: https://www.freeimages.com/photo/emma-puss-1338103. Accessed: 2021-05-20.

57. TAN, Mingxing; LE, Quoc V. EfficientNetV2: Smaller Models and Faster Training. *arXiv e-prints*. 2021. Available from arXiv: 2104.00298 [cs.CV].

Index

A Attached source code

The root directory is divided into two main subdirectories.

A.1 automl

This directory contains the official EfficientDet repository. All code related to EfficientDet models can be found in `automl/efficientdet` directory.

A.2 thesis

This directory contains my implementation of all necessary parts.

Files in this directory are as follows:

- download_model.sh - script for downloading EfficientDet models.
- convert.sh - script for converting models to `.tflite` format.
- run_model_example.sh - script for running official example inference.
- run_tflite.py - script for running inference.
- parse_efficientdet_log.py - script for parsing efficientdet log.
- parse_mobilenet_log.py - script for parsing mobilenet log.
- per_class_analysis.py - script for per-class analysis.
- image_info_test-dev2017 - Information about evaluation images. Necessary for parsing the output logs.

Together with aforementioned files, there are several directories.

c++
This directory contains source code of the measurement application,

along with source code of additional utilities. A `Makefile` is provided for convenient compiling of the application. In `libs` folder, there is a pre-built `libtensorflow-lite.a` archive, which is linked against the application.

images
This directory serves as a place to store images. In the submission, the directory contains only a single example image, which is taken from the official EfficientDet repository.

results
In this directory, all different kinds of results are stored. There are models directories, which all include the `.tflite` model, along with the device evaluation log and COCO evaluation log. Additionally, this directory contains the `.csv` files of per-class analysis results.