Distributed On-line Training for Object Detection on Embedded Devices

Khairul Mottakin

Bournemouth University

A thesis submitted for the degree of
Masters by Research(MRes)

Statement of Originality

I, Khairul Mottakin, declare that the work provided in this thesis, titled "Distributed On-line Training Approach for Object Detection on Embedded Devices," is my own. I declare that:

- This work was completed entirely while pursuing a research degree at this university.
- That any portion of this thesis that was previously submitted for a degree or other qualification at this university or another institution has been clearly disclosed.
- I have consulted other people's published work, which is always fully acknowledged.
- I have quoted from other people's work, and I have always credited the source. This thesis is entirely my own work, with the exception of such quotations.
- I have acknowledged all major sources of assistance.

Signature:

Khairul Mottakin

Date:

14 February, 2022

Abstract

In this thesis, we develop a scalable distributed approach for object detection model training and inference, using low cost embedded devices. Examples of the usage of such an approach is automatic beach litter collection using mobile robots, IoT based intelligent video surveillance system for traffic monitoring, indoor monitoring, crime and violence detection etc. Another important part of this study is using Embedded systems for distributed training, as application of above mentioned scenarios. These devices have brought about a revolution in technology due to the facts that they consume low power, are of a small size and their cost is low per-unit. Since data are distributed over devices and to speed-up the training for object detection on embedded devices, we adopt the concept of Parameter Server based distributed training, where each embedded device works as node or worker and can communicate through Parameter server(s). We develop a distributed training approach keeping in mind the resource constraints e.g. memory limitation, computational power, energy limitation etc for embedded devices. Use of Transfer Learning technique, Data Parallelism and Model Parallelism in this study reduces the resource consumption of such devices. Besides, retraining the model with continuous streaming data helps to improve the accuracy of the model as well as to further reduce resource consumption of embedded devices. Therefore, we combine the distributed training with online learning or incremental learning, where our model not only predicts in real time but also learns in real time. Additionally, this incremental learning approach discards data, once training has been done, thus save huge amount of memory on devices. Concurrent use of Knowledge Distillation and Exemplary Dataset during incremental training exhibits higher accuracy (upto 16%) compare to batch learning, most importantly without forgetting old classes. Our experiment shows that the Distributed Training (using 3-GPUs system) reduces the training time of object detection models by upto 67% comparing with non-distributed system. We test the performance of our Distributed Incremental Training approach using *Fruits* dataset as well as it's practical applicability on our own collected *Cigarette Filter* dataset. Some other associated issues such as Catastrophic Forgetting and Small Object Detection problem have also been studied in this thesis.

Keywords: Parameter Server, Online Learning, Incremental Learning, Distributed Machine Learning, Object Detection, Embedded Device, Transfer Learning, Small Object Detection, Catastrophic Forgetting, Knowledge Distillation

Acknowledgements

I would like to express gratitude to my first supervisor, **Dr. Rashid Bakirov**, for his unwavering support and guidance throughout my research. Working under his supervision has been an incredible experience that has helped me become more organized and improve my research skills. I would also like to thank **Professor Marcin Budka**, my second supervisor, for his unrelenting support and encouragement.

I also would like to express my heartfelt gratitude to **Bangabandhu Science and Technology Fellowship Trust** and the **Government of the People's Republic of Bangladesh** for providing me all kinds of funds and supports including tuition fees, living expenses, return air fare and other associated costs to study higher education in excellent universities like Bournemouth University.

I would want to convey my gratefulness to everyone of the employees at Bournemouth University for establishing a welcoming and productive research atmosphere. Especially, I would want to express my gratitude to Ms. Naomi Bailey, the Research Administrator, for her tireless efforts and support throughout my MRes program at BU.

Last but not the least, I would like to give thanks to my mother, father and all the members of my family for their inspiration and support, which helped me to reach my goals in life including this MRes study at Bournemouth University.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A Deep Neural Network (DNN) training procedure is typically labor-intensive due to the vast amount of training data and weight calculation required. Deep learning comes in for addressing such issues using convolutional transformations while focusing on important features of interest for subsequent training. Given the requirement of a large number of training data to train a DNN, distributed training approaches such as multi-GPU platforms are extensively used to accelerate the training process through parallel execution [27][54][1]. Additionally, it is important for embedded devices to reduce the training time of DNN model through distributed training. Because these devices have limited computational power, memory, and energy. On the other hand, distributed training is also necessary for use cases such as mobile robots to collect garbage to clean up beaches or streets, IoT-based intelligent video surveillance systems for traffic monitoring, indoor monitoring, crime and violence detection etc. In all such scenarios, distributed training is important because the training data are distributed over different embedded devices. From the various distributed system architecture[65], Parameter Server based distributed system has been implemented in this work. This Parameter Server strategy reduces resource consumption on embedded devices by allowing simultaneous usage of data parallelism and model parallelism. In data parallelism, the dataset is divided into 'N' portions, where 'N' is the number of GPUs. Then, these components are allocated to parallel computing systems. Similar to data parallelism, every model in model parallelism is divided into 'N' portions, where 'N' is the total number of GPUs. Then, each model is put on a separate GPU.

Apart from distributed training approach mentioned above, new data can be useful for a model to keep model up to date. Traditionally, this requires training the model from scratch again, which is costly for embedded devices. Online Learning (alternatively known as Incremental Learning, IL) allows training with continuous data without the requirement of training from scratch. This technique saves an enormous amount of space in embedded devices. Because in IL, data can be discarded after the training is completed. We combine both the distributed training and Incremental Learning approaches with an aim to reduce training time and increase the accuracy of the object detection model. However, there is a trade-off between reducing training time and increasing accuracy. In this thesis, we accomplish the reduction of training time and increased accuracy of object detection model by following 2 steps. Firstly,

by implementing Parameter Server based distributed training, and secondly by incorporating distributed training with incremental learning approach. Our experiments showed that it is possible to achieve good accuracy (comparable accuracy for real-time object detection) with reduced training time. Although somewhat similar work has been described in [59] and [25]. A significant difference with our work is that we consider embedded devices crucial for our use-cases mentioned above. Considering significant embedded device usage as well as other related literature, we have concluded that our study—which combines distributed training with IL using embedded devices—is the first experimental work in this field. This is the core contribution of this thesis.

Incremental Learning suffers from Catastrophic Forgetting. It forgets about past tasks (for both classification and regression problems) while training with new data [50] [2][70][24][38]. Our use-case belongs to classification problem. Therefore, in this thesis, we discuss problems associated with classification only. It is to be noted that IL belongs to 2 cases as mentioned in [36], namely, (1) *continuous learning of known classes with new data* and (2) *continuous learning with new data of new classes*. In this work, both cases have been handled by using Knowledge Distillation[30]. To make the IL approach more robust, an Exemplary Dataset is preserved. It consists of some old images from training dataset through augmentation. Overall, our experiments demonstrated that Catastrophic Forgetting is successfully solved for both the cases via utilizing Knowledge Distillation and using Exemplary Dataset. In other words, our Distributed Incremental Training approach is capable of learning new classes without forgetting old classes.

In order to prove the efficacy of our Distributed Incremental Training approach, 3 one-stage state-of-the-art object detection models such as MobileNet-SSD[31], SqueezeNet-SSD[33] and RetinaNet[52] have been trained. The backbones of all the 3 models are pre-trained on Imagenet[17] dataset. Two datasets, namely *Fruits* and *Cigarette Filter* (abbreviated as *Filter*) have been used in this work, to build a fruit picking robot and garbage collector from beach. Our experiment shows that using only Distributed Incremental Training approach (3-GPU system), the training time has been decreased by up to 67% compared with single GPU system. Moreover, the accuracy is increased up to 16% compared with batch training. All the 3 models have been trained and tested for the first scenario (continuous learning of known classes with new data) of Incremental Learning mentioned above. However, regarding the second scenario, we only consider training RetinaNet[52] model by adding a new class for the sake of simplicity.

After discussing the motivation and overview of this thesis, next section mentions the objectives of this work.

1.1 Aims and Objectives

Aims and objectives of this research are as follows:

1. Building a scalable parameter server based distributed training approach fusing with incremental learning technique, with an aim to reduce training time as

well as to achieve a comparable accuracy for real-time object detection.

2. From the academic literature, identify the scope and limitations associated with distributed training and incremental learning approaches.

3. Provide a solution for Catastrophic Forgetting for both the cases of incremental learning mentioned above.

4. Design the whole methodology of Distributed Incremental Learning approach considering the resource constraints of embedded devices. Strategies such as distributed training, Transfer Learning, Incremental Learning help to reduce resource consumption of embedded devices to a great extent.

5. Perform extensive analysis and comparison of the performance of the proposed Distributed Incremental Learning approach by training 3 one-stage state-of-art object detection models.

Chapter 2 discusses relevant literature on distributed systems and Incremental learning. It also discusses the importance of distributed training on embedded devices, architecture and properties of a Parameter Server based system. Then, it describes the 3 object detection models and Single Shot Detection(SSD). Moreover, it explains the concept of Transfer Learning and how it can be applied for our use cases. Chapter 2 ends with discussing our contributions.

In Chapter 3, the methodological pipeline has been discussed, followed by overview of datasets and experimental setup. Then, algorithm of Parameter Server is provided. Next, in-detailed discussion has been carried out on incorporating Incremental Learning and handling Catastrophic Forgetting. Some miscellaneous issues have been discussed briefly such as small object detection problem, class imbalance problem. This chapter concludes with how to transform the best model among the stored models for real-time inference with the help of TensorRT[63] engine.

Chapter 4 provides rigorous analysis of the results by comparing the loss curves, training time and accuracy of the Distributed Incremental approach.

Chapter 5 reiterates and claims the outcome of this study. Then it discusses some limitations of this work. At the end, some potential directions of future works have been mentioned shortly, including Federated Learning.

Chapter 2

Literature Review

2.1 Distributed Systems

Various distributed Deep Learning algorithms have been proposed in the field of distributed computing [12] [42]. However, efficient distribution of regular Machine Learning algorithm is challenging due to distinct communication approaches for embedded devices [34] [62] [71] [75] [82]. In general, the machine learning technique involves two steps, namely, training and prediction, and both of them are not mutually exclusive. Incremental Learning (IL) combines both of the phases by continuously training the model with the new data obtained using a prequential approach. In prequential approach, labels become available after the prediction phase [84].

To design a distributed Machine Learning system, choosing an appropriate topology is very important. That means the structure in which the embedded devices within the cluster are organized. According to [5], there are four possible topologies shown in Figure 2.1, namely, Centralized, Decentralized (Tree), Decentralized (Parameter Server) and Fully Distributed (Peer to Peer). The aggregation process in centralized systems (Figure 2.1a) is rigidly hierarchical and takes place in one central location. Decentralized systems enable intermediate aggregation, either through the use of a replicated model that is continuously updated when the aggregate is broadcast to all nodes, as in tree topologies (Figure 2.1b), or through the use of a partitioned model that is sharded across numerous parameter servers (Figure 2.1c). A fully distributed system (Figure 2.1d) is made up of a network of autonomous nodes that work together to provide the solution without any assigned roles.

We conform to implement Parameter Server based distributed system. The advantages of using a Parameter Server-based distributed system include (1)efficient communication via asynchronous weight updates, (2)the ability to add resources without having to restart the entire calculation, (3)the use of Model Parallelism etc. Unlike centralized systems, decentralized systems such as Parameter Server maintains a set of decentralized workers. Model parameters are stored in one or more Parameter Servers. Workers can read and write parameters as key-value pairs. Architecture and some properties of Parameter Server are discussed later in this chapter.

An open-sourced project called Ray [61] implemented Parameter Server based

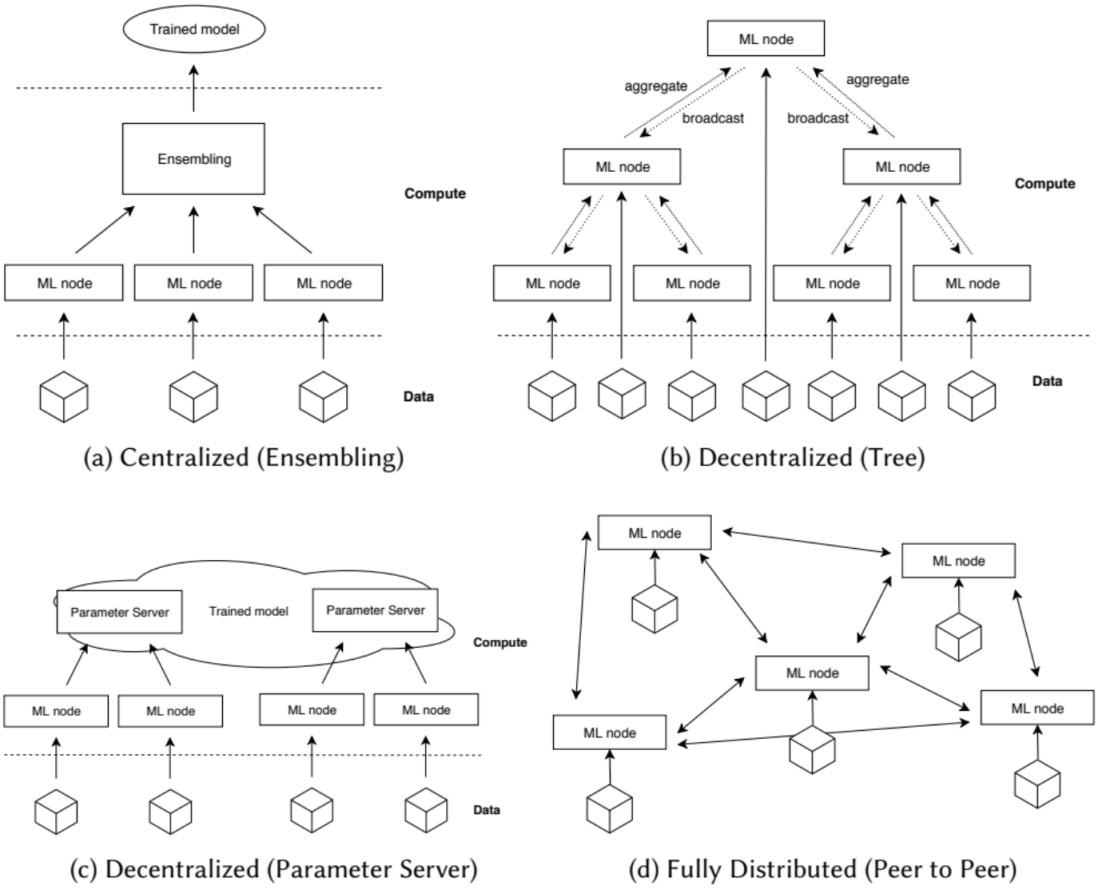


Figure 2.1: Topologies for Distributed ML depending on Degree of Distribution [84]

Reinforcement Learning (RL) training system. However, it particularly focuses on RL paradigm. Authors showed that their framework is capable of achieving better performance than existing approaches for challenging reinforcement learning applications. Apart from Ray[61] and Parameter Server[46], there are some other distributed ML frameworks such as Mahout[4], MLlib(MLBase, currently merged with Apache Spark)[60], GraphLab[57], SystemML[22], DistBelief[16] etc.

In terms of distributed training concept, there are two fundamental ways to partition the problem across multiple machines: Data-Parallel approach and Model-Parallel approach [65]. These two approaches can also be applied simultaneously [86]. In the former case, the data is partitioned into different subsets and all worker nodes apply the same algorithm to different subsets of data. The global model is accessible through the concept of centralization or replication, to produce a unified output. In the latter approach, worker nodes process the entire datasets by operating on different parts of the model. The global model then aggregates all model parts. [10] claimed that data parallelism incurs huge amount of inter-GPU communication overhead. It requires parameter synchronization among the GPUs frequently. In contrast to data parallelism, model parallelism suffers from weight staleness issues. While a worker is computing its gradient, model updates may have happened. As a

result, the gradients are often computed with outdated parameters. These gradients are called stale gradients. Thus, authors of [10] proposed an efficient and robust model parallel approach. This approach effectively handles the weight staleness. However, after analysing the Server-worker communication time using *cProfiler* of Python, our experiments showed that the simultaneous application of both the data parallelism and model parallelism reduce the time of inter-GPU communication. Hence, we conducted the experiments using both the data and model parallelism.

After reviewing important and relevant literature on distributed ML training system, let us discuss the importance of distributed ML system on embedded devices.

2.1.1 Importance and Application of Distributed ML on Embedded Devices

From the literature, it is evident that the importance of distributed ML using embedded systems is vast. According to Figure 2.2, some applications of distributed Deep Learning using embedded systems are intelligent video surveillance system [13] such as crime and violence detection, traffic monitoring [66] [89] [83]; Smart City[85], Smart Industry[45][77]. Additionally, many studies employ Edge Computing, Fog Computing, Internet-of-Things(IoT) by designing distributed Deep Learning systems using embedded systems. These studies concentrate on reducing communication cost and latency. Alternatively, these studies improve the Quality-of-Service(QoS) [90][48][88][44][87][45].

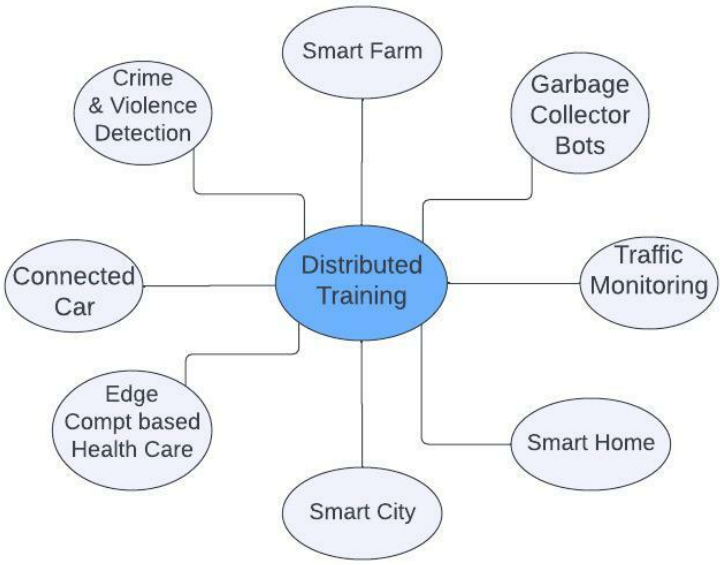


Figure 2.2: Some Applications of Distributed ML on Embedded Devices

[13] proposes an edge computing based Distributed Intelligent Video Surveillance (DIVS) system. Their experiments show that the DIVS reduces huge network communication overhead and latency by migrating network workload to edges. To improve

the balance between the computational power and workload of edge devices of the DIVS system, authors propose a dynamic data migration method. Authors of [18] propose a methodology to distribute the computation (layers) of Convolutional Neural Network (CNN) to the IoT devices. This methodology minimizes the latency. They design the methodology as an optimization problem. By periodical execution of this optimization step, the removal and insertion of IoT components take place in the IoT network configuration. This CNN based distributed platform achieves high QoS and low latency. Because the configuration of the network changes according to the optimization algorithm. Similarly, [79] reduces communication cost by over 20x by locally processing major portion of the sensor data on end-devices instead of offloading the raw sensor data to cloud for processing.

2.1.2 Applying Parallelism in Distributed Training

After discussing the importance of distributed ML on embedded devices, we need to discuss few more issues related to distributed or parallel training.

- **Model Parallelism versus Data Parallelism** As discussed earlier, some neural networks models are required to be split into multiple devices (GPUs). The reason is they cannot fit in memory of a single device. How many layers to put on a device is a contention. The disadvantage of this approach is that while training on a particular GPU is in progress, other GPUs are idle as shown in Figure 2.3.

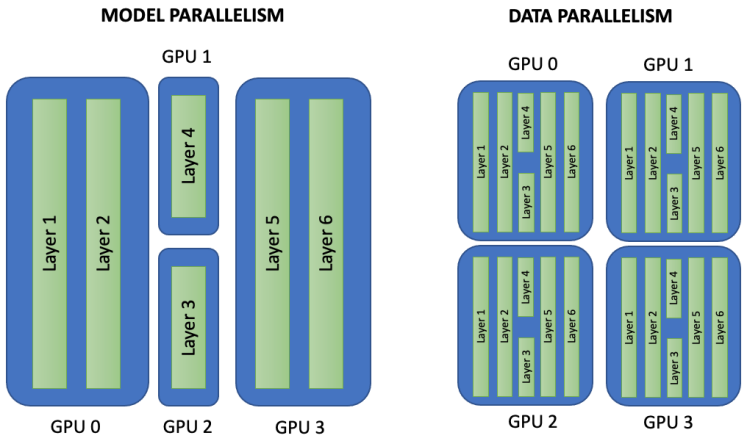


Figure 2.3: Data Parallelism and Model Parallelism[80]

Data parallelism allows us to use the same model for training on every device(s). The distinction is that each device uses different training examples to train the model. Gradients are then calculated on every device and eventually averaged by Parameter Server for a distributed system. Figure 2.3 shows both the Model Parallelism and Data Parallelism approaches.

- **Model Parallelism and Data Parallelism Simultaneously** It is possible to train the model by using both model parallelism and data parallelism. In

fact, we use both of them at the same time while training all the 3 models. The rationale of applying model parallelism is to save memory of embedded devices. Moreover, memory have been saved significantly by applying incremental learning, which is explained in section 3.4.

- **Synchronous versus Asynchronous Distributed Training** Stochastic gradient descent (SGD) is one of the most popular algorithms for training Neural Networks and it tries to find optimal values by iteration. It requires multiple iterations of training. It combines the results of each iteration into the model in order to update gradients for the next iteration. All the iterations can be run either synchronously or asynchronously among multiple devices.

  In synchronous training, all of the devices use their different parts of data obtained from a single mini-batch to train their local model. Then, they send their locally calculated gradients to all devices through server. The model is updated only when all devices have successfully computed and sent their gradients. The updated model is then sent to all nodes along with splits for the next mini-batch. Devices train on non-overlapping splits or subset of the mini-batch, as SGD uses mini-batch of the training data.

  In case of asynchronous training, there is no inter-dependency on any other devices for the updates of the model. In this type of training, devices compute gradients independently and share results through one or more central servers known as *Parameter Servers*. In the centralized architecture, the devices send their locally calculated gradients to the parameter servers. These gradients are collected and aggregated by the servers. One key difference between synchronous and asynchronous training is that, in synchronous training, the parameter servers compute the latest up-to-date version of the model and return the updated models to devices. However, in asynchronous training, parameter servers send gradients to their devices that locally compute the new model. The loop repeats until the training process ends for both types of the training architectures. Figure 2.4 shows the difference between asynchronous and synchronous training approaches.

  Despite of the fact that parallel training speeds up the training process to a great extent, however, it intuitively adds overhead due to slow network or a straggler (slow device). Since in each iteration the updated model needs to be broadcast to all other devices, the number of iterations can be decreased by increasing the mini-batch size without negatively affecting the accuracy of the model.

  In comparison between synchronous and asynchronous SGD, [14] emphasizes on using synchronous SGD with some backup workers to avoid asynchronous noise. In asynchronous SGD, workers update the parameters asynchronously. This leads to slow convergence. In another way, fluctuation in loss curve occurs in asynchronous SGD. This is called asynchronous noise. Asynchronous SGD maximizes the rate of updates with some additional cost due to asynchrony.

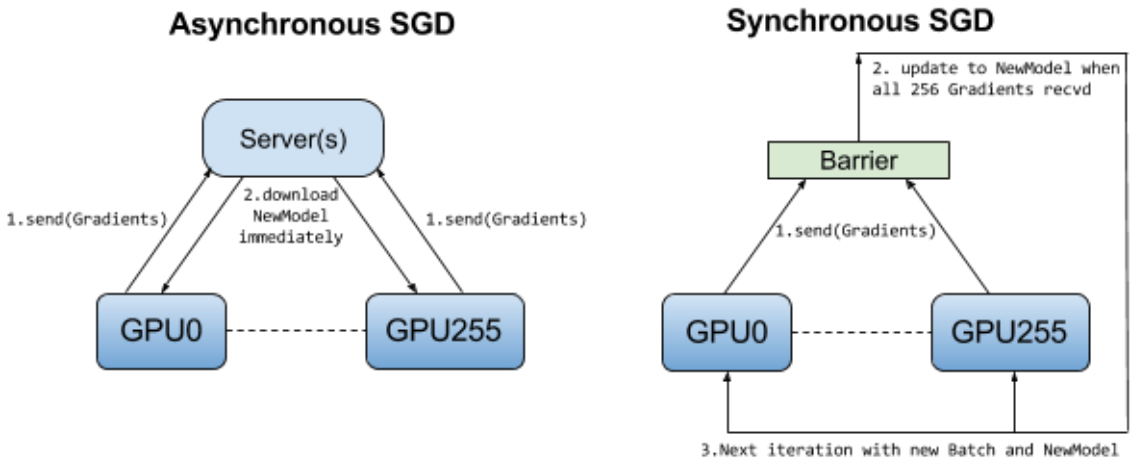


Figure 2.4: Asynchronous and synchronous training with stochastic gradient descent (SGD).[19]

[14], on the other hand, showed that synchronous approach wastes time due to waiting for straggling workers.

The next sub-section describes the architecture of Parameter Server Strategy and some of its properties.

Architecture

According to the architecture described in [47], there can be more than one parameter server grouped into a server group. Each server maintains a partition of the globally shared parameters. Servers can communicate with each other to replicate parameters in order to provide reliability.

Each worker or device usually stores a part of the training data to compute local gradients. Workers communicate only with the parameter servers, obviously not among themselves. They retrieve and update the shared parameters. They can form worker groups. They are allowed to perform any given tasks. A scheduler node handles each worker group by assigning tasks and monitoring their progress. Tasks are done by the workers asynchronously and run in parallel.

In essence, the algorithm starts by broadcasting the model to the workers. During each iteration, each worker computes its own gradients by reading its own split from the mini-batch and then sends those gradients to one or more parameter servers. The parameter servers then aggregate all the gradients from the workers. The new model is then broadcast to all workers for next iterations. In this work, asynchronous SGD is used. The overall architecture and data flow is shown in Figure 2.4 (left image).

Communication Between Worker and Server

According to [46], the shared parameters among workers and servers are represented as (key, value) pair vectors. We implement the Parameter Server based Distributed

Training approach using Distributed Remote Procedure Call (RPC) framework provided by PyTorch[15]. For our case in a loss minimization function, the pair is a Context ID (cid) and it's associated weight/gradients. PyTorch provides Distributed Backward pass to calculate the accumulated gradients under RPC framework. These terms are explained after the Algorithm 1 in Chapter 3.

push and *pull* operations are used to communicate between the workers and servers. According to Algorithm 1, each worker pushes its locally calculated gradient to the servers and pulls the latest weights back. The Parameter Server optimizes network bandwidth and computation by providing range based push and pull operation. In fact, we create global shared parameter reference (*RRef* according to RPC framework) to ease the remote communications.

Asynchronous Task

Each task is issued and handled by Distributed Remote Procedure Call (RPC) framework and executed asynchronously. That means the caller once issue a task, can do further computation. The caller considers a task as finished only when it receives an acknowledgement from callee.

Scalability

The Parameter Server allows us to add new nodes into existing cluster without restarting the running setup.

Fault Tolerance

The Parameter Server internally uses Vector Clock in order to recover from and repair the failures within 1 second. To provide fast recovery, each *(key, value)* pair is incorporated with a vector clock defined by the [47]. Vector clock records the time of each individual node on a particular *(key, value)* pair. It is advantageous for keeping track of gradient accumulation status.

Moreover, model replication and saving the state dictionary of the model help us to handle fault tolerance effectively.

2.2 Online Incremental Learning

As opposed to standard batch learning, our focus is the continuous learning of object detection models. There can be two different scenarios for continuous learning [37]:

1. **Continuous learning for old classes:** This case requires traditional online learning configuration, since more training data are acquired for known classes.

2. **Continuous learning for old and new classes:** It might be a case that we get data for new class in addition to the previous case.

In [40] the authors developed an incremental object detector by expanding original training sample domain incrementally. This incremental object detector is based on probabilistic multi-center embedding system. As authors claimed, it can served as a lifelong learning for object detector. Because it handles continuous stream of unlabelled video data. However, it suffers from model drift. Model drift refers to the degradation of model performance due to changes in data and relationships between input and output variables. To prevent model drifting, authors suggested infrequent human intervention requested by model. This human intervention belongs to another domain of research called *Active Learning* [72][7]. Active Learning selects unlabelled data, so that when labelled, it improves the model most. It makes the process of labelling new data more efficient. Sometimes, active learning also considers to select the most uncertain data for labelling.

[43] develops an incremental object detection system called RILOD. It capables of detecting new object classes without forgetting the old classes. To avoid Catastrophic Forgetting, authors applied Knowledge Distillation [30] to imitate the old classes' behaviour particularly on object classification, bounding box regression and feature extraction from old classes. Additionally, they designed a real-time dataset construction and annotation technique for new classes.

It is obvious that incremental learning suffers from Catastrophic Forgetting. Experiment of [24] suggests to use Dropout while training a model. Dropout tackles the forgetting by remembering the old tasks. Additionally, it can adapt to the new task. This study also showed that the choice of activation function should always be cross-validated. [39] proposed an alternate method to tackle the forgetting based on total absolute signal passing through each connection in the neural network. The problem can also be handled by slowing down the learning of the weights which are important to remember the old class [38]. Alternatively, transfer learning also helps to overcome the forgetting on old tasks [26].

In this work, we applied a combination of different techniques such as Knowledge Distillation[30], Dropout[76], maintaining small exemplary dataset[68], transfer learning and restricting large changes of weights to reduce forgetting.

2.3 Concept and Applicability of Transfer Learning

Transfer Learning is a popular method in the field of deep learning. It leverages the use of a learnt model to solve similar problems. Therefore, it can avoid learning from scratch. In an online article [58], the authors show *how to implement a transfer learning solution for image classification problems*. A pre-trained model is used in transfer learning. It is usually trained on a large benchmark dataset using published architectures such as MobileNet [31], VGG [73], Inception [78] etc. An extensive review has been presented by [8] on performance of pre-trained model on computer vision related problems using ImageNet [17] dataset.

The entire Transfer Learning process can be divided into three steps, namely,

1. **Choosing a pre-trained model:** One can choose a pre-trained model from a wide variety of pre-trained models e.g. ResNet5 [28], MobileNet [31], VGG [73], InceptionV3 [78] etc.

2. **Categorising problem according to dataset Size-Similarity Matrix:** To categorise a computer vision problem, the Size-Similarity Matrix shown in Figure 2.5 takes into account: (1) the size of dataset and (2) its similarity to the dataset in which pre-trained model was trained. Dataset similarity means prior knowledge about the dataset. For example, if we want to detect fruits in an image, then Open Images [41] dataset would be a similar one. Because it has the images of variety of fruits. As a common unwritten rule, a dataset is considered to be small if it has less than 2500 images per class.

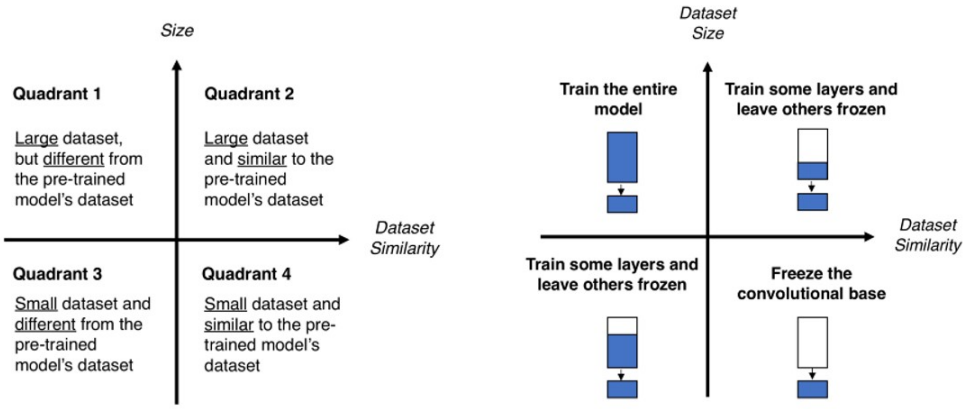


Figure 2.5: Size-Similarity matrix (left) and Map for Decision for Fine-tuning pre-trained model(right) [8]

3. **Fine-tuning the final model:** There are 3 ways to fine-tune a model (shown in Figure 2.6), namely, (1) Train entire model, (2) Train some layers and keep remaining layers frozen and (3) Freeze all convolutional layers. The first strategy requires large dataset and high computational power, to train a model from scratch. To use the second strategy, we need to decide which layers should be frozen. This is problem specific. Usually, more layers are kept frozen to avoid overfitting if we have small dataset and large number of parameters [3]. On the other hand, if we have large dataset and small number of parameters, then the model performance can be enhanced by training more layers for the new task, since there are no issues of overfitting.

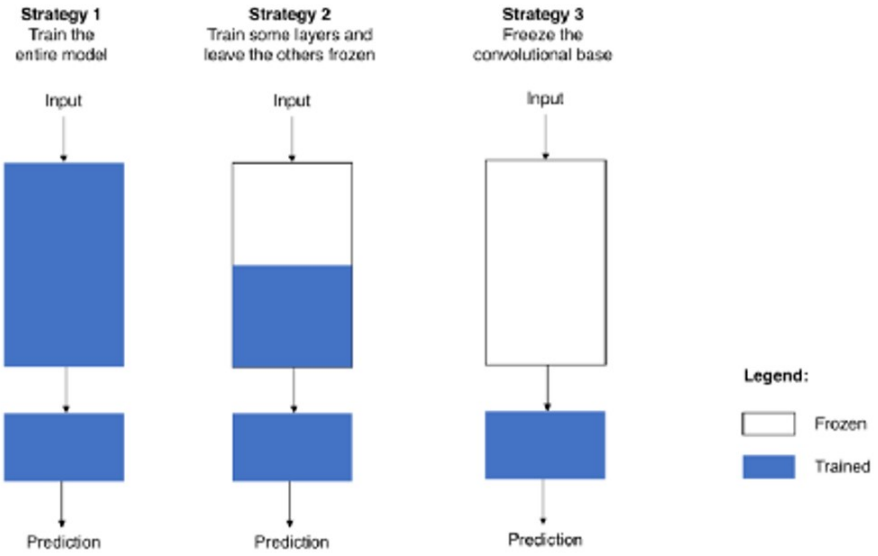


Figure 2.6: Strategies to Fine-tune a Pre-trained Model[8]

The Quadrant 1 of matrix in Figure 2.5 refers to the case of a large dataset which is different from pre-trained model's dataset. This case indicates to choose the first strategy, *train the entire model*, which is shown in Figure 2.5 (right side).

Quadrant 2 represents the case with a large dataset, similar to the pre-trained model's dataset. For this case, the sensible approach should be to follow the Strategy 2 of Figure 2.6. We can train the model as much as we want, due to having large dataset. Since the dataset is similar, we can take the advantage of using the previously learnt knowledge. It requires to train the top layers of convolutional base and the classifier only.

Quadrant 3 is a challenging scenario where we have small dataset which is different from the pre-trained model's dataset. We neither can go deeper into the model due to the risk of overfitting nor we can hold shallow of the model which will not learn much functional. So, trial and error are the solution. It is advised to go deeper into the network. To avoid overfitting, several techniques such as Data Augmentation or Regularization can be applied.

The last segment, Quadrant 4 is the case of having small dataset similar to the pre-trained model's dataset. The probable best option is to follow Strategy 3 mentioned above. According to Strategy 3, we need to remove the last layer (fully connected layer or global average pooling layer) and use the pre-trained model as a fixed feature extractor. Then, we can use the extracted feature to train the new classifier.

The next four sub-sections describe about the MobileNet[31], SqueezeNet[33], RetinaNet[52] and SSD-300[56] architecture.

2.3.1 MobileNet [31]

The MobileNet model is built on depthwise separable convolutions and pointwise convolution to reduce the model size (fewer parameters) and complexity (fewer Multi-Adds). The depthwise convolution filter uses a single filter for each input channel, and the pointwise convolution filter combines the output of depthwise convolution linearly with 1 by 1 convolutions, depicted in Figure 2.7. In the Figure 2.7, we have 5 separate $D_K \times D_K$ spatial convolutions because of 5 channels. Then pointwise convolution is applied which changes the output dimension. These two steps factorization significantly reduce the model size and complexity. In contrast, a standard convolution works in single step, which involves filtering and combining inputs into a new set of outputs.

The computational cost for a standard convolution with Kernel size D_K, Feature map D_F is given by,

$$D_K \times D_K \times M \times N \times D_F \times D_F$$

where M and N are the number of input and output channels respectively.

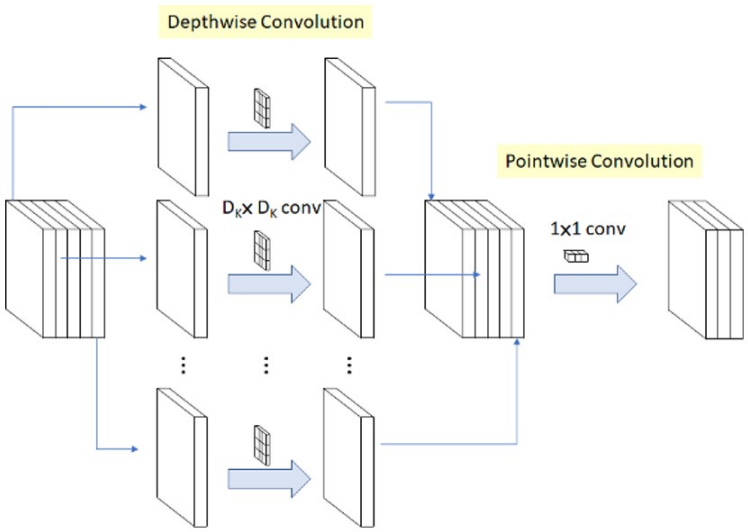


Figure 2.7: Depthwise Convolution with Pointwise Convolution [81]

The computational cost of depthwise convolution with M number of input channels is given by,

$$D_K \times D_K \times M \times D_F \times D_F \tag{2.1}$$

The depthwise convolution considers filtering the input channels only and does not produce new features. To create these new features, pointwise convolution is considered and it's computational cost is as follows,

$$D_F \times D_F \times M \times N \tag{2.2}$$

By combining both the costs of Equation 2.1 and Equation 2.2, the depthwise separable convolutional cost yields,

$$(D_K \times D_K \times M \times D_F \times D_F) + (D_F \times D_F \times M \times N) \tag{2.3}$$

Thus, the overall reduction in computation is,

$$\frac{(D_K \times D_K \times M \times D_F \times D_F) + (D_F \times D_F \times M \times N)}{D_K \times D_K \times M \times N \times D_F \times D_F} = \frac{1}{N} + \frac{1}{D_K^2} \tag{2.4}$$

As an example, if 3×3 kernel size is used, MobileNet architecture uses 8 to 9 times less computation in comparison with standard convolution sacrificing only a small amount of accuracy. Authors of [31] compared performance between full standard convolution based MobileNet and depthwise separable MobileNet on ImageNet dataset. They showed a full convolutional MobileNet yields 71.7% accuracy requiring 4866 million multiplication-additions and 29.3 million parameters, whereas depthwise convolutional MobileNet achieves 70.6% accuracy using only 569 million multiplication-additions and 4.2 million parameters. This indicates a humongous reduction in computation and number of parameters with only 1% loss of accuracy.

Overall, the MobileNet architecture has 28 layers including pointwise and depthwise convolutional layers and has been defined in the following Figure 2.8.

All layers have been passed through a Batch Normalization and ReLU except the final FC layer, which has been feeded into a softmax layer for the purpose of classification. These normalization and nonlinear transformation for all layers (pointwise and depthwise) are compared with a standard convolution layer in Figure 2.9.

2.3.2 SqueezeNet [33]

The goal of this paper is to identify a model to keep very few parameters while preserving accuracy. The approach has been proposed in this paper to take an existing CNN model and compress it in a lossy fashion. The main strategies that have been employed are: (1) Replacing 3 by 3 filters with 1 by 1 filters (2) Decrease number of input channels to 3 by 3 filters (3) Downsample late in the network so that convolutional layers have large activation maps.

The author's intuition is that large activation layer leads to higher classification accuracy. The first two strategies decrease the number of parameters in a CNN preserving accuracy. Third strategy maximizes accuracy on a limited budget of parameters. The author proposed Fire module CNN architecture that successfully employed three strategies.

In the Fire module, a squeeze convolution layer feeds into an expand layer that mix two different size of convolution filters: 1 by 1 and 3 by 3. In the Fire module, the combination of squeeze layer with all 1 by 1 filter, expands layer with 1 by 1 filters and expand layer with 3 by 3 filters. The SqueezeNet begins with one convolution layer and 8 Fire modules and ended with one convolution layer. RELU activations is applied from squeeze and expand layers and Dropout is applied after fire9 module.

To evaluate SqueezeNet[33], the AlexNet model is being used. The goal was to compress AlexNet model that was trained to images using the ImageNet dataset.

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 2.8: MobileNet Architecture *(Reference: [31])*

SqueezeNet[33] is 50 times smaller than AlexNet with equivalent accuracy. The architectural exploration is divided into two main topics: microarchitectural exploration (per-module layer dimensions and configurations) and microarchitectural exploration (high-level end-to-end organization of modules and other layers).

2.3.3 RetinaNet [52]

Object detector are based on two stage proposal-driven mechanism. The first stage generates sparse set of candidate object location and second stage classifies each candidate location as one of the foreground classes or as background using CNN work. However, it is a matter of question if ones-stage detection can give similar accuracy. One-stage detectors are applied over regular dense sampling of possible

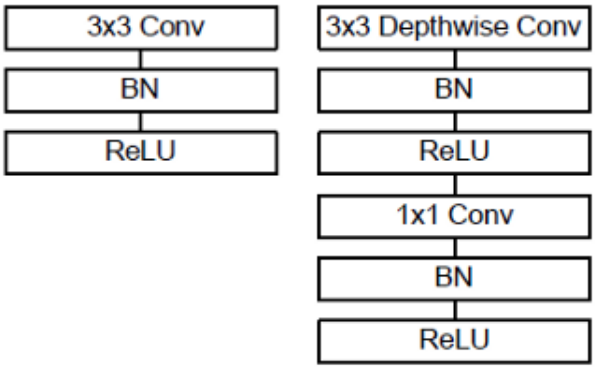


Figure 2.9: Regular Convolution (Left), Depthwise Convolution (Right) with Batch Normalization and ReLU nonlinearity *(Reference: [31])*

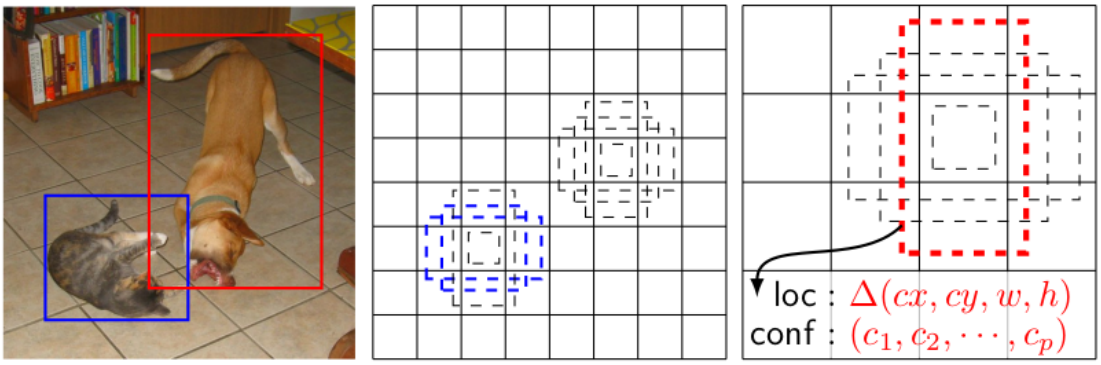
object location which is potentially simpler and faster but accuracy is not near to two stage detectors. This paper investigated the reason of this case. The authors have discovered the extreme foreground-background class imbalance encountered during training of dense detectors is the central cause. To address this problem, the author has proposed to address class imbalance by reshaping the standard cross entropy loss such that it down-weights the loss assigned to well-classified examples.The author has proposed a new loss function that acts more effective alternative to the previous approaches for dealing class imbalance. The novel Focal loss focuses on training a sparse set of hard examples and prevents a vast number of easy negatives. The Focal Loss gives significant high-accuracy than previous one-stage detectors.

2.3.4 Single-Shot Detection (SSD) [56]

SSD-300 is a forward real-time object detection framework that achieves high-accuracy using relatively low resolution input images and increases detection speed. These have been possible since authors of [56] utilize a small convolutional filter capable of predicting object categories and offsets of bounding boxes (fixed set of default boxes). The small convolutional filter uses separate filters for different aspect ratio detections. Then it applies these filters to multiple feature maps for detecting at multiple scales. SSD has three following features:

- **Multiple-Scale Feature Maps:** SSD model has additional convolutional feature layers at the end of MobileNet base network. These auxiliary layers have been reduced in size subsequently and help in detection at multiple scales.
- **Convolutional Predictors:** Each feature layer is capable of producing a fixed set of detection predictions with the help of convolutional filters shown in SSD architecture in Figure 3.2. For example, if we have $x \times y$ feature layer with z channels, then we have standard $3 \times 3 \times z$ *small kernel*, each of which is applied at each $x \times y$ locations and generates a score for object category or offset values for default bounding boxes.

- **Aspect Ratios and Default Bounding Boxes:** Likewise the anchor boxes used in Faster R-CNN [69], SSD uses default bounding boxes with each feature map cell. For each location in a given feature map and for k boxes, there are $(c+4)k$ filters, where c stands for class scores (class scores are calculated for each class if default boxes contain any object belong to that class) and 4 is for offset values for original default box shape. If there are $x \times y$ feature maps, SSD produces $(c+4)kxy$ outputs, although feature maps are of different resolutions. Default boxes have been explained in Figure 2.10. We explain the aspect ratio for bounding boxes of SSD network later on this section.



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

Figure 2.10: *Reference: [56]*

Let us now discuss the training objective (loss function) and other related theoretical aspects of SSD such as Data Augmentation, choosing set of default boxes and different scales, hard negative mining etc. These are crucial while training. During training, matching strategy begins with matching default boxes to any ground truth boxes having Intersection over Union (IoU) higher than a threshold, ideally 0.5. IoU is defined as follows:

$$IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union} \tag{2.5}$$

SSD is able to handle multiple object categories for detection. Since, SSD loss function is a weighted sum of two loss functions, namely, Confidence Loss and Localization Loss, the overall loss function is defined as follows:

$$L(x, c, l, g) = \frac{1}{N}\alpha(L_{loc}(x, l, g)) + \frac{1}{N}(L_{conf}(x, c)) \tag{2.6}$$

N indicates the number of matched default boxes and the loss is 0 if there is no matched boxes found. The Confidence Loss is calculated using Softmax Loss. For multiple class confidence, c, we can define the Confidence Loss,

$$L_{conf}(x, c) = -\sum_{i \in Pos}^{N} x_{ij}^{p} log(\hat{c}_i^p) - \sum_{i \in Neg} log(\hat{c}_i^0) \tag{2.7}$$

where the Confidence Score c for i-th boxes of p categories is calculated by

$$\hat{c}_i^p = \frac{exp(c_i^p)}{\sum_p exp(c_i^p)} \tag{2.8}$$

The Localization Loss is calculated using Smooth L1 Loss [23] on the center (cx, cy) of default bounding box, d and the logarithm of heights, h and widths, w, which is as same as in Fast R-CNN. The Smooth-L1 Loss function is less sensitive to outliers comparing with other error loss functions and it is defined between the predicted box (l) and the ground truth box (g). The α indicates a weight multiplier in Localization Loss which is 1 using cross validation.

$$L_{loc}(x, l, g) = \sum_{i \in Pos}^{N} \sum_{m \in \{cx,cy,w,h\}} x_{ij}^k smooth_{L1}(l_i^m - \hat{g}_j^m) \tag{2.9}$$

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w \qquad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h \tag{2.10}$$

$$\hat{g}_j^w = \log\left(\frac{g_j^w}{d_i^w}\right) \qquad \hat{g}_j^h = \log\left(\frac{g_j^h}{d_i^h}\right) \tag{2.11}$$

For different object scale, the tiling of default bounding boxes is designed in SSD in such a way that each feature map learns that particular scales responsively. In Figure 2.10, there are two feature maps having size of 8×8 and 4×4, each of which capable of learning two different object scales indicated by blue and red default boxes. If there are m feature maps, the scale of default boxes is calculated in SSD framework by:

$$scale_k = scale_{min} + \frac{scale_{max} - scale_{min}}{m - 1}(k - 1), \quad k \in [1, m] \tag{2.12}$$

$scale_{max}$ means the scale for highest layer and is set to 0.9, whereas $scale_{min}$ means the scale for lowest layer and is set to 0.2.

SSD defines 4 different aspect ratios for default boxes, along with aspect ratio of 1 and they are $a_r \in \{2, 3, \frac{1}{2}, \frac{1}{3}\}$. Thus, the width and height for each default boxes can be computed by $w_k^a = scale_k\sqrt{a_r}$ and $h_k^a = scale_k/\sqrt{a_r}$ respectively. In these ways of applying various scales and aspect ratios for default boxes, SSD generates robust set of predictions for several shapes and sizes of input objects.

During training, since the model might produce large number of default boxes and mostly belong to negative class, the class imbalance takes place. This has been tackled by taking highest Confidence Loss negative examples and maintaining the ratio of negative to positive class by maximum 3:1, which shows faster convergence and a firm training.

For Data Augmentation, the designer of SSD model suggest either to use the whole input image, or sampling a patch randomly, or sampling a patch with minimum IoU is 0.1, 0.3, 0.5, 0.7 or 0.9.

Chapter 3

Methodology

3.1 Overview

In this thesis, two areas of ML research have been combined to implement distributed online learning-based object detection model using embedded devices. These are:

1. Distributed Machine Learning Training
2. Online Incremental Learning

Distributed Machine Learning is a widely adopted approach since scaling-up and parallelism could enhance speed of model's training and inference to a great extent. Particularly, it is important for our use-case such as beach litter collection, since training data are distributed over different embedded devices. Training Machine Learning models is a highly data-intensive task. Scaling up is an effective technique to reduce the impact of large volume of input and output on the workload performance by introducing parallelism over cluster of multiple machines. Additionally, resilience against failures of single machine leverages the use of distributed system [20] [74]. The second step, Online Incremental Learning Scheme [35] has been applied, enabling continuous exploration of new training data. This approach helps in an overall increased performance for object detection model. Both of the steps are discussed in detail later in this chapter.

According to the methodological workflow shown in Figure 3.1, Parameter Server Strategy [47] has been opted as distributed system topology. In this strategy, each worker (embedded device) calculates its own gradients using own splitted data and sends those gradients to one or more parameter servers. The parameter servers aggregate all the gradients from the workers and then broadcast the updated model to all workers. An algorithm in section 3.3 has been developed to discuss further details on implementation of Parameter Server Strategy.

The Distributed Training starts with an initial offline object detection model using Transfer Learning. This procedure has been depicted in Figure 3.1 (top figure). For the offline training, we use models pre-trained on ImageNet dataset. The rationale for using this technique is due to small dataset per class and computational power constraints of embedded devices. The Transfer Learning step in our methodology,

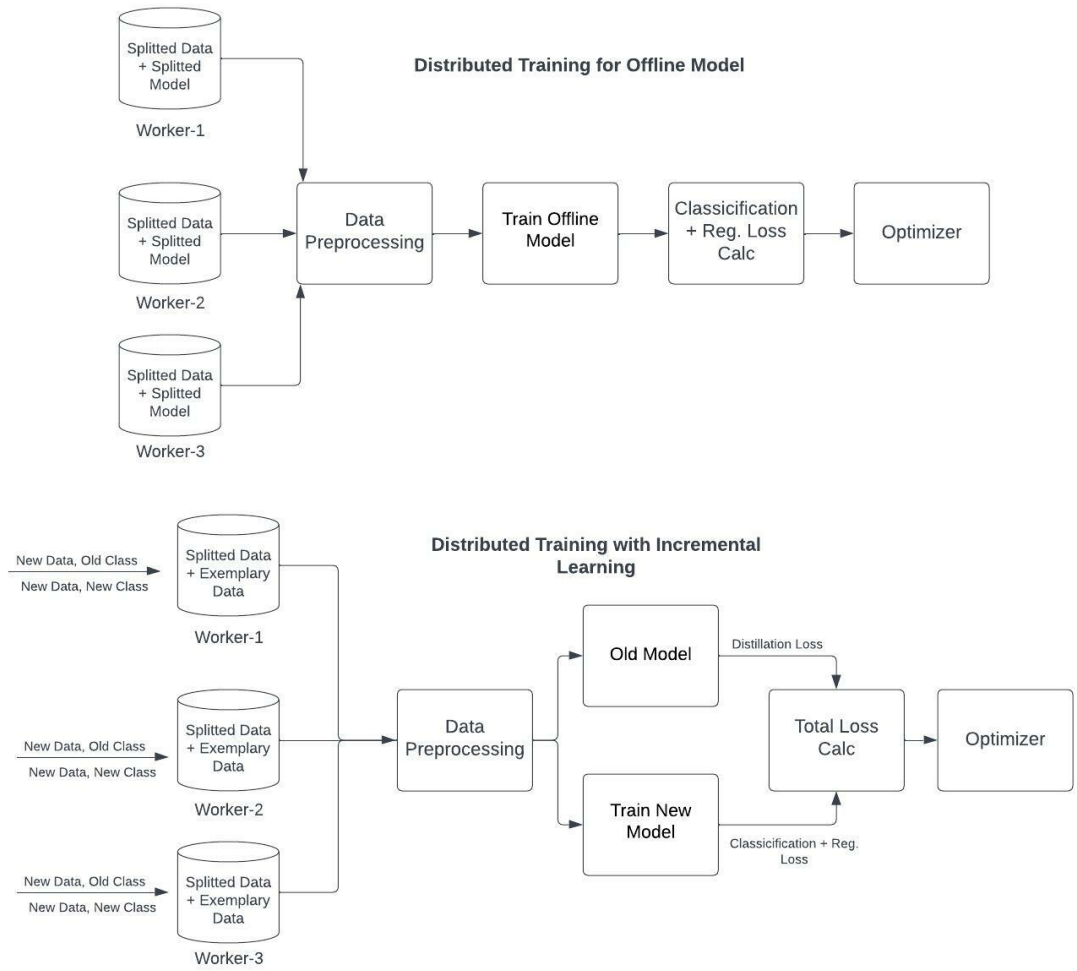


Figure 3.1: Distributed ML Workflow with Offline Training (top figure) and Incremental Learning (bottom figure).

has been elaborated on in section 2.3. Rest of the steps such as Data Pre-processing, training, loss calculation and optimization are typical machine learning workflow. These steps have been elaborated in Algorithm 1.

To increase the performance of the offline model, Incremental Learning(IL) approach is followed. In IL, the new training data (from both old and new classes) are used for re-training and fine-tuning the model. This Incremental Learning workflow is depicted in Figure 3.1 (bottom figure). Unlike typical distributed training approach, as a part of IL approach, we additionally calculate Distillation Loss from old model. All these steps have further been discussed in section 3.4

In this thesis, two datasets have been used, namely *Fruits* and *Filter*, as an application to fruit picking robot and garbage collector respectively. The *Fruits* dataset is a medium size dataset, consisting of 8 classes, 6360 images and 27188 bounding boxes. It is taken from commonly used public benchmark OpenImage [41] dataset. The *Filter* dataset is created by ourselves having only one class and 287 images with

292 annotations. Further details to the datasets are given in Section 3.7.

During all the stages of our methodology, we carefully consider reducing the computational cost and memory requirement since embedded devices are resource constrained. This has been proved by our experiment shown in chapter 4. The methodological pipeline described above and the experiment exhibited in chapter 4, helped us to accomplish our first three research goals mentioned in section 5.1.

Other important issues associated with the methodology such as Small Object Detection Problem and Class Imbalance Problem have been discussed briefly in subsection 3.5.1 and subsection 3.5.2 respectively.

3.2 Pre-trained Object Detection Model using Transfer Learning

We aim to build a parameter server based distributed system, where workers train an object detection model capable of real-time detection on embedded devices. We choose 3 state-of-the-art single stage detectors, namely MobileNet-SSD[31], SqueezeNet-SSD[33] and RetinaNet[52]. Among them, the former 2 models are popular choices that have been designed particularly for mobile and embedded devices. These architectures combine the SSD-300 [56] Single-Shot MultiBox Detector with a MobileNet and SqueezeNet as backbone. The following Figure 3.2 shows the combined architecture of MobileNet and SSD[21]. As mentioned in chapter 1, all 3 models have been trained for incremental learning for new data of known classes. However, for adding new class called **Mango** in *Fruits* dataset, only RetinaNet[52] model has been trained incrementally. The reason behind training RetinaNet only is to test exclusively whether the model is capable of learning new class without forgetting old class by using our distributed incremental learning approach. Further exploration on class incremental training is kept as future work. The pretrained RetinaNet model has ResNet-50 as backbone and trained on ImageNet dataset[17]. It is acquired from https://github.com/fizyr/keras-retinanet and retrained it with *Fruits* and *Filter* datasets using Transfer Learning approach.

It is important to mention here that, among the 4 quadrants shown in Figure 2.5, our case belongs to the Quadrant 3, since *Fruits* and *Filter* datasets are quite different from the ImageNet[17] dataset (which is used for pre-training) and are small or medium in size. Therefore, for MobileNet-SSD[31] model as an example, we train only the SSD, classification and regression layers and keep MobileNet base layers frozen.

3.3 Algorithm: Distributed Training

After discussing the workflow of Parameter Server Strategy shown in Figure 3.1, let us define an algorithm for the Distributed Training.

The Algorithm 1 has been implemented using Distributed Remote Procedure Call (RPC) Framework of PyTorch. The framework easily runs the remote functions (remote communication). These are crucial for distributed training. Its high level API

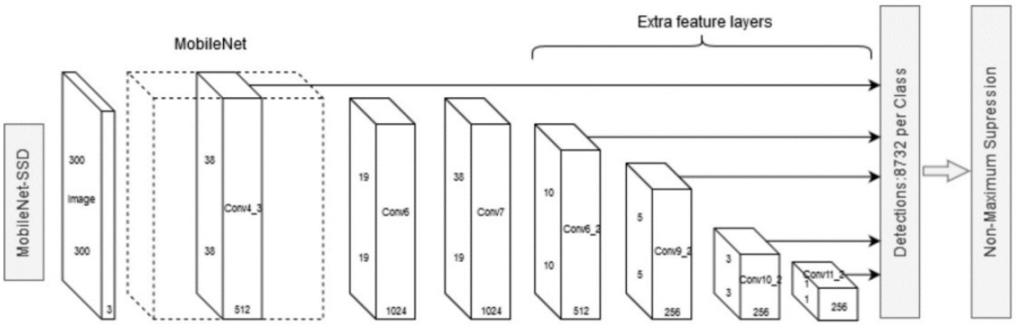


Figure 3.2: MobileNet-SSD Architecture [21]. Extra feature layers are parts of SSD.

creates and handles remote object reference. These object references make it easy to run distributed backward pass to calculate gradients. The distributed Autograd and Optimizer APIs run the distributed backward pass and update the weights accordingly. These two APIs even work with the model parallelism where the models are split across several machines. Overall, the framework provides the following 4 main groups of APIs:

1. **Remote Procedure Call (RPC)** provides function execution on the remote workers. It has 3 sub APIs:
   (a) rpc_sync(): Synchronous RPC, where further execution of codes are not allowed without the return value or reference.
   (b) rpc_async(): Asynchronous RPC, where functions can asynchronously wait for the future once the return value is required.
   (c) remote(): This API is asynchronous and capable of returning reference to the remote return value.
2. **Remote Reference (RRef)** holds a distributed shared reference to either local or remote object (or can be Tensor value) on a remote worker. This RRef can be shared with other workers. It maintains the reference counting in a transparent way.
3. **Distributed Autograd** generates local graph on all workers while forward passing. This graph is used in order to automatically calculate the gradients during the backward pass. Autograd automatically reaches to each worker during the backward pass. With Distributed Autograd functionality, user code

Algorithm 1 Distributed Parameter Server Strategy using Distributed RPC

Input: rank, world_size, master_port, master_address
Output: updated state dictionary of model

Task Scheduler:
function MAIN() ▷ to be spawn by server
    define model ▷ model definition is done in section 3.2
    define forward pass
    Split model to all workers
    Split data to all workers
    for $t \leftarrow 0$ to T **do**
        assign process p to each worker t
        issue WORKER(t) to all workers

Parameter Server:
function PARAMETERSERVER()
    initialize server through RPC
    set $forward()$ function for worker
    aggregate gradients through GET_DISTRIBUTED_GRADIENTS()
    shutdown rpc

function GET_GLOBAL_PARAM_REF()
    for *param* in model.parameters() **do**
        $param\_rrefs \leftarrow [rpc.RRef(param)]$
    return $param\_rrefs$

function GET_DISTRIBUTED_GRADIENTS(cid)
    $grads \leftarrow distributed\_autograd.get\_gradients(cid)$
    return $grads$

Worker w = 1,....,m:
function WORKER(t)
    initialize worker through RPC
    load a part of training data
    load a part of model
    $param\_rrefs \leftarrow GET\_GLOBAL\_PARAM\_REF()$ ▷ download the latest state dictionary of the model
    $dist\_opt \leftarrow DistributedOptimizer(SGD, param\_rrefs, lr)$
    for $i, data, target$ iterate through training data **do**
        with Distributed Autograd Context as cid:
        $model\_output \leftarrow net(data)$
        $loss \leftarrow LossFunc(model\_output, target)$
        $distributed\_autograd.backward(cid, [loss])$
        PARAMETERSERVER.GET_DISTRIBUTED_GRADIENTS(cid)
        $dis\_opt.step(cid)$
    shutdown rpc

does not have to think about how to propagate gradients across RPC boundaries or in what sequence the local autograd engines should be started. This can get very problematic when the forward pass contains nested and inter-dependent RPC operations.

4. **Distributed Optimizer** provides optimizers such as SGD(), Adam() etc. An optimizer is required for each worker. Because the parameter RRefs and gradients are dispersed across different workers during the forward and backward passes. The Distributed Optimizer combines all of the local optimizers from different workers into a single entity.

After discussing the 4 basic APIs of Distributed RPC framework, the next 2 subsections discuss the implementation of Parameter Server Class and Worker Class as well as their methods mentioned in Algorithm 1.

In Algorithm 1, there are four input parameters. The parameter *world_size* stands for total number of workers participating in training, including parameter server(s). The *rank* parameter indicates a unique *rank* for each individual worker. A rank is a unique integer starts from 0 and ends at *world_size-1*. It is used to identify a worker in an RPC boundary. For our case, the *world_size* is 4, where we use 3 workers and 1 parameter server. Hence, rank-0 belongs to single parameter server to run. Rank-1, rank-2 and rank-3 are used to identify worker-1, worker-2 and worker-3 respectively. Other parameters such as *master_address* and *master_port* are used to recognize where the rank-0 (parameter server) process is running and each worker uses these two parameters (master address and port) to run itself by connecting to the parameter server.

For model parallel approach stated in line 5 of the Algorithm 1, Appendix A shows how we split the MobileNet-SSD layers into 3 different GPUs. It is challenging to decide how many layers of a model should be placed in each GPU.

3.3.1 Parameter Server Class

Line 14 of Algorithm 1 describes one of the most important tasks of ParameterServer class, that is the accumulation of distributed or scattered gradients. Parameter Server calculates average of accumulated gradients from all the workers. The difference between Federated Learning (a privacy preserving distributed training approach) and our approach is that the gradients are not encrypted in our case while exchanging with workers. The function GET_DISTRIBUTED_GRADIENTS in line 20 accomplished the gradient accumulation task. This function used the Distributed Autograd of Pytorch API. Distributed Autograd creates *context_id* (an integer autograd context id for which the gradients should be retrieved) for each gradient as well as a graph while forward passing. Therefore, these *context_id* and generated graph can be used for backward pass efficiently. In addition to that, Parameter Server holds the remote object reference of all the parameters of the model. It keeps track of each worker's number of training data, epoch, training time etc.

Another function called GET_GLOBAL_PARAM_REF in line 16 of Algorithm 1 is used to iterate through model parameters and wrap them as local *RRef*. Work-

ers use RPC to call this function, which returns a list of the parameters that need to be optimized. This must be provided as input to the distributed optimizer. Because distributed optimizer needs a list of RRefs for each parameter that has to be optimized.

3.3.2 Worker Class

Each worker runs the same training loop through its entire training data for 250 epochs as depicted in Lines 30 to 36. Apart from implementing MobileNet-SSD, 2 other state-of-the art one-stage object detection models, such as SqueezeNet-SSD and RetinaNet are trained to compare the calculated loss, accuracy and training time while training in a distributed fashion. Multibox loss (classification loss + localization loss) function is used for all models except RetinaNet[52]. Focal loss is incorporated with Multibox loss for RetinaNet model.

3.4 Incorporating Online Incremental Learning Approach

Incorporating online incremental learning with distributed training is the next crucial step to achieve better accuracy for object detection models. In this approach, workers no longer require to store the training data. Data are discarded after the whole training session ends. In addition to that, this approach has an advantage of continuous learning in comparison to static model. According to [37], online learning belongs to 2 scenarios given below. In this study, both scenarios have been implemented separately. When scenario 2 comes (adding new class), we do not incrementally train the model with new data for known class. Similarly, when scenario 1 arrives, we do not add new class header in order to train with new class. We will explore handling both scenarios together in our future work.

1. *Continuous learning of known classes with new data*: refers to Incremental Learning (IL). In IL, weight updates/changes while training for new data is carefully handled by restricting large changes in gradients (known as Weight Constraint). A Weight Constraint is a network update that verifies the weights' size. The weights are rescaled if the size is greater than a predetermined limit or falls within a range. This is achieved by using Dropout[76] and Max-norm regularization. Another way to constraint the weight change is to penalize the large weights. However, penalizing large weights do not force weights to be small. Hence, as suggested by [76] and [29], we set $max\_norm = 3$ while training. We applied online learning in this work by adding more training data splitting into different sessions as done in [7].

   Parameter Server stores the trained model checkpoint. Therefore, online training can resume from last checkpoint by loading the model. In addition to that, Knowledge Distillation loss function is used to prevent Catastrophic Forgetting. This step is described for the 2nd scenario.

2. *Incremental Learning with new classes*: We also consider this case in this study by adding one new class called **Mango** in *Fruits* dataset only. For adding n new classes, along with keeping a copy of old model *N'*, a new model N is created by extending the old model's classification headers. The classification headers should have n more neurons to be able to classify n more classes. For example, for SSD[56], out_channels=6 * num_classes, since there are 6 aspect ratios for one feature map. Weights of the newly added neurons are set randomly. However, weights of new model are set from the corresponding weights of the old model.

   For incremental learning approach, training goes on only with the new data for old or new classes or both. To handle the Catastrophic Forgetting problem, we conform to the idea of Learning without Forgetting (LwF)[51], which uses Knowledge Distillation [30] technique. Knowledge Distillation prevents Catastrophic Forgetting by extracting knowledge from a complex pre-trained neural network called Teacher (in our case, the old model), to train a relatively more compact student model (new model). Typically, this extraction is carried out by making the student model mimic the teacher's responses against the training data. Mathematically, this technique optimizes both the cross-entropy loss on new classes and distillation loss on old classes. This discourages the changes for the output of old classes. In other words, combination of both loss functions make sure that for the same input image, the output for the old classes using the new model is same to the output of the old model.

   In addition to conserve the old model's classification ability, it is important for new model to learn the bounding box label from the old model. Thus the idea of knowledge distillation for object detection is to stimulate both the outputs from classification header and bounding box header of old model to approximate for new model.

   To summarize the above analysis, the loss function for incremental object detection model, suppresses two changes in order to reduce the Catastrophic Forgetting and they are:

   (a) Suppress changes in classification output of the old model

   (b) Suppress changes in bounding box output of the identified objects

   Hence, the goal is to incrementally train an object detection model N, capable of detecting both $m + n$ classes. Here, assume that an old object detection model *N'* is trained on m classes and n new classes is to be added with a training dataset D_{new}. Figure 3.3 shows how the incremental learning method works for new class.

   The loss function defined in equation 2.6 considers the classification loss (confidence loss) and regression loss (bounding box loss or localization loss) for an object detection model. The new loss function for incremental object detection adds two new loss terms, *distilled classification loss* and *distilled regression loss*

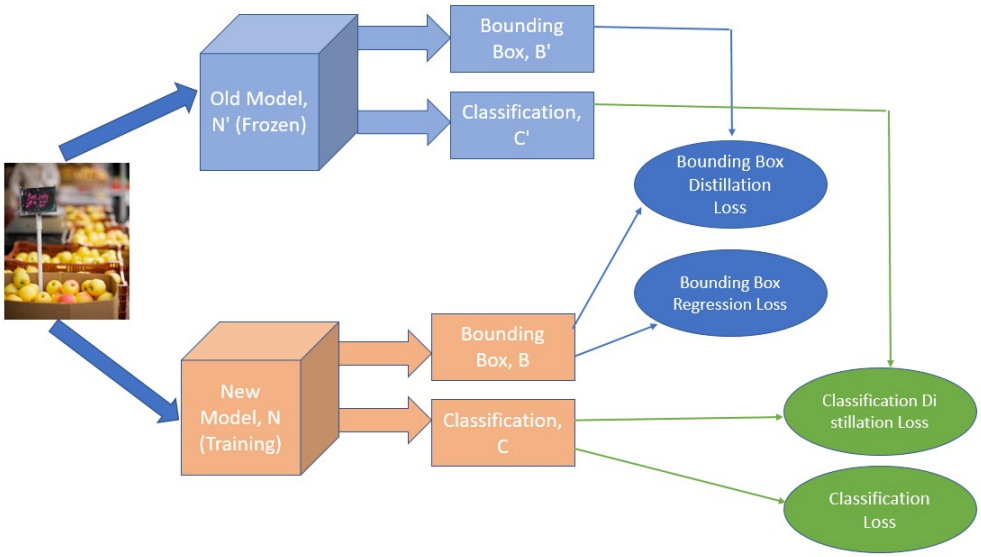


Figure 3.3: Incremental Learning for n new class [43]

from old model's output, and is defined as follows:

$$Loss_{new} = Loss_{class}(Y_n, \hat{Y_n})^1 + Loss_{bbox}(B_n, \hat{B_n}) + Loss_{dist\_class}(Y_0, \hat{Y_0}) + Loss_{dist\_bbox}(B_0, \hat{B_0}) \tag{3.1}$$

where, Y_n = ground truth classification label,

$\hat{Y_n}$ = classification output of new model with n new class,

B_n = ground truth bounding box,

$\hat{B_n}$ = predicted bounding box output of new model,

Y_0 = given the new data, output of old model

$\hat{Y_0}$ = output of new model for old class,

B_0 = predicted bounding box of old model for old class, but with new data,

$\hat{B_0}$ = predicted bounding box of new model for old class

Therefore, the two newly added loss functions are defined as follows:

$$Loss_{dist\_class}(Y_0, \hat{Y_0}) = \frac{1}{m}\sum_{i=1}^{m}(Y_0^i - \hat{Y_0^i})^2 \tag{3.2}$$

and

$$Loss_{dist\_bbox}(B_0, \hat{B_0}) = \sum_{j \in \{x,y,w,h\}} smooth_{L1}(B_0^j - \hat{B_0^j}) \tag{3.3}$$

[1]For RetinaNet[52] model, this Loss function is replaced with Focal loss function

Exemplary Dataset Construction: It is worth to note that no training data from old classes are stored and only new data from old or new classes are used while training. This online incremental learning approach saves a lot of space as well as reduce training time to a great extent. Along with applying Knowledge Distillation, a small exemplary database is maintained to make the model more robust. The exemplary dataset contains few data obtained from old classes through augmentation. This small dataset supplies at least some information about old classes during the training phase and eventually helps the object detection model to gain better accuracy in general. As claimed by [43], a few exemplar data from each class exhibits similar accuracy rather than using all data of old classes. [68] picks images based on the average feature vector of exemplar in such a way that the average feature vector will be nearest to the class average value. This approach adds overhead to the overall training time. Unlike [68], data are chosen randomly for each class for our case. Our experiments exhibit that the randomly chosen exemplary data from each class help to obtain better accuracy. As number of new classes grows incrementally, size of exemplary dataset can be increased. In that case, boundary can be set for exemplary dataset size.

3.5 Handling Miscellaneous Issues

3.5.1 Small Object Detection Problem

Small object detection problem comes from the fact that *Filter* dataset contains small objects. The definition of small objects are not clearly defined in literature. The reasons is, researchers interpret small objects based on different datasets, rather than considering only the size of bounding boxes of objects. [55] defines objects as small if they fill less than 20% of entire image in traffic signs dataset. [9] considers objects as small, if their overlapping area between bounding box and the image is in between 0.08% to 0.58% from 16 by 16 to 42 by 42 pixel image.

The small object detection problem is still not completely solved. Hence, we conform the following strategies in order to tackle the small object detection problem:

- Increasing image capture resolution
- Increasing model's input image resolution
- Tiling input images
- Using Focal Loss function

3.5.2 Class Imbalance Problem

[64] extensively discusses the class imbalance problem in object detection and provides critical views on the solutions. Most commonly known imbalance problem in object detection is foreground-to-background imbalance. It occurs when there is an extreme

inequalities between the number of positive and negative examples. Generally, there are few positive examples where model can extract a lot of negative examples. If this problem is not taken care of, it adversely affects the accuracy of the object detection model as well leads to slow convergence. In case of SSD[56], this problem has been handled by taking highest Confidence Loss negative examples and maintaining the ratio of negative to positive class by maximum 3:1. RetinaNet [52] model uses Focal Loss function in order to deal with class imbalance. It penalizes the contribution of easy examples and gives more emphasize on hard examples using a scaling factor.

3.6 Real-time Inference with TensorRT Engine[63]

From the view point of practical implication, it is important to build a trained object detection model capable of real-time detection. To do that following steps have been taken. Figure 3.4 also illustrates the workflow for real-time inference.

1. **Storing and Loading Model Checkpoint:** After each epoch of training, if the model validation loss is lower than the previous epoch, the model's checkpoint is stored. In this way, model validation is done by storing the best model checkpoint. Later on after the training, the best model is loaded and then is converted to intermediate ONNX (Open Neural Network Exchange) format.

2. **Conversion of Trained Model to ONNX:** PyTorch provides an easy API to export a saved model to ONNX format.

3. **ONNX to TensorRT[63] Conversion:** TensorRT[63] is a Software Development Kit (SDK) that enables high-performance machine learning inference. It can only import a trained model via the ONNX interchange format. It provides efficient and quick inference of an already trained ML model on NVIDIA hardware. *detectnet.py* [21] script has been used to load the ONNX model and convert it for real-time inference.

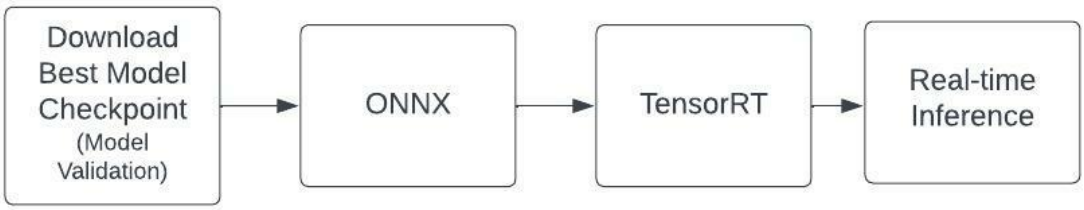


Figure 3.4: Steps for Real-time Inference

3.7 Dataset, Experimental Setup and Data Preprocessing

3.7.1 Dataset

Fruits dataset and it's labels are collected from https://opensource.google/projects/open-images-dataset using a script. For *Filter* dataset, data have been collected by ourselves and labelling has been done using a tool from https://github.com/dusty-nv/jetson-inference/blob/dev/docs/pytorch-collect-detection.md. All data and labels of both datasets are in Pascal VOC format. For online incremental training, initially, we collected the labels from inference step. Frames having object and Intersection Over Union (IoU) > 0.85, are chosen for further incremental training. However, our experiment found that the accuracy of the model is not as good as for real time inference requirement. Therefore, we collected the label of *Fruits* dataset from online and manually annotated the images of *Filter* dataset. We decided to keep this data label automation and further investigation on it as future works.

Table 3.1 describes the data distribution of each class and their annotation for both *Fruits* and *Filter* datasets:

Dataset	Class	Image Count	Annotation Count
Fruits	Apple	1084	3622
	Banana	747	1574
	Grape	846	2560
	Orange	1156	6186
	Pear	250	757
	Pineapple	328	534
	Strawberry	1480	7553
	Watermelon	469	753
	Mango	185	248
Cigarette Filter	Cig Filter	287	292

Table 3.1: *Fruits* and *Filter* Dataset Description (Mango class is used for incremental learning approach)

For each worker, parameter server creates train, validation and test dataset and their annotation with 80:5:15 ratio respectively. By default, TensorPipe backend protocol (which is TCP-based transport) was used for gradient transfer between workers and server. It is integrated with Distributed RPC framework (such as by calling *init_rpc()*) of PyTorch. It has an added advantage of being asynchronous unlike Gloo (Gloo is a distributed backend package provided by PyTorch for distributed CPU training). It is capable of handling large amount of transfer concurrently without interrupting each other. It also can manage different workers with different speeds, which is obvious in reality.

3.7.2 Experimental Setup

We conduct experiments to compare training time per epoch and mean average precision (mAP) for distributed and non-distributed setup. We compare results both for online and batch learning, as well as adding new classes for incremental learning. All the experiments have been carried out using both the Data Parallelism and Model Parallelism.

To measure the accuracy of an object detection model, a metric is used called mean average precision (mAP). mAP is the average of AP. Alternatively, AP is averaged over all classes. According to MS COCO [53], there is no distinction between AP and mAP. In order to understand mAP, it is important to understand how to calculate Intersection Over Union (IoU). IoU is a measurement based on Jaccard Index that considers the overlapping area between model's predicted bounding box and ground truth bounding box. Based on the IoU value, it is determined whether a detection is True or False. A predefined IoU threshold is used (for our case, IoU threshold = 0.5), where $IoU >= 0.5$ indicates a valid detection, otherwise not. The formula for IoU is:

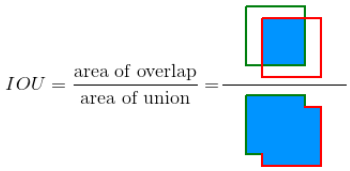


Figure 3.5: Intersection Over Union (IoU)

IoU is calculated for each image. Then detection is considered having $IoU >= 0.5$ and AP is calculated for each class across all images. Finally, mAP is calculated by using mean of the APs over all classes. We use the repo https://github.com/rafaelpadilla/Object-Detection-Metrics to calculate the accuracy of all models.

Measuring training time is straight forward, which is the time (in minute unit) difference between begin and end of an epoch.

Our cluster(3-GPU system) consists of 3 Jetson Nano devices each of which has 472 GFLOPS of compute performance with a quad-core 64-bit ARM CPU, a 128-core integrated NVIDIA GPU, 4GB LPDDR4 memory, low-power with 5W/10W power modes, 5V DC input and run on Linux4Tegra, based on Ubuntu 18.04. These 3 Jetson Nano devices were connected to a small wireless area network with a TP-Link AC1750 WIFI router at an average speed of 450 mbps for 2.4 GHz band. An Edimax 2-in-1 WiFi and Bluetooth 4.0 Adapter was also used for each Jetson Nano. We use one Parameter Server having configuration of Intel Core i5 10th Gen, 3.2 GHz processor, 8GB DDR4-3200 RAM and run on Ubuntu 18.04.

For model selection, three state-of-the-art one-stage object detection models were used to measure the performance of the distributed incremental learning approach. Following Table 3.2 provides a summary of our experiments:

Additionally, 1-GPU based laptop was used to train a baseline model Vanilla-SSD(VGG16-SSD[73]) in order to compare the performance of 1, 2, 3-GPU based distributed systems for both the datasets. The configuration of the laptop is Intel Core

Model	MobileNet-SSD	SqueezeNet-SSD	RetinaNet
Loss Func	Multibox		Focal Loss
Optimizer	SGD		
Epoch	250 (160 for *Filter* Dataset)		
lr	0.001		
Dropout	0.20		
Batch size	4 (3 for online training)		
Evaluation Metric	Training Loss, Validation Loss, Classification Loss, Regression Loss, Accuracy and Training Time		

Table 3.2: Summary of Model Training Parameter and Experiment

i7 11th Gen, 16GB DDR4 RAM and NVIDIA GeForce RTX 3050Ti. No strategies such as Data Parallelism, Model Parallelism, using Examplary Dataset etc. had been applied to train the Vanilla-SSD.

3.7.3 Data Pre-processing

Different data pre-processing steps including *rescaling* of the input images and transformation functions, are carried out before training starts. They are:

- **ConvertFromInts():** to convert the integer image pixel values to float.
- **PhotometricDistort():** perform distortion of images by orderly applying some functions such as changing contrast, brightness, saturation, hue, converting colour from RGB to HSV and vice-versa.
- **Expand():** expands the images' height and width maintaining the aspect ratio.
- **RandomSampleCrop():** randomly crops the expanded images with at least one ground truth bounding boxes.
- **RandomMirror():** transforms an image to its left-right flip.
- **Resize():** resizes an input image from lower dimension to higher and vice-versa.
- **SubtractMeans():** to normalize image pixel values.

Chapter 4

Results and Discussion

4.1 Distributed vs Non-Distributed Training

Using both the data parallel and model parallel approaches for *Fruits* and *Filter* datasets, training loss, validation loss, training time and accuracy have been compared between distributed training (2-GPU and 3-GPU system) and non-distributed training (1-GPU system) for all 3 object detection models. Additionally, loss and accuracy of 3 object detection models have been compared with that of Vanilla-SSD model.

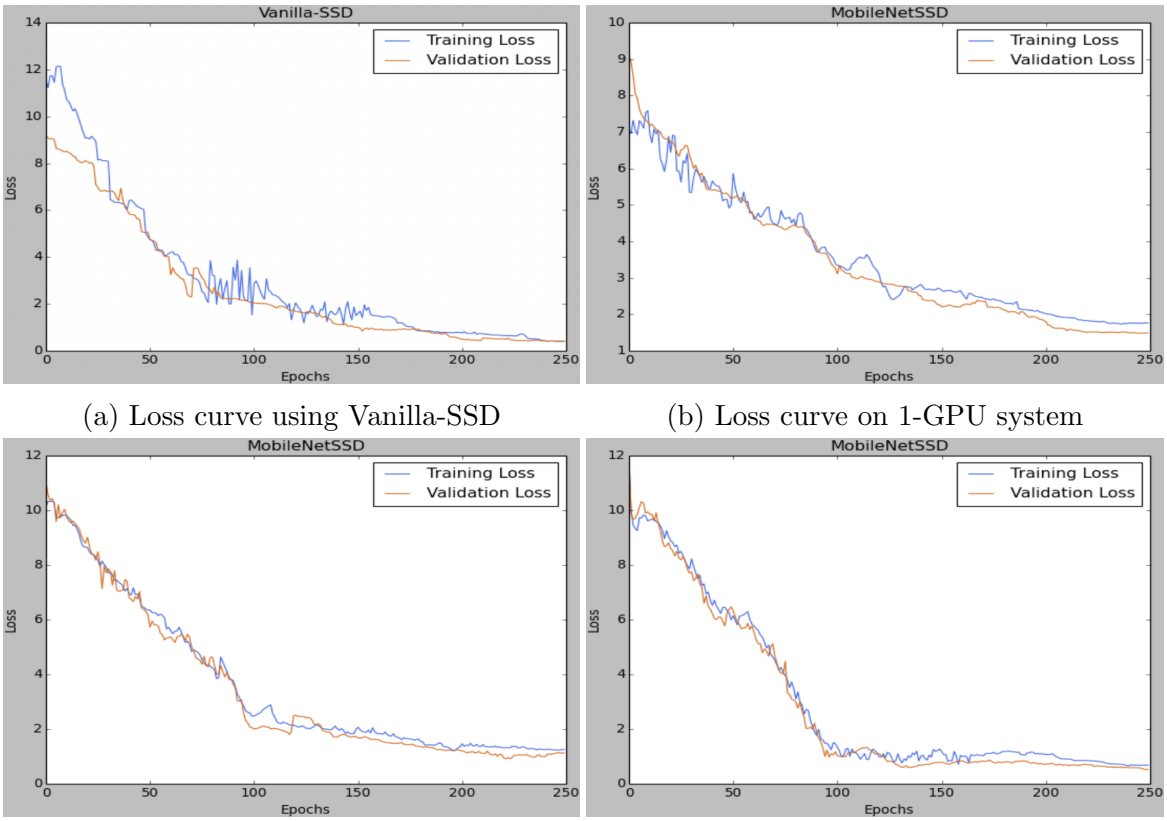
4.1.1 Loss Comparison/Convergence

It is common practice to record training loss and validation loss over time in order to measure how the model is learning. The training loss tells how well the model learns the training data, whereas validation loss says how well the model can fit the new data. For object detection problem, training loss is the summation of classification loss and regression loss for bounding box.

Figure 4.1, Figure 4.2 and Figure 4.3 show the training and validation loss curves while training for 250 epochs using the *Fruits* dataset. 3 different object detection models, MobileNet-SSD[31], SqueezeNet-SSD[33] and RetinaNet[52] were trained on 1-GPU, 2-GPU and 3-GPU system and VGG16-SSD was trained on 1-GPU system. All the training loss curves decrease over time to a point of stability. Also, all the validation loss curves decrease to a point of stability and have a small gap with the training loss. Point of stability and small gap between training and validation loss indicate a good fit of the data. Comparing the loss curve of Vanilla-SSD with the loss curve of MobileNet-SSD, SqueezeNet-SSD and RetinaNet, we can say that the 3 models are trained enough to produce good accuracy for real time inference.

From the dataset representative perspective, both the training and validation datasets represented all the 3 models (All through the thesis, by 3 models we mean MobileNet-SSD, SqueezeNet-SSD and RetinaNet). Alternatively, it can be said that all 3 models successfully learnt the training datasets and have been well validated against validation datasets. Firstly because, there are no large noticeable gaps between both loss curves. Secondly, none the curves have large fluctuation or noise.

However, Figure 4.2b and Figure 4.3b exhibit some noises in validation curve comparing with the training curve.



(a) Loss curve using Vanilla-SSD

(b) Loss curve on 1-GPU system

(c) Loss curve on 2-GPU system

(d) Loss curve on 3-GPU system

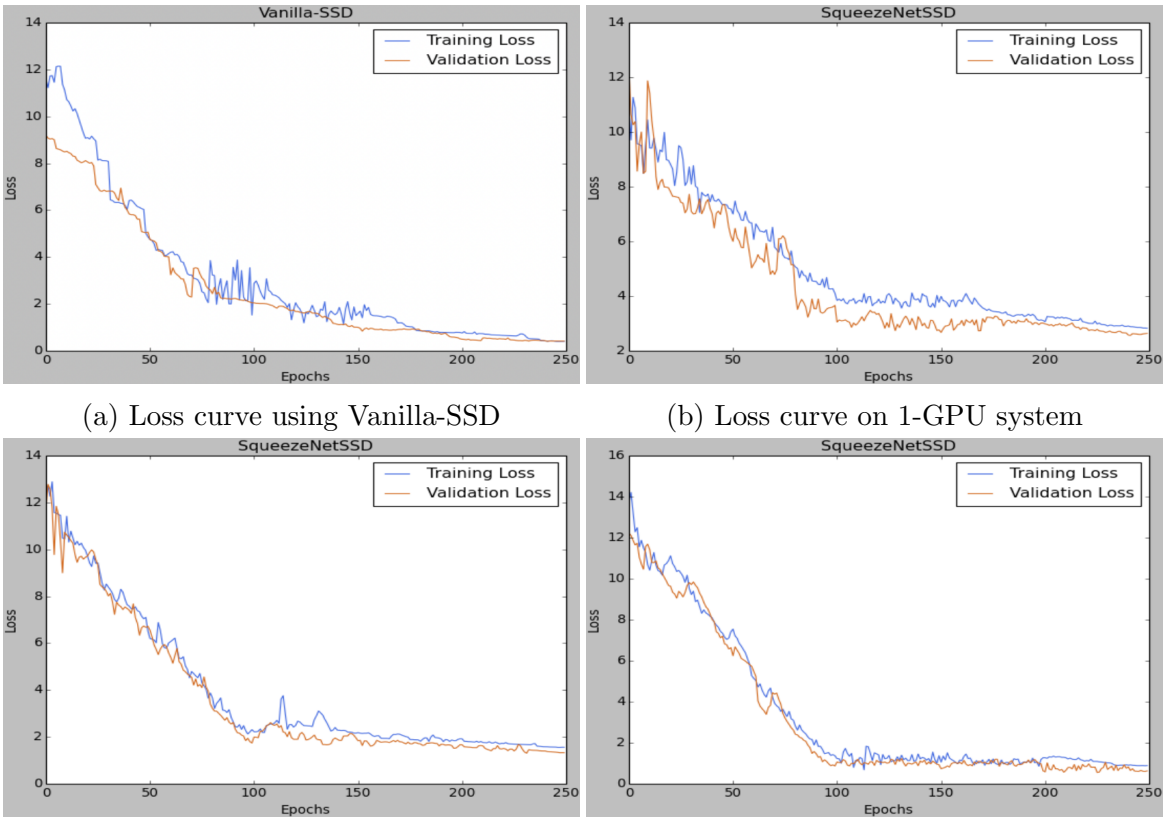
Figure 4.1: Training and Validation Loss Comparison for MobileNet-SSD[31] using Different GPU System and Vanilla-SSD (VGG16-SSD); Dataset: Fruits; Epochs: 250

It is worth to note that for all 3 models, comparing with distributed(2-GPU and 3-GPU system) and non-distributed training(1-GPU system), the former systems begin with larger loss values, which eventually converge to lower loss values than the latter one. An intuition can be, distributed systems initially take some training rounds to be stable in learning, due to asynchronous communication overheads between the GPU and server.

In comparison between 3 models' loss functions, both loss curves of MobileNet-SSD[31] model reside nearby over the training period. Other 2 models' loss curves are not as close as MobileNet-SSD[31].

4.1.2 Training Time Comparison

Training time has been recorded for each epoch in minute unit. Figure 4.4 exhibits the training time required for 250 epochs on various parallelization schemes using the *Fruits* dataset. The experiments were run for all 3 models (we did not compare training time result with Vanilla-SSD, as the configuration of the machine used to



(a) Loss curve using Vanilla-SSD

(b) Loss curve on 1-GPU system

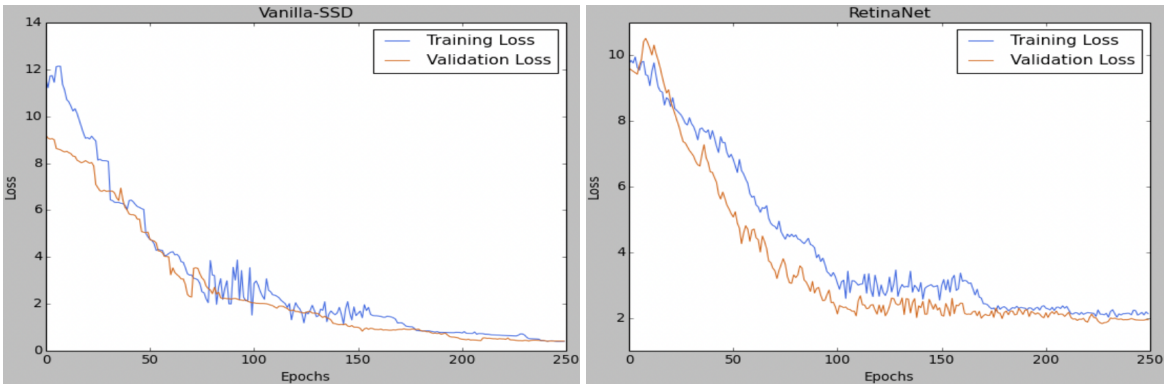
(c) Loss curve on 2-GPU system

(d) Loss curve on 3-GPU system

Figure 4.2: Training and Validation Loss Comparison for SqueezeNet-SSD[33] using Different GPU System and Vanilla-SSD (VGG16-SSD); Dataset: Fruits; Epochs: 250

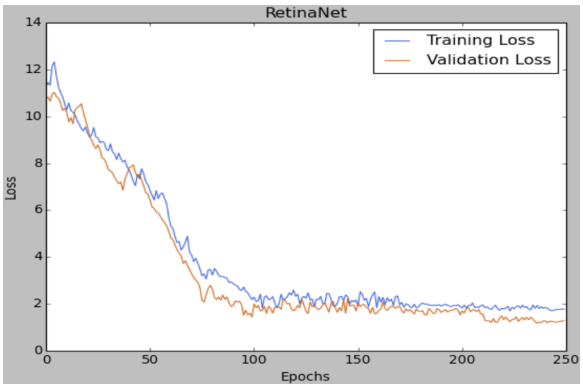
train the model is much higher than Jetson Nano, resulting much lower training time than 3-GPU system). The same experiments were conducted using the *Filter* dataset for 160 epochs and showed in Figure 4.5.

- **Fruits Dataset:** Using *Fruits* dataset, the average training time per epoch for MobileNet-SSD on single-GPU, 2-GPU and 3-GPU systems was 15.753 minutes, 8.177 minutes and 5.170 minutes respectively. From these results it is evident that, on average, the training time has been reduced by 48.09% using 2-GPU system and 67.18% using 3-GPU system comparing with single GPU. Alternately, it can be said that training speed has been increased almost double using 2-GPUs and more than 3-times using 3-GPUs systems comparing with 1-GPU. In comparison between 2 and 3-GPU system, the average training time per epoch from 2 to 3-GPU system has decreased by around 36%, which is because of adding GPUs incur some communications overhead. In case of SqueezeNet-SSD[33], Figure 4.4b says, the average training time using 2-GPU and 3-GPU system has been reduced by 45.26% and 67.25% respectively in comparison with single GPU system. Similarly, there was reduction in training time using distributed training system for RetinaNet model in contrast with non-distributed system. Experiment and Figure 4.4c show that in instance of

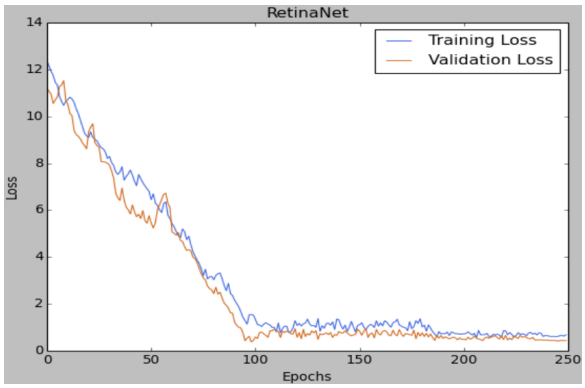


(a) Loss curve using Vanilla-SSD

(b) Loss curve on 1-GPU system



(c) Loss curve on 2-GPU system

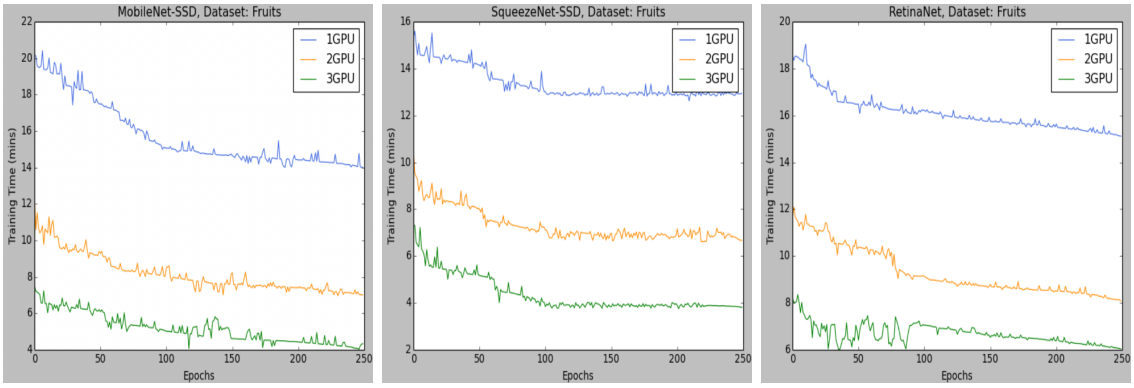


(d) Loss curve on 3-GPU system

Figure 4.3: Training and Validation Loss Comparison for RetinaNet[52] using Different GPU System and Vanilla-SSD (VGG16-SSD); Dataset: Fruits; Epochs: 250

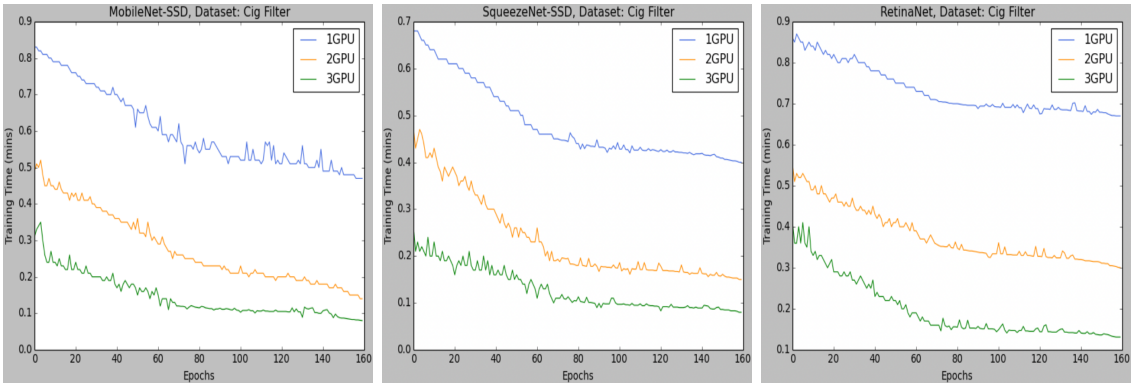
RetinaNet model, the average training time per epoch using 1, 2 and 3-GPU system were 16.13 minutes, 9.31 minutes and 6.68 minutes respectively. Hence the average training time has shrinked by 42.25% and 58.57% using 2-GPU and 3-GPU system respectively comparing with 1-GPU system. Among the 3 object detection models, MobileNet-SSD and SqueezeNet-SSD require less training time (around 5 mins and 4.3 mins respectively) than the RetinaNet model (around 6.7 mins) for distributed training using 3-GPUs.

- **Filter Dataset:** The results of training time (reduction rate) using different distributed systems shown in Figure 4.5a, Figure 4.5b and Figure 4.5c are quite similar to those results of the *Fruits* dataset. The average training time per epoch using 1, 2 and 3-GPU systems were 0.603, 0.279 and 0.146 minutes respectively for the MobileNet-SSD model. Comparing with single GPU, the mean training time for 2-GPU system has been decreased by 53.73%, whereas for a 3-GPU system, it was decreased by 75.78%. For SqueezeNet-SSD[33] model, it was reduced by 51.44% using 2-GPU and 73.55% using 3-GPU, contrasting with the performance of non-distributed system. Comparing with 1-GPU system shown in Figure 4.5c, while training RetinaNet[52] model, the mean



(a) Training Time for MobileNet-SSD (b) Training Time for SqueezeNet-SSD (c) Training Time for RetinaNet

Figure 4.4: Training Time Comparison for MobileNet-SSD, SqueezeNet-SSD and RetinaNet using 1, 2, 3-GPU System; Dataset: Fruits; Epochs: 250



(a) Training Time for MobileNet-SSD (b) Training Time for SqueezeNet-SSD (c) Training Time for RetinaNet

Figure 4.5: Training Time Comparison for MobileNet-SSD, SqueezeNet-SSD and RetinaNet using 1, 2, 3-GPU System; Dataset: Filter; Epochs: 160

training time were reduced by 48.29% using 2-GPU and 72.98% using 3-GPU. Overall, MobileNet-SSD[31] outperforms other 2 models in terms of reducing average training time per epoch for the *Filter* dataset.

From Figure 4.4 and Figure 4.5, it is observable that all the training time curves for 3 models using different parallelisms have downward direction over the course of time. This means that the training time per epoch reduces slowly over the 250 epochs time and 160 epochs time for *Fruits* and *Filter* dataset respectively.

The following table summarizes the average training time required and percentage of reduction comparing with single GPU for 3 models.

Data-set	Models	1-GPU	2-GPU		3-GPU	
		Avg. Time	Avg. Time	% reduction vs 1-GPU	Avg. Time	% reduction vs 1-GPU
Fruits	MobileNet -SSD	15.753	8.177	48.09	5.170	67.18
	SqueezeNet -SSD	13.349	7.307	45.26	4.371	67.25
	RetinaNet	16.126	9.312	42.25	6.681	58.57
Filter	MobileNet -SSD	0.603	0.279	53.73	0.146	75.78
	SqueezeNet -SSD	0.484	0.235	51.44	0.128	73.55
	RetinaNet	0.733	0.379	48.29	0.198	72.98

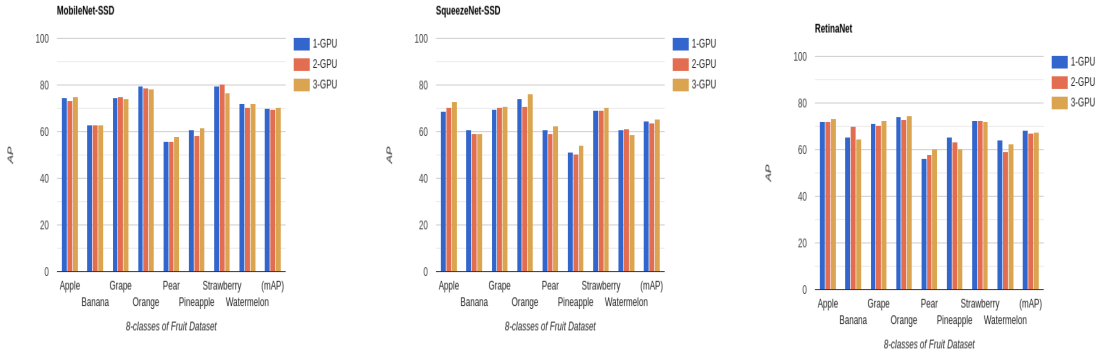
Table 4.1: Summary of Training Time for *Fruits* (250 epochs) and *Filter* (160 epochs) dataset

4.1.3 Accuracy Comparison

As it can be seen from the comparison of Training Time discussed in subsection 4.1.2, distributed training approach reduces training time to a great extent. However, from Table 4.2, Figure 4.6 and Figure 4.7, it can be seen that increasing number of GPUs does not exhibit remarkable contribute in increasing accuracy of the models for both the *Fruits* and *Filter* datasets. mAP of MobileNet-SSD and Squeezenet-SSD using 3-GPUs are slightly greater than the mAP using 1-GPU. Same observation can be found for RetinaNet using *Filter* dataset. Our intuition is that distributed training gives network chances to find more global optimum solution than non-distributed training. Comparing with Vanilla-SSD in Table 4.2, MobileNet-SSD achieves more accuracy for both the datasets. RetinaNet achieves similar accuracy to Vanilla-SSD in case of both datasets. Accuracy of SqueezeNet-SSD is lower than that of Vanilla-SSD. One intuition could be too much squeezing or reducing the depth of the feature map results into loosing accuracy as compared with other 3 models.

Data-set	Models	1-GPU	2-GPU	3-GPU
		mAP	mAP	mAP
Fruits	MobileNet-SSD	70.02	69.38	70.51
	SqueezeNet-SSD	64.41	63.79	65.57
	RetinaNet	68.19	67.26	67.53
	Vanilla-SSD	68.78		
Filter	MobileNet-SSD	74.37	74.52	73.24
	SqueezeNet-SSD	67.53	66.46	66.07
	RetinaNet	68.04	68.92	69.95
	Vanilla-SSD	70.03		

Table 4.2: Summary of mAP for *Fruits*(250 epochs) and *Filter*(160 epochs) dataset



(a) Accuracy for MobileNet - SSD

(b) Accuracy for SqueezeNet -SSD

(c) Accuracy for RetinaNet

Figure 4.6: Accuracy Comparison for MobileNet-SSD, SqueezeNet-SSD and RetinaNet using 1, 2, 3-GPU System; Dataset: Fruits; Epochs: 250

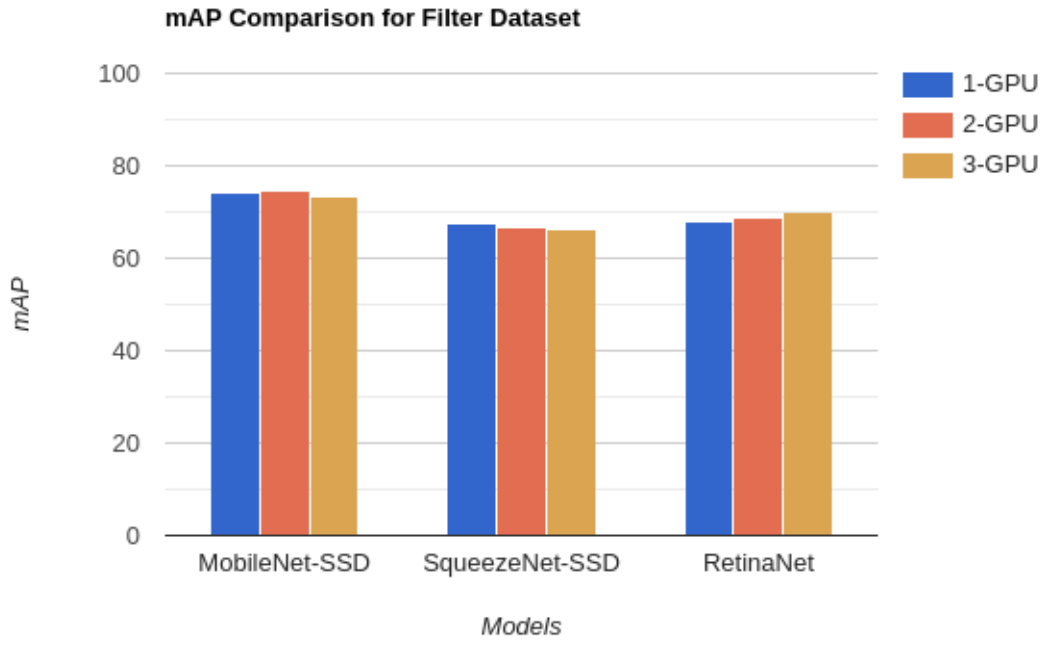


Figure 4.7: Accuracy Comparison using 1, 2, 3-GPU System; Dataset: Filter; Epochs: 160

The next Section 4.2 describes how the distributed system achieves better accuracy through Online Learning as oppose to Static Learning.

4.2 Batchwise Static Learning vs Online Learning

In this section, 1-GPU, 2-GPU and 3-GPU systems have been used for all the online incremental training experiments. The whole datasets *Fruits* and *Filter* have been divided into 4 sessions in order to incrementally train all the 3 object detection models. This can be summarized as follows:

Session	Training Image per Session (Fruits)	Epoch	No. of Exemplary Data	Training Image per Session (Filter)	Epoch	No. of Exemplary Data
S-1	1286	100	5% of 1286 = 65	61	70	5% of 61 = 3
S-2	1286	50	5% of 1286 = 65 + (65 from S-1) = 130	61	30	5% of 61 = 3 + (3 from S-1) = 6
S-3	1286	50	5% of 1286 = 65 + (130 from S-2) = 195	61	30	5% of 61 = 3 + (6 from S-2) = 9
S-4	1287	50	5% of 1287 = 65 + (195 from S-3) = 260	62	30	5% of 62 = 3 + (9 from S-3) = 12
Cumulative (Total)	5145	250	260	245	160	12

Table 4.3: Summary of splitted datasets into different sessions for Online Learning using 3-GPU system

From Table 4.3, total 5145 training images of *Fruits* dataset has been split into 4 sessions (same class distribution as used for distributed training), each session consists of 25% of training data. Remaining 6360 - 5145 = 1215 images are kept for validation and test set, in which same ratios like training set, are applied to split the validation and test sets. Identical approach is followed for *Filter* dataset. First session is trained for 100 epochs where rest of the sessions are trained for 50 epochs using *Fruits* dataset. 5% of training images are kept as exemplary data, obtained through data augmentation from the training set.

4.2.1 Accuracy Comparison

In Figure 4.8 and Figure 4.9, all 3 models show a progressive increase in accuracy after each session of training. Training data for each consecutive session is increasing through Exemplary dataset, which instinctively leverages to achieve better accuracy. Comparing with batch wise training, MobileNet-SSD and RetinaNet yield accuracy increase of 13.05% and 2.40% respectively for *Fruits* dataset. However, SqueezeNet-SSD achieve almost similar accuracy contrasting to batch training. Using *Filter*

dataset, accuracy has been increased by 6.51%, 7.67% and 16.09% respectively for all 3 models. Note that accuracy for Online and Batchwise Training (for both datasets) is shown using 3-GPUs only. Also, it should be noted that, for online and batch wise training, all the 3 models are trained for total 250 epochs (including 4 sessions for online training) using *Fruits* dataset and 160 epochs (including 4 sessions for online training) using *Filter* dataset. The next subsection discusses the training time required for online training.

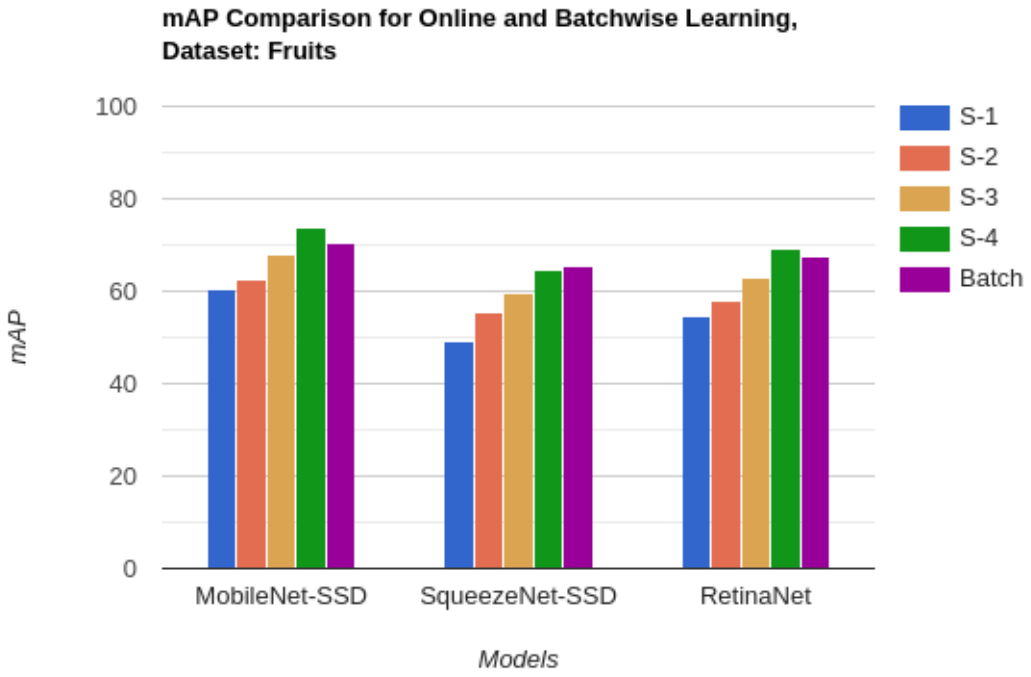


Figure 4.8: Accuracy Comparison between Online Training and Batchwise Training using 3-GPU System; Dataset: Fruits; Epochs: 250

4.2.2 Training Time Comparison

Figure 4.10 and Figure 4.11 show the training time per epoch in minutes required for 4 individual sessions, their cumulative training time and batch wise training for *Fruits* and *Filter* datasets using 1, 2, and 3-GPUs. Both the Figure 4.10 and Figure 4.11 have a common trend of increasing training time for each incremental training session. This is intuitive, since each incremental session has an additional number of exemplary images as well as Distillation loss function to compute.

In essence, it can be said that it is possible to achieve more accuracy using online training comparing with batch wise training. From our experiments, it can be inferred that up to 13% more accuracy can be achieved for a medium size dataset like *Fruits* and up to 16% more accuracy for a small dataset such as *Filter*. Online training incurs some additional training time comparing with batch training. Though, it is

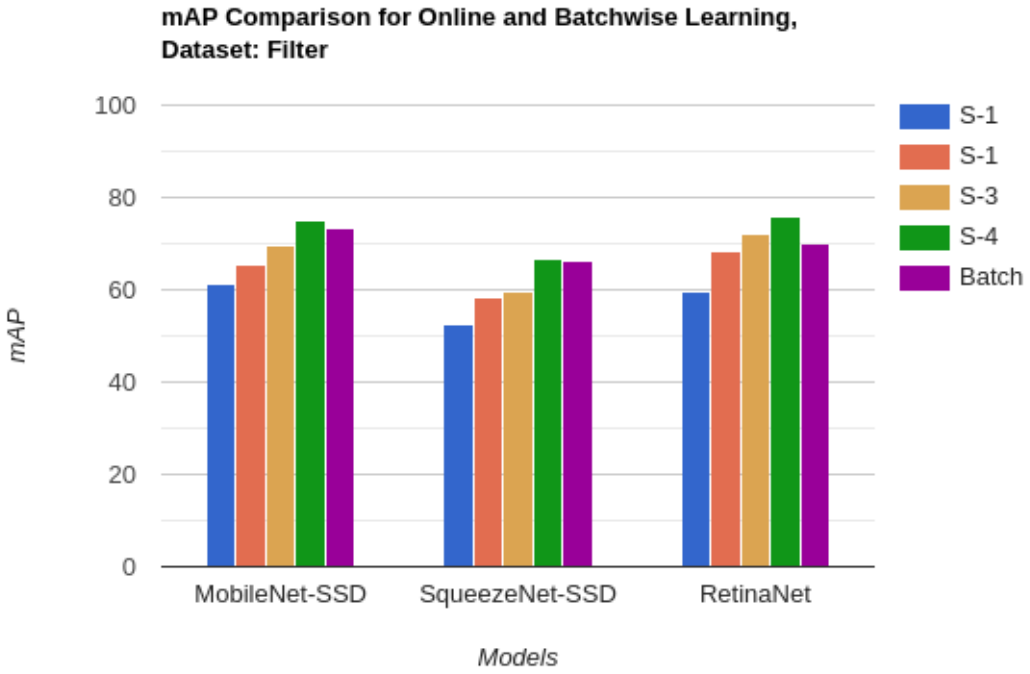
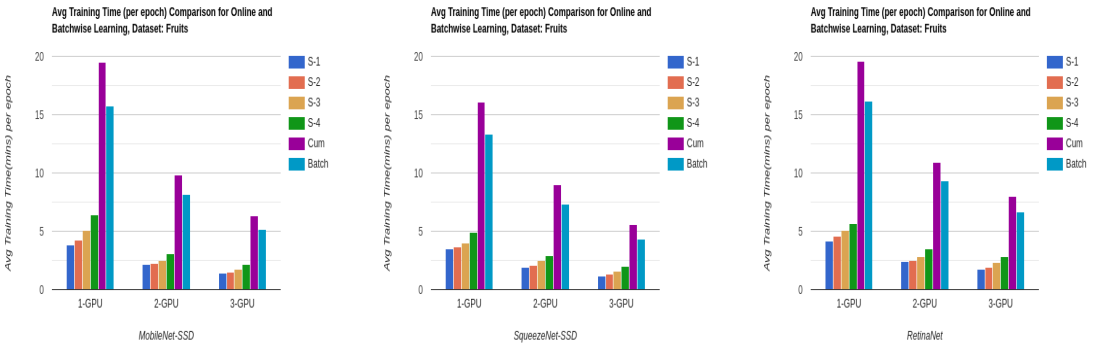


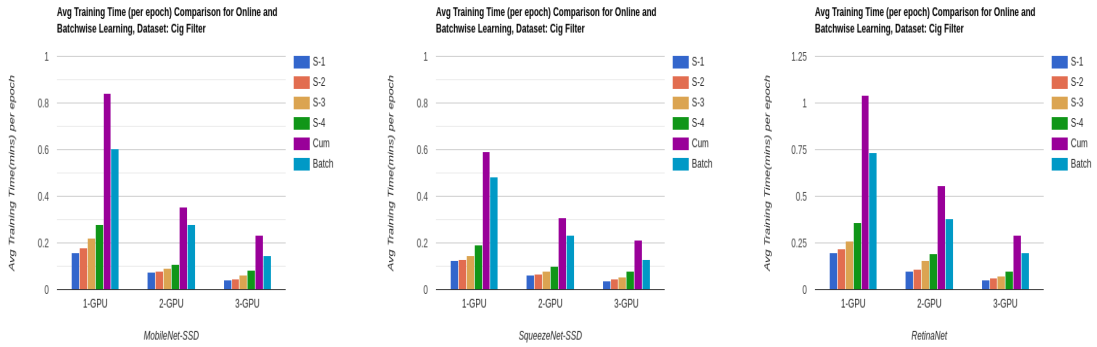
Figure 4.9: Accuracy Comparison between Online Training and Batchwise Training using 3-GPU System; Dataset: Filter; Epochs: 160



(a) Training Time for MobileNet-SSD (b) Training Time for SqueezeNet-SSD (c) Training Time for RetinaNet

Figure 4.10: Online vs Batch Training Time Comparison for MobileNet-SSD, SqueezeNet-SSD and RetinaNet using 1, 2, 3-GPU System; Dataset: Fruits; Epochs: 250

possible to reduce training time for online learning by adding more GPU with some communication overheads.



(a) Training Time for MobileNet-SSD (b) Training Time for SqueezeNet-SSD (c) Training Time for RetinaNet

Figure 4.11: Online vs Batch Training Time Comparison for MobileNet-SSD, SqueezeNet-SSD and RetinaNet using 1, 2, 3-GPU System; Dataset: Filter; Epochs: 160

4.2.3 Accuracy Comparison for Incremental Learning (Adding a new class)

In this section, a new class called **Mango** (having 185 images) has been incrementally trained along with the 8 classes previously trained on 3-GPU system using *Fruits* dataset for 250 epochs. We used Distillation loss function in order to incrementally train the new class. For simplicity, unlike training all 3 models, only RetinaNet[52] model has been trained and tested for this newly added **Mango** class. All through this thesis, a pretrained RetinaNet model (ResNet-50 as backbone and trained on ImageNet dataset[41]) is used and retrained with *Fruits* and *Filter* datasets using Transfer Learning approach.

Class	8-Class(Using Original Loss)	8+1-Class (Using Original Loss)	Catastrophic Forgetting (Original Loss)	Using Knowledge Distillation Loss
Apple	73.22	74.77	19.69	72.63
Banana	64.40	64.23	8.72	65.71
Grape	72.32	74.53	13.31	73.21
Orange	74.51	75.87	16.94	73.74
Pear	60.28	68.16	9.72	67.26
Pineapple	60.62	66.59	11.26	65.18
Strawberry	72.27	73.25	17.45	73.44
Watermelon	62.64	67.48	17.11	67.21
Mango	–	66.83	62.47	70.05
mAP	67.5322	**70.1888**	19.6317	**69.8710**

Table 4.4: mAP Comparison for Adding New Class for RetinaNet[52] model using 3-GPUs systems; Epoch: 250; Dataset: Fruits;

From the Table 4.4, previously trained 8-Class accuracy is shown using the original loss function as defined in Equation 2.6. Secondly, all 8+1-Classes have been trained using the data from both the old and new classes with fine-tuning the old model. This time still original loss function is used and achieved around 70.19 mAP. Thirdly, to measure Catastrophic Forgetting, we trained the model with only the new class using original loss function and tested using test dataset of all classes, resulting mAP equals to 19.63 (old classes are not trained for this case). Finally, the model is incrementally trained using Knowledge Distillation Loss function defined in Equation 3.1 and as depicted in Figure 3.3. This achieved an accuracy of 69.87 mAP which is slightly greater than using original loss function. Therefore, the Distributed Incremental Training approach successfully learnt the new **Mango** class without forgetting other classes.

4.3 Discussion

In [6] authors proposed a resource management framework called Deep-Edge. It leverages parameter server based distributed training at the edge to update DL model. This reduced the epoch time by 1.54 times, where our results exhibited much better reduction in training time. Although, our testbed is quite different from Deep-Edge.

Google Research[32] studied several strategies to tradeoff accuracy for speed and memory utilization of object detection models. Lateral comparisons between models have proven challenging due to differences in base feature extractors such as VGG16, ResNet and hardware and software platforms. According to [32], MobileNet-SSD achieved higher accuracy among the fastest models. Our experiment also claimed the same. MobileNet-SSD outperformed the other 2 models in terms of accuracy and reducing training time.

It is noticeable that the accuracy of SqueezeNet-SSD using 3-GPUs, is lower than the baseline as well as MobileNet-SSD and RetinaNet. However in case of speeding-up training, SqueezeNet-SSD achieved better (almost similar for MobileNet-SSD) speed than baseline and RetinaNet. Therefore, the results of SqueezeNet-SSD in our study are in line with the results explained by the authors of SqueezeNet main paper[33] as well as by [49].

Authors of [67] developed an automated annotation system mounted on a vehicle to collect data of wastes (having Cigarette Filter data) from the streets. Then authors applied deep learning to classify and localize different types of wastes. However, their model is not capable of detecting objects in real time. Their model's accuracy is 63.2% where our experiment showed an accuracy of up to 74.81%, though with different datasets from ours. Moreover, our distributed incremental learning approach is capable of learning new classes without forgetting old class and without training from scratch.

Finally, by combining the distributed training and online learning, both the speed of training and accuracy are improved compare to baseline(VGG16-SSD) model trained on single GPU, without applying any tricks which have been applied to other 3 models. [11] achieved 59.4% mAP with Knowledge Distillation(KD) for VGG16 using PAS-

CAL VOC dataset, which was 54.7% without using KD. Our experiment achieved 69.87% mAP using RetinaNet model with KD, which was 67.53% without KD. Few important aspects of research related to our work, are not explored in this thesis. We mention these limitations in next section and will explore in our future work.

Chapter 5

Conclusions

In this thesis, we implemented a scalable parameter server based distributed training approach particularly designed for object detection including real-time inference for embedded devices. In addition to that, this approach incrementally trains the model for unseen data of existing classes as well as for new class.

In the experiments, we showed that using additional GPUs reduces the training time by around 67% using 3 GPUs compared with single GPU with some overhead due to the increased communication time between the workers and server for distributed training. With the utilization of online incremental training, our experiments exhibited an increase of accuracy up to 16% comparing with batch training using the distributed system. Therefore, incremental training performs better than batch training. After each incremental session of training, we no longer require to store the training data as well as to train from scratch(which is the case for batch training). Moreover, using our incremental learning approach, new class can be trained incrementally without retraining the old classes from scratch. Thus, incremental training saves enormous computation than batch training. We used 3 state-of-the-art one-stage object detectors in order to test the efficacy of our distributed system. From model performance point of views, MobileNet-SSD outperforms the other 2 models, SqueezeNet-SSD and RetinaNet, in terms of both reducing per epoch training time and increasing accuracy.

We rigorously studied the Catastrophic Forgetting and small object detection problems which are associated with our works. Hence, Knowledge Distillation and preparation of a small exemplary dataset helped to overcome the Catastrophic Forgetting substantially. We built a small dataset named *Cigarette Filter* as an application to beach garbage collector. This custom dataset contains small objects, which are difficult to detect. From our empirical study and experiment, it is proved that some strategies such as increasing image resolution, tiling images and most importantly using Focal Loss function (only for RetinaNet model) aided to detect small objects.

5.1 Summary of Contributions

To conclude this study, we can claim that we achieved the goals mentioned in the aims and objectives section of Chapter 1. Therefore, the key contributions are:

1. Incorporating online incremental learning approach with Parameter Server based scalable distributed training approach. By combining both the distributed training and incremental learning, our experiment showed a significant reduction in training time as well as an increased in accuracy compared to batch learning. This is the core contribution of this thesis.

2. Extensive analysis of 3 state-of-the-art one-stage object detection models' performance using distributed incremental learning approach.

3. Simultaneous execution of Knowledge Distillation [30] concept and a small exemplary dataset [68] make our distributed incremental learning approach more robust in dealing with Catastrophic Forgetting problem.

4. Preparing and annotating our own collected *Cigarette Filter* dataset consisting of 287 images, as a part of beach clean-up using robots. This dataset has one class and contains small objects.

5. Along with Focal loss function [52], other strategies mentioned in subsection 3.5.1 substantially solved the challenge of detecting small objects.

5.2 Limitations

Our work has some limitations, and these are:

- For the sake of simplicity, only one Parameter Server and 3-GPUs are used for this experiment. More Parameter servers and workers could have been used in order to further demonstrate the scalibility of the proposed distributed incremental training system.

- In this study, small or medium sized datasets are used. In practical, large dataset may need to be tested for our distributed incremental training approach.

- Our study does not consider some network issues such as improving Quality-of-Service (latency, bandwidth etc.).

- In this thesis, only homogeneous data and embedded devices are used. From practical point of view, data can be heterogeneous and different embedded devices can be used as workers. Moreover, our study completely ignores the encryption of the parameters, where any intruder can temper the parameters. These open a new broad area of research called *Federated Learning*, which is discussed as our future work.

5.3 Future Work

This thesis work unfolds a potential direction of work called *Federated Learning*, as mentioned in the limitation section. We already started working on this area. Below is a glimpse of our future work related to this area.

Federated Learning (FL) refers to a special scenario where edge devices collaboratively train a shared prediction model without transferring their local data to a centralized server, rather model is moved on each edge devices, gets encrypted and then again dispatched to the cloud. A global model in cloud server is formed by integrating all the encrypted models sent from the devices. In this way, the server does not know about the data from devices, thus preserve the privacy of the owner of the data. Finally, edge devices can download the updated model from cloud server which is already under encryption. Moreover, clients (edge devices) do not know about each other's data.

Our focus is on the following three important aspects of FL:

- **Optimal client selection** so as to reduce the number of training rounds to achieve a given accuracy.
- **Improving communication efficiency** through weight compression, model pruning or applying quantization.
- Encryption of parameters through **privacy preserving techniques**.

Appendix A

An Appendix

A.1 Code Snippet

The following code snippet imports the necessary libraries in order to create and spawn the processes. These lines of code correspond to implementation of Lines 6 to 9 of Algorithm 1.

```
1  import argparse
2  import os
3  import time
4  from threading import Lock
5
6  import torch
7  import torch.distributed.autograd as dist_autograd
8  import torch.distributed.rpc as rpc
9  import torch.multiprocessing as mp
10 import torch.nn as nn
11 import torch.nn.functional as F
12 from torch import optim
13 from torch.distributed.optim import DistributedOptimizer
14 from torchvision import datasets, transforms
15
16 processes = []
17 world_size = args.world_size
18 if args.rank == 0:
19     p = mp.Process(target=run_parameter_server, args=(0, world_size)
       )
20     p.start()
21     processes.append(p)
22 else:
23     # Get data to train on
24     train_loader = torch.utils.data.DataLoader(
       dataset= train_dataset,              train=True,
              transform=train_transform,    target_transform=
       target_transform,
25              batch_size=4,             shuffle=True,
26     )
27
28     test_loader = torch.utils.data.DataLoader(
```

```
            dataset=test_dataset,
            train=False,
            transform= test_transform, target_transform=
    target_transform
            batch_size=4,
            shuffle=True,
    )
    # start training worker on this node
    p = mp.Process(
        target=run_worker,
        args=(
            args.rank,
            world_size, args.num_gpus,
            train_loader,
            test_loader))
    p.start()
    processes.append(p)

for p in processes:
    p.join()
```

Listing A.1: Creating process for each node

In Listing A.1, for example, one parameter server running with $world\_size = 4$, the shell command would be *python rpc_parameter_server.py –world_size=4 –rank=0*. Afterwards, the workers can be launched by the command *python rpc_parameter_server.py –world_size=4 –rank=1* and so on for each worker.

```
# Ref: https://github.com/marvis/pytorch-mobilenet

#Putting 26 layers of MobileNetV1 into 1st GPU
class MobileNetV1_SSD(nn.Module):
    def __init__(self, num_classes=1024):
        super(MobileNetV1_SSD, self).__init__()
    device_ids = ["cuda:0", "cuda:1", "cuda:2"]

        def conv_bn(inp, oup, stride):
            return nn.Sequential(
                nn.Conv2d(inp, oup, 3, stride, 1, bias=False),
                nn.BatchNorm2d(oup),
                nn.ReLU(inplace=True)
            )

        def conv_dw(inp, oup, stride):
            return nn.Sequential(
                nn.Conv2d(inp, inp, 3, stride, 1, groups=inp, bias=
    False),
                nn.BatchNorm2d(inp),
                nn.ReLU(inplace=True),

                nn.Conv2d(inp, oup, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup),
                nn.ReLU(inplace=True),
            )
    device = torch.device(device_ids[0])
```

```
        self.model = nn.Sequential(
            conv_bn(3, 32, 2),
            conv_dw(32, 64, 1),
            conv_dw(64, 128, 2),
            conv_dw(128, 128, 1),
            conv_dw(128, 256, 2),
            conv_dw(256, 256, 1),
            conv_dw(256, 512, 2),
            conv_dw(512, 512, 1),
            conv_dw(512, 512, 1),
            conv_dw(512, 512, 1),
            conv_dw(512, 512, 1),
            conv_dw(512, 512, 1),
            conv_dw(512, 1024, 2),
            conv_dw(1024, 1024, 1),
        ).to(device)
        self.fc = nn.Linear(1024, num_classes).to(device)


#Putting 8 layers of SSD into 2nd GPU
    device = torch.device(device_ids[1])
    SSD_layers = ModuleList([
      Sequential(
        Conv2d(in_channels=1024, out_channels=256, kernel_size=1),
        ReLU(),
        Conv2d(in_channels=256, out_channels=512, kernel_size=3,
   stride=2, padding=1),
        ReLU()
      ),
      Sequential(
        Conv2d(in_channels=512, out_channels=128, kernel_size=1),
        ReLU(),
        Conv2d(in_channels=128, out_channels=256, kernel_size=3,
   stride=2, padding=1),
        ReLU()
      ),
      Sequential(
        Conv2d(in_channels=256, out_channels=128, kernel_size=1),
        ReLU(),
        Conv2d(in_channels=128, out_channels=256, kernel_size=3,
   stride=2, padding=1),
        ReLU()
      ),
      Sequential(
        Conv2d(in_channels=256, out_channels=128, kernel_size=1),
        ReLU(),
        Conv2d(in_channels=128, out_channels=256, kernel_size=3,
   stride=2, padding=1),
        ReLU()
      )
      ])
    self.model = nn.Sequential(SSD_layers).to(device)

#Putting rest layers of SSD into 3rd GPU
```

```
77      device = torch.device(device_ids[2])
78
79      regression_headers = ModuleList([
80          Conv2d(in_channels=512, out_channels=6 * 4, kernel_size=3,
    padding=1),
81          Conv2d(in_channels=1024, out_channels=6 * 4, kernel_size=3,
    padding=1),
82          Conv2d(in_channels=512, out_channels=6 * 4, kernel_size=3,
    padding=1),
83          Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=3,
    padding=1),
84          Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=3,
    padding=1),
85          Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=3,
    padding=1),
86      ])
87
88      classification_headers = ModuleList([
89          Conv2d(in_channels=512, out_channels=6 * num_classes,
    kernel_size=3, padding=1),
90          Conv2d(in_channels=1024, out_channels=6 * num_classes,
    kernel_size=3, padding=1),
91          Conv2d(in_channels=512, out_channels=6 * num_classes,
    kernel_size=3, padding=1),
92          Conv2d(in_channels=256, out_channels=6 * num_classes,
    kernel_size=3, padding=1),
93          Conv2d(in_channels=256, out_channels=6 * num_classes,
    kernel_size=3, padding=1),
94          Conv2d(in_channels=256, out_channels=6 * num_classes,
    kernel_size=3, padding=1),
95      ])
96      self.model = nn.Sequential(regression_headers).to(device)
97      self.model = nn.Sequential(classification_headers).to(device)
```

Listing A.2: Splitting MobileNet-SSD model into different GPUs

In Listing A.2, in case of MobileNet-SSD, it is reasonable to place first 26 layers from MobileNet into first GPU, then 8 layers of SSD on second GPU and the remaining layers (both classification headers and regression headers) on the third GPU. The rationale of this splitting is that MobileNet layers are used for forward pass only. On the other hand, SSD layers are both used for forward and backward passes.

Bibliography

[1] Martın Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Wickliffe C Abraham and Anthony Robins. "Memory retention–the synaptic stability versus plasticity dilemma". In: *Trends in neurosciences* 28.2 (2005), pp. 73–78.

[3] Abbas Raza Ali, Marcin Budka, and Bogdan Gabrys. "Towards meta-learning of deep architectures for efficient domain adaptation". In: *Pacific Rim International Conference on Artificial Intelligence*. Springer. 2019, pp. 66–79.

[4] Apache. *Apache Mahout*. http://mahout.apache.org/. [Online; accessed 21-May-2021]. 2021.

[5] Paul Baran. "On distributed communications networks". In: *IEEE transactions on Communications Systems* 12.1 (1964), pp. 1–9.

[6] Anirban Bhattacharjee et al. "Deep-Edge: An Efficient Framework for Deep Learning Model Update on Heterogeneous Edge". In: *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*. IEEE. 2020, pp. 75–84.

[7] Clemens-Alexander Brust, Christoph Käding, and Joachim Denzler. "Active learning for deep object detection". In: *arXiv preprint arXiv:1809.09875* (2018).

[8] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. *An Analysis of Deep Neural Network Models for Practical Applications*. 2017. arXiv: 1605.07678 [cs.CV].

[9] Chenyi Chen et al. "R-CNN for small object detection". In: *Asian conference on computer vision*. Springer. 2016, pp. 214–230.

[10] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. "Efficient and robust parallel dnn training through model parallelism on multi-gpu platform". In: *arXiv preprint arXiv:1809.02839* (2018).

[11] Guobin Chen et al. "Learning efficient object detection models with knowledge distillation". In: *Advances in neural information processing systems* 30 (2017).

[12] Jianguo Chen et al. "A Bi-layered Parallel Training Architecture for Large-scale Convolutional Neural Networks". In: *CoRR* abs/1810.07742 (2018). arXiv: 1810.07742. URL: http://arxiv.org/abs/1810.07742.

[13] Jianguo Chen et al. “Distributed deep learning model for intelligent video surveillance systems with edge computing”. In: *IEEE Transactions on Industrial Informatics* (2019).

[14] Jianmin Chen et al. *Revisiting Distributed Synchronous SGD*. 2017. arXiv: 1604.00981 [cs.LG].

[15] Torch Contributors. *Distributed RPC Framework*. URL: https://pytorch.org/docs/stable/rpc.html. (accessed: 01.01.2021).

[16] Jeffrey Dean et al. “Large scale distributed deep networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1223–1231.

[17] J. Deng et al. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255.

[18] Simone Disabato, Manuel Roveri, and Cesare Alippi. “Distributed deep convolutional neural networks for the internet-of-things”. In: *IEEE Transactions on Computers* (2021).

[19] Jim Dowling. *parameter server based training*. https://www.oreilly.com/content/distributed-tensorflow/. [Online; accessed 05-Sept-2021]. 2020.

[20] Elmootazbellah Elnozahy et al. “A Survey of Rollback-Recovery Protocols in Message-Passing Systems”. In: *ACM Computing Surveys* 34 (June 2002). DOI: 10.1145/568522.568525.

[21] Dustin Franklin. *jetson-inference*. https://github.com/dusty-nv/jetson-inference. [Online; accessed 19-July-2021]. 2020.

[22] Amol Ghoting et al. “SystemML: Declarative machine learning on MapReduce”. In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 231–242. DOI: 10.1109/ICDE.2011.5767930.

[23] Ross Girshick. *Fast R-CNN*. 2015. arXiv: 1504.08083 [cs.CV].

[24] Ian J Goodfellow et al. “An empirical investigation of catastrophic forgetting in gradient-based neural networks”. In: *arXiv preprint arXiv:1312.6211* (2013).

[25] Hongtao Guan et al. “A Distributed Class-Incremental Learning Method Based on Neural Network Parameter Fusion”. In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2019, pp. 257–264. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00049.

[26] Steven Gutstein and Ethan Stump. “Reduction of catastrophic forgetting with transfer learning and ternary output codes”. In: *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2015, pp. 1–8.

[27] Aaron Harlap et al. “Pipedream: Fast and efficient pipeline parallel dnn training”. In: *arXiv preprint arXiv:1806.03377* (2018).

[28] K. He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[29] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. 2012. arXiv: 1207.0580 [cs.NE].

[30] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. “Distilling the knowledge in a neural network”. In: *arXiv preprint arXiv:1503.02531* (2015).

[31] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[32] Jonathan Huang et al. “Speed/accuracy trade-offs for modern convolutional object detectors”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7310–7311.

[33] Forrest N. Iandola et al. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size*. 2016. arXiv: 1602.07360 [cs.CV].

[34] Yangqing Jia et al. “Caffe: Convolutional architecture for fast feature embedding”. In: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pp. 675–678.

[35] Christoph Käding et al. “Fine-Tuning Deep Neural Networks in Continuous Learning Scenarios”. In: (Mar. 2017), pp. 588–605. DOI: 10.1007/978-3-319-54526-4_43.

[36] Christoph Käding et al. “Fine-tuning deep neural networks in continuous learning scenarios”. In: *Asian Conference on Computer Vision*. Springer. 2016, pp. 588–605.

[37] Christoph Käding et al. “Fine-tuning deep neural networks in continuous learning scenarios”. In: *Asian Conference on Computer Vision*. Springer. 2016, pp. 588–605.

[38] James Kirkpatrick et al. “Overcoming catastrophic forgetting in neural networks”. In: *Proceedings of the national academy of sciences* 114.13 (2017), pp. 3521–3526.

[39] Alexey Kutalev. “Natural Way to Overcome the Catastrophic Forgetting in Neural Networks”. In: *arXiv preprint arXiv:2005.07107* (2020).

[40] Alina Kuznetsova et al. “Expanding object detector’s horizon: Incremental learning framework for object detection in videos”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 28–36.

[41] Alina Kuznetsova et al. “The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale”. In: *IJCV* (2020).

[42] Matthias Langer et al. “MPCA SGD—A method for distributed training of deep learning models on spark”. In: *IEEE Transactions on Parallel and Distributed Systems* 29.11 (2018), pp. 2540–2556.

[43] Dawei Li et al. “RILOD: Near real-time incremental learning for object detection at the edge”. In: *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 2019, pp. 113–126.

[44] He Li, Kaoru Ota, and Mianxiong Dong. “Learning IoT in edge: Deep learning for the Internet of Things with edge computing”. In: *IEEE network* 32.1 (2018), pp. 96–101.

[45] Liangzhi Li, Kaoru Ota, and Mianxiong Dong. “Deep learning for smart industry: Efficient manufacture inspection system with fog computing”. In: *IEEE Transactions on Industrial Informatics* 14.10 (2018), pp. 4665–4673.

[46] Mu Li et al. “Scaling distributed machine learning with the parameter server”. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 583–598.

[47] Mu Li et al. “Scaling distributed machine learning with the parameter server”. In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014* (2014), pp. 583–598.

[48] Peng Li et al. “Deep convolutional computation model for feature learning on big data in internet of things”. In: *IEEE Transactions on Industrial Informatics* 14.2 (2017), pp. 790–798.

[49] Yuxi Li et al. “Tiny-DSOD: Lightweight object detection for resource-restricted usages”. In: *arXiv preprint arXiv:1807.11013* (2018).

[50] Zhizhong Li and Derek Hoiem. “Learning without forgetting”. In: *IEEE transactions on pattern analysis and machine intelligence* 40.12 (2017), pp. 2935–2947.

[51] Zhizhong Li and Derek Hoiem. “Learning without forgetting”. In: *IEEE transactions on pattern analysis and machine intelligence* 40.12 (2017), pp. 2935–2947.

[52] Tsung-Yi Lin et al. *Focal Loss for Dense Object Detection*. 2018. arXiv: 1708.02002 [cs.CV].

[53] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV].

[54] Yujun Lin et al. *Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training*. 2020. arXiv: 1712.01887 [cs.CV].

[55] Li Liu et al. “Deep learning for generic object detection: A survey”. In: *International journal of computer vision* 128.2 (2020), pp. 261–318.

[56] Wei Liu et al. “SSD: Single Shot MultiBox Detector”. In: *Lecture Notes in Computer Science* (2016), pp. 21–37. ISSN: 1611-3349. DOI: 10.1007/978-3-319-46448-0_2. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2.

[57] Yucheng Low et al. “Graphlab: A new framework for parallel machine learning”. In: *arXiv preprint arXiv:1408.2041* (2014).

[58] Pedro Marcelino. *Transfer learning from pre-trained models*. URL: https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751. (accessed: 01.09.2020).

[59] Christian Mayer, Ruben Mayer, and Majd Abdo. "StreamLearner". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems* (June 2017). DOI: 10.1145/3093742.3095103. URL: http://dx.doi.org/10.1145/3093742.3095103.

[60] Apache Spark MLLib. *Apache Spark MLLib*. https://spark.apache.org/mllib/. [Online; accessed 21-May-2021]. 2021.

[61] Philipp Moritz et al. *Ray: A Distributed Framework for Emerging AI Applications*. 2018. arXiv: 1712.05889 [cs.DC].

[62] David Newman et al. "Distributed algorithms for topic models." In: *Journal of Machine Learning Research* 10.8 (2009).

[63] NVDIA. *TensorRT engine*. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html. [Online; accessed 15-Aug-2021]. 2021.

[64] Kemal Oksuz et al. "Imbalance problems in object detection: A review". In: *IEEE transactions on pattern analysis and machine intelligence* (2020).

[65] Diego Peteiro-Barral and Bertha Guijarro-Berdiñas. "A survey of methods for distributed machine learning". In: *Progress in Artificial Intelligence* 2.1 (2013), pp. 1–11.

[66] Udaya LN Puvvadi et al. "Cost-effective security support in real-time video surveillance". In: *IEEE Transactions on Industrial Informatics* 11.6 (2015), pp. 1457–1465.

[67] Mohammad Saeed Rad et al. "A computer vision system to localize and classify wastes on the streets". In: *International Conference on computer vision systems*. Springer. 2017, pp. 195–204.

[68] Sylvestre-Alvise Rebuffi et al. *iCaRL: Incremental Classifier and Representation Learning*. 2017. arXiv: 1611.07725 [cs.CV].

[69] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 91–99. URL: http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf.

[70] Fiona M Richardson and Michael SC Thomas. "Critical periods and catastrophic interference effects in the development of self-organizing feature maps". In: *Developmental science* 11.3 (2008), pp. 371–389.

[71] Peter Richtárik and Martin Takáč. "Distributed coordinate descent method for learning with big data". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 2657–2681.

[72] Ozan Sener and Silvio Savarese. "Active learning for convolutional neural networks: A core-set approach". In: *arXiv preprint arXiv:1708.00489* (2017).

[73] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[74] Alexander Smola and Shravan Narayanamurthy. "An Architecture for Parallel Topic Models". In: *Proc. VLDB Endow.* 3.1–2 (Sept. 2010), pp. 703–710. ISSN: 2150-8097. DOI: 10.14778/1920841.1920931. URL: https://doi.org/10.14778/1920841.1920931.

[75] Alexander Smola and Shravan Narayanamurthy. "An architecture for parallel topic models". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 703–710.

[76] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[77] Wen Sun et al. "Double auction-based resource allocation for mobile edge computing in industrial internet of things". In: *IEEE Transactions on Industrial Informatics* 14.10 (2018), pp. 4692–4701.

[78] Christian Szegedy et al. *Rethinking the Inception Architecture for Computer Vision*. 2015. arXiv: 1512.00567 [cs.CV].

[79] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. "Distributed deep neural networks over the cloud, the edge and end devices". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 328–339.

[80] Jordi TORRES.AI. *Data parallel and Model Parallel*. https://towardsdatascience.com/deep-learning-on-supercomputers-96319056c61f. [Online; accessed 20-Apr-2021]. 2020.

[81] Sik-Ho Tsang. *review on MobileNet*. https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69. [Online; accessed 20-Feb-2021]. 2020.

[82] Konstantinos I Tsianos, Sean Lawlor, and Michael G Rabbat. "Communication/computation tradeoffs in consensus-based distributed optimization". In: *arXiv preprint arXiv:1209.1076* (2012).

[83] Maria Valera et al. "Communication mechanisms and middleware for distributed video surveillance". In: *IEEE transactions on circuits and systems for video technology* 21.12 (2011), pp. 1795–1809.

[84] Joost Verbraeken et al. "A survey on distributed machine learning". In: *ACM Computing Surveys (CSUR)* 53.2 (2020), pp. 1–33.

[85] Huaming Wu et al. "Collaborate Edge and Cloud Computing With Distributed Deep Learning for Smart City Internet of Things". In: *IEEE Internet of Things Journal* 7.9 (2020), pp. 8099–8110. DOI: 10.1109/JIOT.2020.2996784.

[86] Eric P Xing et al. “Strategies and principles of distributed machine learning on big data”. In: *Engineering* 2.2 (2016), pp. 179–195.

[87] Shanhe Yi et al. “Lavea: Latency-aware video analytics on edge computing platform”. In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. 2017, pp. 1–13.

[88] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. “Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2348–2359.

[89] Yi Zhou et al. “Fast automatic vehicle annotation for urban traffic surveillance”. In: *IEEE Transactions on Intelligent Transportation Systems* 19.6 (2017), pp. 1973–1984.

[90] Zhenyu Zhou et al. “Robust mobile crowd sensing: When deep learning meets edge computing”. In: *Ieee network* 32.4 (2018), pp. 54–60.