ACKNOWLEDGEMENT

I would like to take this opportunity to thank Dr Geoff Merrett and PhD scholar Amin Sabetsarvestani for guiding me towards this project. This was a completely new adventure, and their guidance and corrections were an invaluable resource.

I would also like to thank the entire MSc Artificial Intelligence lecturing staff members for imparting me with their knowledge and philosophy of Artificial Intelligence.

I also acknowledge the use of “IRIDIS High-Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work”.

1 INTRODUCTION

1.1 OVERVIEW OF THE INTERNET OF THINGS

The Internet of Things (IoT) [1] can be defined as broad-spectrum, internetworked embedded devices, collectively achieving specific use cases. There are no well-defined set of objectives that are classified as only suitable for IoT; however, there are characteristics of the use cases under which IoT devices are generally deployed, which govern the nature of the devices. For example, IoT devices, in general, tend to be implemented in large numbers and are interconnected to a backbone infrastructure [2] over a wired or wireless interface. Due to the large number of devices which can be deployed, there are constraints imposed on per-device cost, and this restricts the capabilities of the individual devices [3]. In effect, the devices tend to have only a few MB of memory and limited computational ability. Another important constraint is the power consumption of the devices. IoT devices can be deployed in sparse locations, such as forests; in such circumstances, power consumption becomes a significant constraint.

Figure 1 shows the essential sections of the IoT infrastructure. The devices are the front-end sensors which gather information from the environment they interact with. The gateway is an aggregation point that collates information from various devices and pushes data into the cloud infrastructure. The cloud is where the data is stored, processed and combined with alternate sources [1].

1.2 Overview Of Deep Neural Networks

Deep Neural Networks (Figure 2) [4] are Multilayer Perceptron with multiple hidden layers between the input and output. In recent times, these networks have achieved paramount feats in visual processing tasks [5]. There have been numerous state-of-the-art models created since the year 2012, such as AlexNet, VGG, GoogleNet, ResNet, RCNNs etc. [5]. Most of these models are specially designed for high accuracy [6], while few are designed to operate in resource constrained environment [7].

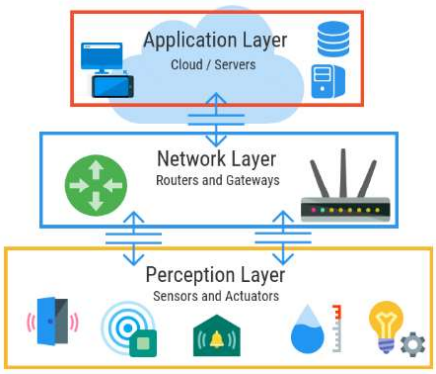

Figure 1: IoT infrastructure (reprinted from: [1])

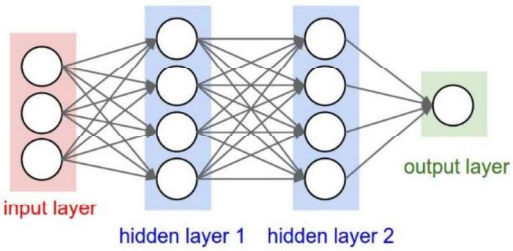

Figure 2: Deep Neural Network (reprinted from [8])

Training Deep Neural Networks requires enormous datasets and are system resource and time exhaustive. However, it is possible to transfer trained parameters from one network to the other. This is referred to as transfer learning [9]. For example, networks trained on extensive dataset such as the ImageNet [10] can transfer their learned parameters for convolutional layers to another network for feature extraction, while fully connected or classification layers can be trained from scratch. This technique not only

facilitates faster learning; but also improves the accuracy and regularisation of the network.

Testing/Inferencing with Deep Neural Networks are also system resource and time exhaustive. It is possible to use GPUs to speed up the process. The basic idea is that GPUs can parallelise simple vector operations during backward/forward propagation and thus utilising GPUs during training or testing can significantly reduce the processing time.

1.3 MOTIVATIONS AND OBJECTIVES

IoT devices are pervasive and can generate a large amount of data, while deep neural networks can be used to perform state-of-the-art data analytics. However, there exists a contention of resource requirements in between. In view, there are numerous research approaches underway to optimise neural network for IoT devices. One of the approaches is model "partitioning" which envisages a distributed deep neural network.

In view, this project aims to investigate a mechanism to distribute Deep Neural Networks in hardware resource constrained IoT infrastructure. The main objectives of the project are as follows:

1. To distribute a deep neural network into smaller scalable nets such that they can be utilised on IoT devices.

2. To aggregate this distributed inferencing to produce higher accuracies than possible by the network in isolation.

3. To spread this distribution and aggregation mechanisms across the various layers of the IoT infrastructure.

4. To train this distributed model in a unified manner on available compute resources.

1.4 DISSERTATION STRUCTURE

The dissertation is segregated into Five Sections. The First Section Introduces the subject theme, states the Project Motivation and Dissertation Structure. The Second Section discusses the Literature and the background research conducted before starting the project. The System Design and design considerations are discussed next in Section Three. Section Four then presents the project outcome and analyses the results. Finally, the Conclusion, Future Work and Project Management aspects are outlined in Section Five. Overall, the dissertation is structured in such a way that initial chapters act as a reference to later detailed discussions.

2 LITERATURE REVIEW AND RESEARCH

2.1 INTERNET OF VIDEO THINGS (IOVT)

Internet of Video Things [11] are a specific class of IoT devices which have visual sensors. These devices tend to have more memory and compute resources available to them and have unique characteristics of sensing, storage and transmission which are divergent from contemporary IoT devices, however, cannot still process video streams locally and are required to transmit real-time video streams to a gateway for processing [11]. These attributes of IoVT devices imposes considerable constraint on the network infrastructure within which they operate, not only network bandwidth but also a requirement of high uptime [11]. For example, IoVT devices currently used for any image classification or object detection transmit out real-time video streams to a gateway. The gateway then aggregates these video streams and forwards them to a cloud service for inferencing and analysis (Figure 3) If the network collapses momentarily, the inferencing task is hindered. Hence, this approach only provides a workaround solution to the lack of local inferencing capability on IoVT devices.

One solution to this challenge is to perform some minimal inferencing locally [12] on the device and based upon specific criterion forward the actual data on-demand basis. This way, in case of network failures, the infrastructure does not entirely breakdown. The devices can pre-roll during this downtime

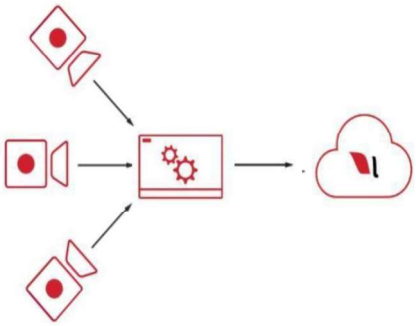

Figure 3: IoVT Infrastructure (reprinted from: [13])

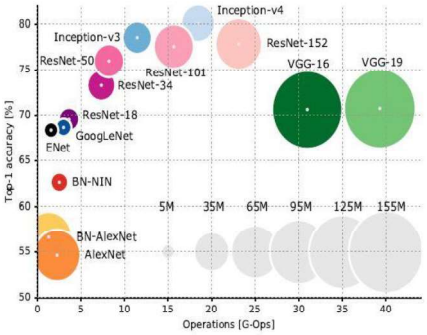

Figure 4: Top1 vs. operations, size/parameters (reprinted from: [5])

2.2 TRADE-OFF: DEEP NEURAL NETWORKS VS IOT DEVICES

To train/test effectively, deep neural networks require system configuration such as 16GB high-speed GDDR5 RAM and Nvidia 1080p GPUs [4]. This becomes quite prohibitive for resource constraint IoT devices, which typically operate with Megabytes or even Kilobytes of memory without any notion of a GPU as such [1].

Figure 4 shows the number of Gflops required for a forward pass of various networks. The size of the circle depicts the number of parameters. We can observe that models like VGG have parameters of the order of 155 million, whereas GoogleNet is similar in accuracy while the number of parameters is reduced to 5 million. Assuming each parameter occupies a float datatype of 4Bytes, then the amount of memory required is approximately 20MB and approximately 5 G-Ops for a single forward pass. For Odroid XU4 (Cortex-A15) with approximately 4.0 DMIPS/MHz/core running at 1200MHz and four ARM A15 cores (4 * 1200 * 4 ~= 20 G-Ops), every forward pass shall require

approximately 0.25 seconds minimum, that is without considering operating system and floating-point overheads. This is quite prohibitive for any real-time image processing task running at 30fps.

2.3 COMBINING DEEP NEURAL NETWORKS AND IOT DEVICES

As shown in section 2.2, the extent of resources required for deep neural networks for even a single forward pass is quite prohibitive for embedded IoT devices. There have been various methods proposed to circumvent these issues. Some of these are discussed below:

1. **Pruning** [14]**:** In this approach, the number of parameters in a network is optimised. There have been various methods proposed such as sparsity regularises to remove redundant layers, channels and filters etc.

2. **Quantisation** [14]**:** In this technique, the parameters of the model are approximated to lower number of bits (16 or 8, instead of 32-bit float). This saves a lot of memory and computation; however, effects the overall performance of the network.

3. **Low-Rank Factorization** [15]**:** Most of the complexity of a neural network can be attributed to convolutional operations. This technique performs low-rank factorisation utilising statistical methods such as Singular Value Decomposition (SVD). One drawback of this method is that the decomposition operations are non-trivial and since these are performed

on a per-layer basis, i.e. not appreciating the inter-layer dependencies, the overall training and accuracies of the model are affected.

4. **Partitioning** [16]**:** In this technique, a neural network model is broken down into smaller sections. These smaller models can then be easily executed on resource constraint devices. The key idea here is that the partitioned models should be able to perform meaningful computations within the latency and resource constraints of IoT infrastructure, while a larger model with more resource requirements can be executed on the server system. This also forms the basis of distributed deep learning, where multiple smaller models are right-sized and distributed on smaller devices, while the significant chunk of the network is executed elsewhere. Further, this technique is especially beneficial for IoVT devices. The end devices can be used to gather data from multiple orientations, which improves the accuracy and fault tolerance of the system [12].

2.4 CONVOLUTIONAL NEURAL NETWORKS

Convolutional [4] and Fully connected networks are the essence of contemporary neural network applications. While convolutional networks are more extensively applied to image processing, fully connected networks are more ubiquitous [5]. In the context of this project, we only review convolutional networks and especially 1x1 conv and how it can be used in place of a fully connected layer. Figure 5 shows a convolutional network in operation.

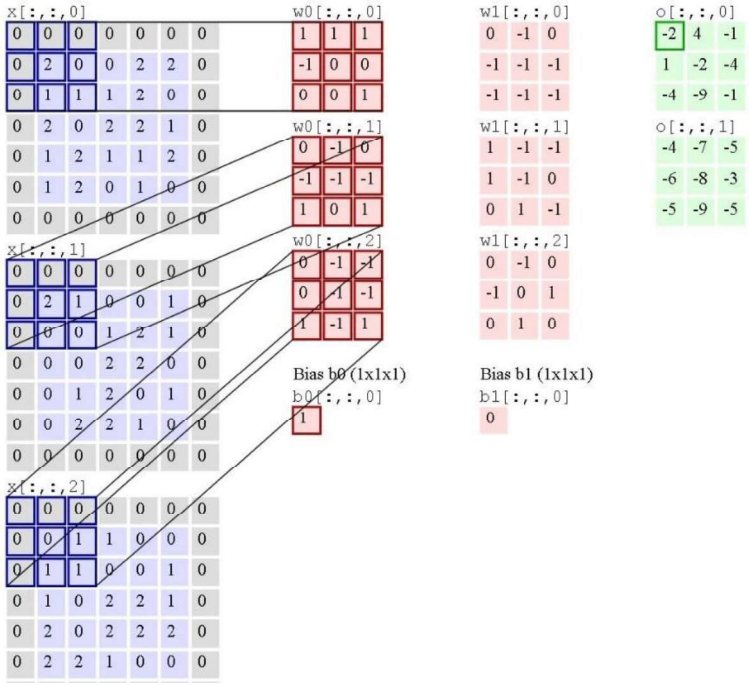

Figure 5: Convolutional Network Operation (reprinted from: [8])

The leftmost blue sections are the input image channels, consider as three channels RGB. The bolded blue sections are known as the "receptive field"; they directly influence the output data. The central red elements are the filter weights with bias; in this case, we say the filter shape is 3x3x2-s-1. The rightmost green sections are the outputs. To generate the output of "-2", the weight elements are multiplied with the corresponding inputs are added together. For this example, "-2" is generated as follows:

$F_R^{W_1} = 0*1+0*1+0*1+0*(-1)+2*0+0*0+0*0+1*0+1*1=1$

$F_G^{W_1} = 0*0+0*(-1)+0*0+0*(-1)+2*(-1)+1*(-1)+0*1+0*0+0*1=-3$

$F_B^{W_1} = 0*0+0*(-1)+0*(-1)+0*0+0*(-1)+1*(-1)+0*1+1*(-1)+1*1=-1$

$B = 1$

Thus, the output is:

$O[0,0] = F_R^{W_1} + F_G^{W_1} + F_B^{W_1} + B = -2$

Similarly, both the outputs blocks are computed for both the filter channels. What should be noted here is the number of output channels is not dependent on the number of inputs channels but is dependent on the number of filter channels. That is, there are two filters and thus, two outputs. There are other concepts like stride (shown like s-1), padding, etc., to maintain simplicity these are not discussed here.

Now consider the case that we are using a 1x1 filter. In that case, the input and output, both have the same shape (7x7 in this case). However, the output shall have only two channels. This is the application of dimensionality reduction techniques to convolutional networks.

```
14 conv     512  3 x 3 / 1     7 x   7 x 256   ->     7 x   7 x 512
15 conv     512  3 x 3 / 1     7 x   7 x 512   ->     7 x   7 x 512
16 conv     512  3 x 3 / 1     7 x   7 x 512   ->     7 x   7 x 512
17 dropout       p = 0.50               25088  ->  25088
18 conv      10  1 x 1 / 1     7 x   7 x 512   ->     7 x   7 x  10
19 avg                         7 x   7 x  10   ->    10
20 softmax                                           10
```

Figure 6: 1 x 1 for CIFAR-10

Now in Figure 6, we see that after the 1x1 filter, an average pool is applied. This converts the 7 x 7 x 10 filter to just 10 outputs suitable for a Softmax layer.

This shows that by using 1 x 1 conv filters and avgpool, we avoided the need for a fully connected layer. There are various advantages to this technique, for example, dimensionality reduction as discussed previously, then the number of parameters is less in this approach, further another essential

benefit is that output dimension is independent of the input dimension and also since we are using avgpool, the input size can be dynamically varied. Training a network with variable input sizes makes it more robust at inference time if the input image size varies [17]. These ideas are used in YOLO Object detection model that we discuss in section 2.7.

2.5 DISTRIBUTED DEEP NEURAL NETWORKS

The Universal Function Approximation theorem [18], [19] states that, under mild assumptions, any neural network with at least one hidden layer and a finite number of neurons can be used to approximate any continuous function. However, as the complexity of the function increases, the width of such networks increases exponentially [20]. Another approach is to increase the depth of the network [21]. This leads to better accuracy in the supervised learning task [4]; however, as the depth increases, we encounter vanishing and exploding gradient problem [22]. One of the most modern techniques to circumvent this issue is to use "identity shortcut connections" as shown in [23]. It is not apparent; however, these "shortcuts" form the basis of many state-of-the-art deep neural networks such as GoogleNet, YOLO, etc., including distributed deep neural networks [24]. In [24], the authors present the idea of early exits from a network and demonstrate that not all elements of a dataset are required to pass through the entire network, and most can be classified with a much smaller network. They further argue that these early exits can add to regularisation and mitigate the vanishing gradient issue. Figure 7 shows the simplest BranchyNet [24] structure. As can be seen, there are multiple early exit points added. These exit points can be single or

multi-label classifier, or even object detection networks such as YOLO. The key idea the authors present is training all the exit points jointly and suggests an objective function as follows [24]:

$$L_{branchynet}(\hat{y}, y; \theta) = \sum_{n=1}^{N} w_n L(\hat{y}_{exit_n}, y; \theta) \quad (1)$$

Where, $\hat{y} = softmax(z) = \frac{exp(z)}{\sum_{c \in C} exp(z_c)}$ OR $sigmoid(z) = \frac{1}{1+exp(-z_j)}$, for multi-label classification. The equation (1) describes a weighted sum of the losses from all the exit points. During forward pass the losses generated from the various exit points are added together, while during a backward pass, the errors are moved back through the network and gradients added where the network bifurcated. Finally, the weights are updated to minimise the losses using ADAM [25] optimiser. Performing joint training ensures consistent behaviour of the network exit points for the same dataset. Further, [12] shows an application of BranchyNet, where the network is distributed across IoT infrastructure hierarchies such as the edge, devices and the cloud. The authors propose using edge devices as information encoding pipeline. The key idea is by partitioning a neural network into smaller sub-sets, it should be possible to perform inferencing in a distributed manner, and the smaller networks on the devices should be less resource-intensive than the full network. This approach has many advantages; for example, consider three IoVT devices capturing data from multiple orientations (Figure 3). Instead of streaming the entire data real-time to a cloud-based server, the devices perform minimal classification locally and

stream if certain conditions are met. This way, the authors demonstrate 20x network bandwidth savings.

Further, distributed inferencing also adds to fault tolerance and improves the performance of the network. Finally, it should be noted that this form of distributed neural network inferencing is different from distributed training utilising multiple GPUs. In distributed inferencing, both training and inferencing could be distributed; however, distributed inferencing is the fundamental idea.

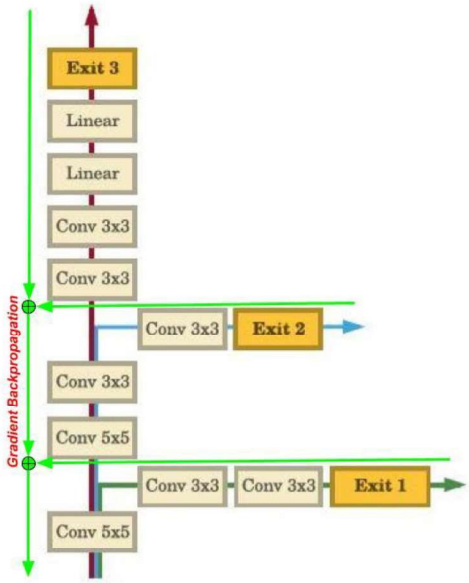

Figure 7: BranchyNet (adapted from: [24])

2.6 OBJECT DETECTION

Object detection is a sub-field of image recognition, where multiple semantically similar or different objects in an image are identified and localised using bounding boxes. This field has various applications, such as face recognition, autonomous vehicles, security etc. Figure 8 shows the outcome of the YOLO object detection model. There are mainly two approaches to object detection. In Region-CNN (R-CNN) [26], the entire image is scanned multiple times by a sliding window approach. This way, in

each iteration a region of the image is proposed (bounding box) and forward passed through an image classifier network.

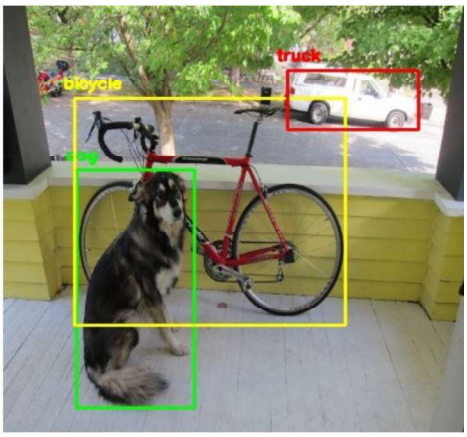

Figure 8: Object Detection (reprinted from [27])

This has various limitations, such as scanning of the image multiple times leads to very high latencies. To give an idea, R-CNN generates about 2000 region proposals using a state-space search algorithm called "selective search". This leads to a turnaround time of 40 seconds on modest GPUs.

Further, the selective search algorithm does not have any learnable parameters, thus may perform poorly on varied datasets. There have been multiple improvements suggested with Fast/Faster R-CNN [6] [28]. However, the general impression of this approach is, R-CNNs are considered as re-purposed classifier, not a dedicated object detection algorithm. One advantage of this method is it leads to good accuracies when compared to other contemporary methods. Figure 9 shows the two-step approach in Faster R-CNN architecture. On one branch bounding box proposals are generated while in the ROI layer, the feature maps along with the region proposals are used to predict the class for each.

The second approach to object detection is single shot detectors [29]. Both YOLO [30] and SSD [29] are of this category. The key idea here is both

bounding boxes and classes are predicted directly from feature maps in one step.

These types of networks have a backbone classifier for feature extraction and an object detection head for bounding box and class prediction. Instead of proposing perfect bounding boxes, the network first assigns fixed grid cells to the entire image and predicts class probabilities for each box. This is as shown in Figure 10. Another essential concept these networks use is that of anchor boxes.

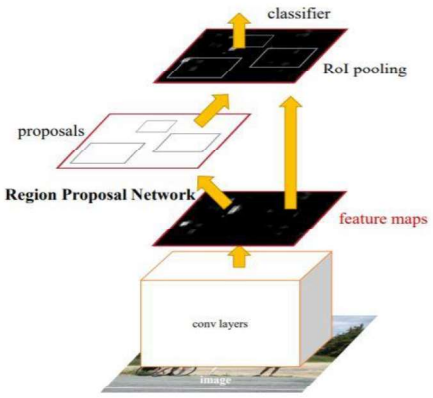

Figure 9: Faster R-CNN (reprinted from: [6])

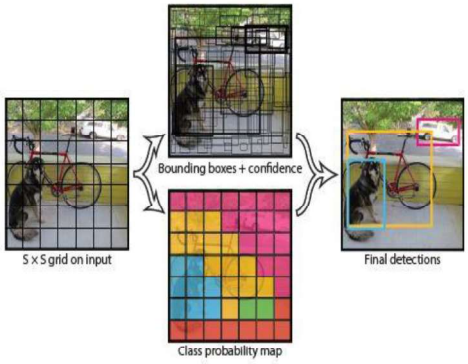

Figure 10: Grid and Confidence (reprinted from: [31])

Anchor boxes (statistical priors) define standard shapes and sizes of objects found in nature. These were defined after large scale cluster analysis of datasets [29]. For example, a car is expected to be horizontal rectangular, while a person is expected to be vertical rectangular. Multiple anchor boxes are assigned to each grid box. The network then predicts adjustments (resizes) to the anchor boxes to

maximize Intersection of Union (IoU, section 2.12.1) with the ground truth label and predicts the class probabilities simultaneously. One difference between YOLO and SSD is that SSD assumes fixes anchor box height/width ratios, while YOLO determines anchors by running k-means clustering algorithm separately on training dataset. This helps YOLO perform better on known datasets; however, if a new dataset is required to be used, then new anchor boxes should be generated for the specific dataset.

2.7 YOLO OBJECT DETECTION MODEL

You Only Look Once [30], or YOLO is one of the state-of-the-art object detection models. As stated previously (section 2.6), it is a single shot detector and thus produces bounding box and class predictions simultaneously. The advantages of YOLO is that it is incredibly fast (up to 150 fps) and lightweight. Therefore, YOLO can be used for real-time object tracking. There are multiple versions of YOLO from YOLOv1 to YOLOv3 and a lightweight 'Tiny' version which can be used on embedded single board computers. The YOLOv3-tiny model is as shown in Table 1. There are in total 8858734 parameters, and each parameter is 4B float, generating an overall size of the weights as 35434936B (~34MB). Adding 20B file header, the parameter size of YOLOv3-tiny is exactly 35434956B [32] (~34MB).

The model generates 13 x 13 grid cells and 3 anchor box per grid, the prediction layer for YOLO-Tiny thus outputs 13 x 13 x 3 x 5. The number 5 denotes 'x', 'y', 'w', 'h' and 'class confidence'. The multi-part loss function for YOLO is as follows [31].

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \quad (2)$$

$$+\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \quad (3)$$

$$+\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \quad (4)$$

$$+\sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \quad (5)$$

Here, equation (2), represent the regression setup for learning the 'x' and 'y' coordinates. For each grid $i \epsilon 0 \rightarrow (S\ x\ S)$, for each anchor box $j \epsilon 0 \rightarrow B$, the one-hot vector $\mathbb{1}_{ij}^{obj}$, represents the box prediction with the highest IOU with the ground truth. Similarly, equation (3), represents the same for the width and height, however, adds a square root to account for the fact that small deviations in large boxes should be given less weight than smaller boxes. Equation (4), is regression setup for IOU predictions, where 'C' represents the IOU confidence score for each box. The parameter λ is used as a weight for model stability. $\lambda_{\text{noobj}} = 0.5$ is the weight when no object is present and $\lambda_{\text{coord}} = 5$ otherwise. Finally, equation (5), represents loss for each class prediction.

Layer	Type	Num. Filters	Size	Num. params	Num. Bias	Num. B. Norm	Input	Output	BFLOPs
0	conv	16	3 x 3 / 1	432	16	48	416 x 416 x 3	416 x 416 x 16	0.150
1	max		2 x 2 / 2				416 x 416 x 16	208 x 208 x 16	
2	conv	32	3 x 3 / 1	4608	32	96	208 x 208 x 16	208 x 208 x 32	0.399
3	max		2 x 2 / 2				208 x 208 x 32	104 x 104 x 32	
4	conv	64	3 x 3 / 1	18432	64	192	104 x 104 x 32	104 x 104 x 64	0.399
5	max		2 x 2 / 2				104 x 104 x 64	52 x 52 x 64	
6	conv	128	3 x 3 / 1	73728	128	384	52 x 52 x 64	52 x 52 x 128	0.399
7	max		2 x 2 / 2				52 x 52 x 128	26 x 26 x 128	
8	conv	256	3 x 3 / 1	294912	256	768	26 x 26 x 128	26 x 26 x 256	0.399
9	max		2 x 2 / 2				26 x 26 x 256	13 x 13 x 256	
10	conv	512	3 x 3 / 1	1179648	512	1536	13 x 13 x 256	13 x 13 x 512	0.399
11	max		2 x 2 / 1				13 x 13 x 512	13 x 13 x 512	
12	conv	1024	3 x 3 / 1	4718592	1024	3072	13 x 13 x 512	13 x 13 x1024	1.595
13	conv	256	1 x 1 / 1	262144	256	768	13 x 13 x1024	13 x 13 x 256	0.089
14	conv	512	3 x 3 / 1	1179648	512	1536	13 x 13 x 256	13 x 13 x 512	0.399
15	conv	255	1 x 1 / 1	130560	255	0	13 x 13 x 512	13 x 13 x 255	0.044
16	yolo								
17	route 13								
18	conv	128	1 x 1 / 1	32768	128	384	13 x 13 x 256	13 x 13 x 128	0.011
19	up sample		2x				13 x 13 x 128	26 x 26 x 128	
20	route 19 8								
21	conv	256	3 x 3 / 1	884736	256	768	26 x 26 x 384	26 x 26 x 256	1.196
22	conv	255	1 x 1 / 1	65280	255	0	26 x 26 x 256	26 x 26 x 255	0.088
23	yolo								
	Total Params			8845488	3694	9552			

Table 1: YOLOv3-Tiny

Another interesting feature to notice is that the model does not contain any fully connected layer; instead, it uses a 1x1 conv layer to reshape the net. For example, consider the layer '22', it has 256 inputs and 255 outputs, i.e. there are 255 filters for each of the 256 inputs. Thus, feature maps of the

shape 26x26x255 is feed into the YOLO layer. The absence of the fully connected layers also allows the network to be reshaped for different image sizes (multi-scale) during training. This makes YOLO more robust to multiple image sizes during inference.

2.8 ACTIVATIONS: SIGMOID AND SOFTMAX

The Sigmoid (Figure 11) or logistic activation function is given by equation (6) [33]:

$$f(s_i) = \frac{1}{1+e^{s_i}} \tag{6}$$

This is used to squash the output vectors independently within the range (0, 1). This means that if applied to different vectors, then each receive independent probabilities. This can be useful for multi-label classifiers discussed later. Another essential function is the Softmax function shown in equation (7) [4].

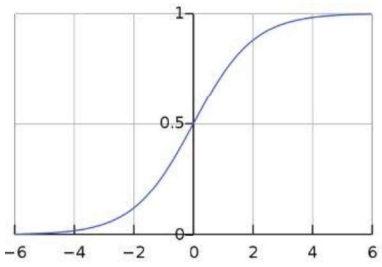

Figure 11: Sigmoid

$$f(s_i) = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \tag{7}$$

This function squashes the vector in the range (0, 1); however, the vectors do not remain mutually exclusive of each other.

2.9 LOSS FUNCTIONS: CROSS ENTROPY AND FOCAL LOSS

For the binary classification problem, the cross-entropy loss is given by equation (8) [34].

$$CE = \sum_{i=1}^{C'=2} t_i log\ (s_i) = -t_1\ log(s_1) - (1 - t_1) log\ (1 - s_1) \quad (8)$$

Considering two classes, the ground truth labels are either 0 or 1. Activation function such as sigmoid or softmax are applied before the cross-entropy loss. Finally, Focal Loss [35] was proposed to improve the classification accuracy of highly imbalanced detection dataset. The fundamental intuition behind focal loss is that it dynamically increases the weight of classes which are less in number and decreases the weight of classes which are more in number. This way, focal loss balances the loss contribution from different classes. Focal loss and its gradient is given by equation (9) and (10) [35].

$$FL(p_t) = -\alpha_t(1 - p_i)^{\gamma} log\ (p_i) \quad (9)$$

$$\frac{dFL}{dx} = y(1 - p_t)^{\gamma}(\gamma p_t\ log(p_t) + p_t - 1) \quad (10)$$

Where γ is a focussing parameter and $\alpha_t$is known as a balancing parameter. Generally, the values of the constants are set to 2 and 0.25, respectively.

2.10 MULTI-LABEL CLASSIFICATION

Multi-Label [36] classification is a generalisation of single-label classification wherein there are multiple labels associated with every classification task.

In simple terms, each instance of the dataset is associated with multiple-labels, and the classification task is to identify all the objects in an image. This is different from the object detection task, where there are bounding boxes associated with each object is also detected. In multi-label classification, there are no bounding boxes only semantically separate objects without any localisation information. In softmax approach, that we detailed in section 2.8, the output predictions are not independent of each other, i.e. they sum to a '1', however in multi-label classification the primary assumption is each predicted class is mutually exclusive of each other and thus need not sum to unity. There are a few approaches to multi-label tasks:

1. **Reframing as Binary Classification:** In this technique, the multi-label classification task is converted to a binary classification task. For example, multiple neural networks can be trained to classify different set of labels. If label L1, L2 & L3 are associated with Image1, then there can be three different models, each trained on L1, L2 or L3. Finally, the outputs of the binary classification can be combined to give an illusion of multi-label classification. One disadvantage of this technique is that there are now three separate networks and associated parameters. However, as these are binary classifiers, the performance of the models is better.

2. **Label Powerset Transformation:** In this technique, the three labels are transformed into a binary classifier as follows: L0 [0,0,0], L1 [0,0,1], L2 [0,1,0], L3 [0,1,1], L4 [1,0,0], L5 [1,0,1], L6 [1,1,0], L7 [1,1,1]. Thus, instead

of a three-class prediction task, it is now an eight-class binary prediction task. One advantage of this technique is that it considers interrelationships between labels. For example, it is possible that a person and a car are classified together with higher probability than a car or a person separately.

3. **Multi-Label Ground Truth:** This is the simplest form of multi-label classification. In this technique, a single network is used to generate multiple outputs, and each output is feed into a binary classifier such as a binary cross-entropy. The main disadvantage of this method is that it does not consider inter-label relationships and uses a single network to generate all the labels; this results in lower performance of the model. Table 2 shows the multi-label predictions and the corresponding labels. In this case, there are three classes, independently trained using a single network binary cross-entropy loss function.

	Probabilities		
Predictions	0.8	0.1	0.7
Ground Truth	1	0	1

Table 2: Multi-Label Classifier

2.11 FEATURE PYRAMID NETWORK (FPN)

One disadvantage of earlier versions of YOLO was that they were not good at detecting small objects. That was because the feature maps would become too small by the time the detection layer is reached. To circumvent this issue, YOLOv3 introduced two types of multi-scale learning. One of them

allows YOLO [30] net to be reshaped on the fly while training. This enables YOLO to learn from images of multiple sizes. Another type is a multi-scale prediction technique, namely Feature Pyramid Network (Figure 12) [37]. This technique was already used in R-CNNs [6]; however YOLO adapted it differently. For example, in Table 1, there are two YOLO layers, 16 and 23. The layer 16 is used to detect larger objects, while layer 23 is used for smaller objects. The feature maps from both the layers are then merged to incorporate more semantic information but also detect small objects. One disadvantage of this technique is that it makes YOLO layers interdependent.

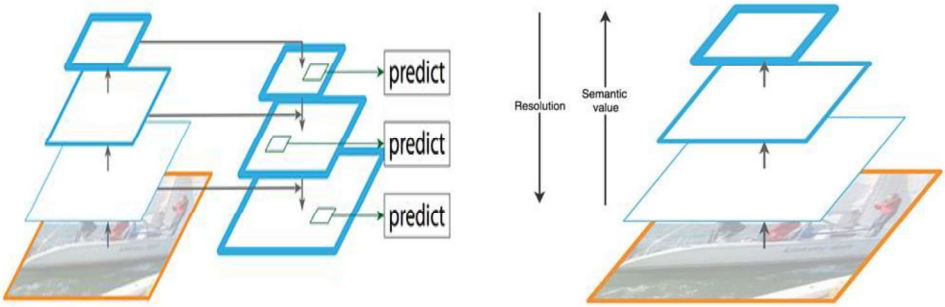

Figure 12: Feature Pyramid Network (reprinted from: [37])

2.12 METRICS FOR OBJECT DETECTION

2.12.1 Intersection of Union (IOU)

The intersection of union is used to measure bounding box precision during object detection. To define the IOU, we consider two boxes, as shown in Figure 13, the ground truth box and the predicted box.

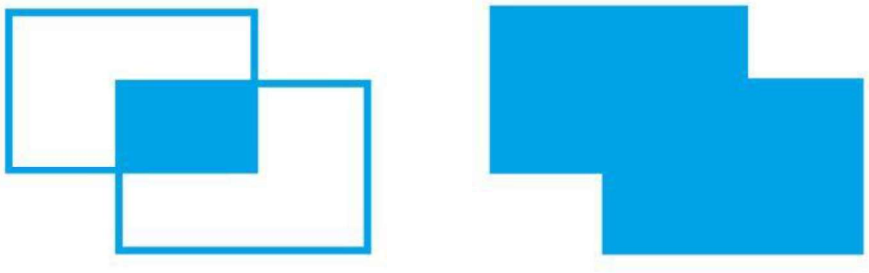

Figure 13: Intersection of Union (reprinted from: [21])

The intersection is given by the area common between them, and the union is the entire area covered by the boxes jointly. Then the IOU is given by:

$$IOU = \frac{Area\ of\ Overlap}{Area\ of\ Union} \tag{11}$$

The IOU is used to set up the threshold for object detection of true positive cases.

2.12.2 Mean Average Precision (mAP)

The primary metrics used for object detection is the mean average precision (mAP). To understand this, we need to investigate precision and recall. Suppose there are 20 objects in an image of a specific class C. Out of these, suppose a detector can predict 10 of them as class C. And out of the 10, 8 are correct (true positives) and 2 are false positives. Then precision is 8/10 and recall is 8/20. The precision and recall are given by:

$$Precision = \frac{truepositive}{truepositive + flasepositive} \tag{12}$$

$$Recall = \frac{truepositive}{truepositive+flasenegative} \quad (13)$$

This means, that for precision to be high, there should be low false positives, and for recall to be high, there should be low false negatives. Precision is also known as false-positive rate and recall is also known as false-negative rate. Finally, there is an inverse relationship between precision and recall which also depend upon model detection threshold. For example, if we set the detection threshold too high, then less true positive classes are detected, i.e. high precision but low recall, on the other hand, if we set low threshold, then more classes are identified, i.e. high recall but low precision. To estimate the average precision, the precision-recall curve is plotted by varying the threshold. AP is calculated for all classes, as shown in equation (14), and mAP is calculated over all IOU thresholds.

$$AP = \frac{1}{11}\sum_{r\epsilon[0,0.1\ldots1]} Precision(Recall_i) \quad (14)$$

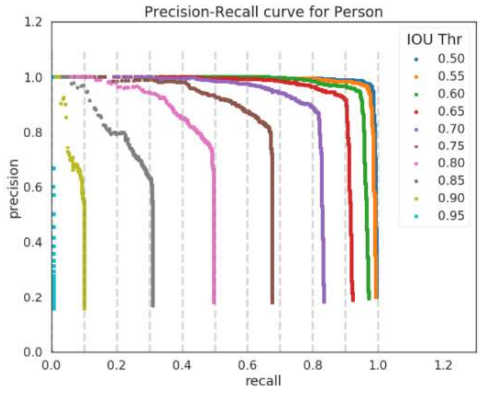

Figure 14: mAP calculation (reprinted from: [38])

Figure 14 shows the precision/recall plot for different IOU thresholds. For the Pascal VOC dataset, IOU of 0.5 is considered as a positive detection. For COCO, this is set to 0.7. This project utilised an online tool to calculate mAP [39].

3 SYSTEM DESIGN

3.1 SYSTEM OVERVIEW

The key idea of distributed neural network is to pre-process some amount of information close to the sensors [2]. Figure 15 shows the overall system in operation. A robot is being monitored from three different orientations, Front (left), Back (right) and Side (down). Depending upon the training dataset, the left orientation may result in the highest accuracy. This decision can be taken on the device itself. When a server requires to display a new image, it requests only inference information from the three devices. After receiving inference information from any of the devices, the server re-calculates the best orientation and requests image data from the corresponding client. This way, all devices are not required to stream the entire real-time data to the server.

Figure 16 and Figure 17 shows the server and client state machine. On the server-side, a new thread is spawned for every client. Initially, the server waits for a client (S1) to connect over Berkeley socket, when connected, the server initiates a FETCH_INFERENCE_INFO (S2) request and waits for the client to respond (S3). When the client responds, the server calculates the entropy of the information and compares with information available from other devices (S4). If the inference leads to the highest information entropy, then the server initiates a FETCH_IMAGE_DATA request (S5) to the device. When the device sends image data (S6), the server pushes it through the

server neural network and displays the inference data (S7). Finally, the server thread returns to state S2 and again requests for more inference information from the device. From the device perspective, it just waits for server requests and processes the commands from the server. In-state S0, the client captures video images from a UVC camera and pushes the image through the client neural network to generate inference information. In-state S1, it waits for the server data requests, the client then switches to state S2 or S3, depending upon the request from the server. Finally, after sending the information in S4, it returns to state S1.

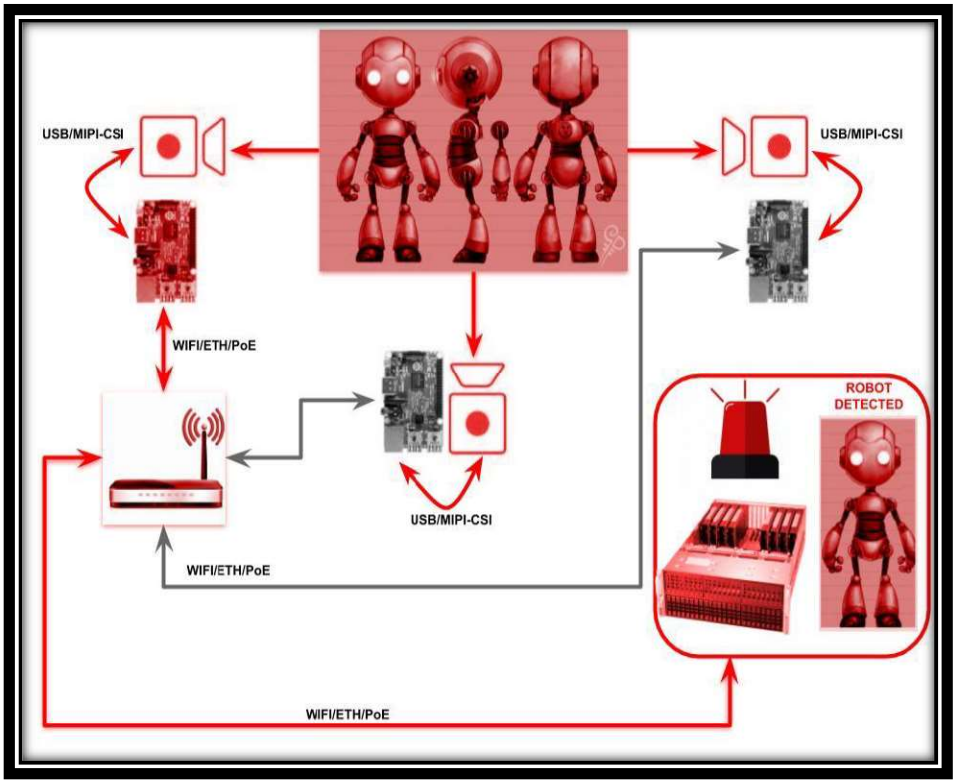

Figure 15: Overall System Design (adapted from: [40])

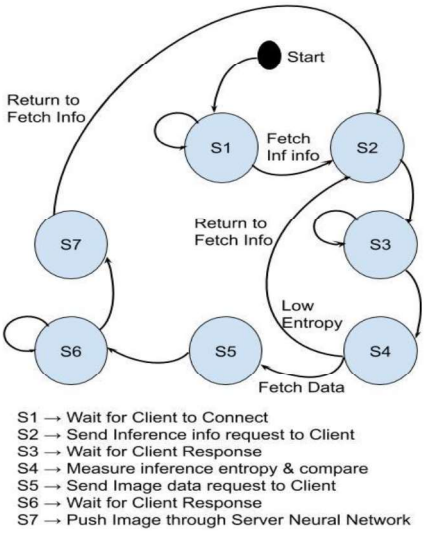

Figure 16: Server State Machine

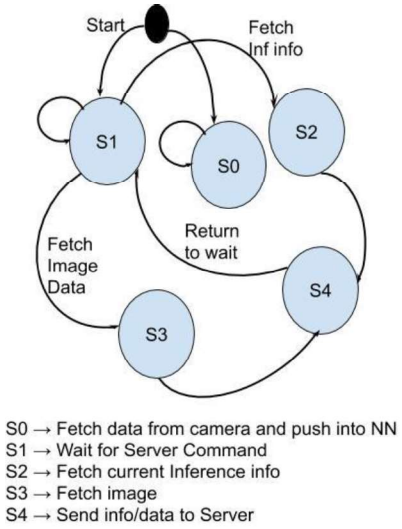

Figure 17: Device State Machine

3.2 REPRODUCING BRANCHYNET

The BranchyNet [24] article suggested adding early exits in a model to partition it for distributed inferencing. The key idea presented is that not all layers of a network are required for predicting all images. Most images are simple enough to be predicted by a smaller network.

As a starting point of this project, it was attempted to reproduce these claims. It was noted that YOLOv3 [32], which is based upon the darknet neural network framework [27] also supported CIFAR-10 [41] classifier networks. Hence, for paper reproduction, a small CIFAR-10 dataset-based network was modified with early exit. The modified network is, as shown in Figure 18. The network branches from layer→10 onwards and exits at layer → 12. The "route" layer is used to bypass this branching and preserves the

main trunk path. The network exits were jointly trained as discussed in section 3.6, except that the joint training was between the losses generated by the two Softmax layers. As expected, Figure 18 clearly shows the early exit predictions are with slightly less accuracy; however, the number of FLOPS and time required for the prediction is also lower.

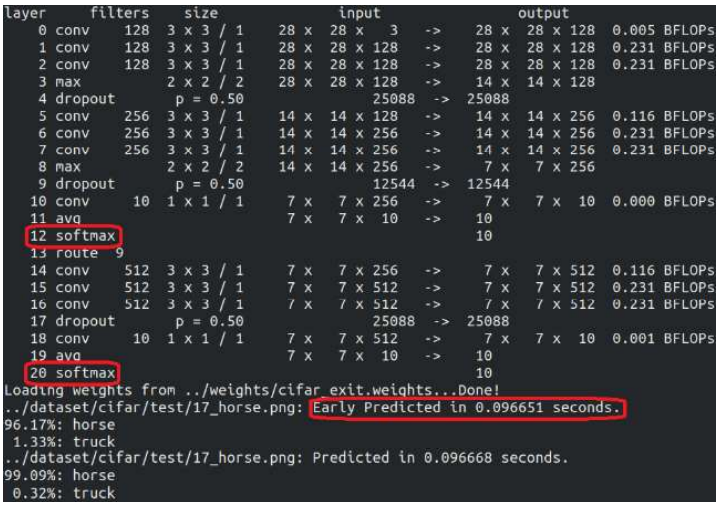

Figure 18: Reproducing BranchyNet with CIFAR-10 [41]

3.3 CHOICE OF OBJECT DETECTION MODEL

To shortlist an object detection model, various architectures were considered, and their features compared. However, the information available through literature were very varied. There was no consistent benchmark available, for example, some of the parameters that could be varied are, single-stot/multi-shot, VOC/COCO dataset, IOU threshold, light/tiny variants, GPU/CPU type, backbone classifier, official/un-official

implementation etc. Ideally, all the parameters considered should be empirically examined. However, this would have consumed considerable time. Given the time constraints for the project, Table 3 was constructed on a best effort basis from varied resources.

	FPS	Memory Footprint (MB)	FC Layers Present	Extra Packages	Accuracies (mAP)	Feature Pyramid	Focal Loss	FLOPS (Bn)	Odroid XU4 Users
Faster R-CNN	**5 (GPU)** [29]	**100** [42]	**Yes** [6]	**Yes**	**78.8%** [6]	**Yes** [6]	**No** [6]	**181.12** [7]	**No**
SSDLite	**5 (CPU)** [43]	**50** [44]	**No** [44]	**Yes**	**68%** [45]	**Yes** [44]	**No** [44]	**1.14** [7]	**No**
Retina Net	**11** [30]	**140** [46]	**No** [47]	**Yes**	**77%** [48]	**Yes** [30]	**Yes** [30]	**114** [47]	**No**
YOLOv3	**33 (GPU)** [30]	**236** [30]	**No** [30]	**No**	**82%** [49]	**Yes** [30]	**No** [30]	**65.66** [47]	**No**
YOLOv3-Tiny	**1 (CPU)** [50]	**34** [30]	**No** [30]	**No**	**66%**	**Yes** [30]	**No** [30]	**5.56** [30]	**Yes** [51]

Table 3: Comparing Object Detection Models

The colour coding indicates showstopper (red), workable (orange) and good (green), levels of comparison. For example, any red box in a row renders the row un-usable; hence, there is no point evaluating the parameters any further. The orange box indicates that the model in the row might still be usable. The green box indicates that the model in the row meets the

minimum requirement. The criterion to assign a box to a colour is based upon experience with embedded systems and not derived empirically. For example, Faster RCNN which can execute a forward pass with 5fps on a GPU will likely be very slow on an ARM Cortex-A15 processor. Below, each column is extrapolated:

1. **FPS:** Most information available for fps was for models run on GPU based platforms. However, only for SSDLite [44] and YOLOv3-tiny [50], some information was available for ARM-based CPUs. As we will be using a distributed neural network (section 2.5) which partially executes on an embedded device without a GPU, it was necessary to estimate model performance on an ARM CPU. Further, as expected, Faster RCNN is not usable at all for embedded devices without a GPU.

2. **Memory Footprint (MB):** This column captures the approximate memory required for the model. It indirectly captures the number of parameters, as well. This is necessary to estimate the memory that must be allocated on the device.

3. **FC layers present:** The third column is if the network has fully connected layers. This is required because fully connected layers prevent multi-scale training by varying in input image size dynamically. This improves the model's response to varied image sizes during inferencing.

4. **Extra Packages:** Next, the Extra Package column is comparing if the model implementation requires external packages such as python and dependencies. This is useful in estimating the amount of disk space and execution latencies for the model. It was found that only YOLO models were entirely implemented in 'C', and thus there was no dependency on a python port for the platform. Further, 'C' is the choice of programming language for most embedded systems with the least amount of runtime overhead whereas Python is an interpreter which generates machine code on the fly.

5. **Accuracies (mAP):** This column shows the recorded accuracies for all the models. The measurement criterion is mAP discussed previously. Although YOLOv3 outperforms all other models, however, it is unusable due to the other red boxes shown in Table 3.

6. **Feature Pyramid:** The feature pyramid network is discussed in section 2.11. Although it increases the multi-scale inference capability of the network, it renders the network challenging to split for distributed inferencing. For example, in the YOLOv3-tiny model, there are two YOLO exit layers. It was envisaged that a third YOLO layer could be added earlier in the network. However, the YOLO layers do not operate independently; specifically, the earliest layers are most suitable to detect small objects only. Thus, an object detection model with feature pyramid network cannot be split easily like a classification model.

7. **Focal Loss:** The Focal loss [35] is discussed in section 2.9. It was found that the VOC dataset was unbalanced, and focal loss can be used to improve model performance under such circumstances. Only, RetinaNet uses focal loss as part of its official implementation; however, it is not usable due to other red boxes in the row.

8. **FLOPS (Bn):** The billion flops criterion is a measure of the model complexity. In YOLO, this data was available on a per-layer basis. This provided some insight into splitting the model and can be useful to investigate a FLOPS vs accuracy metric.

9. **Odroid XU4 Users:** Since we shall be investigating a distributed system across multiple Odroid XU4 targets, it was necessary to study any literature available. It was found that only YOLO models were previously attempted to be executed on Odroid XU4 targets.

Final Impression: The final impression after comparing all the models was that either of YOLOv3-tiny or SSDlite could be suitable for the project. However, there was no literature for SSDlite executed on Odroid XU4. Further, due to the use of python in SSDlite, it was decided that YOLOv3-tiny to be the most appropriate model to be extrapolated.

3.4 TRAINING AND VALIDATION DATASET

The dataset used for training in this project was the 20 class Pascal VOC 2007+2012 [52], while for validation, VOC 2007 and a multi-orientation

dataset [53] was attempted to be used. The ground truth label for VOC consisted of five elements: 'class', 'x', 'y', 'w', 'h'.

Figure 19 shows the VOC dataset classes. These are grouped into multiple categories. The choice of this dataset is for the following reasons:

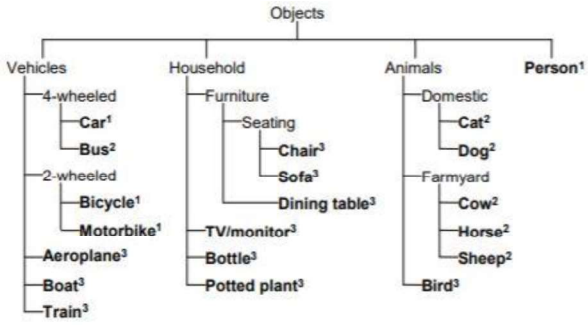

Figure 19: VOC Classes (reprinted from: [52])

1. The VOC dataset is suitable for both classification and detection. We shall later see that we use a classifier for splitting the model for distributed inferencing.

2. An unpublished report [38] indicated classification accuracies in the range 60% to 70%. This was within the range of the baseline YOLOv3-Tiny detection model (Table 3).

3. The size of the dataset was relatively small with about 17125 images for VOC 2012 and 9963 images for 2007. This was useful to train the models quickly (~12 hrs).

One issue noticed later in the project was the classes in VOC dataset were imbalanced. This shall be further discussed in section 4.

3.5 DISTRIBUTING THE YOLOV3-TINY MODEL

The key idea of distributed inferencing is to partition a neural model into a smaller network, which can be executed on a resource constraint device and a more extensive network, which can be executed efficiently on a larger system. Figure 20, shows only the backbone classifier for YOLOv3-Tiny model. The detailed model was shown in Table 1. To distribute the model into device and server sections, a multi-label classifier was added after the backbone classifier, as shown in Figure 21.

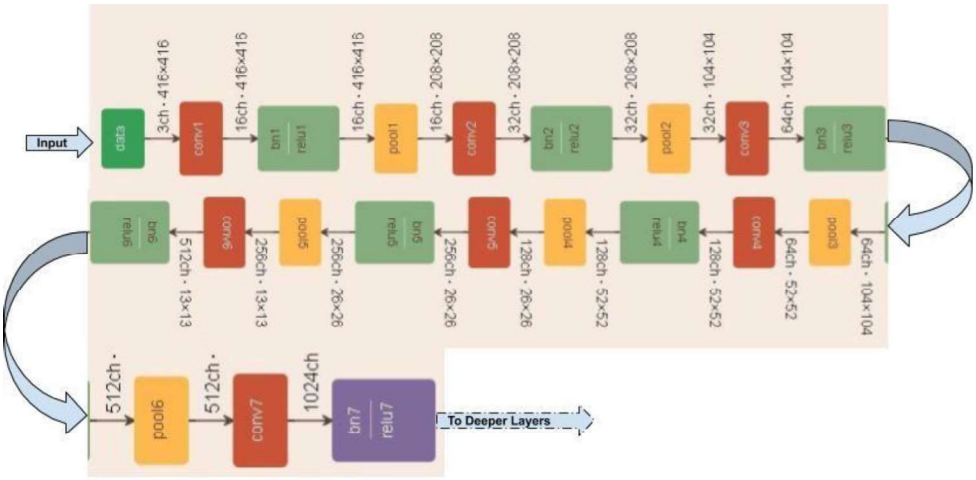

Figure 20: YOLOv3-Tiny Classifier Backbone (Generated from Netscope [54])

To implement the multi-label classifier, first, a 1x1 conv was added after the sixth maxpool layer or after layer-11 as per Table 1. The output of the 1x1 convolution is fed into a sigmoid to squash the feature vectors in the range 0→1. Finally, a binary cross-entropy loss function was applied for each of the 20 class outputs for the VOC dataset. Following are the design decisions that were taken to split the model:

1. Initially, it was attempted to split the model using another YOLO detection layer earlier in the network. It was envisaged that an early YOLO layer could be used as an early exit, similar to a classifier approach as described in [24]. However, due to feature pyramid network [37] in YOLOv3, it was not possible to utilise a YOLO layer in isolation. The early YOLO layer will be only capable of detecting small objects with less semantic value.

2. In another approach, a Softmax was added to partition the model. The class label selected for each image was that of the largest bounding box in the ground truth label. For example, Table 4, depicts a ground truth label for the VOC dataset in the format "class", "x", "y", "w", "h". To select a class against which to train, the largest bounding box in the image was selected (w * h) for classification. However, the drawback of this method was, by using Softmax, it was only possible to classify one object in the image.

Sl.No	Cls	X	Y	W	H
1	14	*0.509915014164*	*0.51*	*0.974504249292*	*0.972*
2	11	*0.344192634561*	*0.611*	*0.416430594901*	*0.262*

Table 4: Ground truth label

3. Finally, it was envisaged that a multi-label classifier as the most optimal approach for an object detection model. For example, in Table 4, both classes 14 and 11 were selected as the ground truth. The losses were calculated, as shown in Table 5 and Table 6.

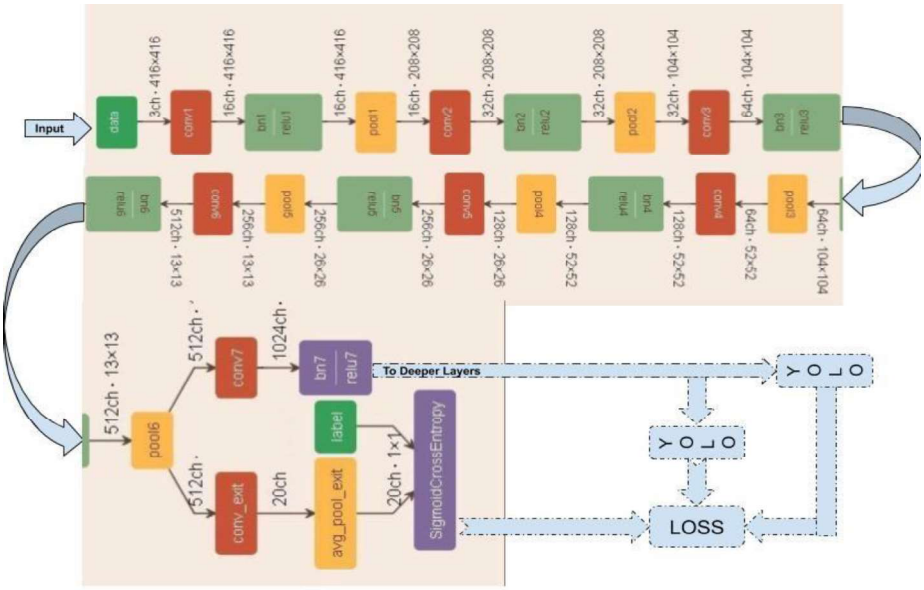

Figure 21: Intermediate Exit (Generated from Netscope [54])

Table 5: Ground Truth

0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0

LOSS FUNCTION

.1	.1	.3	.3	.1	.5	.7	.2	.2	.5	.9	.8	.1	.3	.1	.3	.5	.1	.2	.2

Table 6: Sigmoid output for each class

4. Also, it should be mentioned that the layer →11 was chosen as the preferred site for the early exit from the model. We can observe from Table 1, that the BFLOPS until this layer → 11 is 2.145 compared to the entire model with BFLOPS of 5.567. The lower FLOPS should increase the FPS to 10. Initially, layer → 9 was chosen as the early exit (with 1.746 BFLOPS); however, the classification accuracy with 9 layers was degraded. This was due to the minimal size of the network.

3.6 JOINT TRAINING WITH CLASSIFIER AND DETECTOR

As discussed in section 2.5, [24] presents the critical ideas of jointly training all the exit points from a network, as shown in Figure 21. Also, as discussed in 3.5, a multi-label classifier was used for the early exit. The joint loss was computed as per equation (1), combining the losses from the classifier and the two YOLO layers. The loss output of YOLO was shown in 2.7. As suggested in [24], the weights for the individual losses were the same. Following are the design considerations for training the model:

3.6.1 Adapting the Yolov3-Tiny Model

The original YOLOv3-Tiny model was designed for use with the COCO dataset [55]. Some of the hyper-parameters, such as IOU threshold, number of classes, iterations, anchor boxes and network structure, were not suitable for use with the VOC dataset. For example, the IOU threshold and number of classes of COCO dataset is set to 0.7 and 80 respectively, whereas for the VOC dataset it is set to 0.5 and 20 respectively. These parameters were adapted, and the results discussed in section 4.2. It was also attempted to adapt the network structure and anchor boxes of the original model for VOC dataset based upon YOLOv2 (which is suitable for VOC). However, it did not yield better results and needed to be further investigated.

3.6.2 Transfer Learning

YOLO provides two model configuration options to support, transfer learning, “dontload” and “stopbackwards”. The default YOLOv3 model was

trained on the COCO dataset [55], and the weights were transferred. The "dontload" option allows selectively not loading transferred weights into specific layers of a model, and the "stopbackwards" stops backpropagation of loss from the layer it is enabled backwards. Referring to Figure 21, the "dontload" option was applied to "conv_exit". The reason being that this was the only layer with parameters that were added to the original YOLOv3-Tiny model; thus, there was no weight initialisation possible when weights from the original model was transferred. The "stopbackwards" was applied to the layer "pool6". Thus, backpropagation from the layer "pool6" to "conv1" were disabled. This significantly reduced training time from days to about 13 hours. Although with this approach, some overfitting was observed; however, that was overcome by early training exit [56].

3.6.3 Advantages of Joint Training

The joint training, as proposed in [24] has specific benefits as against training the device and server network separately. As the entire network is trained in unison, the gradients and losses from all the network exits are generated for the same randomly chosen image in the dataset. Thus, if the partial network on a device generates a specific feature map at layer →11, then the full network shall also produce the exact feature map at layer →11 and forward it further into the network. If the networks were trained separately, then the feature maps generate at a specific layer for the device and server models will be different, this might result in different inferencing by the device and the server.

3.7 DISTRIBUTED INFERENCING WITH CLASSIFIER AND DETECTOR

For distributed inferencing, the model was split as discussed in section 3.5. However, for inferencing, the loss generated from the different layers are not used; hence, only the sigmoid outputs are required. The model (Figure 21) until the "SigmoidCrossEntropy" was pushed into three Odroid XU4 targets, as shown in Figure 15. The entire model was also placed on the server. The client-server communication was shown in section 3.1. Briefly, the devices capture data from a camera and push it through the smaller client neural network. Upon receiving a server request, the device forwards the inferencing information or the corresponding image to the server for further processing. The server upon receipt of an image pushes the data through the server network and displays the detected objects (refer section 3.1).

4 RESULTS AND ANALYSIS

Overall, the project had two different aspects required to be analysed. One is product level system performance such as FPS, Fault Tolerance, Automatic Camera Orientation Switching (ACOS) etc. Second is the modified YOLOv3-Tiny object detection network performance. For the first, we designed two experimental setups to extrapolate different high-level system performance. For the latter, we first evaluate the original YOLOv3-Tiny network performance, then show how our modifications impact the same. Finally, we also discuss a few applications of the project.

4.1 SYSTEM PERFORMANCE

Two experimental setups were envisaged to analyse system performance, each better suited to extrapolate different aspects of the system.

4.1.1 Target Emulation

In this approach, the behaviour of the system was emulated by executing the YOLO network on a standard laptop. Multiple devices were emulated by running multiple instances of the darknet [27] application. A network loopback interface was used to establish communication between all the devices and server, as shown in Figure 22. USB cameras were attached to each device instance.

This approach helped to avoid the latencies observed with ODROID XU4 devices and was useful in visualising the core operating principles of the

system. For example, by covering one or more cameras, we observed automatic camera orientation switching, fault tolerance and sensor integration in the system. If camera-2 was the current optimal orientation for inferencing, then by covering it, the active camera switched to the next most optimal high-entropy direction. Similarly, in regular operation, if one of the orientations provided more information, then that orientation was automatically selected by the server.

Figure 24 shows the entire system in operation. The first three images, C0, C1 and C2 [53] show the three-camera orientation. The fourth image is the output from the modified YOLOv3-Tiny detector; it shows that four people have been detected. The image C2-DETECTOR shows the probabilities of detection. The next image (C2-CLFR) shows the distributed multi-label classifier output for the selected camera orientation (C2). Finally, the last two pictures show the classifier output of the cameras which were not selected (C0, C1).

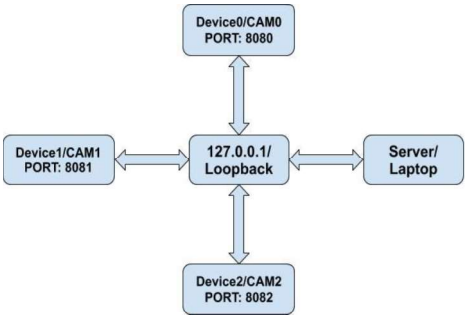

Figure 22: Emulated devices on Laptop

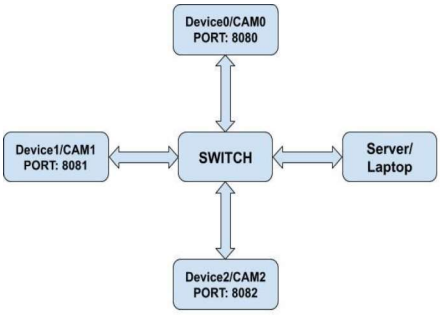

Figure 23: Devices on ODROID XU4

Figure 24: Output from Emulated Targets

It should also be observed that the classifier was able to see the bicycle, car and motorbike; however, the detector missed them. This situation can be improved by utilising a larger, more accurate detector on the server.

The automatic camera orientation switching (ACOS) can be useful in surveillance systems. For example, instead of monitoring multiple video feeds, the user might only track one interface, the choice of the most optimal feed decided by the type of information from a specific orientation.

4.1.2 ODROID XU4

In this approach, the distributed YOLO network (section: 3.7) was pushed into the ODROID XU4 devices, and instead of a loopback interface, a network switch was utilised to establish communication between the server and devices. This was useful in visualising a real system in operation and challenges associated with it. The setup was the same as Figure 15, and a simplified version is as shown in Figure 23. We observed that the system behaviour was the same as discussed in section 4.1.1; however, the latencies in framerate was apparent. The frame rate was measured to be 0.1 fps per device. Thus, for an image change to be updated, it took minimum of 10 seconds and a maximum of 20 seconds, with an average of 15 seconds. As per theoretical calculations (section 2.2), a frame rate of 4 fps was expected. The root cause of the issue was, during forward pass of the network the floating-point operations on the device was running very slow. This indicated the need for a dedicated effort to improve DNN execution on embedded

platforms. There is already research in this direction, such as Quantisation [14], Binarization [57] etc.

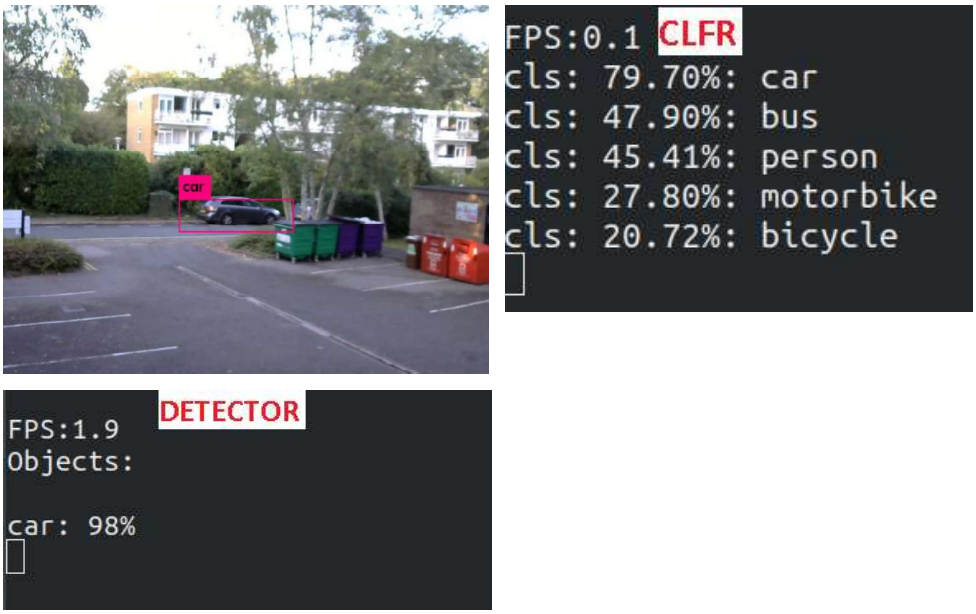

Figure 25: Output from ODROID XU4

Figure 25 shows the output from the Odroid XU4 target. We can observe that the XU4 device is operating with very low FPS (0.1). However, the overall system is operating as expected. The multi-label classifier on the Odroid device can classify a car with probability of almost 0.8. The detector has also detected the car with a high probability of 0.98.

4.2 ORIGINAL YOLOV3-TINY VALIDATION

The original YOLOv3 model, as available in [30], was trained with the COCO [55] dataset. However, for this project, the model was adapted for the VOC [52] dataset. Thus, there was a need for a baseline YOLOv3-Tiny model trained and validated on the VOC dataset. For this purpose, the network was adapted as discussed in section 3.6.1.

The outcome of the YOLOv3-Tiny model trained for the VOC dataset is shown in Figure 26. The mAP of 66.29% was obtained. However, there was an issue of overfitting observed, as shown in Figure 27. We can observe that the mAP increases rapidly from 1000 to 5000 iterations; however, gradually starts dropping after about 15000 iterations. To overcome this issue, the idea of "early stopping" [56] was applied. This meant that the model training was stopped after about 10000 iterations and weights with the best mAP was obtained at 8000 iterations.

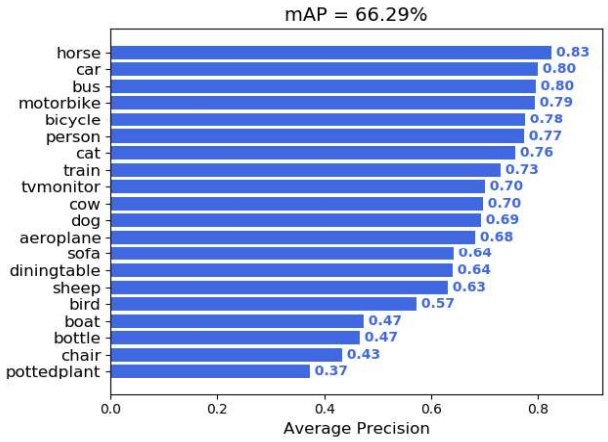

Figure 26: YOLOv3-Tiny Baseline mAP (generated from: [39])

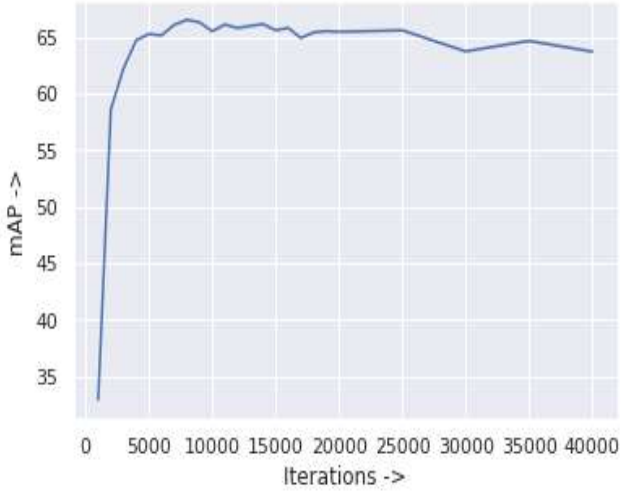

Figure 27: Overfitting with baseline model (generated from: [59])

4.3 MODIFIED YOLOV3-TINY VALIDATION

As discussed in section 3.5, the YOLOv3-Tiny model was split into two sections to be distributed partially on devices and server. Multiple approaches were adopted for this split; however, some performed better than others. For example, in the Softmax approach, we added a softmax exit from layer→11. However, it was found that using a single label Softmax classifier was not suitable for an object detection model. The Softmax only predicted a single class, whereas an object detection dataset consists of multiple classes and objects per image. Although the softmax classifier exit can still be applied in focussed use cases. For example, if object detection was used as a person detector. In that case, a Softmax classifier can be used as an intermediate exit to classify if a person was detected locally on the device. In another approach, we added an intermediate YOLO exit. However, it was found that the YOLO detection layers do not work in isolation due to the use of Feature Pyramid Networks. Thus, an early exit with YOLO would only be able to detect small objects, and larger objects would also tend to cause incorrect predictions. Finally, we adopted a Multi-Label classifier, as outlined in section 2.10. This was found to be the most suitable approach as it could predict multiple objects at once, and the classifier was also lightweight in complexity as it only predicted the classes and not the bounding boxes. Thus, making it more suitable for on-device execution.

Next, the model inference aggregation was performed using Berkley Sockets as discussed briefly in section 3.1. The socket connections were utilised to

conditionally transfer inference or image data for selective processing at the server. However, the implementation employed a polling mechanism to probe the devices in a round-robin fashion. This caused unnecessary network bandwidth usage. Ideally, an interrupt/trigger mechanism is required to be set up, such as transferring data only on motion detection.

Finally, the performance of the network was attempted to be measured under two different scenarios:

4.3.1 Single-Orientation Dataset

The VOC2007 [58] dataset was used to measure the performance of the model from a single orientation. It was hypothesized that a multi-label classifier early in the network should prime the network to extract better feature maps as the losses during training shall be forced to reduce early in the network. These optimised feature maps should improve the detection performance of the network as the depth increases. However, it was observed that on the contrary, the detection performance deteriorated. Upon investigation, it was found that the VOC dataset [58] was highly imbalanced. To circumvent this issue, Focal Loss [35] was added to the multi-label classifier instead of binary cross-entropy. Figure 28, shows the class imbalance in the VOC dataset and Figure 29 & Figure 30, shows the mAP without and with using focal loss, respectively. Comparing with the baseline mAP, as shown in Figure 26, we can observe a slight improvement in model performance from 66.29% to 66.60%.

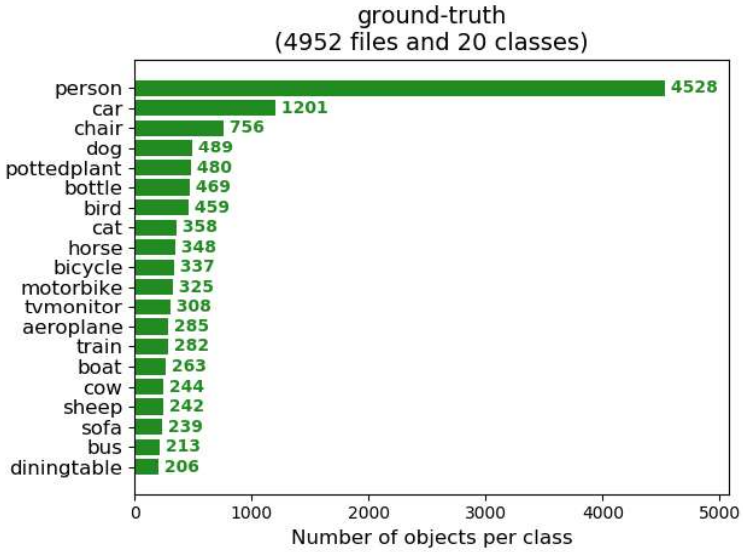

Figure 28: Ground Truth Class Imbalance(generated from: [39])

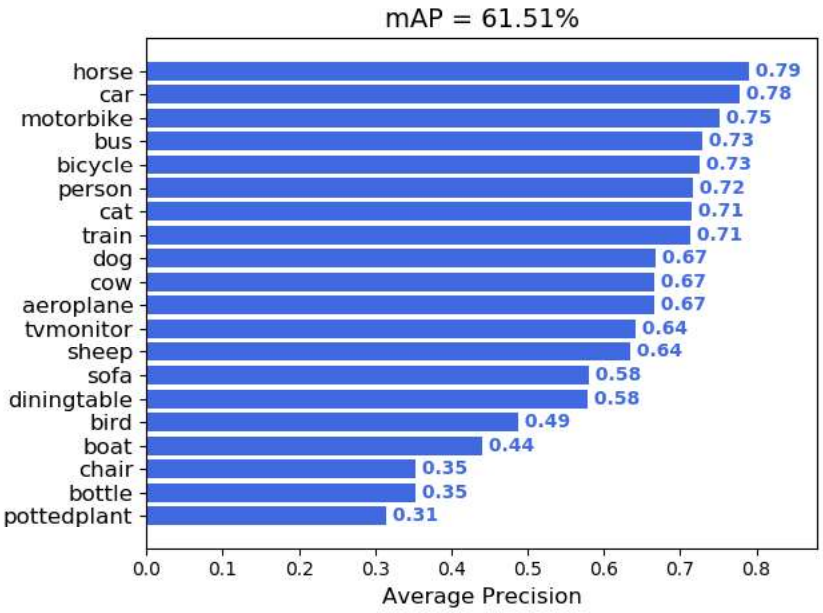

Figure 29: mAP without Focal Loss(generated from: [39])

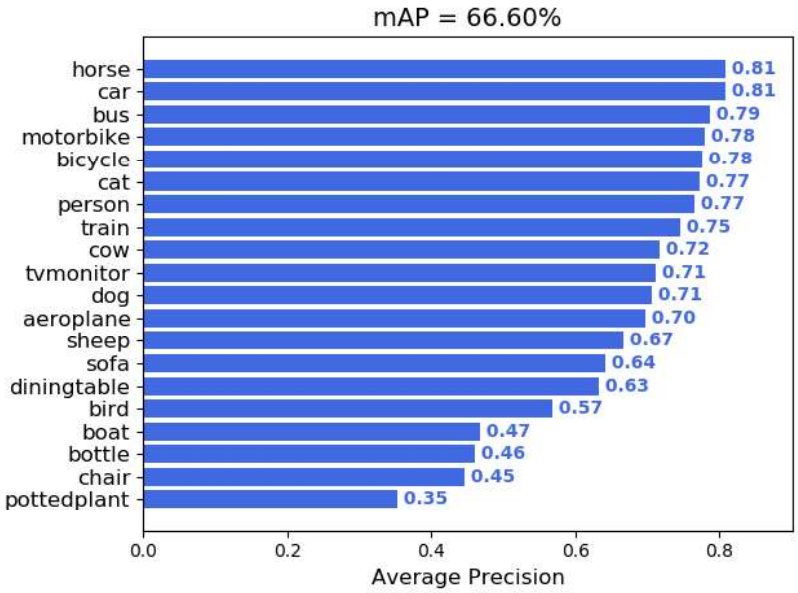

Figure 30: mAP with Focal Loss(generated from: [39])

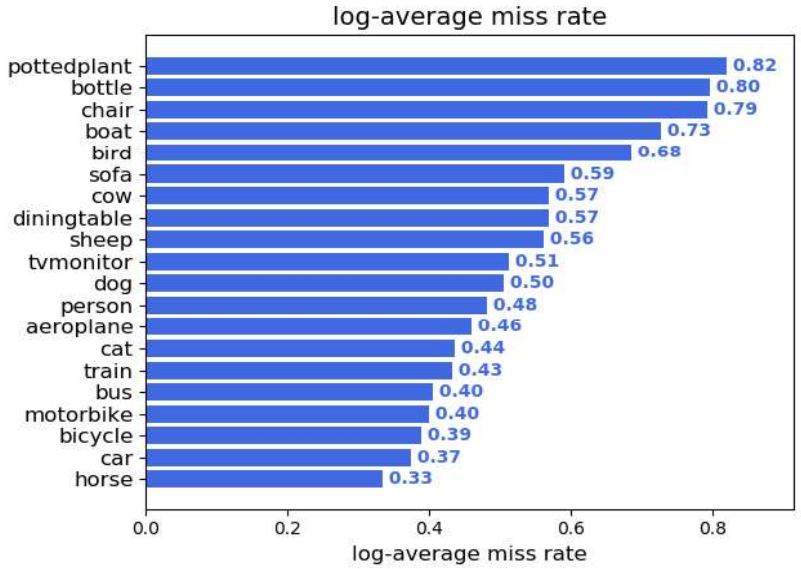

Figure 31: Log Average miss rate without focal loss (generated from: [39])

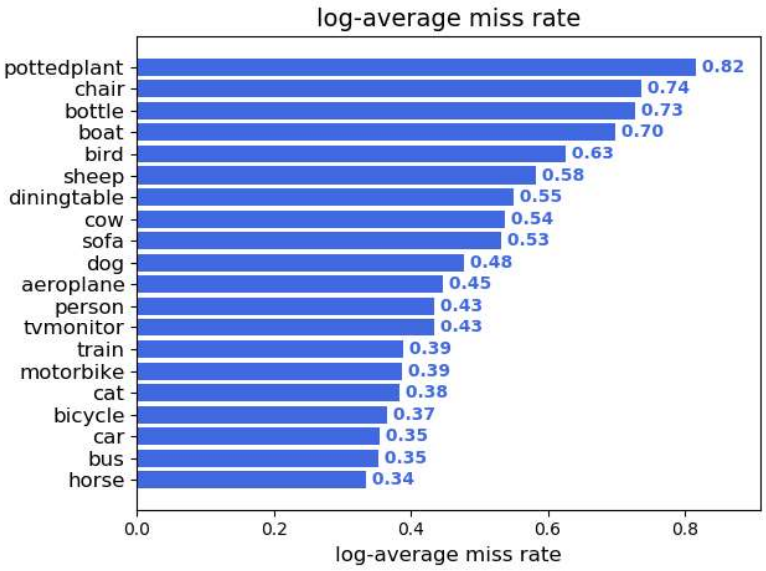

Figure 32: Log Average miss rate with focal loss (generated from: [39])

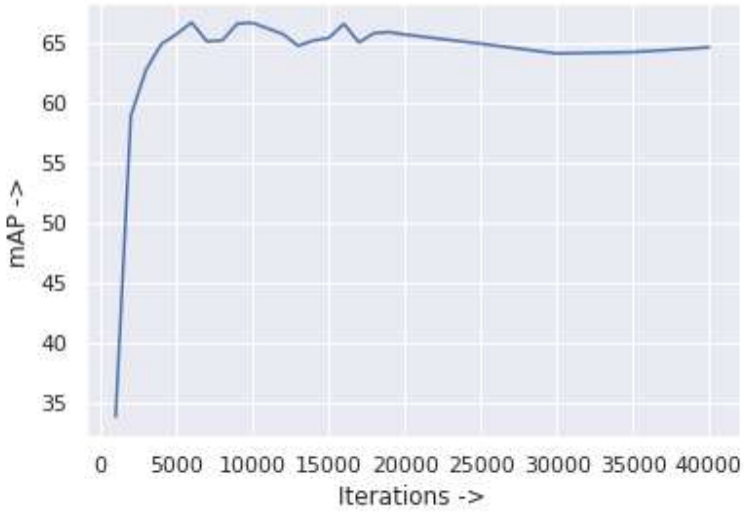

Figure 33: Overfitting in modified YOLOv3-Tiny (generated from: [59])

Figure 31 and Figure 32 shows the Log average miss rate without and with focal loss. It can be observed that almost all classes have improved performance.

Further, it should be noted that the modified YOLOv3-Tiny network also exhibited overfitting, as shown in Figure 33. To circumvent the issue, “early stopping” was applied, and the best weights were extracted at 9000 iterations.

4.3.2 Multi-Orientation Dataset

One of the challenges this project faced was a way to prove that distributed inferencing through multiple orientations could improve model performance. This necessitated the requirement of a dataset which was multi-orientation, multi-class and multi-object detection. There was only one dataset found which satisfied all the requirements. The same conclusion was also noted in [53], which introduced the first such dataset. The use of this dataset was recorded as recently as 2017 [12]. However, the dataset consisted of only 242 annotated ground truth labels with three classes: bus, car and person. The quality and size (380 x 244) of the images were also not suitable for the YOLOv3 model. Figure 34 to Figure 36, shows the detected objects from the distributed YOLOv3-Tiny network. As can be seen, the network missed objects (car/persons) which are high in the mAP, as shown in Figure 30. Overall, due to challenges in the available dataset and time constraints, the performance improvement of the distributed DNN could not

be demonstrated. It can only be inferred that with multiple orientations and a proper dataset, the performance should only improve.

Figure 34: Cam C0 View

Figure 35: Cam C1 View

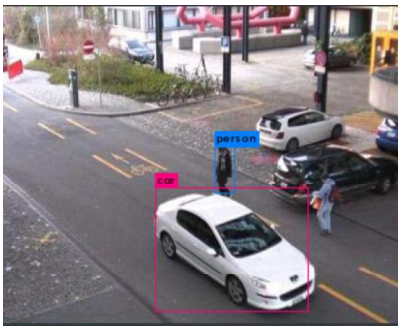

Figure 36: Cam C2 View

4.4 PROJECT APPLICATIONS

The fundamental idea of distributed deep learning presented in this project applies to innumerable use cases. However, instead of describing specific applications, it is more appropriate to map to various topologies of any distributed system. For example, this project presents an implementation of a distributed deep neural network in Star topology. However, there are various topologies that can be implemented, their advantages, disadvantages compared, and use cases realised.

Following are two most commonly realised topologies and a few examples for each:

1. **Star Topology:** A star topology is as shown in Figure 40. It is relatively easy to design & implement and maps conveniently to the hierarchical nature of the IoT infrastructure. However, there are some

disadvantages, such as if the central aggregator fails, then the entire network collapses. On the other hand, the topology is easily scalable; multiple devices can be added and removed on-the-go. Some example use case could be a surveillance system demonstrated as part of this project. This could be further enhanced to track objects/people across multiple cameras (e.g. airport security) with the ACOS feature discussed in section 4.1.1.

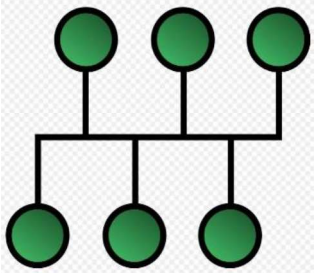

Figure 37: Bus (reprinted from: [60])

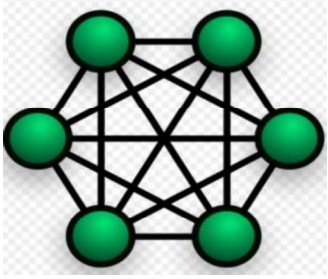

Figure 38: Fully Connected (reprinted from: [60])

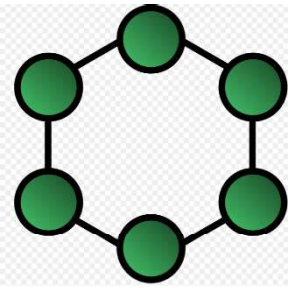

Figure 39: Ring (reprinted from: [60])

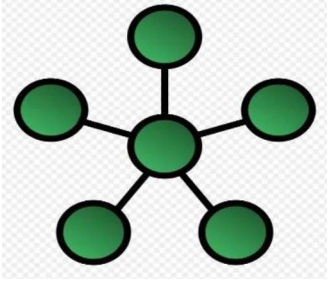

Figure 40: Star (reprinted from: [60])

2. **Fully Connected Topology:** In a fully connected topology, all nodes are interconnected and are at the same hierarchical level. Due to the massive number of independent interconnections required, this

topology is impractical as is. However, this structure can be modified to suit the requirements of specific use cases such as partially connected mesh networks. An example use case could be swarm robotics where each robot may communicate with other entities based upon its proximity. The same can also be applied to autonomous vehicle collision avoidance, where if multiple objects are in near vicinity, automatic evasive action could be taken.

5 CONCLUSIONS

With the ever-increasing data generated from ubiquitous sensors, there is a general trend in pushing computation from the cloud to edge devices [2]. One of the key enablers in this domain are deep learning models such as Object detection and Image Classification, which tend to be very resource intensive. Thus, there is a trade-off in computational resources required between deep learning models and available resources on edge devices. In view, this project aimed at distributing the YOLOv3-Tiny Object detection model in a hierarchical manner across multiple levels of devices with different computational capabilities. In the process, the project faced numerous challenges and proposed solutions for few. For example, due to dataset issues, the project could not categorically prove that distributed inferencing could increase the accuracy of the deep neural network. However, the project was able to show that even without multiple orientation dataset, just by structural changes (section: 4.3.1) and using focal loss, the mAP of the YOLOv3-Tiny model increased than baseline. Thus, with data augmentation from multiple orientations, the performance of the network must improve. Other than that, the project faced issues with latencies of the ODROID XU4 devices, however, was able to demonstrate the critical business logic of the system by emulating the device operation on a laptop.

5.1 FUTURE WORK

While executing the project, various possible optimisations were noticed. Some of these were related to enhancing system performance while others were improvements to the YOLO object network. In this section, we provide pointers to possible investigations to further improve the overall system.

1. **Improving Device Latencies:** There needs to some focussed research in improving latencies of executing a deep neural network on an embedded platform. There are many research directions such as Quantisation [14], Pruning [14], Binarization [57] etc.

2. **Utilising MALI GPU:** The Mali GPU can be used to execute the neural network models on the device. However, Mali GPU supports OpenCL, whereas YOLO supports the use of only CUDA libraries. It would be an extensive design, implementation and testing effort to replace CUDA library with OpenCL.

3. **Validation Dataset:** A proper validation dataset should be generated to measure the performance improvements due to distributed inferencing. Some work was already done in [53]. This dataset can be expanded, and more labels added for better validation.

4. **Multi-Label Classifier:** Improvements to the multi-label classifier type can be investigated. For example, instead of using the simplest form of