# Improving the Inference Quality of Real-Time Object Detection using Deep Learning on Resource Constrained Hardware

Jayjun Lee*
*Department of Electrical and Electronic Engineering*
*Imperial College London*
London, United Kingdom
* jl7719@imperial.ac.uk

David Thomas
*Department of Electrical and Electronic Engineering*
*Imperial College London*
London, United Kingdom
dt10@imperial.ac.uk

***Abstract*—This study explores the ways of utilising Deep Learning (DL) and quantization techniques to perform real-time inference on resource-constrained hardware environments such as an FPGA and a Raspberry Pi with limited computational power compared to a cluster of GPUs. This study investigates a variety of CNN architectures with varying accuracy and inference time, which are in a trade-off relationship, as a more robust solution than that of rule-based traditional object detection approaches. The DL object detector aims at detecting balls of 5 different colours (red, green, blue, pink, yellow), which is trained using manually collected and labelled ball dataset by myself (published as a DOI). Starting from minimal CNNs to state of the arts CNNs, such as ResNets, and finally EfficientDet by Google that uses compound scaling and BiFPN, a number of scaled architectures are trained and evaluated, to find an optimal architecture that satisfies both the robust accuracy and inference time requirements as well as why that might be the case.**

## I. Introduction

A lot of research has been conducted on how well DL methods and deep neural networks can be applied to detecting objects of interest using computer vision [1]. With GPUs being widely used for parallelised processing of repetitive computation tasks such as a convolution filtering operation in a convolutional layer, it has come to my interest of how neural network can optimally achieve accuracy in object detection and still maintain a low latency and inference time to work real-time at a high camera frame rate. Therefore, in this research project, I aim to explore the ways how a model could be designed and trained, perhaps to reduce its size in trainable parameters or the architecture depth and width as well such that a model with high performance can fit onto several types of resource constrained environment such as an FPGA and a Raspberry Pi. The project uses the Intel DE10-Lite FPGA and the Raspberry Pi Model 3 to explore different ways to fit a DL model onto the resource-constrained environment and improving the inference quality and its robustness.

## II. Performance Metrics of an Object Detection Algorithm

Choosing a set of metrics to evaluate the performance of an object detection algorithm is important for comparison purposes of different algorithms of neural network architectures. In this section, several performance metrics are introduced to evaluate and quantify the performance of the object detection models to be discussed later.

### A. Accuracy and Robustness Metrics

One of the most important metrics in its sole purpose of object detection would be how accurately it can detect an object and draw a bounding box around it as well as classifying the object. In this study, such performance is measured by the training, validation and test losses from loss functions used to train the DL models by backpropagation. Commonly L1 loss is used for a bounding box regression and cross entropy loss is used for a classification task. On the other hand, the robustness factor is not an easy metric to quantify, yet this will be counted towards the accuracy metric by collecting a dataset that would require the models to be robust enough to have low validation and test losses after training. The process of collecting the ball dataset is discussed in Section 4-1.

### B. Speed Metric

Another important aspect of evaluating the performance of an object detection algorithm is to measure how fast an algorithm or a neural network can detect an object. As object detection is based on a camera image stream, the frame rate at which the processor can handle the computation is relevant especially for real-time purposes. Such a timing constraint would correspond to the clock rate of a programmed FPGA and the inference time of a compiled neural network in a Raspberry Pi that determines such a speed metric or the frame rate at which the whole process could be run at. A real-time object detection would work at the frame rate of 30 - 60 frames per second (fps) and hence is a requirement that the models in this project must meet.

### C. Resource Utilisation Metric

A quantification of resource utilisation gets significant especially in a resource-constrained hardware where the possibility of the amount of resource acting as a limiting factor is

more prevalent than other powerful multi-core CPUs with large gigabytes of RAM or GPUs, which are specialised for parallelised processing. The amount of resource utilised can be quantified using the number of trainable parameters in a CNN along with the number of floating-point multiplications required per forward pass for inference. For an Intel FPGA, an additional metric can be found by using resource utilisation analysis from Intel Quartus software as an example to ensure that it does not exceed the constraint of the FPGA.

### *D. Accuracy, Resource and Speed Trade-offs*

One of the hardest elements in implementing a real-time DL object detector with a low inference time and a high frame rate is the fact that all of these metrics are improved at a cost of losing some performance in the other (accuracy vs resource speed). As an example, to improve the accuracy of a model, a CNN could be scaled much larger in depth, which would be infeasible due to limitations in resources and speed and vice versa. Therefore, it is important to note that a lightweight DL model should be designed and trained to learn to have a high accuracy and to be optimized to minimise the trade-off that might have incurred to the performance.

## III. Rule-based Object Detection Algorithm and its Problems

The project is an extension or an improvement of the vision subsystem from the Mars Rover project in Year 2 where a rule-based (hard-coded) version of real-time object detection ( 45 fps) was written on the FPGA in Verilog. Figure 1 displays the assembled Mars rover.

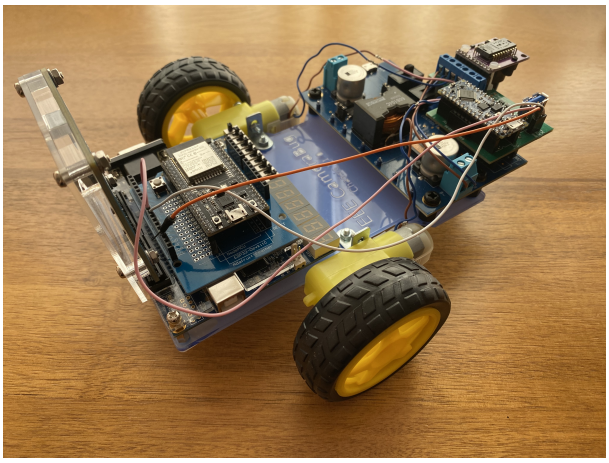

Fig. 1. The Mars rover with the vision module at the front for perception.

This works by a set of rules that involves applying a 3x3 Gaussian filter on the image for weighted averaging to remove background noise and subsequently passing through a set of activation ranges in colour spaces such as the RGB and HSV to classify each pixel as a candidate for a class (colour) of an object (ball). A sample image of the working algorithm is shown in Figure 2, where the bounding boxes are not perfect although they are functional. In reality, it only works in specific dark light settings that essentially eliminates most of the sources for the noisy pixels.

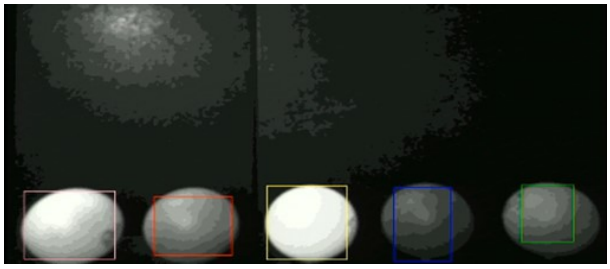

Fig. 2. Sample image from rule-based hard coded object detection algorithm.

The problem with this rule-based approach was that dealing with individual pixels using a set of fixed rules was particularly susceptible to being in different light settings and to frequent variation in the background as the rover navigated and would only work in certain light conditions. Due to the lack of robustness, this study therefore targets at designing a DL model that can maintain the frame rate while improving the inference quality compared to that of the rule-based object detection algorithm and its robustness with respect to the variation in the background and other noise in the pixel stream from a camera.

## IV. Object Detection using Convolutional Neural Networks (CNNs)

CNNs, also known as ConvNets, are often used for image classification and object detection because the convolutional layers that form the CNNs are great spatial feature extractors from low level patterns to higher level features that are formed by low level ones. These features can be shared and convolution filtering leads to reductions in dimensionality that the information can be compacted into feature maps.

### *A. Manual Dataset Collection*

One of the inevitable steps in training using supervised learning of DL models is to acquire a labelled dataset to train them on. For the object detection of the balls, I had to manually spend time taking individual photos in different light settings, which I thought it would help the model learn the variation in light setting better and still be able to detect objects. I had to go through the process of labelling the bounding boxes and the colours of the balls for about 500 images. The dataset is formed of three different categories of light settings: bright, normal and dark background images for each of the target objects of the colours red, green, blue, pink and yellow. The image dataset is shown in Figure 3.

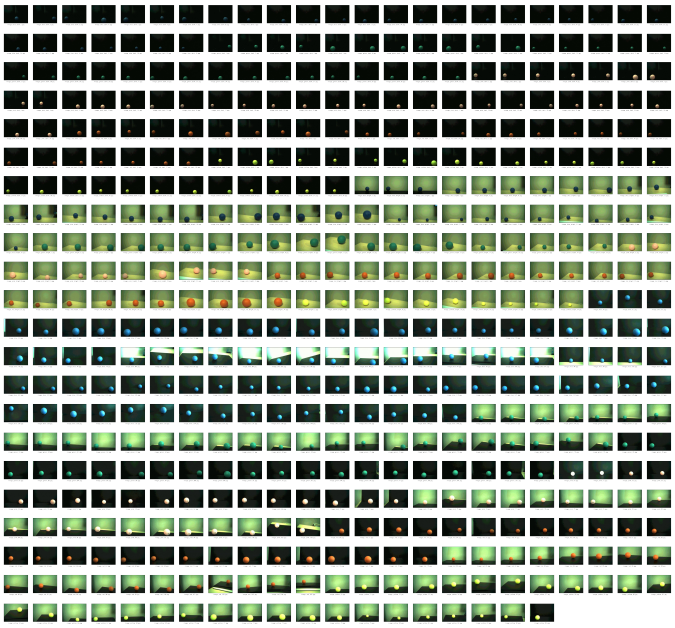

Fig. 3. Ball dataset collected by myself (500 images).

### B. Design of CNNs and End to End Training

Once the dataset was acquired, a suitable CNN architecture had to be designed. The simple and minimal model I came up with – considering the accuracy and the size of the model – consists of two convolutional layers followed by two fully connected layers. Another thing to consider was the input image shape or size in its width and height since the larger the input image size, the more computation required as the convolution filter would have to stride over the image even more. Therefore, the trade-off in such computation cost and the resolution of the image was identified and the input image size of 320 x 240 (width x height) was chosen.
The CNN model is end-to-end trained, which means that an image input to the model would output the direct regression of the bounding boxes as well as the classification of the class of the object. The model is coded and trained using the DL framework PyTorch. With the limited amount of the dataset, one of the major issues during training was that the model would easily overfit to the training batch that the validation loss would no longer improve with the training loss as in Figure 4(a).

To account for overfitting, the importance of data augmentation was considered that the image training set loaders were improved to randomly flip the images horizontally and crop them such that the model does not just learn the same images repeatedly. This approach helped preventing the model from overfitting to the training dataset as shown in Figure 4(b). To add on, one of the interesting things to note was that since the convolutional layers are shared for the bounding box regression part and the ball colour classification part, it often seemed to learn only one task better and the other worse and not learn them at the same time. However, once it had gone through enough epochs, it was able to achieve an acceptable level of the accuracy (90%) in regression and classification.

The simple CNN had approximately 16 million floating point operations per inference with about 260,000 trainable parameters as shown by the summary in Figure 5.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 9, 58, 78]           3,276
              ReLU-2          [-1, 9, 58, 78]               0
         MaxPool2d-3          [-1, 9, 29, 39]               0
            Conv2d-4         [-1, 18, 13, 18]           4,068
              ReLU-5         [-1, 18, 13, 18]               0
         MaxPool2d-6          [-1, 18, 6, 9]               0
            Linear-7                [-1, 256]         249,088
              ReLU-8                [-1, 256]               0
            Linear-9                  [-1, 5]           1,285
           Linear-10                  [-1, 4]           1,028
================================================================
Total params: 258,745
Trainable params: 258,745
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.88
Forward/backward pass size (MB): 0.77
Params size (MB): 0.99
Estimated Total Size (MB): 2.64
----------------------------------------------------------------
```

Fig. 5. Summary of the simple CNN architecture by layers and trainable parameters.

### C. Scaling on Resource-constrained Hardware

From the simple CNN from Section 4-2, several other CNNs were scaled by altering the depth, width and height by using different number of channels and filter sizes in the convolutional layers as well as by changing the filter size. It is important to note that the CNN was scaled in attempt to experiment and improve the accuracy and inference time in accordance with the constraints in different hardware environments: Intel DE10-Lite FPGA and Raspberry Pi 3. In the following, the environment specific specifications and resources available will be discussed.

*1) DE10-Lite FPGA:* The Intel DE10-Lite FPGA has 144 18-bit by 18-bit multipliers (dsp blocks), 50k programmable logic elements and 1638 Kbits of M9K memory for RAM and ROM. These numbers specify the resource constraint that limits the design of the neural network that could be deployed to solve the object detection problem. Before entering the world with resource-constraints, the asymptotic behaviour was analysed if there were to be no hardware constraints and what would be possible if so. One could conceive of building a CNN of whatever depth one would prefer, depending on how complex the problem is to detect the objects. Another option would be to use the state of the arts CNN that is known for having a great feature extracting convolutional layers where the final fully connected layers could be retrained with the rest frozen (parameters stays constant). Now that these upper bounds of what could be possible without a constraint was identified, the next step would be to consider an estimate of how many floating-point multiplications such a state of the art network like ResNets would require.

*2) Raspberry Pi 3:* A Raspberry Pi 3 Model B has a quad-core ARM Cortex A53, a cluster 1.2 GHz Broadcom BCM2837 64-bit CPU and 1 GB of RAM. To optimize and estimate the performance of neural network inference, finding out what kind of layers are efficient on the Raspberry Pi, especially regarding the instruction set architecture and cache bandwidths, which will be directly linked due to the large number of parameters to be loaded for the dot product of matrices. The CPU on the Raspberry Pi is not tailored for lots of repetitive dot product and with the limited RAM, it is likely to be memory-bound than compute-bound during inference.

To first try out the PyTorch-trained model of the simple CNN, Open Neural Network Exchange (ONNX) was used to convert the PyTorch model to ONNX model then to TF model and finally to TFLite. The TFLite model generated was deployed on Raspberry Pi but the performance on the accuracy metric was insufficient. This model achieved an inference time of about 30 ms on a Raspberry Pi 3, which means it can be run at a frame rate of around 30 fps and yet more improvements were needed on designing a more robust model that can learn to deal with variations.

The following variations were tested out by training as shown by the learning curves in Figure 6. Figure 6(a) illustrates how the train and validation losses improve with respect to the increase in the number of channels of the convolutional filters, where validation loss cannot keep up with the train loss. Figure 6(b) displays the learning curve when the depth of the CNN architecture is deepened by an extra convolutional layer improving the validation loss below the train loss. Figure 6(c) also presents an effective scaling method, which is to scale the size of the filter by enlarging the width and height for each layer. The scaled architectures and the resource usages by trainable parameters is summarized in Figure 7.

As an attempt to combine these various efforts, a compound scaling that uses all scaling means to improve the architecture is used where the respective resultant learning curves are shown in Figure 8(a). With the same number of epochs ran, the compound scaled model tends to perform better in terms of both the train and validation losses by about 5000. For a comparison and to reinforce the importance of preventing overfitting to training data by data augmentation Figure 8(b) presents the learning curve without data augmentation, which is significantly better in terms of the loss metrics but is overfitting to the training data rather than generalising to the dataset.

### *D. State of the Arts ConvNets and its Limitations*

After a few extra attempts on scaled variants of the simple CNN, which did not have a big difference in accuracy when deployed on Raspberry Pi, the focus shifted towards trying to achieve near-perfect robustness and accuracy before meeting the resource and inference time requirement. One great way of doing this is transfer learning from a pre-trained state of the arts model and to add on this is a common and standard thing to do, especially if the dataset available was small, which indeed was the case [2]. Transfer learning means to utilise the pre-trained models which are trained on millions of images in a dataset like ImageNet such that it has learned to have great feature extractors in the convolutional layers, which can be used by later layers. And to use the already-trained layers, the layers can be frozen that the backpropagation does not affect the parameters but the final fully connected layers, which are meant to be trained from an initialized state.

A model is trained by transfer learning the ResNet-34 (34-layer CNN) and deployed on the Raspberry Pi. Compared to the learning curves in Figure 3, 4 and 5 of different models including the simple CNN and other scaled CNNs, the ResNet-34 performs significantly better in terms of the train and validation losses within the same number of epochs as in Figure 9.

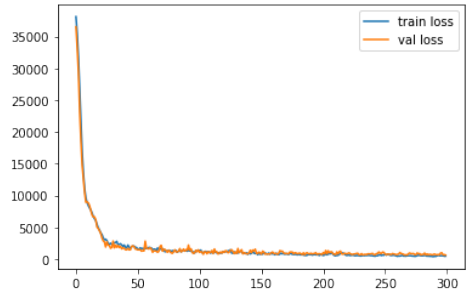


Fig. 9. Learning curve of ResNet-34.

It achieved satisfactory training, validation and test losses but the inference time had vastly increased up to about 3000 ms (equivalent to 3 seconds delay), that the real-life verification of the object detector could not be done properly. The inference time was the limiting factor in the case of transfer learning state of the arts ConvNets. There needed to be a more lightweight and thus efficient yet something scalable and with inter-layer connections to retain certain information that may be lost by convolutional layers' characteristic of reducing dimensionality and by strides, which may leave out certain pixels or patterns.

### *E. Optimization using Quantization Techniques*

While seeking for optimization techniques for making a trained neural network less compute-intensive either by reducing the number of computations such as by pruning or by not using floating-point multiplications, which tend to be expensive on CPUs, but int-8 multiplications. Pruning is a technique that removes unnecessary or negligible connections between neurons to reduce the number of computations. The latter approach is called quantization, where a number of approaches are available: quantization-aware training and post-training quantization. These methods were still an active area of research in academia and asymptotically, these would allow the DL models to be reduced in size by 4x, reduced in memory bandwidth for loading and storing weights and biases parameters by 2x – 4x as well as faster inference by 2x – 4x

due to overall savings in memory bandwidth and computation cost. However, as mentioned earlier, these optimizations to reduce the amount of computation resource required and speed is traded at some significant loss of accuracy that these need to be applied delicately and properly.

## V. Compound Scaling and Feature Network in EfficientDet

The final attempt was to apply a model called EfficientDet with the EfficientNet backbone architecture and weighted bi-directional feature pyramid network (BiFPN) as its feature network [3]. While my sequential and step-by-step improvements on the minimal simple CNN was done at a single-dimensional scaling such as depth-scaling or width-scaling, this particular approach uses a compound scaling, which utilises all of depth, width and resolution-scaling to improve the model performance. This model also bases its feature extraction using the FPN, which consists of a bottom-up forward pass pathway (like the simple CNN) and a separate top-down pathway with lateral connections in between the bottom-up and top-down pathways to share and retain information as in Figure 10 that might have been lost in the process as mentioned at the end of Section 4-4 as a possible direction for an improvement.

Once the EfficientDet was used for training and deployed on the Raspberry Pi, it started to display some reasonably fast inference and a feasible object detector. The average inference time for the EfficientDet-Lite0 was about 1000 ms and the EfficientDet-Lite2 was about 2700 ms. The word 'Lite' comes from the model being on TFLite and being quantized where applicable as a form of optimization to fit the resource-constraints. The EfficientDet-Lite object detectors were able to achieve a highly accurate and robust object detection that is better than the rule-based object detector which can only work in dark background as shown in Figure 11.

To return to where this research project started, Figure 12 provides the DL version of object detection for a comparison between the rule-based version from Figure 2 and to visualise how both perform on the similar images. The DL version is better at placing the well-fitted bounding boxes and since the DL approach is based on thresholds that only some bounding boxes with high probability of detection are shown. It has far less glitches as it is based on pattern recognition rather than a direct pixel calculation.

Fig. 12. Performance of DL object detection on Raspberry Pi on a similar image used in Figure 2.

## VI. Room for Improvements

Although the study presents the bright side of the project, there were some room for improvements identified as well as some obstacles on the way of designing, finding, implementing and deploying an optimal model.

### A. Biased Dataset

I started this project by collecting the ball dataset, purely out of my experience from the rule-based computer vision that directly works (only in the optimal light setting) from the individual pixel values that are Gaussian filtered to minimise noise. I initially thought by collecting images of certain categories of light settings (bright, normal, dark), it would help neural networks to generalize better but it turns out that was not the case despite image augmentation and possibly due to the nature of CNNs that tends to generalize poorly to small image transformations [4]. Moreover, not all images in the dataset were perfect and needed some manual removal of certain images due to the misleading representation of an object for its class. In Figure 13, the sample images of the misrepresentation of the balls are shown. Without the prior knowledge or label of the images, even for humans these images are not clear enough to be used for training the neural network as it would learn from such misrepresentations. Therefore, it is important to pre-process the training dataset and to really understand the data that is available and being used.

### B. Limitations in Adapting CPUs for DL Operations/Purposes

For the Raspberry Pi, on a low level, the CPU is running the inference of the trained DL model from the compiled code of the model. It is important to realise that without a GPU – specialised for parallel processing – the CPU has to take on the burden and it is a compiler's job to adapt such common DL operations for multiple-core CPUs. However, having seen that the simple CNN achieving about 30 fps at maximum, set a very low frame rate despite the simplicity and the relative size of the neural network and therefore the number of computations required per inference. This analysis on the near-simplest model with highest expected frame rate achieving about 30 fps present the problem and limitation on

the computational power of the hardware itself, which can be improved somehow and not endlessly rely on the software stack to optimize the computations.

### C. A Need for AI Accelerators

This leads to why AI accelerators such a Google TPU, Google coral USB accelerator and NVIDIA GPUs take part as an important part in training the neural networks as well as in inference [5]. It is important to be able to minimise the resource needed to deploy a DL model at a satisfactory accuracy level, however, given that the resource-constrained hardware can actually be the bound in a roof-line model of the software versus hardware, it is necessary to improve the hardware. In other words, there is a limit set by the hardware, where at some point the improvements in software will start to saturate and no longer have a noticeable effect on a fixed resource-constrained hardware. Therefore, I believe that it is important to develop even powerful hardware devices for DL applications and even an optimized DL software for the hardware available at last. As for the EfficientDet-Lite0 on the Raspberry Pi, even with a single Google coral USB accelerator, the inference time is reduced significantly with much higher frame rate to highlight the need for AI accelerators that are purely for DL operations.

### D. Github and Publication of Dataset

The self-acquired ball dataset of approximately 500 images and labels is published as a digital object identifier (DOI) via zenodo and on Github repository where all other codes are available as shown in Figure 14.

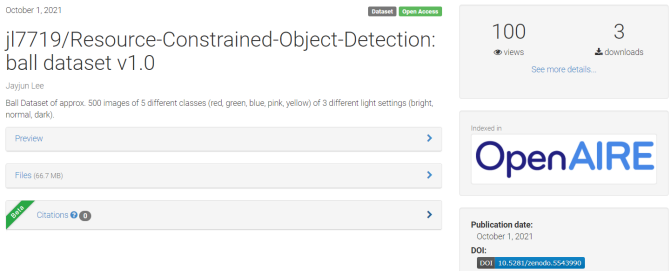


Fig. 14. Zenodo webpage screenshot of the published DOI of the ball dataset.

## VII. Conclusions

To conclude, a self-driven research project was carried out, from scratch and with no dataset at all that the ball dataset images had to be taken and labelled before even starting to design the DL models. Once the dataset was collected, after analysing what may or may not be possible on the resource-constrained hardware, the minimal CNN and maximal CNN (state of the art) were trained to experiment with asymptotic behaviours. The minimal CNN had the problem of low accuracy while the maximal CNN had the problem of high inference time and hence a low frame rate to be used in real-time. Therefore, an optimal solution that belongs somewhere in the midway of the minimal and maximal CNNs, the EfficientDet is used for the final object detection model. Regarding the problems of the minimal and maximal CNNs, methods of scaling (width, depth, resolution) and methods of optimization (quantization-aware training and post-training quantization) were explored, arriving at the conclusion that EfficientDet-Lite, which uses compound scaling and feature pyramid network along with post-training quantization allows to achieve an optimal solution that can work in real-time as an object detector that meets the requirements in the accuracy, resource and speed metrics only with the addition Google coral USB accelerator (to improve inference time), which is also explored why it is an important part of the research area on hardware for DL and DL for hardware. As a mandatory step of this research project, the dataset was first collected manually and later it was published as a DOI for an open access for whom wishes to continue with further work on the research problem.

## References


[1] Arxiv.org. 2019. Deep Learning vs. Traditional Computer Vision [online] Available at: ¡https://arxiv.org/ftp/arxiv/papers/1910/1910.13796.pdf¿ [Accessed 7 September 2021].

[2] Pytorch.org. n.d. Transfer Learning for Computer Vision Tutorial — PyTorch Tutorials 1.9.1+cu102 documentation. [online] Available at: ¡https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html¿ [Accessed 11 August 2021].

[3] Tan, M., Pang, R. and Le, Q., 2020. EfficientDet: Scalable and Efficient Object Detection. [online] arXiv.org. Available at: ¡https://arxiv.org/abs/1911.09070¿ [Accessed 10 September 2021].

[4] Jmlr.org. 2019. Why do deep convolutional networks generalize so poorly to small image transformations? [online] Available at: ¡https://jmlr.org/papers/volume20/19-519/19-519.pdf¿ [Accessed 12 September 2021].

[5] Coral. n.d. USB Accelerator — Coral. [online] Available at: ¡https://coral.ai/products/accelerator¿ [Accessed 17 September 2021].