UNIVERSITY OF NOTTINGHAM MALAYSIA



EDGE COMPUTING TECHNIQUES FOR CLASSIFICATION BASED MULTI-TASK INCREMENTAL LEARNING

DUBE SWARAJ SUNILKUMAR, M.Eng.

A DISSERTATION SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING AT THE UNIVERSITY OF NOTTINGHAM MALAYSIA CAMPUS IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Abstract

The number of Internet of Things (IoT) edge devices are exponentially on the rise that have both computational capabilities and internet connectivity. IoT devices are the ones that collect massive amounts of data which can then be analyzed and can also be used to train machine learning models for various applications. However, edge devices, generally do not possess the same resources as the cloud in terms of computing power and storage. Such limitations of edge devices often lead to relying largely on the cloud for further data processing. Consequently, large transmission costs occur with respect to the data size and number of edge devices connected to the cloud. Furthermore, real-world data always evolves over time, hence machine learning needs to be learning continuously to adapt to such evolving data, resulting in large data transmission over time.

The main focus of this research is to reduce data transmission costs between edge devices and the cloud in an incremental learning scenario while retaining deep neural network model performance and to reduce the energy consumption of the learning process on multiple edge devices.

Firstly, a novel pre-training data sampling algorithm is presented for edge-cloud systems whereby this algorithm can automatically select a subsample from a training distribution right at the edge device before training even starts while maintaining the incremental learning performance of a Convolutional Neural Network (CNN) in terms of its classification accuracy. This means that if this data sampling technique is executed on edge devices, it will not require the transmission of all data samples to the cloud. This can save both the transmission cost and the training cost on the cloud while the model performance is largely retained. Results show that after performing the developed pre-training data sampling technique, the classification performance of Convolutional Neural Network (CNN) models in the context of incremental learning remains within 3% while the transmission costs are reduced by up to 18%.

Secondly, a novel method to extract only the important weights of a fully connected neural network is also presented in this report. The method can also further reduce the transmission costs between the edge devices and the cloud. Experiments show that not all the weights learned by a neural network in the cloud are useful. So, it is actually not required to send all the learned weights back to the edge devices. When only the useful weights are transmitted back to the edge devices, this will be beneficial especially in the case of incremental learning where the size of a neural network can continuously grow. Results show that by implementing the proposed extraction method, the transmission costs can be reduced by at least 35%. The proposed extraction method can also extract useful weights/parameters faster than the baseline by a factor of at least 1.9.

Finally, this research also discovers that when multiple edge devices are participating in carrying out multi-task learning, the overall energy consumption across all devices is highly dependent on the edge device training order i.e., which edge device should start the training and which edge device should end the training. This is due to the heterogeneity of the edge devices i.e., some edge devices can be much more computationally powerful than the other edge devices. An introductory algorithm that can determine the edge device training order to minimize the end-to-end energy consumption across all the participating edge devices is developed. Results show that by following the introductory algorithm in choosing the correct order of edge devices for training, the overall end-to-end energy consumption can be reduced by a factor of at least 1.6.

Acknowledgements

I would like to express my sincere gratefulness to my supervisor Dr. Hermawan Nugroho for providing supervision, support, key ideas, and scientific freedom to pursue my research interests. Dr. Hermawan became my supervisor right towards the beginning of the COVID-19 Pandemic which meant we had to start our work together virtually, however, despite being in a lockdown, he was quickly able to understand my research work and provided key ideas based on which this entire research is based. I received full support from him in terms of meetings, the direction that I needed, the correctness of my approach, and the scientific publications as well.

I would also like two of my former supervisors: Dr. Khor Jeen Ghee and Dr. Wong Yee Wan. The idea of carrying out research in the domain of edge computing and deep learning was Dr. Khor helped lay out the foundation of my research and encouraged me to carry out research in the domain of edge computing and deep learning based on current and future technological trends. Following this, Dr. Wong fueled my research by providing insights into how edge computing and deep learning can be integrated with continuous learning, this was the period I was able to precisely detect research gaps in this domain and started my actual experiments. It was after this phase that Dr. Hermawan took over and guided me to get this research to the point it is at today. I would like to thank my internal assessor Dr. Tomas Maul who reviewed my annual reports and gave me extremely valuable feedback as to how I can improve my work and he made sure I'm on the right path in terms of my progression.

I would also like to thank the University of Nottingham Malaysia for providing me with the dean's scholarship that waived 100% of the tuition. This support is what enabled me to start my Ph.D. and carry on with my research for the duration of the program.

List of Publications

A total of 3 publications are published based on the research work presented in this thesis. The details are as follows:

Chapter 3

- Journal publication: **S. Dube**, YW. Wong, H. Nugroho. "Dynamic Sampling of Images from Various Categories for Classification Based Incremental Deep Learning in Fog Computing". In: PeerJ Computer Science. PeerJ, 2021 (https://peerj.com/articles/cs-633/).
- Conference publication: **S. Dube**, YW. Wong, JG. Khor, H. Nugroho. "Novel Approach of Class Incremental Learning on Internet of Things (IoT) Framework". In: Advances in Intelligent Systems and Computing. Springer, 2021 (https://link.springer.com/chapter/10.1007/978-3-030-70917-4_34).

Chapter 4

- Journal publication: **S. Dube**, YW. Wong, H. Nugroho. "A Novel Approach of IoT Stream Sampling and Model Update on the IoT Edge Device for Class Incremental Learning in an Edge-Cloud System". In: IEEE Access. IEEE, 2021 (https://ieeexplore.ieee.org/document/9354152).

The work presented in chapter 3 has been published in a journal and a conference paper. The work presented in chapter 4 has been published in another journal, thus strengthening the novelty and validity of the research work carried out. The novelty and the details of these chapters are discussed later in the thesis.

Table of Contents

Abstract……ii

Acknowledgements……iii

List of Publications……..iv

List of Tables……..vii

List of Figures……..ix

1. Overall Thesis Introduction and Layout……1
1.1. Background……1
1.2. Motivation and Problem Statement...……..2
1.3. Aims and Objectives……...3
1.4. Research Scope……...4
1.5. Research Contribution and Outline……4
1.6. Research Significance……....5

2. Literature Review and Related Work……7
2.1. Background……7
2.2. Search Strategy……...9
2.3. Deep Learning for Edge Computing……11
2.3.1. Compression in Neural Networks……..12
2.3.2. Speeding Up Execution of Neural Networks……13
2.3.3. Parallel and Distributed Deep Learning……15
2.3.4. Computational Offloading……..16
2.3.5. Data Sampling……17
2.4. Types of Incremental Learning……19
2.4.1. Supervised Incremental Learning……...19
2.4.2. Unsupervised Incremental Learning……..19
2.4.3. Incremental Reinforcement Learning……20
2.5. Challenges Of Incremental Learning in Edge Computing……...20
2.5.1. Catastrophic Forgetting……..20
2.5.2. Dealing with High Dimensional Data……22
2.5.3. Transmission Costs Between the Cloud and The Edge Devices……22
2.5.4. Training Without Labels……23
2.5.5. Dealing with Data Distribution Shifts……24
2.5.6. Hardware……25
2.6. Summary..……25
2.7. Research Direction and Problem Interdependency..……26

3. Sampling of Data from Novel Classes for Class Incremental Learning……29
3.1. Background……..29
3.2. Methodology……31
3.2.1. System Architecture and Sample Elimination Algorithm……..32
3.2.2. Communication Between the Edge Device and The Cloud……...36
3.2.3. Incremental Learning on The Cloud……..37

3.3. Experiments……..38
3.4. Results……..40
3.5. Discussion……51
3.6. Summary..……....53

4. Parameter Extraction of Artificial Neural Networks In Class Incremental Learning……..55
4.1. Background……..55
4.2. Neural Network Weight Extraction Algorithm……56
4.3. FitCNN++……60
4.4. Experiments……..62
4.5. Evaluation on CIFAR-100 using MnasNet……..62
4.6. Evaluation on Stanford Dogs using MobileNetV2……..65
4.7. Evaluation on ImageNet using ShuffleNetV2……..67
4.8. Evaluation on CUB-200 using SqueezeNet……69
4.9. Effect of Weight Extraction on Training Accuracies……...71
4.9.1. Training Accuracy on Evaluation of CIFAR-100 Using MnasNet……72
4.9.2. Training Accuracy on Evaluation of Stanford Dogs Using MobileNetV2……..74
4.9.3. Training Accuracy on Evaluation of ImageNet Using ShuffleNetV2…...76
4.9.4. Training Accuracy on Evaluation of CUB-200 Using SqueezeNet……...77
4.10. Effect of Partition Size on The Weight Extraction Time……79
4.11. Summary..……80

5. Effects of Distributed Learning for Multi-Task Learning on Energy Consumption……...82
5.1. Background……..82
5.2. Distributed Multi-Task Learning……83
5.3. On Device Neural Network Training for Larger Than Memory Training Samples……..85
5.4. Edge Device Training Order Selection Algorithm……...86
5.4.1. Total Neural Network Prediction Time on Edge Devices……....…..90
5.4.2. Algorithm For Selecting Edge Device Training Order to Minimize Energy Consumption……..92
5.4.3. Communication Between Edge Devices……94
5.5. Experiment Setup for Performing Distributed Multi-Task Learning……...95
5.6. Evaluation on a Dataset with a High Number of Classes……...……..96
5.6.1. 70-30 Split……..96
5.6.2. 50-50 Split……..99
5.7. Evaluation on a Hybrid Dataset……..……101
5.8. Edge Device Training Order Selection Algorithm Results……103
5.9. Effect of Edge Device Training Order on The Model Accuracy……...105
5.10. Discussion…….…....107
5.11. Summary..……..108

6. Conclusion and Future Work……..109

Bibliography…….....113

List of Tables

Table 2.1. Classification accuracies of different CNN models evaluated on ImageNet [1] and Snapshot Serengeti [2] Dataset……..8
Table 2.2. High level summary of topics that are related to both deep learning and edge computing.……11
Table 2.3. Summary of the research works focusing on incremental learning and IoT……26
Table 3.1. Learning settings and datasets used for the experiments in this chapter…….38
Table 3.2 Results of performing data sampling using CIFAR-100 [3] and MnasNet [4]……41
Table 3.3. Results of performing data sampling using CUB-200 [5] and SqueezeNet [6]…..41
Table 3.4. Results of performing data sampling using Stanford Dogs [7] and MobileNetV2 [8]…….42
Table 3.5. Results of performing data sampling using ImageNet [1] and ShuffleNet V2 [9]…43
Table 3.6. Standard deviation of the reduction in the number of training samples, reduction in the training time on the server and the difference in classification accuracies after applying SEA……..44
Table 3.7. Incrementally learning 10 classes at a time from CIFAR-100 [3] using SqueezeNet [6]…….47
Table 3.8. Incrementally learning 10 classes at a time from CIFAR-100 [3] using ShuffleNet V2 [9]……47
Table 3.9. Incrementally learning 20 classes at a time from CIFAR-100 [3] using SqueezeNet [6].……47
Table 3.10. Incrementally learning 20 classes at a time from CIFAR-100 [3] using ShuffleNet V2 [9].……...48
Table 3.11. Incrementally learning 40 classes at a time from CUB-200 [5] using SqueezeNet [6].……49
Table 3.12. Incrementally learning 40 classes at a time from CUB-200 [5] using ShuffleNet V2 [9]…..……..49
Table 3.13. Incrementally learning 50 classes at a time from CUB-200 [5] using SqueezeNet [6]……..……...50
Table 3.14. Incrementally learning 50 classes at a time from CUB-200 [5] using ShuffleNet V2 [9].……...50
Table 4.1. Useful weights extracted and the time taken via FitCNN++ (Evaluating CIFAR-100 [3] using MnasNet [4]).…….62
Table 4.2. Useful weights extracted and the time taken via FitCNN+ (Evaluating CIFAR-100 [3] using MnasNet [4]).……63
Table 4.3. Useful weights extracted and the time taken via FitCNN [10] (Evaluating CIFAR-100 [3] using MnasNet [4]).…….63
Table 4.4. Useful weights extracted and the time taken via random weight extraction (Evaluating CIFAR-100 [3] using MnasNet [4]).……63
Table 4.5. Useful weights extracted and the time taken via magnitude weight extraction (Evaluating CIFAR-100 [3] using MnasNet [4]).……63
Table 4.6. Useful weights extracted and the time taken via FitCNN++ (Evaluating Stanford Dogs [7] using MobileNetV2 [8]).……65
Table 4.7. Useful weights extracted and the time taken via FitCNN+ (Evaluating Stanford Dogs [7] using MobileNetV2 [8]).……65
Table 4.8. Useful weights extracted and the time taken via FitCNN [10] (Evaluating Stanford Dogs [7] using MobileNetV2 [8]).……...65

Table 4.9. Useful weights extracted and the time taken via random weight extraction (Evaluating Stanford Dogs [7] using MobileNetV2 [8])..66
Table 4.10. Useful weights extracted and the time taken via magnitude weight extraction (Evaluating Stanford Dogs [7] using MobileNetV2 [8])..66
Table 4.11. Useful weights extracted and the time taken via FitCNN++ (Evaluating ImageNet [1] using ShuffleNetV2 [9])..67
Table 4.12. Useful weights extracted and the time taken via FitCNN+ (Evaluating ImageNet [1] using ShuffleNetV2 [9])..67
Table 4.13. Useful weights extracted and the time taken via FitCNN [10] (Evaluating ImageNet [1] using ShuffleNetV2 [9])..67
Table 4.14. Useful weights extracted and the time taken via random weight extraction (Evaluating ImageNet [1] using ShuffleNetV2 [9])..68
Table 4.15. Useful weights extracted and the time taken via magnitude weight extraction (Evaluating ImageNet [1] using ShuffleNetV2 [9])..68
Table 4.16. Useful weights extracted and the time taken via FitCNN++ (Evaluating CUB-200 [5] using SqueezeNet [6])..69
Table 4.17. Useful weights extracted and the time taken via FitCNN+ (Evaluating CUB-200 [5] using SqueezeNet [6])..69
Table 4.18. Useful weights extracted and the time taken via FitCNN [10] (Evaluating CUB-200 [5] using SqueezeNet [6])..70
Table 4.19. Useful weights extracted and the time taken via random weight extraction (Evaluating CUB-200 [5] using SqueezeNet [6])..70
Table 4.20. Useful weights extracted and the time taken via magnitude weight extraction (Evaluating CUB-200 [5] using SqueezeNet [6])..70
Table 4.21. Weight extraction times with respect to different partitions of the weight difference distribution..79
Table 5.1. Possible ways of training a neural network incrementally across various devices...87
Table 5.2. Power, time, and energy consumption across edge devices during distributed multi-task learning. The MacBook Pro having 70 classes and the Jetson Nano having the remaining 30 classes of CIFAR-100 [3] dataset..98
Table 5.3. Power, time, and energy consumption across edge devices during distributed multi-task learning. The MacBook Pro having 50 classes and the Jetson Nano having the remaining 50 classes of CIFAR-100 [3] dataset.....100
Table 5.4. Power, time, and energy consumption across edge devices during distributed multi-task learning. The MacBook Pro having 100 classes of CIFAR-100 [3] and the Jetson Nano having 10 classes of Stanford Dogs [7] dataset..103
Table 5.5. The actual and predicted neural network training times on edge devices.....104
Table 5.6. The actual and predicted energy consumption on edge devices.....104
Table 5.7. Multi-task learning classification accuracies across multiple edge devices with respect to different edge device training orders. 50 classes each of the CIFAR-100 [3] dataset are placed on the MacBook Pro and the Jetson Nano.....105
Table 5.8. Multi-task learning classification accuracies across multiple edge devices with respect to different edge device training orders. 70 classes of the CIFAR-100 [3] dataset are placed on the MacBook Pro and the remaining 30 classes placed on the Jetson Nano.....106
Table 5.9. Top – 5 classification accuracies of the SqueezeNet [6] model trained on the CIFAR-100 [3] dataset incrementally using 10 classes at a time in a single-incremental-task learning scenario and multi-task learning scenario.....106

List of Figures

Figure 2.1. High level overview of edge intelligence and incremental learning…….9
Figure 2.2. Total number of publications over the last 10 years in the domain of (a) edge computing, (b) edge intelligence, and (c) incremental learning…….10
Figure 2.3. (a) pruning of non-useful weights in a neural network and (b) pruning of non-useful neurons in a neural network……..14
Figure 2.4. Representation of distributed intelligence across various edge devices…….16
Figure 2.5. Incremental learning system partitioned between an IoT edge device and cloud……17
Figure 2.6. Unsupervised learning of raw unlabelled streaming data for incremental learning…….23
Figure 2.7. Sudden concept drift (observed in multi task incremental learning)……24
Figure 3.1. Incremental Learning System Split Between A Fog Device and A Centralized Server.……..31
Figure 3.2. High level Overview of Sample Elimination Algorithm (SEA) on the Fog Device……..……32
Figure 3.3. The proposed system flowchart. Incremental learning system partitioned between an IoT edge device and the cloud…….34
Figure 3.4. Overall change in the number of samples transmitted to the server, the training time on the server, and the classification accuracies after applying SEA……45
Figure 3.5. Total samples transmitted to the cloud (a) using SqueezeNet [6], (b) using ShuffleNet V2 [9]. Training time on cloud (c) using features from SqueezeNet [6], (d) using features from ShuffleNet V2 [9] during evaluation on CIFAR-100 [3] dataset……...48
Figure 3.6. Total samples transmitted to the cloud (a) using SqueezeNet [6], (b) using ShuffleNet V2 [9]. Training time on cloud (c) using features from SqueezeNet [6], (d) using features from ShuffleNet V2 [9] during evaluation on CUB-200 [5] dataset……...50
Figure 3.7. Initial cross entropy loss values of samples from CIFAR-100 [3] dataset on SqueezeNet [6] when learning 10 classes at a time using the same learning settings as shown in Table 3.1 when evaluating CIFAR-100 [3]. Red points represent samples whose entropy values increased after training and blue points represent samples whose entropy values decreased after training.……52
Figure 4.1. Overview of the weight extraction procedure……56
Figure 4.2. Distribution of the weight difference between the classifier on the cloud before and after training when learning (a) the first 10 classes (b) last 10 classes of the CIFAR-100 [3] dataset with rehearsal (training all samples accumulates so far) using the SqueezeNet [6] model.……...57
Figure 4.3. Partitioning the weight difference between the post-trained and pre-trained neural network into multiple sections.……61
Figure 4.4. Testing accuracy of the neural network at every incremental training round when evaluating CIFAR-100 [3] using various weight extraction algorithms……64
Figure 4.5. Testing accuracy of the neural network at every incremental training round when evaluating Stanford Dogs [7] using MobileNetV2 [8] using various weight extraction algorithms.……66
Figure 4.6. Testing accuracy of the neural network at every incremental training round when evaluating ImageNet [1] using ShuffleNetV2 [9] using various weight extraction algorithms…….68
Figure 4.7. Testing accuracy of the neural network at every incremental training round when evaluating CUB-200 [5] using SqueezeNet [6] using various weight extraction algorithms…71

Figure 4.8. Relation between the training accuracy of the classifier with respect to the number of iterations when evaluating CIFAR-100 [3] using MnasNet [4] after learning (a) 20 classes, (b) 40 classes, (c) 60 classes, (d) 80 classes, (e) 100 classes……73
Figure 4.9. Relation between the training accuracy of the classifier with respect to the number of iterations when evaluating Stanford Dogs [7] using MobileNetV2 [8] after learning (a) 40 classes, (b) 80 classes, (c) 120 classes.……75
Figure 4.10. Relation between the training accuracy of the classifier with respect to the number of iterations when evaluating ImageNet [1] using ShuffleNetV2 [9] after learning (a) 2 classes, (b) 4 classes, (c) 6 classes……..76
Figure 4.11. Relation between the training accuracy of the classifier with respect to the number of iterations when evaluating CUB-200 [5] using SqueezeNet [6] after learning (a) 50 classes, (b) 100 classes, (c) 150 classes, (d) 200 classes……79
Figure 5.1. Basic high-level diagram of multi-task learning……84
Figure 5.2. Overview of multi-incremental task learning taking places across multiple edge devices.……84
Figure 5.3. Major steps involved in the complete edge device order selection algorithm…....88
Figure 5.4. Accumulated training time when features of images are saved while training….91
Figure 5.5. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 70 classes of CIFAR-100 [3] are first learnt on the MacBook Pro and the remaining 30 classes of CIFAR-100 [3] are then learnt on the Jetson Nano…….97
Figure 5.6. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 30 classes of CIFAR-100 [3] are first learnt on the Jetson Nano and the remaining 70 classes of CIFAR-100 [3] are then learnt on the MacBook Pro……...97
Figure 5.7. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 50 classes of CIFAR-100 [3] are first learnt on the MacBook Pro and the remaining 50 classes of CIFAR-100 [3] are then learnt on the Jetson Nano……..99
Figure 5.8. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 50 classes of CIFAR-100 [3] are first learnt on the Jetson Nano and the remaining 50 classes of CIFAR-100 [3] are then learnt on the MacBook Pro…….....100
Figure 5.9. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 100 classes of CIFAR-100 [3] are first learnt on the MacBook Pro and 5 classes of Stanford Dogs [3] are then learnt on the Jetson Nano……..102
Figure 5.10. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 5 classes of Stanford Dogs [7] are first learnt on the Jetson Nano and 100 classes of CIFAR-100 [3] are then learnt on the MacBook Pro…….…..102

Chapter 1

Overall Thesis Introduction and Layout

1.1. Background

In the computer vision domain, deep learning has shown a great amount of success and in some tasks even surpassing the level of human accuracy [11]. A lot of this success has been obtained in an offline setting whereby all the data is already present on a machine before training starts. Another interesting observation is the rapid development in different types of convolutional neural networks (CNNs) and the classification accuracies produced by each of them. The era of CNNs started with the success of AlexNet [12] in 2012 which had a classification accuracy of 84.7% on ImageNet [1] and merely 3 years later, ResNet-152 [11] was able to achieve a classification accuracy of 96.5%. Apart from the complex model architectures, this success is attributed to two main factors: big datasets and hardware accelerators. Deep learning models are traditionally trained on big datasets and on powerful hardware accelerators such as graphics processing units (GPUs) and tensor processing units (TPUs). and the developed models are then deployed into production.

High volumes of data are produced on edge devices such as mobile and internet of things (IoT) devices [13]. As per Ericsson's report and prediction, 45% of the total internet data will be generated by IoT devices [14]. In such scenarios, it is impractical for such high amounts of data to be transmitted to a centralized machine or a cloud computing system for further processing due to high bandwidth demands and transmission costs. Transmission costs are associated with the dimensionality of the data being dealt with. For e.g., data such as simple temperature sensor values have a very low dimension since each value is simply a floating-point number, however, data such as high-resolution images contain a large number of pixels just for one single image. For such high dimensional data such as images, transmissions costs to a centralized machine are very high. To tackle the issue, edge computing is proposed with the main objective of carrying out all or a fraction of computation on the edge devices itself as an effort to reduce the communication costs and the computation load on the cloud. This inadvertently leads to an increased quality of experience (QoS).

However, in the real world, data keeps evolving and is produced in large volumes. Furthermore, keeping and processing large datasets is computationally expensive. Training deep learning models on large datasets requires a lot of power and computation time. Therefore, the general practice of training a deep learning model statically on large amounts of data is not a good practice and not applicable to most of the real-world cases. The training methodologies can become uncommon in the future due to their nature.

In order to realize Artificial Intelligence (AI) for edge computing, numerous solutions are being developed such as designing efficient hardware accelerators [15], machine learning techniques for reducing memory and computation footprint such as knowledge distillation [16], model pruning [17]–[21], model quantization [19], [21]–[24], software development such as PyTorch [25], TensorFlow [26], etc. While such developments have been great so far, there are not many edge computing techniques that have been established for incremental learning.

Incremental learning is a branch of machine learning whereby models are trained to continuously learn from new data while retaining the old knowledge. This form of learning is inspired by how humans learn since we have the ability to learn new tasks without forgetting the old ones. The biggest objective of incremental learning is to minimize catastrophic forgetting. Catastrophic forgetting refers to a phenomenon seen in deep neural networks in which the networks tend to forget old tasks when learning new tasks because the parameters that are learned for the new tasks affect the parameters that were learned for the old tasks.

We found out that work in incremental learning has mostly been carried out on centralized powerful machines and any new novelties in the field of incremental learning have been thought of from a deep learning point of view. Given the fact that data keeps changing and the fact that large amounts of data are generated at the edge devices, edge computing and incremental learning have to be considered together to realize the true potential of deep learning and edge computing as well. The research progress made by combining both the paradigms are highly limited due to the current trends of incremental learning and edge computing alone. We believe that improving and adapting incremental learning must also be thought of from an edge computing perspective, or vice versa.

As mentioned before that incremental learning has to continuously learn from new data, this means that deep learning models must also be trained and updated continuously. If the training is too computationally expensive to be carried out on edge devices, then it must be carried out on the cloud whereby the training samples need to be transmitted to the cloud. This will produce transmission costs. If the model training takes place on the cloud, then the trained model must be transmitted back to the edge device which will again incur transmission costs.

The main motivation for using edge computing is to reduce the computational load from the cloud. This is performed by offloading a certain amount of intelligence from the cloud to the edge devices which are often the closest to the data sources. This allows the edge devices to carry out local processing and thus reduces the dependency on the cloud to a large extent.

1.2. Problem Statement

Problem 1: Incremental learning is a special case of deep learning where data is being produced continuously unlike traditional deep learning methods where all the data is collected beforehand. Therefore, in the case where data is continuously streaming in, it is impractical to continuously transmit such data to the cloud especially in the presence of multiple edge devices as it can lead to an enormous waste of bandwidth. A solution is required in which an edge device can filter data for training before the training even starts on the cloud. The filtration needs to be completed in a way that the model performance remains significantly unaffected while reducing transmission costs to the cloud considerably.

Problem 2: Once the training has been completed on the cloud, the parameters of the updated neural network are needed to be transmitted back to the edge devices. Due to a large number of these parameters, transmitting all of the parameters back to the edge device can cause a high transmission cost. Additionally in incremental learning, the size of the neural network can also grow with respect to time resulting in an increased transmission cost over time as the network grows. Neural networks are known to have a very large number of parameters that can easily extend to the order of millions. Therefore, a solution that can quickly determine important parameters of a trained neural network and transmit only these important parameters back to the edge devices is required.

Problem 3: Problems 1 and 2 are applicable to a single edge device connected to the cloud. However, multiple heterogeneous edge devices are generally connected to the cloud and are also participating in the training. These multiple edge devices can have access to different data sources from which the neural network must learn. In such scenarios, the incremental learning model must learn from these data samples obtained from different edge devices. Depending on the application, the data samples stored on different edge devices, cannot be allowed to be shared for e.g., records of patients in different hospitals. Therefore, the learning must be done on the edge devices. In the case of multi-task learning, the neural network needs to visit each of the edge devices one at a time as part of the learning process. The order in which edge devices participate in the training contributes to the overall energy consumption across all edge devices because different edge devices can have different processing capabilities. An algorithm is required that can determine the order in which edge devices should participate in the training such that the overall energy consumption across all edge devices can be minimized.

1.3. Aims and Objectives

There are 3 main aims of this research work. The details are as follows:

Aim 1: Reduce the transmission cost when transmitting training samples from edge devices to the cloud.

- Objective 1: Develop an algorithm to automatically select and transmit only a subset of training samples of a class to the cloud for neural network training.
- Objective 2: Ensure that even after applying the data sampling algorithm, the model performance remains the same while achieving significant transmission reduction costs.
- Objective 3: Ensure this solution can be adapted to different CNN models, datasets, and hyperparameters.

Aim 2: Reduce the transmission cost when transmitting parameters of the trained neural network from the cloud to edge devices.

- Objective 1: Implement an algorithm to automatically select useful parameters of a trained neural network without performing any iterative re-training.
- Objective 2: Improve the speed of the FitCNN [10] algorithm which is the baseline neural network parameter extraction algorithm.
- Objective 3: Ensure this solution can be adapted to different CNN models, datasets, and hyperparameters without affecting the performance of the trained neural network.

Aim 3: Study the effect of end-to-end energy consumption across multiple edge devices participating in multi-task learning.

- Objective 1: Demonstrate how multi-task learning can be performed using multiple edge devices without having the edge devices share data with each other or the cloud.
- Objective 2: Demonstrate the differences in the end-to-end energy consumption and the training time with respect to different edge device training orders.
- Objective 3: Develop a basic algorithm that can run on the cloud to determine the order in which the edge devices need to participate in the training so that the end-to-end energy consumption across all edge devices is minimized.

1.4. Research Scope

There are various types of deep learning models such as convolutional neural networks, long short-term memory, autoencoders, generative adversarial networks, deep belief networks, Boltzmann machines, Hopfield networks, deep feedforward networks and more. In this research, we focus on image classification which is a subdiscipline of computer vision. Hence the deep learning architectures used throughout this research are convolutional neural networks and deep feedforward networks.

In the incremental learning domain, there are 2 different types of incremental learning namely: single-incremental-task learning and multi-task learning [27]. Single-incremental-task learning and multi-task learning both refer to learning new tasks over time while retaining knowledge on the previously learned tasks. However, in multi-task learning, output neurons are grouped into separate classification layers whereas, in single-incremental-task learning, all the output neurons are grouped into one single classification layer. In multi-task learning, the weights learned for a given task are usually not changed when the neural network learns new tasks, however, in single-incremental-task learning, the parameters representing the previous tasks can get modified as new tasks are learned.

Any artificial intelligence (AI) problem can be classified into various categories: supervised learning, semi-supervised learning, unsupervised learning, reinforcement learning, meta-learning and more. Generally, supervised learning requires labels for the training AI models, semi-supervised learning requires only a small portion of labels for learning and unsupervised learning requires no labels at all for learning. Large amounts of data are generated via edge devices and the data is generally not annotated. In this research, however, we still focus on supervised learning as the work offers one of the earliest insights: to demonstrate the potential of the combination of edge computing and incremental learning.

1.5. Research Contribution and Outline

There are 3 contributions that are provided in this research which are listed as follows:

- To tackle problem 1, a novel pre-training data sampling algorithm is developed that can discard a certain number of samples from a training distribution on the edge device itself before neural network training starts regardless of whether the training takes place on the edge device or the cloud. This leads to a reduction of transmission costs by 15-23% while keeping the classification accuracy of the model within 2% as compared to using all of the training distribution. **Journal Publication: S. Dube et al.** [28].

- To tackle problem 2, an algorithm that can select important weights of an ANN without iterative pruning and re-training is developed. Though iterative pruning and re-training can be added to prune away more parameters and still retain the model performance [21], this research focuses on how to prune parameters before performing iterative pruning and re-training. Though this has already been done in [10], however, as compared to [10], the weight extraction algorithm presented in this chapter reduces the number of iterations of finding the pruned weights by at least 70%. **Journal Publication: S. Dube et al.** [29].

- To tackle problem 3, a framework is proposed that demonstrates how multi-task learning can be carried out across multiple edge devices without letting the edge devices share any training data with neither the cloud nor the other edge devices. This is shown in chapter 5 which shows that the overall energy consumption of all devices depends on the order in which the edge devices carry out training.

Chapter 2 provides the survey for state-of-the-art incremental learning techniques, edge computing techniques for deep learning, and deep learning techniques for edge computing. This chapter also gives a background for convolutional neural networks, datasets, hardware, and software used in this research.

Chapter 3 provides a novel data sampling algorithm that reduces both the transmission and neural network training costs in the context of class incremental learning. This chapter shows how this data sampling algorithm can perform under various CNN architectures, datasets, and hyperparameters.

Chapter 4 provides an algorithm for choosing the important ANN parameters without iterative pruning and re-training. This chapter is based on [10] whereby the weight extraction algorithm presented in [10] is improved in this chapter in terms of the computation time in finding the same number of pruned parameters of an ANN.

Chapter 5 provides a framework for multi-task learning across distributed edge devices without sharing any training data. This chapter provides insights into the overall energy consumption across all edge devices with respect to various edge device training orders.

Chapter 6 provides a summary of the research, limitations, and future research directions.

1.6. Research Significance

This section describes the novelty and the research significance of the solutions presented in chapters 3, 4, and 5.

Novelty 1: The solution being proposed in chapter 3 is basically performing data sampling on training data distributions. In deep neural network training, data sampling has already been investigated [30]–[34] whereby the goal is to accelerate the training time and also to not use training samples that are no longer contributing to the training of the neural network for e.g., entropy value of a sample is no longer reducing after several training iterations. However, for this to be achieved, training has to first be started. The solution proposed to solve problem 1 is different in the sense that this solution performs data sampling before neural network training starts, the challenge here is to sample data from a training distribution without even a single iteration of both forward and backward propagation and still retain the model performance. As mentioned, achieving success here will lead to a reduced transmission cost and an accelerated neural network training time on the cloud.

Novelty 2: The solution proposed in chapter 4 is a continuation of the work presented in [10]. This work presents an algorithm that selects the useful weights of a neural network after training. This is because not all the weights learned after training are useful in making inferences, therefore, if only the useful weights can be extracted then only these weights need to be transmitted back to the edge devices. This again leads to a reduction in the transmission cost from the cloud to the edge device. The solution presented in chapter 4 performs the same

task as presented in [10], however, the only difference is that this chapter proposes an improved algorithm that is a lot faster in extracting the useful weights from the trained neural network.

Novelty 3: The solution presented in this chapter demonstrates how multi-task learning can be performed across multiple-edge devices without having edge devices share their local data. Research in incremental learning has mostly been carried out on centralized computers [35]–[40] with no emphasis on edge computing especially in scenarios where multiple heterogeneous edge devices participate in training. While the multi-task incremental learning method has already been developed [37], the novelty of this chapter lies in the fact that it demonstrates that the order in which the edge devices participate in the training has a big impact on the overall energy consumption. A basic regression algorithm is proposed in this chapter that can decide the edge device training order. The significance of this is that the edge device training order chosen can minimize the overall energy consumption during neural network training across all edge devices.

Chapter 2

Literature Review and Related Work

2.1 Background

In the real-world, data keeps evolving and is produced in large volumes. Furthermore, keeping and processing large datasets is expensive. Training a deep learning model on a large dataset requires a lot of power and computation time. This is why the general practice of training a deep learning model statically on large amounts of data is not a good practice and is not applicable to most of the real-world cases. Therefore, such training methodologies will become uncommon in the future as such models will have to cope with evolving data.

For real-world applications, incremental learning strategies must be developed in which deep learning models can learn from new data over time. Incremental learning is a strategy of machine learning in which input information is persistently utilized to amplify the existing model's information (further training of the model) [41]. Incremental learning itself has a few challenges and the main challenge is known as catastrophic forgetting [42], [43] which refers to the problem of forgetting old knowledge while new knowledge is learned. The main objective of many incremental learning studies is to reduce the amount of catastrophic forgetting. Another challenge in incremental learning is to utilize the information picked up from past data that can offer assistance to learn new data and how to keep the knowledge from the data that have been learned previously. Generally, researchers group incremental learning into three types: example incremental learning [10], class incremental learning [35], attribute learning.

The best and easiest way to tackle catastrophic forgetting is to carry out rehearsal by using all the previous data with the new data and training the model again. Though this approach alleviates catastrophic forgetting, it is not an efficient solution because it requires the storage of all of the previous data which leads to several problems such as:

1) It is memory demanding to store all of the old previous data.
2) A longer training time because new data mixed along with the old data will lead to a bigger dataset and it only gets bigger as more and more new data is observed.
3) There may be cases where the data cannot be stored in the raw form because of privacy issues. Human beings are known to learn incrementally but do not remember everything that is ever learned. Humans do not need to revise all the old tasks learned in order to learn a new skill, which is why incremental learning models must also be able to learn with minimum or even no rehearsal of data to learn new data and retain old knowledge.

Implementation of incremental learning in an Internet of Things (IoT) framework is an important step because of the challenges of IoT itself. The 4 V's of big data are very well known in the world of IoT which are as follows: Volume, Variety, Velocity, Veracity. The data that IoT devices collect all have at least one of the 4 V's. Volume refers to the amount of data being generated. Variety refers to the different forms of data such as images, video, and social media content. Velocity refers to the rate at which data is being collected by edge devices. Veracity refers to the uncertainty of the data, for example, not all of the data collected by edge

devices contain knowledge. In the context of incremental learning models, the edge devices must be able to learn from large amounts of data and train in the presence of data variety (i.e., a concept drift whereby the distribution of the data changes). For veracity, not all of the data is useful, therefore, incremental learning methods must be able to identify which data should be ignored, either before training and/or during the training because during incremental learning if at any point the learning gets corrupted then the subsequent learning may also get corrupted [27].

In a real-world edge computing scenario, the data that sensors collect are highly dynamic and unlabelled. Given the massive number of edge devices and the fact that each device collects data, it will not be practical to label all the data just to perform supervised learning. This is because data labelling will take a lot of time and the time taken to label the data is directly proportional to the amount of data which in the case of edge devices is huge. Furthermore, even if the data is labelled manually, there is a chance that a number of samples are labelled incorrectly which can affect the final performance of the model [2].

Table 2.1. Classification accuracies of different CNN models evaluated on ImageNet [1] and Snapshot Serengeti [2] Dataset.

	Classification Accuracy (%)		
Dataset	**AlexNet** [12]	**GoogleNet** [44]	**VGGNet** [45]
ImageNet [1]	80%	83%	93%
Snapshot Serengeti [2]	54%	62%	72%

When deep learning models are built using a training set, the same models can face very different data in the real-world environment that can lead to poor accuracy. Speaking from a computer vision perspective, in the real-world, the images may not contain the entire object, varying lighting conditions throughout the day, the lighting may vary even more depending on the weather conditions, objects can be at varying distances from the camera. Table 2.1 shows the accuracy of AlexNet [12], GoogleNet [44] and VGGNet [45] on two different data sets: ImageNet [1] and Snapshot Serengeti dataset [2]. It can clearly be seen that the classification accuracies of each model drastically reduce when inferring images from a more complex dataset with varying lighting conditions such as the Snapshot Serengeti dataset [2].

Figure 2.1 shows a list of topics at a high level revolving around incremental learning and edge computing. The blocks that are highlighted are the topics that are discussed in this chapter, the blocks that are not highlighted are topics that have already been discussed in other review s. This chapter discusses the core incremental learning challenges, data transmission challenges between edge devices and the cloud especially in the context of high dimensional data, and the challenges of applying incremental learning to edge computing. The reason why edge computing and incremental learning are discussed together in this chapter is that majority of the data is collected using edge devices, hence, some form of intelligence must be available on the edge device to process this data without having to transmit all of the data to the cloud. Such data can be evolving in nature as well which is why incremental learning is also discussed as this form of learning can deal with learning new data.

The primary aim of this chapter is to highlight the main challenges of incremental learning and its implementation towards edge computing. A number of concepts will be reviewed to understand incremental learning and edge computing and how they can be merged together. Such concepts include catastrophic forgetting, distributional shifts, memory handling in incremental learning, different types of incremental learning that are supervised and unsupervised incremental learning and their techniques as well dealing with imbalanced datasets, importance sampling, and deep learning for resource-constrained edge devices which are discussed from section 2.3 to section 2.5. Section 2.6 concludes this chapter and Section 2.7 discusses possible research topics that can stem out of the combination of edge computing and incremental learning.

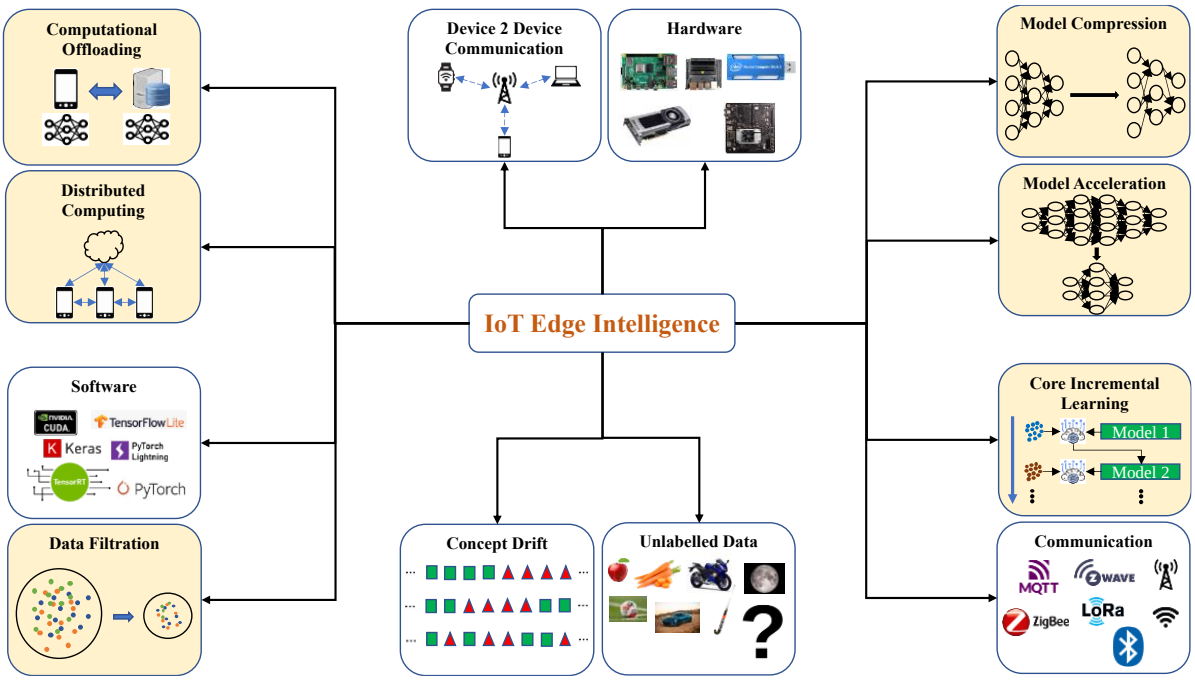


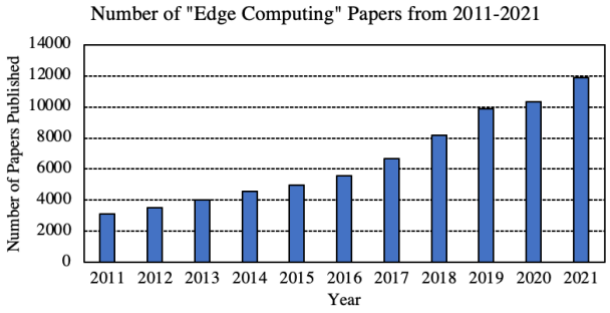
Figure 2.1. High level overview of edge intelligence and incremental learning.

2.2. Search Strategy

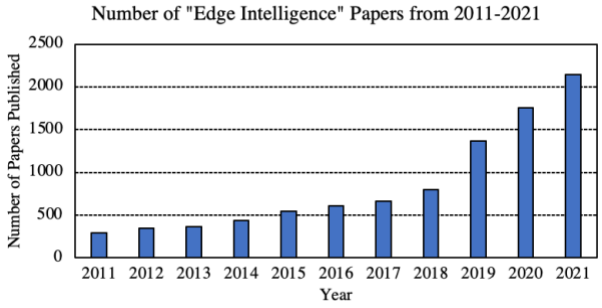
To search for the related s, the Web of Science is used as the search engine. The keywords used are as follows: “edge intelligence”, “machine learning”, “deep learning”, “edge”, “deep learning”, “IoT”, “internet of things”. On the Web of Science, only “review articles” are searched. All of the papers are searched from 2017-2021 (5 years). For obtaining the review s, one of the filters on Web of Science is set to “review articles”. A total of 14 specific and related reviews are found from 2017 to 2019 [15], [46]–[58]. A total of 23 specific and related reviews are found from 2020 to 2021 [59]–[81]. Two recent technical review papers about the confluence of incremental learning and edge computing are also found [27], [82].

As it can be seen that the number of review papers in edge intelligence is a lot more than the review papers about incremental learning and edge computing combined. Furthermore, the number of edge intelligence review papers from 2020 to 2021 [59]–[81] (2 years) are much higher than the review papers published in 2017-2019 [15], [46]–[58] (3 years) which shows the increasing interest in this topic. The review papers about edge intelligence mostly describe the techniques of how machine learning and deep learning can be offloaded from the cloud to

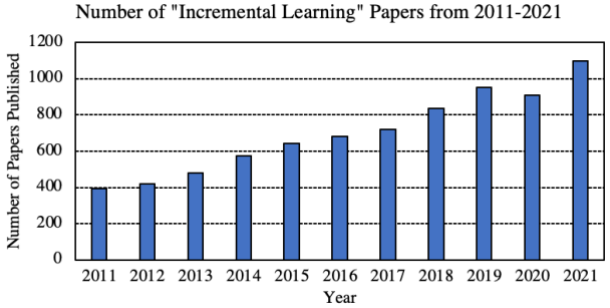
the edge devices, some of these review papers are also inclined towards specific applications such as blockchain, cybersecurity, and smart cities. demonstrating current and future potential of edge intelligence. However, in the case of incremental learning and edge computing, there are currently extremely limited review papers. It is clear that the edge intelligence domain is gaining momentum as there is an urgent need to enable intelligence on edge devices to reduce dependency on the cloud. However, due to the fact that data is hardly static and can change from time to time, AI models must be continuous/incremental in nature, therefore, the next wave of edge intelligence will have to dive into continuous/incremental learning. Based on our search, a good review highlighting the confluence of incremental learning and robotics has been published recently [27] whereby robotic devices can be viewed as powerful edge devices. However, this work talks about incremental learning only in the context of robotics. On the contrary, this chapter discusses about incremental learning in the context of edge computing.



(a)



(b)



(c)

Figure 2.2. Total number of publications over the last 10 years in the domain of (a) edge computing, (b) edge intelligence, and (c) incremental learning.

Figure 2.2 shows the total number of publications in edge computing, edge intelligence, and incremental learning instead of just the review s. The purpose of this figure is to highlight the overall research trend in these topics and to highlight the year-on-year growing research interest in these topics. From Figure 2.2, it can be seen that right from 2011, the number of publications about edge computing has increased year after year most likely due to the increased research output because of the rise in the number of IoT devices. A similar trend is observed in the output of review papers about edge intelligence, however, the number of publications is a lot lesser than that of edge computing. This could be because, for a long time, a lot of emphasis has been placed solely on machine learning/ deep learning development without much resource constraints because these developments have mainly been carried out using powerful servers. The number of incremental learning publications reported in Figure 2.2 are related only to machine learning, deep learning, edge computing, and edge intelligence. The number of publications about incremental learning are a lot higher than that of edge intelligence up until 2018, however, after 2018, the number of publications regarding edge intelligence are a lot higher than incremental learning. Due to these technological trends, a review is much needed highlighting both incremental learning and intelligence at edge devices and how they can be merged together.

2.3. Deep Learning for Edge Computing

Since it is important for deep learning models to move from the cloud-centric servers back to IoT devices, it is also important to make adjustments to the deep learning models to be able to run on devices where memory and power are limited. The main requirements for running deep learning models on edge devices are that the size of the model should be small enough to run efficiently and must also have a fast inference execution speed. Compressing neural networks can enable faster execution and less memory footprint which can be very helpful for edge devices. Table 2.2 shows a high-level summary of the technologies that revolve around deep learning and edge computing.

Table 2.2. High level summary of topics that are related to both deep learning and edge computing.

Work	Summary	References	Potential contributions to incremental learning for IoT
Compression	Quantization of weights, gradients to low bit precision values.	[21]–[23], [83]–[90]	Models with a low memory footprint can allow edge devices to analyse a large number of samples from a stream and look out for data distribution shifts.
Neural network execution speed	Accelerate the inference and training speed of neural networks.	[4], [6], [8], [9], [16]–[18], [20], [91]–[100]	Faster processing of continuous streaming data input.
Parallel and distributed deep learning	Dividing big data and/or models into small chunks	[101]–[105]	Data and model parallelism can be carried out on both the cloud and edge devices. This can be useful when dealing

	and carrying out training.		with multiple sources of streaming data.
Computational offloading	Split tasks between different devices Reduce latency and bandwidth.	[10], [106]–[114]	Monitoring data distribution shifts at edge devices to aid incremental learning.
Data filtration	Accelerate neural network training by using less samples while maintaining model performance	[10], [30]–[34], [106], [107], [115], [116]	Intelligent data sampling at IoT edge devices can lead to cheaper transmission costs to the cloud.

2.3.1. Compression in Neural Networks

In deep learning, compression is mainly used to reduce the size of the model. In the context of transmitting data from the edge device to the cloud, the output of a given neural network layer can be compressed and then sent to the cloud after which it can be decoded. The advantage is that if the output feature of a specific layer is very large then, sending the entire output feature map without compression can be expensive. Another form of compression which is vital for neural network models is known as quantization. The purpose of quantization is to bring down the memory size of the neural network while maintaining or even improving the performance of the model in terms of its accuracy. Since the weights and biases are by default represented using 32-bit floating-point numbers, quantization approximates it by representing the weights and biases using low bit-width numbers. For example, in a standard Artificial Neural Network (ANN), if the first layer has 1000 neurons and the second layer has 1000 neurons as well, then the number of connections between these layers are 1,000,000 so assuming that each weight is represented in a 32-bit floating-point format then the total number of bits that will be used for storing these weights will be 32 million bits. If the weights are represented using an 8-bit floating-point, then the total number of bits to represent the same number of weights will be reduced to 8 million bits (indicating a reduction in storage size by a factor of four). Apart from the weights and biases, activation functions can also be quantized using low bit width. Quantization however can lead to information loss which is why re-training is performed iteratively to recover the accuracy loss.

By default, deep learning software frameworks such as TensorFlow [26] use 32-bit floating-point representation to store a number. Researchers show that weights and activations that are represented using 8-bit format results in only a very slight drop in accuracy [21], [23]. The main advantage of this process is bandwidth and storage reduction. In programming, integer data types are generally faster in comparison to floating-point data types. There is also research being carried out to use weights that are ternary or even binary [23], [84]–[86] because the convolutional layers can be executed without using the multiplication operation where only addition and subtraction are needed. So far, the reason why a 32-bit floating-point format has been used is because of the large range of numbers that it can store.

Most research focuses on reducing the precision of weights as compared to reducing the precision of activations, quantizing the activations almost always affects the accuracy of the final model, when quantizing both weights and activations of a model, DoReFa [22] causes an 8% top-1 accuracy loss and XOR-NET [86] causes a massive 12% top-1 accuracy drop. Some works have proposed quantizing down to 2-bit ternary mode and 1-bit binary mode [23], [84]–

[87]. Due to the presence of specialized hardware platforms such as FPGA, ASIC, GPU, quantization itself does not guarantee benefits by running it on any platform and producing similar performances. The energy efficiency and performance that can be achieved due to this technique highly depends on the hardware.

Quantized neural networks are the solution to creating memory-efficient models. In [22], [88] modified training schemes are proposed. Since quantization is about using fewer bits to represent information, this can also lead to a loss of information thus causing training errors. Therefore, such methods take into consideration the estimation error of the previous layer whereby the quantization depends on the accumulated error. The work in [22] starts with a trained 32-bit floating-point baseline and quantized part of a model at a time after which re-training occurs to recover the accuracy loss that occurred due to quantization. This is done because floating-point multiplication operations can be replaced with binary bit shift operators. This process repeats until all the non-zero weights are quantized.

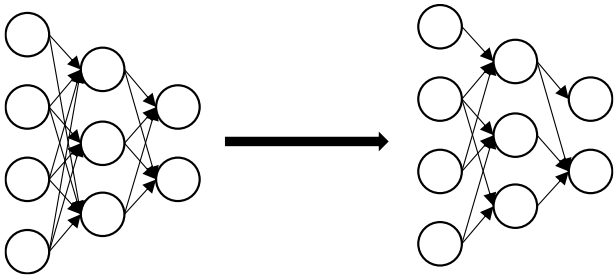
In wide reduced-precision networks, the precision of activation maps are reduced along with model parameters, and at the same time to compensate for the loss incurred due to reduced precision, the number of filter maps are increased in each layer thus making it a wide network(more filters means more features will be extracted) [89]. Though increasing the number of filter maps induces more compute operations, the number of compute bits needed per operation is a lot less than what is required when working with full precision (32-bit floating points). One of the main motivations behind such a development is because activation maps in neural networks consume a high memory footprint. Mishra *et al.* [89] also report that specialized accelerated hardware such as FPGA, ASIC can take advantage of low-precision operations thus providing efficiency gain. GPU however is unable to take advantage of low-precision operation below 8 bits.

Vector quantization for compression of the parameters of the model is used to deal with CNN model storage issues [90]. Vector quantization performs better than matrix factorization in terms of compressing the model [90]. More focus has been put on reducing the storage instead of speeding up the execution of the model which is why emphasis is put on the dense fully connected layers of a neural network which almost always have the majority of the parameters in a neural network for example the model described by Zeiler *et al.* [83] has 90% of their weights in the fully connected layers.

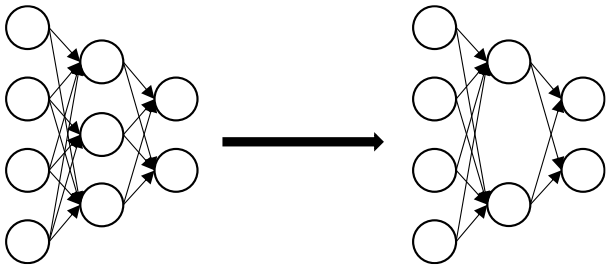
2.3.2. Speeding Up Execution of Neural Networks

Large CNNs such as VGGNet [45] or ResNet-152 [117] have many layers and thus can be very expensive for edge devices to run such networks locally which is why CNNs such as MobileNet [99] & v2 [8], SqueezeNet [6], ShuffleNet V1 [100] & v2 [9], MnasNet [4] are proposed that are efficient forms of CNNs for mobile and embedded device applications. While the architecture of all these networks is different and achieve speedups in their own ways, the most common strategy for reducing computational cost is using techniques such as depth-wise convolution, group convolution, and point-wise convolution, all of which are efficient ways of performing convolutions on images.

Methods such as BranchyNet [91] use early-exit methods which allow for the early-exit for samples that already have high inference confidence before reaching the final output layer of the model. This way, the input sample does not need to be processed by every layer in the model thus saving lots of computational costs. After the model training, the input sample is fed into BranchyNet [91] which analyses the output at each exit point by using an entropy to measure the confidence of the branch. If the confidence is high that is above a certain threshold, then inferencing will terminate at that particular exit branch.



(a) Neural network weight pruning.



(b) Neural network neurons pruning.

Figure 2.3. (a) pruning of non-useful weights in a neural network and (b) pruning of non-useful neurons in a neural network.

Knowledge distillation [16] is another common method of coming up with a smaller neural network. The intuition behind this approach is to train a small model that is capable of mimicking a large model that has a high accuracy. This system replicates a teacher-student setting whereby the teacher is the large model, and the student is the small model that is to be trained by learning from the teacher, the student model is also referred to as the distilled model. The teacher model transfers the knowledge to the student model by the way of minimizing a loss function [106]. The cross-entropy is calculated between the student and the teacher model and this is then added to the loss of the student model. Using this overall loss function, the student model is thus trained via backpropagation. Using this approach, the student model not only learns from its own ground truth labels but also from the loss output by the teacher.

The work in [92]–[95] increases the neural network training speed by varying the learning rates throughout training (adaptive learning rate). It is also possible to cut away certain connections between neurons because after training there may be some connections that do not carry much weight and contribute to the final output, thus when certain connections are pruned, a smaller number of multiplications need to be carried out [17], [18], [20], [21] thus increasing the execution speed of neural networks. Figure 2.3 shows a basic representation of pruning in neural networks.

Redundancies are observed in convolutional filters/kernels [96] that are then used for speeding up the test-time evaluation of CNNs. Close approximations of the kernels are formulated so that it reduces computational costs. Denton *et al.* [96], achieve a speedup by a factor of 2x is achieved when running convolutional layers on both CPU and GPU while keeping the accuracy within 1% of the baseline accuracy. It is also shown that such methods can reduce the memory footprint in the first two layers by 2-3x and 5-13x in the fully connected layers via singular value decomposition. Similar works such as [97] that carry out the convolution operation in the Fourier domain using Fast Fourier Transform (FFT) achieve a speed-up factor of 2x and work such as [98] employ low-rank decomposition of tensors to speed up the evaluation of CNN. [97] also provides a method to compress the first convolutional layer which has been reported many times to be highly sensitive. It is also reported that if [97] is combined with quantization methods and/or operates in the Fourier domain, it can lead to further performance gains. [98] presents a scheme that achieves CNN speed up by exploiting cross-channel or filter redundancy to construct a low-rank version of filters that is a rank-1 filter in the spatial domain. They report a 2.5x speedup with no accuracy loss and a speedup of 4.5x with 1% accuracy loss by using a pre-trained CNN.

2.3.3. Parallel and Distributed Deep Learning

The main purpose of this research area is to accelerate deep learning training. Due to the rise in the amount of data and the complexity of the deep learning models themselves, high-performance computing is often needed in order to train large amounts of data which is time-consuming. In order to accelerate the process, model training must be parallelized.

There are two main forms of parallelism: data and model parallelism. In the former, data is distributed across more than one machine and in the latter, the model is split across multiple machines [101]. For incremental learning, these techniques can be beneficial when dealing with very large amounts of streaming data and/or incrementally learning new concepts from various data streams parallelly.

Federated learning [102] is an area of research in decentralized training that is using the local data on the edge devices. There is only one common model shared on the edge devices. Each edge device will then train the model using the local data and only the updated model from each device is sent to the cloud for a global model aggregation. The aggregated model is then again sent back to the devices for further learning. Bonawitz *et al.* [102] greatly reduces communication to the cloud and also ensures data privacy by training the model on the device and only transmitting the encrypted updated model to the cloud instead of sending the data to the cloud. However, there are a number of downsides if Bonawitz *et al.* [102] is to be applied in the incremental learning context. For example, the training time taken on the edge devices when learning patterns from high dimensional data such as images will be longer. Gao *et al.* [118] report that even for a simple dataset like CIFAR-10 [3], it takes 8 hours 41 minutes to train a MobileNetV1 [99] for 1 epoch on a Raspberry Pi which is considered a high-end edge device whereas if the model was split between the Raspberry Pi and the cloud whereby some layers are run on the edge device and some on the cloud, the training time for 1 epoch was reduced down to 2.5 hours which is why when dealing with high dimensional data such as images, local training of the entire model may not be feasible due to the sheer processing power that is required for training which can only be met by powerful hardware such as the GPUs which generally reside on the cloud.

Generally, there are two main challenges when dealing with distributed deep learning: the first is the communication between multiple computing nodes which causes overhead, and the second is the modification of the optimization algorithm which many researchers tend to avoid. Figure 2.4 shows a basic representation of distributed intelligence. Communication is an issue because of the presence of multiple compute instances that are needed to carry out some processing and for this to happen, the information must be passed from one machine to another which causes an overhead. The modification of optimization algorithms is challenging because of the synchronization of the optimization across multiple compute instances instead of only one single machine.

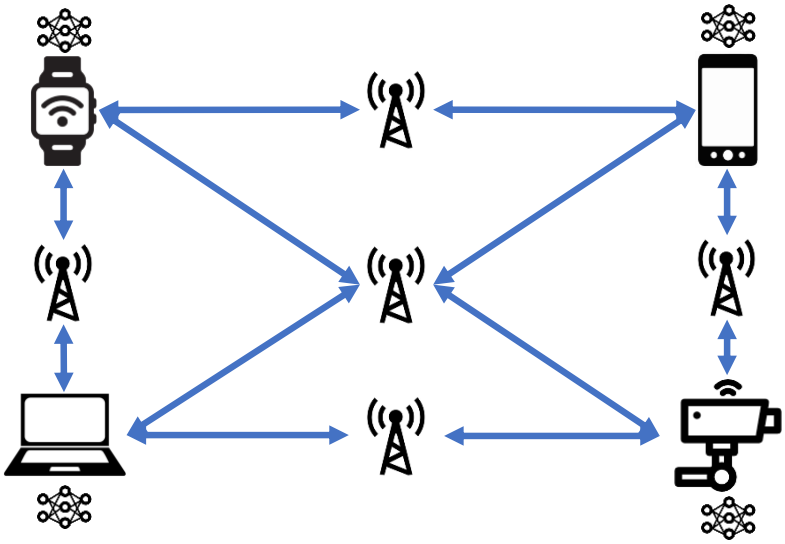


Figure 2.4. Representation of distributed intelligence across various edge devices.

The work in [93]–[95] present modified optimization algorithms in order to train deep neural networks in a distributed manner. Frameworks are also present for distributed training that enables the users to modify only a few lines of code to enable distributed training by enabling efficient communication between GPUs [105].

2.3.4. Computational Offloading

When dealing with deep learning, high dimensional data, and large volumes of data, it may not be possible for edge devices to independently carry out model training due to the sheer demands of model training. It becomes very important to make use of a platform that hosts powerful hardware accelerators to carry out the overall training process.

Several methods rely on the cloud for training their model [10], [108]–[110]. This leads to another research area that has received a lot of attention and is known as computational offloading. Computational offloading is mainly used to transfer computationally expensive tasks to powerful external platforms such as cloud, hardware accelerator, grid, or cluster. There are two types of computational offloading methods: static offloading and dynamic offloading. In static offloading, each platform carries out specific tasks for example, in the case of deep learning, this means running a fixed set of layers of a model at each device involved in the entire system [10], [106], [107]. However, in the case of dynamic offloading, it is a much smarter way of deciding which platforms run expensive tasks and which platforms run easier tasks. Certain tasks are carried out by either the edge or the cloud or other external platforms

depending on various variables such as device resources, available bandwidth, network latency, DNN model size, and data size [121], [122].

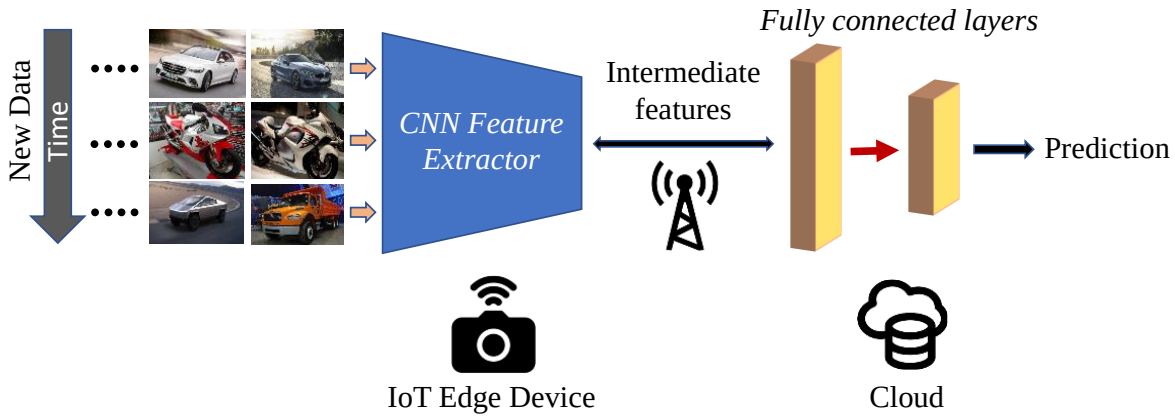


Figure 2.5. Incremental learning system partitioned between an IoT edge device and cloud.

In the context of deep learning, computational offloading means running certain layers of the deep learning model on the IoT edge device and the other remaining layers on a server [10], [107]. Figure 2.5 shows a basic scenario of an incremental learning system in an offloaded setting. Edge devices are typically used for feature extraction of data and the servers (typically residing on the cloud) are used for model training. To put it simply, computationally expensive tasks are run on the cloud whereas the edge devices are used for reducing the computational load on the cloud [111]–[114].

Systems with computational offloading mechanisms are ideal for incremental learning because edge devices can then be used for tasks such as feature extraction, detecting drifts in data [123]–[126], and data filtration. The cloud can focus on model training. However, real-time algorithms must be designed properly, especially for resource-constrained edge devices. Such algorithms must also be unsupervised in nature because raw data that arrives at edge devices, most probably are without labels.

2.3.5. Data Sampling

For any deep learning model, the majority of the learning takes place during the initial training phase after which a number of samples could be ignored, and the deep learning model does not need to be trained on those particular samples. Not only does this method result in unchanged accuracy of the model being trained but this also leads to accelerated training. In the context of incremental learning, the reason why data filtration is important is not so much because of accelerated training but more about the stability of the learning of the incremental learning process. Since incremental learning learns in a streaming manner, it is very important that the learning does not get corrupted at any incremental learning stage otherwise the subsequent training stages may get corrupted as well [27]. Consequently, at every incremental learning stage, any sample that is no longer contributing to the improvement of the model should be discarded.

Usually, a training set consists of several training instances but some of them may not be not be useful for further processing, therefore, it is possible to obtain acceptable model performance without such samples and this process is known as instance selection. There are

2 types of instance selection methods: wrapper and filter methods [127]. Wrapper methods are based on accuracies obtained by classifiers, therefore, instances that do not contribute to the overall accuracy are discarded. Filter methods rely on a selection function which are not based on classifiers. Some of the most common wrapper based methods are as follows: Support Vector Machines (SVM) are known to be instances selectors since only the support vectors are used to discriminate among classes, SV-kNNC (Support Vector k-Nearest Neighbour Clustering) [128] applies SVM over the training set and then uses the k-means algorithm for clustering the support vectors and retains only such instances belonging to homogeneous clusters (clusters where all instances belong to the same class). k-means algorithms are also widely used for instance selection based on clustering. A common way of doing so is to run an unsupervised k-means clustering algorithm on the training instances and given a selection rate, instances can be selected from the centres and the borders of the clusters [129]. Filter based methods are commonly based on clustering where one idea consists of splitting a training set into a few clusters and the selected instances will be the centres of the clusters [130]. The same-class nearest clusters can be merged and then the centres from the new merged clusters can be selected [131].

Kawaguchi *et al.* [115] propose a technique known as Ordered SGD whereby once the training accuracy is higher than a certain threshold which means that the model now understands the general dataset well, from that point onwards, they choose the top-q samples with the highest losses where q is the size of the dataset at the beginning of training and as the training accuracy continues to improve, the size of q is reduced by a factor of 2. Average top-k loss is also used by [116] however, it is unusable for deep learning and similar other non-convex problems. The work presented in [30]–[34] all commonly perform stochastic gradient descent with importance sampling (select only the useful samples for backpropagation).

In the edge computing context, data sampling can be performed both on the edge device and the cloud. Most of the data sampling takes place during training the model as mentioned and thus this process can take place on the cloud, however, there are a few research works that can be used to perform data sampling on the edge device as well such as [106], [107] whereby in [26], a boss model which is the original deep neural network and an assistant model is also trained which is a lightweight version of the boss model. The assistant model is the key in [26] whose role is to perform importance sampling that deduces the important samples with respect to the current boss model. Since loss computation can be an expensive process, the assistant model is lightweight so that it can infer the importance quickly at a cheaper cost and also filter out unwanted samples based on a certain loss threshold, which is adaptive with respect to the boss model. Although [26] has not been applied in an edge computing environment, however, the assistant model could be run on the edge device and the boss model could run on the cloud, the key challenge will be the communication of the weight updates from the cloud to the IoT edge device. Liu *et al.* [10] propose a method for data sampling on the IoT edge device in an incremental learning scenario. The method only learns the same task incrementally therefore the data filtration method is simple. If the sample of the same task on the edge device results in a low inference confidence, it implies that the model does not understand the sample properly and thus the sample is sent to the cloud for further training. If the sample confidence is more than the defined threshold then there is no need to transmit the data to the cloud.

Song *et al.* [107] propose an incremental and unsupervised training method that can tackle big raw IoT data. In the cloud, an unsupervised neural network is first trained and using this network, transfer learning is performed to create a target inference network. For example, after training an unsupervised neural network, the first n layers of the network are transferred to the

first n layers of the target network, this target network then only needs to train the remaining layers with a limited amount of labelled data. The target inference network is deployed to the edge device for carrying out predictions and the unsupervised network is also deployed to the edge device for performing data diagnosis. In an incremental learning setting, the new data is referred to as the training data because it is being used to improve the knowledge of the model thus in this case, the training data is split into two categories: recognized class and unrecognized class. To reduce the data movement between edge device and cloud, the deployed unsupervised network on the edge device checks whether the new data can be correctly recognized or not. The edge device (IoT device) sends the data to the cloud for further incremental learning only if the data is not recognizable and this is how data movement is reduced between the edge device and the cloud.

2.4. Types of Incremental Learning

Incremental learning refers to machine learning models which have the ability to continuously learn from new data and improve the overall model representation. Single-Incremental-Task learning refers to learning from multiple classes but using one a single joint classification layer. This means that when new classes are learnt, the weights associated with old classes can be modified. Multi-Task incremental learning refers to learning completely new classes but using new dedicated classifiers for each of the tasks for e.g., learning to classify cars first and then learning to classify airplanes [27]. All of these techniques exist in the form of supervised, unsupervised, and reinforcement learning.

2.4.1. Supervised Incremental Learning

In supervised learning techniques, labels of training data and test data are given and known. A lot of incremental learning research has been conducted with labels of the tasks already known [35], [38], [39], [132]–[134]. One major downside of this technique is in its application in edge computing. This is because it is not feasible for humans to manually annotate large amounts of raw data collected by edge devices. There is a demand for semi-supervised or even fully unsupervised incremental learning algorithms to solve these challenges.

2.4.2. Unsupervised Incremental Learning

There has been much development in this area but not as much as compared to supervised incremental learning. In this type of model learning, training and test data do not have labels, which is well suited for edge computing and Big Data applications. In unsupervised incremental learning, there is a heavy usage of generator networks for generating artificial images/data for the following reasons: fewer data, imbalanced dataset, privacy concerns. When there is less data or imbalanced datasets, generative networks can produce more samples to increase the size of the dataset and balance out the dataset, respectively. For privacy concerns, in some applications, the data cannot be allowed to be stored because of privacy concerns therefore artificial samples are generated for rehearsal thus eliminating privacy concerns. Unsupervised deep learning models can also be used for classification mainly through the use of autoencoders [27], [36], [135], [136].

For edge devices, unsupervised methods can be used for monitoring concept drift in data distributions. Correctly detecting drifts in data is very important for incremental learning

because failure to do so results in degradation of the model learning performance [27], [137]. Unsupervised concept drift detection is an exciting research area and lots of work has been conducted in this area [123]–[126] using various clustering methods and also using one-class learning. However, there is not much work available in unsupervised concept drift detection mechanisms for high dimensional data such as images.

2.4.3. Incremental Reinforcement Learning

Reinforcement learning (RL) is a learning model where the model takes action in an environment, the updated parameters will be based on reward and penalty points. The learning is carried out by maximizing cumulative rewards i.e., rewards for taking correct actions and penalties for taking wrong actions. The reason why RL is mentioned is because RL environments do not have access to all of the data before training. Data continues to arrive over time based on the RL agent taking actions after which rewards are computed therefore this learning paradigm can be compared to incremental learning in which case data also arrives over time. RL has mostly been applied in games and also in [138]–[140]. RL has two main challenges, firstly, performing relevant extraction of information from input data. Secondly, policy learning for solving specific tasks. Policy learning is a much more challenging task in the context of embedded agents [141], [142].

2.5. Challenges of Incremental Learning in Edge Computing

This section discusses the challenges that arise when incremental learning is integrated with edge computing such as dealing with high dimensional data (i.e., images), transmission load between the edge devices and the cloud, training without labels, and novelty detection i.e., the ability for the models to detect new data to reduce/eliminate catastrophic forgetting.

2.5.1. Catastrophic Forgetting

Catastrophic forgetting is a problem of forgetting old knowledge while learning the new knowledge and is the main problem when learning incrementally using deep learning models and has been a long-standing research problem [42], [138], [143], [144]. Lesort *et al.* [27] state that there are three main approaches to tackle catastrophic forgetting: the first is through dynamic architectures which means the original architecture at the start of the training continues to get modified throughout the incremental learning phase, the second approach is regularization and the third and the most common approach is rehearsal. These approaches are presented in more detail below:

1) *Dynamic Architectures for Alleviating Catastrophic Forgetting*: Most commonly, the model for the old task is saved and a new model is added to the overall network for learning new tasks [37] without forgetting by not changing the model of the previous task. This method is immune to catastrophic forgetting, but it is highly memory-demanding because a new model has to be created to learn a new task. With the addition of a new model per task, there is also a huge increase in the number of parameters per task. [35], [132] use additional neurons in the final classification layer to learn a new task, compared to [37], this approach yields a lot fewer new parameters per task. These works however do not address that there must be a finite upper limit to the number of parameters since the model cannot keep growing indefinitely due to hardware memory

constraints regardless of any hardware. Dual architectures are also another technique in this domain. [145] makes use of two growing neural networks whereby both the categories and the category instances are learnt. [36], [146] use short term memory modules to replay the information into the long term memory modules that often contain the main neural network model, [146] however requires the storage of the entire dataset during training, and [36] uses generative replay as part of rehearsal instead of saving all of the previous data in its raw format. [147] also presents similar dual memory architecture that also uses pseudo-rehearsal for memory replay with the help of a variational autoencoder. Other types of dual architectures are also reported in [148], [149].

2) *Regularization Approaches for Alleviating Catastrophic Forgetting*: Incremental learning methods tend to focus more on the new task being learnt therefore, there is more focus on new tasks compared to the previous tasks which is what causes forgetting, Dropout [150], L2 regularization [151], early stopping are some of the common regularization techniques that are used to prevent the neural network from overfitting by not allowing weights to become too dependent on features. Similarly, in the context of incremental learning, regularization approaches are typically used to reserve weights in order to retain prior knowledge thus reducing catastrophic forgetting. Regularization approaches usually apply changes to loss functions whereby regularization terms are added to loss functions that will ultimately penalize changes of weights corresponding to old tasks [38]. Zenke *et al.* [152] have a similar approach whereby synapse importance is computed, and the most relevant synapses of a neural network are penalized so that new tasks can be learnt without affecting the old learnt tasks. Since the parameters of a model are modified by computing derivatives of the loss with respect to the parameters and modifying the weights of the model based on this gradient, [153], [154] provide a similar but slightly different regularization technique whereby forgetting is alleviated by using this gradient to modify certain weights by a larger degree as compared to other weights as a measure to reduce catastrophic forgetting.
3) *Memory Rehearsal for Alleviating Catastrophic Forgetting:* This is one of the most common approaches to tackle catastrophic forgetting whereby samples of the past tasks are saved and used together with the new samples for learning. The previous data can be saved in raw format or only features of the data is stored. This way new knowledge is learnt along with old task whenever a new task needs to be learnt. The biggest disadvantage of this approach is that memory needs to be reserved for most or even all of the previous tasks learnt which not only is memory demanding but the training time keeps increasing as more and more tasks need to be rehearsed. This can also introduce privacy issues. A selected number of exemplars for the previous tasks for rehearsal are kept [35]. In order to solve two of the above mentioned problems that is data privacy, saving original images of old tasks, generative networks are used. Generative models learn for a given distribution and are used to generate artificial samples based on that knowledge, the target is to produce artificial images that are almost indistinguishable from the original data distribution. In the context of incremental learning, this can be used to generate more data in the case of imbalanced datasets, generate artificial samples based on the original distribution and use them for rehearsal instead of saving original old data. Generative Adversarial Networks (GANs) [135], [155]–[157] and autoencoders [36], [147], [158]–[160] are used to perform regeneration. Apart from unsupervised feature representation learning, generative models can also be used for classification for example [161] use an autoencoder as a classifier in their incremental learning model. The autoencoder is trained on each task after which the samples for

each task are passed only through the encoder to obtain latent vectors and a mean is then computed as a way to represent the task for classification purposes. Saving original data from old classes for rehearsal for incremental learning can lead to a privacy issue which is why generative models such as GANs are used to learn from such data and then generate artificial samples that are similar to the original data distribution. This way, the old data no longer needs to be preserved thus mitigating privacy issues.

2.5.2. Dealing with High Dimensional Data

In the computer vision domain, it is known that the better the resolution of images, the better the accuracy of the final model because high-resolution images contain much richer information compared to the same image represented with a lower resolution. Though increasing the resolution of images will result in better model performance, however, it will create a problem for resource-constrained edge devices. Suppose the image resolution is high, it means that the spatial dimensions of the images are higher. The higher the spatial dimensions, the higher the processing power consumption as more convolutional, pooling operations are needed. This leads to learning being highly time-consuming where not only will it be necessary to make sure that the edge device does not run out of power but also hardware failure does not occur. Computer vision itself has many other applications such as visual tracking [162], [163], object recognition [164]–[166].

2.5.3. Transmission Costs Between The Cloud and The Edge Devices

This challenge is directly brought into light with respect to the dimensionality of the data being dealt with. Suppose the data is high dimensional, it is very unlikely that the training can be performed solely on edge devices which is where cloud assistance becomes necessary. This is because the cloud can perform heavy computation much better than an edge device. Even after forward propagating such data through a CNN feature extractor on edge devices, it still yields a feature map that has large spatial dimensions. This is challenging especially if there are a large number of images to be transmitted to the cloud [167], [168]. One solution that can be adapted is to continue storing the feature maps of images on the edge device and instead of sending feature maps one by one to the cloud, all of the stored feature maps can be transmitted to the cloud at once at regular intervals so that multiple connections need not be made. However, the random-access memory (RAM) limitation of the edge device is another issue that must be taken care of. When RAM becomes scarce, there is a slowdown in the computer speed because the OS will allocate RAM space to resource-hungry programs. However, when RAM space starts to run out, the OS will move some of the program's memory to disk thus the OS now needs to move the data more frequently between disk and RAM. Accessing a disk is slower than accessing RAM.

Here a dilemma arises, it can be seen that sending feature maps to the cloud frequently will not cause RAM scarcity, but it will cause large transmission loads. On the other hand, storing multiple feature maps on the edge device will ease out the transmission loads but will cause RAM scarcity on the edge device and cause the overall program to slow down. Therefore, it is important to strike a balance between the transmission load and RAM scarcity of edge devices. This concept is one of the factors related to offloading scenarios [169]–[175].

2.5.4. Training Without Labels

There are a number of things that embedded edge devices are expected to do such as exploring the environments on their own, knowing which tasks to learn and which ones to ignore. Edge devices must be capable of knowing when the task changes. There are a few existing approaches with embedded devices that are capable of exploring environments on their own [176]–[178]. For every task to be learnt, the data can be collected in a stream-based manner and then be used for training. The problem arises when it comes to labelling the data samples. Collecting raw and unlabelled data can result in a large number of samples being collected and the larger the dataset, the more time-consuming it would be to label each and every sample especially when there is a limited labelling budget.

Figure 2.6. illustrates unsupervised learning in an incremental setting. In cases where tasks appear in a sequential manner then the only thing that must be taken care of is the detection of a new task because in this case a task label can automatically be assigned to a particular task and the label can be incremented for the next task. For example, for the first task, a label of value 1 can be assigned and when a new task is detected then a label of value 2 can be assigned to it, and so on. Here, the only challenge of labelling instances in sequential task learning is the detection of the new task. However, if lots of samples from various classes are collected in an unordered manner, the process of labelling can get more complex, and this is where a human annotator may be required.

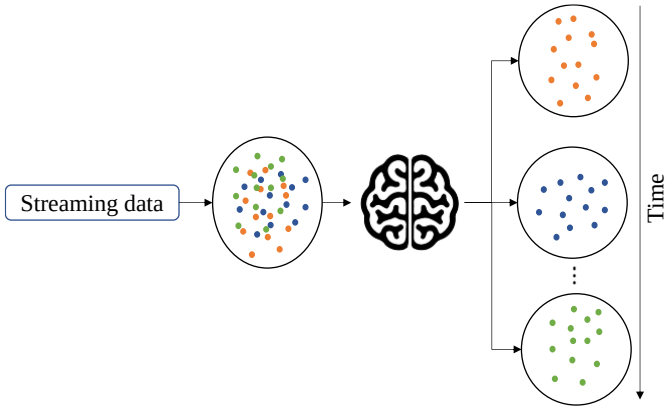


Figure 2.6. Unsupervised learning of raw unlabelled streaming data for incremental learning.

Active learning is a research area in the machine learning field that aims to learn from a large number of unlabelled samples with only a few labelled samples/data [179]. This is a research area that is not very much looked into but holds the key to unlocking the potential of learning with large amounts of unlabelled data, especially in the context of edge computing. Active learning is a method for identifying samples that are in the most need of labels which can improve the learning of the model. There are a number of ways that this is performed. For a simple example, active learning is achieved by training a model on some amount of labelled instances and the model then makes predictions on the unlabelled samples. If the model is very confident about a given sample, then the prediction made by the model can be considered as a label for that instance but if the model is not confident about the sample after making the prediction, then this is where the sample must then be labelled by the human annotator. Active learning still needs a lot of work especially for algorithm development in the case of incremental learning where new data continues coming without labels.

2.5.5. Dealing with Data Distribution Shifts

In the real-world, concept drift is a common scenario whereby the data distribution changes over time. A simple example is facial recognition as human beings do not look the same forever. So, a good facial recognition system must adapt to the changes in the facial structure over time in an incremental manner. The facial recognition system considers changing data of the same task since the face remains. However, in the case that category of the data can change over time, it means that the system will deal with a completely novel class/task and this also falls under distributional shift. There are a few approaches such as [180], [181] that are used to detect changes in the distribution. Figure 2.7 shows a scenario of sudden concept drift whereby completely new concepts arrive over time. There are various types of data drifts such as sudden drift, gradual drift, incremental drift, and recurring drift, all of which are explained in detail in [182].

Deep learning models are not usually capable of computing uncertainty of input data, for example, if a given model is trained to classify cats and dogs at a very high accuracy, if an input is fed that is very far away from the original training data distribution then deep learning

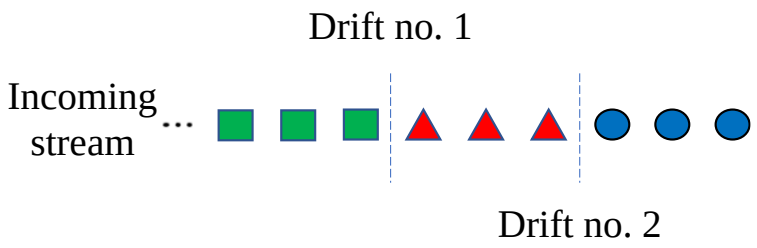


Figure 2.7. Sudden concept drift (observed in multi task incremental learning).

models still go ahead and make a prediction on the sample. For example, if the input image is of a laptop and this sample is fed into a model that is trained to classify cats and dogs, then the model may still make a prediction with a high confidence pointing the input to one of the outputs. This should not be the case, instead, the model should be able to output an uncertainty measure that indicates that this model has not been trained on such samples. The reason why this capability is so important especially in the context of incremental learning in edge computing is because of the change of distribution of data as mentioned. If an IoT edge device that has a deep learning model encounters a new task, then with the help of uncertainty measure features, the device should be able to detect that a new task has been detected. There are a few methods that have been developed to compute uncertainty for classification and regression-based tasks [183]–[187]. Monte Carlo dropout is the most commonly used technique to estimate uncertainty. Dropout is used both during training and testing time. It can be seen as creating an ensemble of models without instantiating separate models in memory. Neurons are randomly deactivated to create a new parameter distribution at every forward pass. Therefore, in Monte Carlo dropout, when an input arrives, forward propagation takes place on the model a few times to obtain a confidence score at every forward pass. Using the average of all the outputs, if the variance/entropy is very large then this indicates high uncertainty and could indicate a new sample.

In incremental learning, when real concept drift occurs, that is a scenario where a completely new task is observed and must be learnt. The IoT edge device itself must be able to detect

whether a change has occurred or not. One of the most common approaches to solve this problem is to perform cluster analysis which is a technique that is used to group similar data together. There are numerous clustering techniques applied in deep learning to group data [188]–[192]. Clustering algorithms must have access to a fairly large amount of data in order to properly learn representations to differentiate new incoming samples from all of the data to perform grouping. This becomes an issue when dealing with both resource-constrained devices and high-dimensional data. Clearly, not all of the incoming data can be stored on the IoT edge device just for the sake of clustering. Researchers must develop a way to effectively select the most important set of data samples from streaming data which can then be used for comparing clusters. Clustering-based methods usually involve computing distances between data points that can be computationally expensive in the case of high dimensional data and a high number of data samples.

Other alternative methods for performing data drift detection is by detecting outliers. One way to do it is by making use of autoencoders. Autoencoders can be used for reconstructing the input samples accurately. If the new data distribution is similar to the data that the autoencoder is trained on, the reconstruction error will be small. However, for a novel sample, the reconstruction error can be quite high in which case it can be deduced that a novel sample has been encountered [193]–[198]. Another issue is for fast streaming data, the classifier must quickly be updated as soon as a drift is detected and to conduct such tasks using neural network based methods on IoT edge devices may not be feasible.

2.5.6. Hardware

The majority of the progress that has been made in the field of incremental learning has been conducted on a centralized computer with hardware accelerators. With all the hardware accelerators provided by cloud service providers, it is easy to make use of powerful GPUs available, and also the cost is dependent only on the amount of usage. Such features are very user-friendly in terms of carrying out experiments. However, buying hardware can be expensive and along with this, the failure of embedded devices is another reason why it is difficult and time-consuming for researchers to carry out experiments on such devices.

Incremental learning is highly computationally expensive. Such tasks simply cannot be conducted on resource-constrained devices which is another reason why the roles of edge devices have not been investigated in the field of incremental learning.

2.6. Summary

The importance of this chapter is highlighted in Figure 2.1 whereby it can clearly be seen that though there are a large number of papers that are related to either incremental learning or edge computing, there are extremely few papers that highlight the applications of incremental learning in edge computing. As seen in this chapter, research is being done to come up with lightweight models that are more suitable for embedded devices. New hardware devices are also being developed to support running deep learning models on edge devices. When dealing with IoT-generated data, using incremental learning becomes mandatory because IoT-generated data keeps evolving. When researchers want to apply incremental learning in edge computing, they must be aware of the technical components of this research area that focus on integrating deep learning with embedded systems such as model compression/quantization, techniques for speeding up inference of deep learning models, and parallel/distributed

computing in deep learning and computational offloading (all of which are discussed in this review). This chapter also discusses the different types of incremental learning strategies i.e., supervised, unsupervised, and reinforcement learning. Out of all the three, a lot of work has been done on supervised incremental learning. This chapter also studies the challenges of applying incremental learning to edge computing such as dealing with high dimensional data, transmission costs of data between the cloud and IoT edge devices, training without labels, unsupervised data drift detection. In an edge computing context, the high dimensionality of data and transmission costs are related challenges because the higher the dimensionality of the data, the better the performance of the model but the larger the transmission costs in cases where both the edge device and the cloud are involved which is why more research needs to be carried out to tackle these two challenges. Unsupervised novelty detection is also discussed because in edge computing environments, embedded systems that are very close to the data source have very small or even no supervision which calls for the need for the ability to deal with data without labels at the edge. Furthermore, the IoT edge devices must be able to detect novel data without labels.

2.7. Research Direction And Problem Interdependency

All of the subsections above provide a general overview of the topics that lie at the intersection of edge computing and deep learning. After an extensive literature review, we conclude that in terms of individual contributions, the fields of edge computing and deep learning are producing very high research output separately. However, the field of incremental learning is relatively new compared to the other research topics in deep learning. Due to the fact that incremental learning is still in a research phase, a lot of the experiments in incremental learning are being carried out on centralized machines that are typically servers with specialized hardware accelerators. There is limited extensive research that combines both incremental learning and edge computing. This chapter provides an insight into the mixture of these technologies and how they can work with each other. Table 2.3 shows some of the research works that have tackled the problem of performing incremental learning in an Edge-Cloud setting.

Table 2.3. Summary of the research works focusing on incremental learning and IoT.

Reference	Application	Objective	Architecture	Dataset modality
[10]	Image classification	- Accuracy - Latency	Edge-Cloud	Images
[107]	Image classification	- Accuracy - Training time	Edge-Cloud	Images
[199]	Abnormal object detection of power transmission lines at the edge	- Reducing bandwidth - Latency - High accuracy	Edge-Cloud	Images
[200]	Advanced Metering Infrastructure Energy data stream processing	- Training time	Edge-Cloud	Power demand data

		- Inference time			
[201]	Smart traffic management	-	Accuracy	Edge-Cloud	Vehicle MAC ID, location, time

While all of these works are very good attempts into performing distributed incremental learning, however, these works majorly focus on single incremental task learning whereas the research presented in this thesis focuses on multi-task incremental learning. Based on all of the research gaps observed when thinking about combining edge computing and incremental learning, the problem this research tries to solve is reducing the communication costs between edge devices and the cloud which involves reducing the number of samples being transmitted from the edge device to the cloud and it also involves reducing the model size being transmitted from the cloud to the edge devices as mentioned in Chapter 1. These challenges will be addressed in Chapter 3 and Chapter 4.

The scope of this research is as follows: throughout this research, a CNN will be used as a deep learning model and the CNN will be trained only for image classification purposes. The pre-trained feature extractors of CNN models such as SqueezeNet [6], MobileNetV2 [8], ShuffleNetV2 [9], etc. will be used for extracting the features from incoming images, and only the classifier which is in the form of fully connected layers will be trained. This is because training an entire CNN on edge devices can be highly computationally expensive which is why the weights of the pre-trained feature extractors will not be updated on the edge devices. Throughout the remaining chapters, only the CNNs designed for embedded devices/mobile devices will be used since deep learning models are placed on the edge devices for all of the work in this research.

It is very critical to understand how the problems being tackled by this research are linked to each other, which is why the problem interdependency is listed out as follows:

1. As mentioned that due to the nature of evolving data and the rise of IoT edge devices, the communication costs and the computational load on the cloud increases. Therefore, the first problem to tackle in this scenario is to reduce the transmission cost from the edge device to the cloud. This is because transmitting less information to the cloud reduces the transmission costs and it means the cloud will have less information to process as well thus easing the computational load. This is why a pre-neural network training algorithm is being investigated in this research that can samples training samples at the IoT edge device itself in order to achieve transmission cost reduction and computational cost reduction. However, solving this problem leads to another problem as mentioned below.

2. In terms of neural network training, after achieving a reduced transmission cost, the cloud still needs to carry out neural network training using fewer samples. Though this helps with reduced computational cost, however, after the training is completed, the parameters of the trained neural network may need to be sent back to edge devices. As mentioned, this again leads to a high transmission cost from the cloud to the edge devices with respect to the number of parameters of a neural network and the number of edge devices. This is why a neural network parameter extraction algorithm is being investigated in this research because with the presence of this algorithm, the

transmission costs can be saved when transmitting the trained neural network parameters back to edge devices.

3. After solving problem 1 and problem 2, the solutions to those problems have to be scaled up to multiple edge devices. This naturally leads to the question of how multi-task learning can be performed in the presence of multiple participating edge devices. To solve this problem, the training needs to start on one of the edge devices can continue to the other edge devices. However, the challenge here is to figure out which edge device should start and end the training because the order in which the training out on the edge devices can have a big impact on the overall energy consumption across all the edge devices. This is why an algorithm is being investigated that can determine the edge device training order such that the end-to-end energy consumption across all edge devices can be reduced.

Though this chapter mentioned about adopting unsupervised learning going forward with enabling incremental learning for edge computing, this thesis focuses on the supervised learning domain. This is because the problems that this thesis tries to tackle are considered new in this setting especially in the data sampling domain, therefore, it must first be investigated whether this is possible and if yes, then, what are the benefits. Hence, to take a strong step forward to solve the problem of pre training data sampling, the problem of pre training data sampling must first be investigated from a supervised learning angle just to be able to show that the gains that can be achieved by doing pre training data sampling along with an automated sampling rate embedded into the data sampling algorithm. Only after gaining solid results with this approach would be the right time to look into unsupervised approaches.

Chapter 3

Sampling of Data from Novel Classes for Class Incremental Learning

3.1. Background

This chapter introduces a technique to dynamically eliminate a certain number of samples from a class distribution on the edge device itself to solve the problem of the high transmission cost of transmitting images to the cloud to the cloud. The motivation to do so is to reduce the transmission cost from the edge device to the cloud and to reduce the training time on the cloud. The objective is to reap both of these benefits while maintaining the classification accuracy of the model at every incremental training round. The development of this idea is inspired by techniques such as Dropout [150] and model pruning [19], [21].

As mentioned in chapter 2, the motivation of model pruning is to use only a subset of parameters of the entire model and still retain the accuracy of the model. It is important to note that to achieve this target, the model has to be trained first only after which pruning of unwanted weights can take place. Similarly, the motivation of data sampling is to use only a subsample of the data distribution to learn the patterns of the entire data distribution whereby to accelerate neural network training, certain training samples that are no longer assisting in the convergence of the model being trained are eliminated after certain training epochs. This is mentioned in section 2.3.5 in Chapter 2, whereby data sampling is only being performed during training or after training is done but none of the samplings occurs before training starts. This chapter proposes a data sampling algorithm for the edge device which can eliminate a certain number of images before training begins on the cloud, this way, not only is the neural network training time reduced but the transmission cost from the edge device to the cloud is also reduced.

In the class incremental learning context, new classes can appear over time. New classes can represent new tasks hence, these tasks must be learned incrementally by neural networks. In the context of IoT, if these new samples are encountered by the IoT devices, these samples must be transmitted to the cloud for training especially in the case of high dimensional data such as images. Training of high-dimensional data has to be carried out at the cloud because edge devices do not have the resources to do so. The dependence on the cloud for training leads to a transmission cost problem because the new streaming samples must be transmitted to the cloud before training even begins. This transmission cost is directly proportional to the number of samples and the dimensionality of the features.

This is why a mechanism needs to be in place at the IoT edge device that can filter a certain number of samples without affecting the model performance on the cloud while largely reducing the transmission costs. Transmitting fewer samples to the cloud can also lead to faster training time. However, the biggest challenge is that no neural network can have any knowledge about novel classes because training has not occurred yet. For a neural network to understand a certain class, the neural network must first be trained on the given distribution. Therefore, in the context of IoT and data sampling, the task here is to filter a certain number of samples from novel classes before transmission to the cloud and still maintain the model performance before training even begins.

The nomenclature that is used throughout this chapter is shown below.

Nomenclature	
c	Class index
L^c	Set of losses of all samples sorted in ascending order belonging to class c
n_c	Number of samples in class c just before transmission from IoT edge device to cloud
W'	Newly initialized weights of the SoftMax layer on the IoT edge device
b'	Newly initialized biases of the SoftMax layer on the IoT edge device
x_i^c	Feature map output of image i belonging to class c
y_i^c	One-hot label corresponding to x_i^c
l_i^c	Cross entropy loss corresponding to x_i^c
$\varphi(\cdot)$	SoftMax function
${\mu_L}^c$	Median value of set L^c
${\sigma_L}^c$	Deviation that denotes by how much the values of L^c deviate from ${\mu_L}^c$
$\mathbb{I}(\cdot)$	Indicator function
${L_{\text{cutoff}}}^c$	Cut-off loss of set L^c
V^c	Set containing loss values smaller than ${L_{\text{cutoff}}}^c$
τ^c	Total number of samples to be discarded from n_c
t_{mb_0}	Time taken to forward propagate the first mini batch of images on IoT edge device
t_{mb_n}	Time taken to forward propagate a mini batch of images on IoT edge device after first mini batch
lab_{str}	Label string format
FM_n	Feature map tensor at index n of feature maps of a mini batch
D_n	Depth of feature map tensor of FM_n
W_n	Width of feature map tensor of FM_n
H_n	Height of feature map tensor of FM_n
S_n	Size of FM_n
t	Incremental training round t
$\emptyset(x_t)$	Output of feature extractor of sample x belonging to incremental training round t
E_t	Set of all features at every incremental training round up until round t
OS	Operating System
RAM	Random Access Memory
M_{ec}	String containing feature maps and labels

	transmitted from the IoT edge device to the cloud
M_{ce}	String containing W_{best} and I_{best} transmitted from cloud to the IoT edge device
NS	No Sampling
SEA	Sample Elimination Algorithm
KWT [202]	Kruskal Wallis Test
MT [203]	Median Test
WRST [204]	Wilcoxon Rank Sum Test

3.2. Methodology

In this chapter, the data sampling algorithm that takes place on the IoT edge device is discussed that largely reduces training time on the cloud and transmission cost to the cloud in exchange for a very small change in the model performance. This is done by counting the number of samples that can be discarded per training class distribution followed by eliminating such samples in a randomized order. Figure 3.1 shows the proposed system block diagram. At a basic level, the role of the edge device is to perform data sampling and the role of the cloud is to train a neural network in an incremental learning manner.

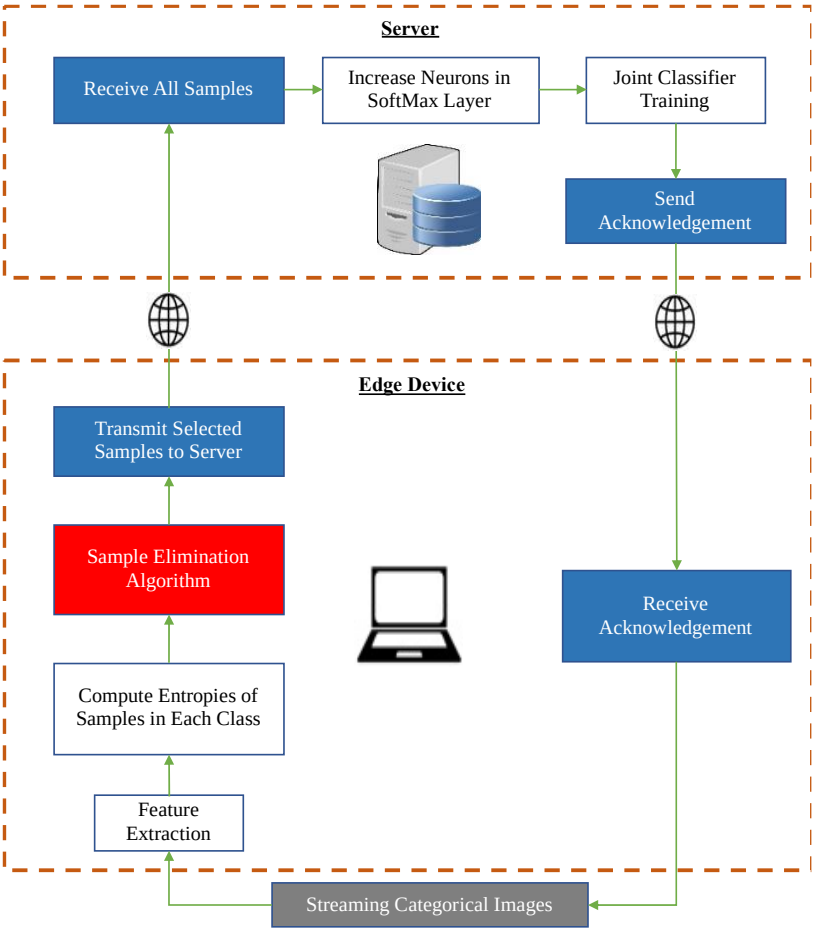


Figure 3.1. Incremental learning system split between a fog device and a centralized server.

3.2.1. System Architecture and Sample Elimination Algorithm

The Sample Elimination Algorithm (SEA) is proposed in this chapter where the objective of this algorithm is to discard certain training samples from a training distribution on the IoT edge device and to reduce the communication load, in exchange for a very small difference in the incremental learning performance. This can improve the efficiency of a deep learning model partitioned between the cloud and an IoT edge device while retaining the incremental learning performance. Although data sampling may result in slight accuracy changes, in all the experiments below, the differences in the classification accuracies are observed. This is because when a given deep learning model is trained on separate occasions with the same hyperparameters and model architecture, the model will yield slightly different classification accuracies. This is due to the random initialization of the model weights. So, because of this natural stochastic property of neural network training in which slightly different classification accuracies are yielded every time a model is run, a slight difference in accuracy can also be observed due to data sampling on the IoT edge device. The size of the test dataset can also vary for different datasets. Consequently, incremental learning performance can also vary with respect to data sampling. However, the focus of this chapter is not to propose the acceptable margin as this is problem dependant, the focus of this chapter is to see the differences in the model classification accuracies.

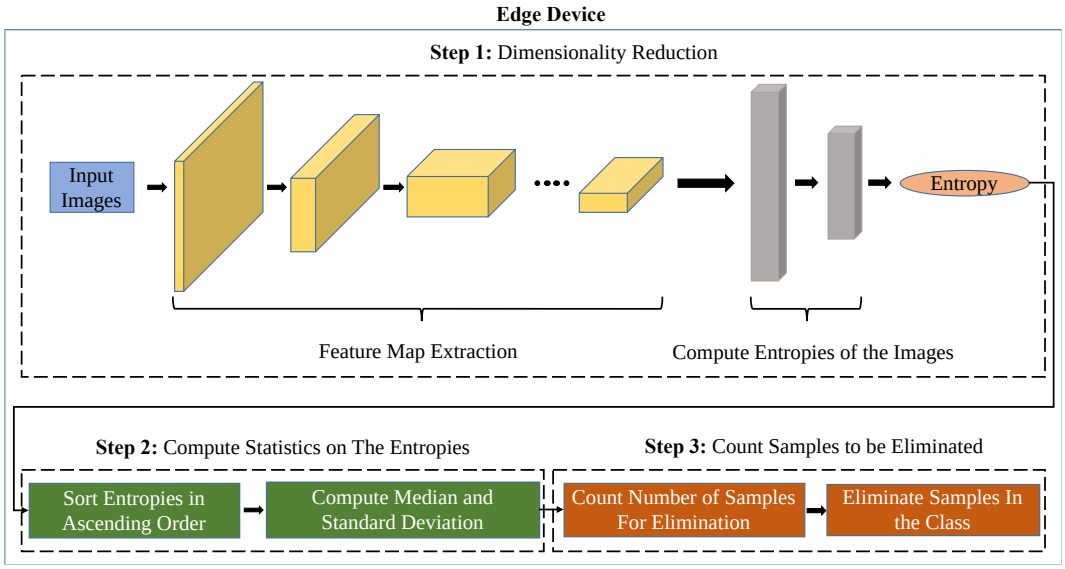


Figure 3.2. High level overview of Sample Elimination Algorithm (SEA) on the fog device.

Figure 3.2. shows the pipeline of the Sample Elimination Algorithm (SEA). Images and even image features are known as high dimensional data which are computationally difficult to operate on especially on resource-constrained edge devices. Therefore, to perform data sampling, the dimensionality of the data needs to be reduced. From a high-level perspective, two stages of dimensionality reductions are employed before performing data sampling. Firstly, the input images are forward propagated through convolutional layers of a CNN to obtain feature maps of the images. This is followed by forward propagating the feature maps through an ANN that produces a SoftMax output which is then used to compute entropies of the images. During this process, each high-dimensional image is converted to a 1-D entropy value. Each training class distribution will have a set of entropies and it is this entropy set that

is used to perform data sampling (step 3) as explained below. Figure 3.3. shows the detailed flowchart of the proposed system.

The design is capable of running on multiple IoT edge devices. The blocks highlighted in Figure 3.3 are the contributions. To explain the methodology from a high-level perspective, an ImageNet [1] pre-trained feature extractor is run on the IoT edge device along with a classifier. Feature extractors pre-trained on ImageNet are chosen due to the fact that such models have been trained on a large-scale dataset such as ImageNet, thus they can act as generic models to extract features from incoming images. The classifier on the IoT edge device, however, is used only for inference based on the output of the feature extractor. The training of the classifier takes place on the cloud. The weights associated with the newly added neurons in the SoftMax layer of the classifier are randomly initialized on both the IoT edge device and the cloud for the new classes to be learned. The learning rate of the feature extractor is set to 0 i.e., the CNN feature extractor layers are frozen. This indicates that the feature extractor parameters will not be modified any further and implies that once backpropagation takes place on the cloud, there is no need to transmit the gradients back to the IoT edge device.

At every incremental training round, n number of classes are trained together at a time. Data sampling takes place per class therefore, all samples in each class are forward propagated through the model to obtain the loss values of all the samples. The samples with very low loss values with respect to the losses of training samples are counted on the IoT edge device based on which data sampling takes place. After forward propagating the selected samples through the feature extractor of the CNN, the output feature maps of the selected samples are converted from tensor to a string format and stored on the IoT edge device. As the tensor converted strings are stored on the IoT edge device, RAM consumption increases which can slow down the processing on the IoT edge device, and as soon as a slowdown in the forward propagation time is detected, the tensor strings are transmitted to the cloud via the Transmission Control Protocol (TCP/IP). Otherwise, transmission takes place once data sampling has been performed for a given class. The cloud waits for the incoming data streams from the edge device and converts the strings into tensor feature maps and saves them. The training on the cloud only starts once all the classes belonging to an incremental training round have arrived. As new classes are sent to the cloud for training, if an imbalanced dataset is detected then the minority classes are oversampled using the Synthetic Minority Over-sampling Technique (SMOTE) [205]. The features of all the previous classes are used as exemplars for incremental learning. So, when learning new tasks, all the features of the previous classes are used together with the features of the current samples belonging to the new classes to train the classifier on the cloud.

Once all the data of a particular incremental training round has been sent to the cloud, all the tensor feature maps on the cloud are used to train the classifier. During the classifier training on the cloud, the only part of the classifier that changes dynamically is the SoftMax classification layer on the cloud i.e., new neurons are added to the SoftMax layer based on the number of new tasks to be learned after every incremental training round. Once the classifier is trained, weight extraction is performed whereby only the useful weights of the trained model are chosen to be transmitted back to the IoT edge device. Together with the useful weights, the indices of the useful weights are also transmitted to the IoT edge device. These indices denote the exact connections of the classifier at the IoT edge device that must be updated with the useful weights received from the cloud.

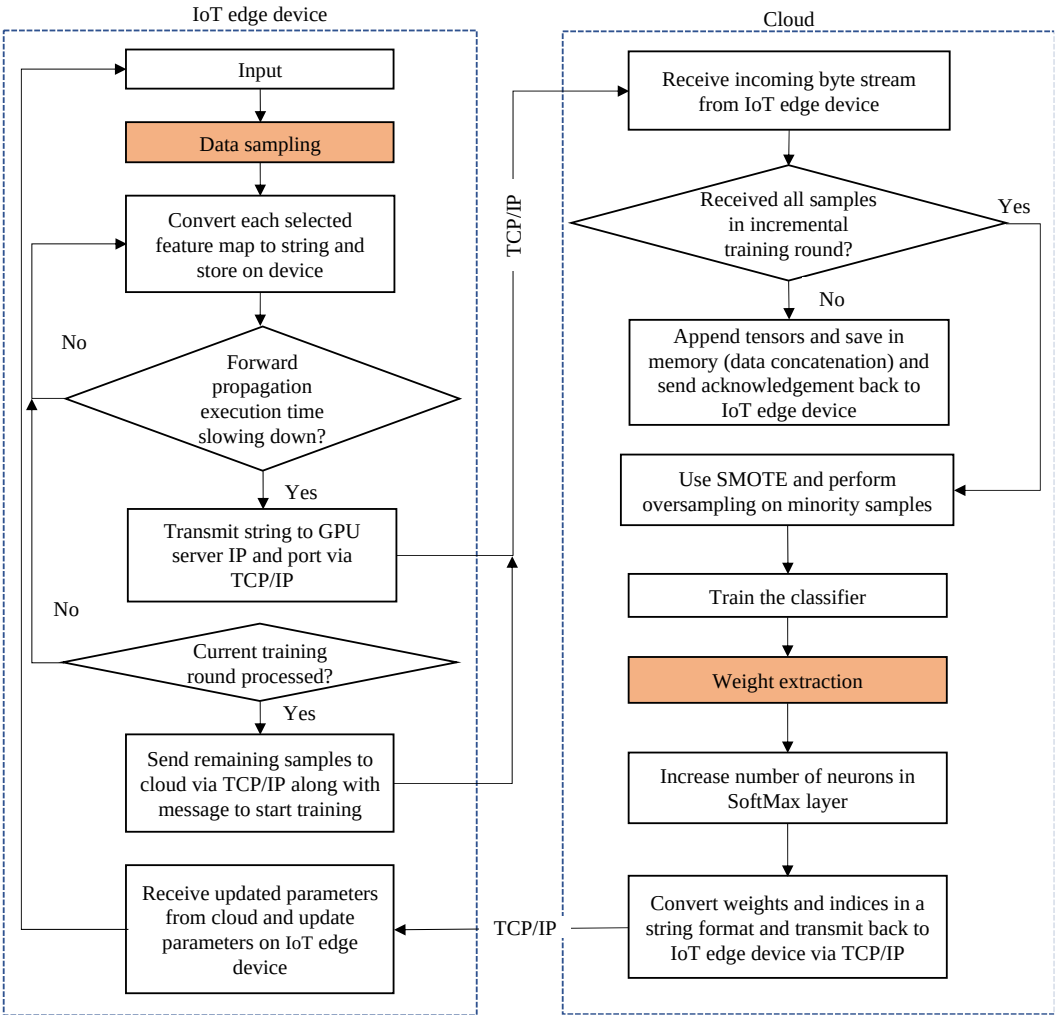


Figure 3.3. The proposed system flowchart. Incremental learning system partitioned between an IoT edge device and the cloud.

Here, the details of the Sample Elimination Algorithm (SEA) algorithm are presented that explains Figure 3.2. Let c be the class index of the current class being processed, n_c be the number of samples in each class thus the number of losses of all samples in class c is also n_c. Consider a set L^c which contains all the losses for all the samples in the current training class, such that $L^c \in \{l_0^c, l_1^c, l_2^c, \ldots l_{n_c}^c\}$ where L^c is sorted in ascending order. L^c is computed using the cross-entropy loss function and ${\mu_L}^c$ is the median of the losses of n_c samples. In SEA, a pre-trained feature extractor is used with a single-layer neural network whose weights are completely randomly initialized for data sampling. The formulas used to discard training samples at the IoT edge device are described one step at a time below.

$$ {l_i}^c = -\sum_j (y_i^c)_j \cdot \log_e(\varphi(W' x_i^c + b'))_j \tag{3.1} $$

In Equation 3.1, the cross-entropy values of incoming samples are calculated as the first step of the SEA algorithm where x_i^c is the feature map i of class c coming into the SoftMax layer, y_i^c is the one-hot label corresponding to x_i^c. W' and b' are the weights and biases of the new randomly initialized n neurons in the SoftMax layer on the IoT edge device where n is the new number of classes to be learned. $\varphi(\cdot)$ is the SoftMax function. j is the index of the newly added

neurons (in the SoftMax layer). ${l_i}^c$ is the loss of sample i of set L^c.

$$ {\sigma_L}^c = \sqrt{\frac{\sum_{i=1}^{n_c}({l_i}^c - {\mu_L}^c)^2}{n_c}} \tag{3.2} $$

In Equation 3.2, the term ${\sigma_L}^c$ that denotes by how much the metrics ${l_i}^c$ differ from the median (${\mu_L}^c$) of the distribution of the set L^c of class c. The median value of L^c is used instead of the mean in order to protect the SEA algorithm from skewed entropy values in L^c. For example, a skewed value can be an extremely high entropy value in L^c that would largely affect the mean value of L^c. This could greatly increase the value of ${L_{\text{cutoff}}}^c$ in Equation 3.3 which would lead to an extremely high data sampling rate and thus affect the learning performance of the model on the cloud. However, the median of L^c does not get affected by such skewed values. As a result, the data sampling rate remains stable as per Equation 3.3 and Equation 3.4. This is the second step of the SEA algorithm.

$$ {L_{\text{cutoff}}}^c = \underset{\forall {l_j}^c \in L^c}{\arg\max} \left({l_j}^c \cdot \mathbb{I}\left({\mu_L}^c - {l_j}^c > {\sigma_L}^c \right) \right) \tag{3.3} $$

In Equation 3.3, $\mathbb{I}(\cdot)$ is the indicator function whose output is 1 if the condition inside the indicator function is true or else the output is 0. The term ${L_{\text{cutoff}}}^c$ is the cut-off loss which also tells us about the number of samples with low loss values with respect to the training distribution. Determining the value of ${L_{\text{cutoff}}}^c$ is the third step of the SEA algorithm. The total number of samples to be discarded from class c (τ^c) is expressed in Equation 3.4 which is the final step of the SEA algorithm. In Equation 3.4, let V^c be a set containing all loss values smaller than ${L_{\text{cutoff}}}^c$.

$$ \tau^c = \lfloor \omega^c \cdot (\textstyle\sum_{l \in V^c} 1) \rfloor \tag{3.4} $$

In Equation 3.3, if the distance between the low loss values in L^c and ${\mu_L}^c$ is greater than ${\sigma_L}^c$, then all such samples that satisfy this condition are counted. This process repeats iteratively until a sample is encountered where the distance between its loss and the median is smaller than the standard deviation which means the largest loss value in L^c that satisfies Equation 3.3 is considered to be the cut-off loss (${L_{\text{cutoff}}}^c$). ${L_{\text{cutoff}}}^c$ is then used to count the number of samples to be discarded i.e., the number of values in L^c that have values less than ${L_{\text{cutoff}}}^c$ is denoted by τ^c in Equation 3.4 i.e., the number of samples to be discarded. ω^c lies within 0 - 1 and controls the sampling rate.

For each class, the loss distribution L^c is sorted in ascending order. ${\sigma_L}^c$ denotes how far apart the values in L^c differ from ${\mu_L}^c$ in general. However, in such distributions, there can be values in L^c whose distance from ${\mu_L}^c$ is less than ${\sigma_L}^c$ and there can also be values in L^c whose distance from ${\mu_L}^c$ is greater than ${\sigma_L}^c$ due to the variations in the images per class which also means varying entropies of images per class as well. Images with high loss values can be beneficial for neural network training because a high loss value implies a lot of weights will have to be fine-tuned thus improving the generality of the model as compared to images with low loss values. Images with already low loss values cannot have their values greatly reduced as compared to images with high loss values. Hence, the low loss values in L^c that vary from ${\mu_L}^c$ with respect to ${\sigma_L}^c$ must be counted which is why Equation 3.3 has been designed. By doing

so, the number of samples to be discarded from a class just before transmission to the cloud can be determined.

3.2.2. Communication Between the Edge Device and The Cloud

Initially, the IoT edge device has no data. However, during training, when new class data arrives, such data is forward propagated through a CNN feature extractor. The output feature map is then converted to a string format and saved in a buffer along with its respective label. The same process is applied to the other incoming mini-batches of images. All the output feature maps of images are concatenated to a buffer that stores all the tensor converted strings. This buffer (M_{ec}) is what is transmitted to the cloud via the TCP/IP protocol. The format of M_{ec} is shown in Equation 3.7. The description of "act" is as follows: "$a!$" (for data concatenation), "$t!$" (for training), "$d!$" (for process termination).

Along with the feature maps, the associated labels must also be transmitted to the cloud. Since each feature map has one unique label (denoted by "lab" in Equation 3.5), let *N* be the total number of feature maps and thus the label format is shown in Equation 3.5.

$$\mathrm{Lab}_{\mathrm{str}} = \mathrm{lab}_n, \mathrm{lab}_{n+1}, \ldots \mathrm{lab}_{n+N} \tag{3.5}$$

A single feature map will have a depth of one and the same width (W_n) and height (H_n) as the overall feature map of an image. The string format of a single feature map can be expressed as shown in Equation 3.6.

$$\mathrm{FM}_n = \mathrm{val}_{n,i}, \mathrm{val}_{n,i+1}, \mathrm{val}_{n,i+2}, \ldots \mathrm{val}_{n,S_n} \tag{3.6}$$

All values are comma-separated when the feature maps and the labels are converted to a string and each feature map string is separated by the character 'k'. The overall message format is written as shown in Equation 3.7.

$$M_{ec} = D_n, W_n, H_n, < \mathrm{lab}_{\mathrm{str}} >, \mathrm{FM}_n, k, .. \mathrm{FM}_N, < \mathrm{act} > ! \tag{3.7}$$

On the cloud, the incoming stream is accepted, and the reading operation continues until the end of message character '$!$' is detected. Once the end of message character is detected, the action character ("act") is obtained, and based on that action, an appropriate process is carried out. For example, if the action character is $'a'$ then tensor concatenation is carried out, and if the action character is $'t'$ then training is carried out, and if the action character is $'d'$ then the program on the cloud stops executing. At the end of every incremental training round on the cloud, only the useful weights of the trained classifier are sent back to the IoT edge device. Upon receiving this message, the IoT edge device then proceeds with processing the next batch in the dataset i.e., converting the images and labels to strings. This process continues until all the samples in the dataset are processed.

In Equation 3.7, the end of message character ("act") is appended at the end of the tensor converted string because the TCP/IP protocol is not a message-based protocol but a stream-based protocol. It means that there is no guarantee that all the bytes that are transmitted will arrive at the recipient. So, the recipient must keep listening for the incoming string from the IoT edge device and only stop reading when the end of message character '$!$' is encountered.

The character '*!*' is always appended at the end of the message, therefore if this character is encountered, the recipient device knows that all the transmitted bytes from the sender have been received.

Synchronization between the IoT edge device and the cloud is ensured by the following steps: firstly, when the IoT edge device transmits feature maps of the images to the cloud, the IoT edge device waits for a reply from the cloud. Secondly, if the reply is not received from the cloud, the IoT edge device will not carry out any other processes. Thirdly, the cloud will keep listening for incoming data and will only proceed once the end of message character is read (*'!'*).

The amount of memory available in the IoT edge device hardware is limited and this is a key factor to consider in distributed processing scenarios. Since the CNN feature extractor output of the input images at the IoT edge device is stored, this may lead to RAM scarcity which causes a slowdown in the feature extraction time. This is because in any modern operating system (OS) when a program requires more RAM, the OS will allocate the required memory to that program. However, when RAM starts to run out, the OS will move some of the program's memory to disk. In other words, the OS now needs to move the data more frequently between disk and RAM, resulting in a slower response time. For a given deep learning model, if it is required to store more feature maps on the IoT edge device, more memory will be consumed with respect to the number of samples, resulting in a slowdown of the IoT edge device feature extraction. To account for this scenario, a very simple algorithm is formulated in Equation 3.8.

$$\text{Transmit} = \begin{cases} \text{yes}, (t_{\text{mb}_0} - t_{\text{mb}_n}) > t_{thresh} \\ \text{no}, \text{otherwise} \end{cases} \tag{3.8}$$

In Equation 3.8, the time taken to forward propagate the first mini-batch is recorded (denoted by t_{mb_0}). If the difference between the time taken to process subsequent mini-batches (denoted by t_{mb_n}) and t_{mb_0} exceeds t_{thresh}, it indicates that the processing speed of the IoT edge device is slowing down. In such a scenario, all feature maps, stored on the IoT edge device thus far, are sent to the cloud and subsequently, the memory on the IoT edge device will be cleared. t_{thresh} is the acceptable time difference between the time taken to forward propagate the first mini-batch and the time taken to extract the subsequent mini batches.

3.2.3. Incremental Learning on The Cloud

In this work, only the SoftMax classification layer grows in size. The number of neurons added to the SoftMax layer at every incremental training round is equal to the number of new tasks to be learned at every training round. Once the feature extractor outputs are sent from the IoT edge device to the cloud, these features are saved on the cloud and as the training process continues, these features are used as exemplars. The exemplar set is constructed following Equation 3.9 below.

$$E_t = \emptyset(X_t) \cup \emptyset(X_{t-1}) \cup \emptyset(X_{t-2}) \cup \ldots \emptyset(X_0) \tag{3.9}$$

In Equation 3.9, let X be all the samples of an incremental training round and $\emptyset(\cdot)$ be the output of the CNN feature extractor and t be the incremental training round. E_t is defined as the

exemplar set at incremental training round t and represents the new set of samples being fed into the classifier. The new batch of feature maps and their respective labels are randomly shuffled after every epoch to maintain the classification accuracy. Next, the combined feature maps (E_t) are passed through the fully connected layers. During the backpropagation process, only the fully connected layers are updated (the feature extractor layers are frozen). This process of training is known as joint training and is one of the most common techniques to alleviate catastrophic forgetting.

3.3. Experiments

This section describes how experiments are conducted to evaluate the performance of the proposed data sampling algorithm (SEA). Table 3.1 shows the datasets, models, and hyperparameters used. Apart from CIFAR-100 [3], all the datasets are imbalanced in nature. Only the RGB images that are present in each of these datasets are chosen and all the images in these datasets are resized before training begins as shown in Table 3.1. All the hyperparameters have been randomly chosen to show that SEA can perform under any learning setting. Achieving the state-of-the-art accuracy for incremental learning is not the goal of this work, thus, the hyperparameters shown in Table 3.1 have not been fine-tuned. However, when working with ImageNet [1], two fully connected layers in the classifier are used, therefore, for the first layer which has 1000 neurons, a dropout probability of 0.30 is used and for the second and the final layer (SoftMax classification layer) of the classifier, a dropout probability of 0.20 is used.

Table 3.1. Learning settings and datasets used for the experiments in this chapter.

Learning Settings	**Datasets**			
	CUB-200 [5]	**Stanford Dogs** [7]	**CIFAR-100** [3]	**ImageNet** [1]
Training epochs	25	30	15	3
Number of fully connected layers	1	1	1	2
Dropout probability	0.25	0.30	0.20	0.30,0.20
Learning rate	0.0015	0.0002	0.0001	0.0035
Batch size	16	32	64	128
Number of classes learned at a time	50	40	20	3
Image size	90 x 90 x 3	85 x 85 x 3	32 x 32 x 3	100 x 100 x 3
CNN model	SqueezeNet [6]	MobileNetV2 [8]	MnasNet [4]	ShuffleNet V2 [9]
Train/Test split	5994/5794	12000/8580	50000/10000	6397/1728

SEA is compared with 4 baseline methods: No Sampling (NS) and 3 non-parametric statistical methods: Wilcoxon Rank-Sum Test (WRST) [204], Median Test (MT) [203], and Kruskal-Wallis Test (KWT) [202]. Firstly, WRST [204], MT [203], and KWT [202] are not built on top of SEA and secondly, these methods are non-parametric in nature that do not assume any data distribution in advance. Furthermore, these methods are very strong in comparing statistical differences between 2 groups whereas KWT can compare statistical differences across multiple groups. Data sampling leads to the formation of 2 groups i.e., one group

containing all samples and the second group containing only the selected samples for training which is why these non-parametric methods are ideal for comparison with SEA. WRST [204], MT [203], and KWT [202] are applied to L^c for performing data sampling. The formulas to compute WRST [204], KWT [202], and MT [203] are shown in Equation 3.10-3.15, Equation 3.16-3.18, and Equation 3.19-3.25 respectively.

$$\mu_w = \frac{n_A \cdot (n_A + n_B + 1)}{2} \tag{3.10}$$

$$\sigma_w = \sqrt{\frac{n_A \cdot n_B (n_A + n_B + 1)}{12}} \tag{3.11}$$

$$z = \frac{R - \mu_w}{\sigma_w} \tag{3.12}$$

$$p = \frac{1}{2} \cdot \left[1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right] \tag{3.14}$$

$$\vartheta = \begin{cases} 1, & p < \alpha \\ 0, otherwise \end{cases} \tag{3.15}$$

In Equation 3.10-3.15, n_A and n_B represent the number of elements in data distributions A and B, respectively. μ_w and σ_w denote the mean and the standard deviation. R is the rank sum of the elements in data distribution A. z presents the z score and p represents the p-value. α is the statistical significance value which is set to 0.05 in the experiments.

$$n_T = \sum_{i=1}^{k} n_i \tag{3.16}$$

$$H = \left[\frac{12}{n_T \cdot (n_T + 1)} \cdot \sum_{i=1}^{k} \frac{R_i^2}{n_i}\right] - 3 \cdot (n_T + 1) \tag{3.17}$$

$$\vartheta = \begin{cases} 1, & H > X^2 \\ 0, & otherwise \end{cases} \tag{3.18}$$

In Equation 3.16-3.18, k represents the total data distributions that are being compared, k is set to 2 in this case since two data distributions are being compared i.e., the distribution with all entropy values of the class being processed and the distribution with the selected entropy values of the class being processed. n_i is the total samples in a data distribution k and n_T is the total samples across all the data distributions. R_i is the rank sum of all elements in data distribution k. X^2 is the inverse of the p-value for a chi-square distribution where p is set to 0.05.

$$W = \sum_{i=1}^{n_A} \mathbb{I}(A_i > \widetilde{\mu_{AB}}) \tag{3.19}$$

$$X = \sum_{i=1}^{n_B} \mathbb{I}(B_i > \widetilde{\mu_{AB}}) \tag{3.20}$$

$$Y = \sum_{i=1}^{n_A} \mathbb{I}(A_i < \widetilde{\mu_{AB}}) \tag{3.21}$$

$$Z = \sum_{i=1}^{n_B} \mathbb{I}(B_i < \widetilde{\mu_{AB}}) \tag{3.22}$$

$$N_{AB} = W + X + Y + Z \tag{3.23}$$

$$X^2 = \frac{N_{AB} \cdot \left(W \cdot Z - X \cdot Y - \frac{N_{AB}}{2}\right)^2}{(W+X) \cdot (Y+Z) \cdot (W+Y) \cdot (X+Z)} \tag{3.24}$$

$$\vartheta = \begin{cases} 1, & X^2 > X^2_{\alpha,k-1} \\ 0, & otherwise \end{cases} \tag{3.25}$$

In Equation 3.19-3.25, $\widetilde{\mu_{AB}}$ is the median of all the elements in distribution A and B combined. W and X are the number of elements in distribution A and B respectively that are greater than $\widetilde{\mu_{AB}}$. Y and Z are the number of elements in distribution A and B respectively that are less than $\widetilde{\mu_{AB}}$. N_{AB} is the total number of elements in distributions A and B combined. X^2 is the chi-square value. $X^2_{\alpha,k-1}$ is the chi-square value at significance level α, k-1 is the degree of freedom where k is the number of data distributions. In all non-parametric tests, if ϑ is 1, it means there is a statistical difference that is detected between distributions A and B thus all the samples corresponding to the entropies in distribution A are transmitted to the server. If ϑ is 0, it means there is no statistical difference between distributions A and B.

SMOTE [165] is used to create new samples to tackle the problem of class imbalance. In Equation 3.26, the formula to generate a new sample is shown.

$$p' = p_1 + [\lambda \cdot (p_2 - p_1)] \tag{3.26}$$

In Equation 3.26, p' is the new point, p_1 and p_2 are any 2 random points in the dataset. λ is a uniformly random generated number between 0 and 1 i.e., $\lambda \sim U([0,1])$. In these experiments, p_1 and p_2 are chosen randomly per class to balance the number of samples across all classes.

All accuracies reported in the experiments are the top-5 accuracies (rounded off to 2 decimal places) and averaged by running the respective experiment three times. However, a top-1 accuracy is reported when working with the ImageNet [1] dataset. Classification accuracies are obtained using the test set of each dataset. The PyTorch [25] library is used to program the model and the experiments. The input images are normalised by converting the RGB images from a range of 0 to 255 to become 0 to 1. The NVIDIA Tesla K80 GPU is used as the server and a desktop computer with an i7 processor is used as a fog device to carry out all the experiments.

3.4. Results

In this section, the results of SEA are presented and discussed. Table 3.2, Table 3.3, Table 3.4, Table 3.5 show the number of samples selected before incremental training starts, the time taken to train the classifier, and the classification accuracies at every incremental training round when evaluating different datasets using various CNN architectures. This is with respect to the learning settings shown in Table 3.1 using all the data sampling algorithms along with no data sampling. The numerical values in bold are the ones that correspond to the best data sampling algorithm out of SEA, WRST [204], MT [203], and KWT [202].

Table 3.2 Results of performing data sampling using CIFAR-100 [3] and MnasNet [4].

Data Sampling Algorithm	**Number of Classes Trained**				
	20	40	60	80	100
	Samples Needed for Training				
NS	10000	10000	10000	10000	10000
SEA (Proposed)	**8578**	**8461**	**8406**	**8453**	**8485**
WRST [204]	9379	9393	9394	9393	9395
MT [203]	8742	8751	8759	8767	8757
KWT [202]	9260	9267	9279	9272	9271
	Training time (s)				
NS	*4.36*	*8.38*	*12.32*	*16.48*	*20.50*
SEA (Proposed)	**3.49**	7.22	**10.84**	**13.91**	**17.80**
WRST [204]	4.15	8.02	11.81	15.29	18.90
MT [203]	3.87	**7.18**	10.95	14.22	18.30
KWT [202]	3.94	7.76	11.99	16.21	19.51
	Classification Accuracy (%)				
NS	*85.35*	*75.25*	*69.73*	*66.66*	*64.08*
SEA (Proposed)	**84.75**	**74.25**	69.01	65.93	**63.59**
WRST [204]	84.74	74.23	**69.15**	**66.36**	63.58
MT [203]	84.50	74.13	68.55	65.26	62.67
KWT [202]	84.55	74.18	69.08	66.16	63.39

For CIFAR-100 [3], regardless of the data sampling algorithm, the classification accuracies obtained at every incremental training round are very similar to the ones obtained where no data sampling takes place. However, the differences are observed in the number of samples needed for training and the training time at every incremental training round. SEA requires the least number of training samples and the least amount of training time with the exception of the training time taken when learning 40 classes using MT. Overall SEA performs better than other data sampling algorithms.

Table 3.3. Results of performing data sampling using CUB-200 [5] and SqueezeNet [6].

Data Sampling Algorithm	**Number of Classes Trained**			
	50	100	150	200
	Samples Needed for Training			
NS	*1498*	*1495*	*1500*	*1498*
SEA (Proposed)	1381	1345	1347	1346
WRST [204]	1068	1066	1070	1069
MT [203]	**723**	**720**	**725**	**717**
KWT [202]	986	974	979	979
	Training Time (s)			
NS	*4.16*	*8.33*	*12.49*	*16.69*
SEA (Proposed)	3.74	7.45	11.05	14.85
WRST [204]	2.92	5.75	8.61	11.70
MT [203]	**1.99**	**3.88**	**5.82**	**7.92**
KWT [202]	2.70	5.34	7.95	10.74
	Classification Accuracy (%)			

NS	*51.48*	*43.36*	*33.09*	*29.67*
SEA (Proposed)	**49.67**	**40.88**	**32.05**	**27.77**
WRST [204]	47.95	37.37	30.82	25.51
MT [203]	44.56	32.54	25.80	20.85
KWT [202]	47.59	35.06	28.83	24.59

CUB-200 [5] dataset has a total of 200 classes but does not have a fixed number of images per class. However, the number of images per class is very small which is the main reason why this dataset is chosen to test SEA. Discarding information from a small data pool can lead to a big information change in the overall data pool which is why SEA is tested on this dataset to really observe its effectiveness.

For CUB-200 [5], SEA is able to reject the least amount of samples as compared to other data sampling techniques whereby the least number of samples needed for training arises when using MT. However, the classification accuracies obtained using SEA are the best out of all the data sampling methods. Furthermore, the classification accuracies obtained at every incremental training round are within 3% of the baseline (NS). The fact that WRST [204], MT [203], and KWT [202] discard so many samples indicates that these statistical methods are only able to conclude a significant difference between the overall metric distribution and the selected distribution once a lot of samples have been discarded. Clearly, for datasets which has a small number of samples per class such as CUB-200, SEA is much better suited than other data sampling methods.

Table 3.4. Results of performing data sampling using Stanford Dogs [7] and MobileNetV2 [8].

Data Sampling Algorithm	**Number of Classes Trained**		
	40	80	120
	Samples Needed for Training		
NS	*4000*	*3999*	*4000*
SEA (Proposed)	3505	3440	3477
WRST [204]	3403	3410	3411
MT [203]	**2879**	**2877**	**2871**
KWT [202]	3318	3319	3316
	Training Time (s)		
NS	*8.93*	*17.13*	*25.09*
SEA (Proposed)	7.77	15.43	22.15
WRST [204]	7.68	15.29	22.14
MT [203]	**6.45**	**12.78**	**18.96**
KWT [202]	7.39	14.61	21.32
	Classification Accuracy (%)		
NS	*62.89*	*52.13*	*44.78*
SEA (Proposed)	61.78	**51.66**	**43.17**
WRST [204]	62.11	49.99	43.14
MT [203]	60.88	48.96	41.27
KWT [202]	**62.39**	51.27	43.16

Stanford Dogs [7] dataset has a total of 120 classes where the number of samples in each dataset is different. However, the number of samples in almost every class exceeds 100 images. For

Stanford Dogs [7], the number of samples needed for training and the training time is the least when using MT. Even though MT has a faster training time and transmits fewer samples to the cloud than any other data sampling algorithm, SEA still leads to having a higher classification accuracy than the rest of the data sampling algorithms for 2 out of 3 incremental training rounds, furthermore, classification accuracies obtained after applying SEA are within 2% of the baseline at every incremental training round.

ImageNet [1] is a well-known large-scale dataset which is why it is very important to apply data sampling algorithms on a dataset that has a large number of samples per class. ImageNet [1] has a very large number of classes, however, since the focus is to see how data sampling performs on classes with a large number of samples, only a total of 6 classes from ImageNet [1] are used. However, these 6 classes are among the top 10 classes with the most number of samples in the ImageNet [1] dataset. The classes used are as follows: n01882714 (koala bear, kangaroo bear, native bear), n02086240 (Shih-Tzu), n02094433 (Yorkshire terrier), n02138441 (meerkat), n02279972 (monarch butterfly, Danaus plexippus), and n09428293 (seashore).

Table 3.5. Results of performing data sampling using ImageNet [1] and ShuffleNet V2 [9].

Data Sampling Algorithm	**Number of Classes Trained**		
	2	4	6
	Samples Needed for Training		
NS	*1868*	*2435*	*2631*
SEA (Proposed)	**1540**	**2066**	**2223**
WRST [204]	1785	2340	2533
MT [203]	1698	2246	2436
KWT [202]	1767	2322	2513
	Training Time (s)		
NS	*1.00*	*2.04*	*3.01*
SEA (Proposed)	**0.64**	**1.75**	**2.51**
WRST [204]	0.70	1.97	2.91
MT [203]	0.69	1.91	2.75
KWT [202]	0.70	1.97	2.86
	Classification Accuracy (%)		
NS	*97.01*	*95.07*	*86.37*
SEA (Proposed)	97.22	**95.45**	85.27
WRST [204]	97.01	94.80	83.48
MT [203]	97.22	94.42	**86.83**
KWT [202]	**97.44**	93.31	83.71

For ImageNet [1], SEA manages to utilize the least number of training samples and the least training time as compared to all the other data sampling algorithms. Furthermore, all classification accuracies obtained irrespective of the data sampling algorithm are within 3% of the baseline, therefore, SEA is the overall better performing algorithm in this experiment. The reason why the training time is so small is that the experimentation is carried out using only 6 classes of ImageNet [1] and the training epochs are very small thus there is not a very high training time. Despite reporting top-1 accuracies when working on ImageNet [1] dataset, the

classification accuracies that are obtained at every incremental training round are very high and this is because all the models mentioned in Table 3.1 are already pre-trained on this dataset.

From Table 3.2, Table 3.3, Table 3.4, Table 3.5, it can be seen that SEA is the best performing data sampling algorithm, however, the performance and effectiveness of SEA are further examined. Table 3.6 shows the standard deviation in the classification accuracies, training time, and the number of samples discarded at the fog device with respect to NS. The standard deviation of the number of samples transmitted to the server ($\text{Samples}_{\text{dev}}$), the training time of $\text{ANN}_{\text{server}}$ on the server ($\text{Train}_{\text{dev}}$), and the classification accuracies (Acc_{dev}) are calculated as shown in Equation 3.27, Equation 3.28, and Equation 3.29 respectively.

$$\text{Samples}_{\text{dev}} = \left\lfloor \frac{\left|\sum_{t=1}^{T} \text{Samples}_{\text{SEA}}^{t} - \sum_{t=1}^{T} \text{Samples}_{\text{NS}}^{t}\right|}{\sqrt{2}} \right\rfloor \tag{3.27}$$

$$\text{Train}_{\text{dev}} = \frac{\left|\sum_{t=1}^{T} \text{Train}_{\text{SEA}}^{t} - \sum_{t=1}^{T} \text{Train}_{\text{NS}}^{t}\right|}{\sqrt{2}} \tag{3.28}$$

$$\text{Acc}_{\text{dev}} = \frac{1}{T} \cdot \sum_{t=1}^{T} \frac{\left|\text{Acc}_{\text{SEA}}^{t} - \text{Acc}_{\text{NS}}^{t}\right|}{\sqrt{2}} \tag{3.29}$$

In Equation 3.27, Equation 3.28, and Equation 3.29, T represents the total number of incremental training rounds, at the incremental training round t, $\text{Samples}_{\text{SEA}}^{t}$ represents the number of samples transmitted from the fog device to the server, $\text{Train}_{\text{SEA}}^{t}$ represents the training time of the classifier on the cloud, and $\text{Acc}_{\text{SEA}}^{t}$ represents the classification accuracy. In Equation 3.27 and Equation 3.28, using SEA and NS, the total number of samples transmitted to the server and the total training time of the classifier are computed, respectively. The standard deviation between SEA and NS is then computed for the metrics: total samples transmitted to the server and the total training time on the server. At every incremental training round, a classification accuracy is obtained, therefore, in Equation 3.29, the standard deviation of the classification accuracy between SEA and NS is computed at every incremental training round and then the average of all the standard deviations is computed.

Table 3.6. Standard deviation of the reduction in the number of training samples, reduction in the training time on the server and the difference in classification accuracies after applying SEA.

Datasets	**Standard deviations between NS and SEA**		
	Samples transmitted	**Server training time**	**Accuracies**
CIFAR-100 [3]	1077	15.87	0.50
CUB-200 [5]	101	5.32	1.27
Stanford Dogs [7]	1115	4.10	0.75
ImageNet [1]	781	0.82	0.40

Table 3.6 shows that the standard deviation of the classification accuracies is less than 1 for all datasets except CUB-200 [5] which also has a very small standard deviation of 1.27 even though the number of samples per class in this dataset is very low. For such small standard deviations in the classification accuracies, a very large standard deviation in the reduction of the number of training samples is achieved along with a decent standard deviation in the

reduction of the training time after applying SEA. The learning settings for evaluating each dataset are different which is why the standard deviations in the training time reduction are different for each dataset i.e., not directly proportional to the reduction in the number of training samples. However, the main point here is that the results in Table 3.6 support the hypothesis that after using SEA, the classification accuracies are very similar to the ones obtained without data sampling while reducing the training time and the number of samples.

In this work, the model is trained incrementally by carrying out joint training, and by doing so, it is shown that SEA can also work for static training. In joint training, all the samples of the new classes are concatenated together with all the samples of the old classes, and the model is then trained. However, if observed closely, joint training is simply a series of static training sessions. For e.g., for the first rows of Table 3.2, Table 3.3, Table 3.4, and Table 3.5, no training has taken place but when the training begins, the model is trained on all the available data and it can be seen that even after SEA is applied, the classification accuracies are very similar thus proving that SEA is not only useful for training for also for traditional static training of deep learning models.

Apart from the standard deviations, in Figure 3.4, the overall reduction rate is shown for the sample transmission to the server and the training time of the classifier on the server after applying SEA. The average change in the classification accuracies after applying SEA is also shown. From Figure 3.4, it can be seen that when evaluating SEA on several datasets, the sample transmission reduction rate to the server and training time on the server is greater than 13% and 11% respectively in all cases. The maximum reduction in sample transmission to the server and training time on the server achieved is 15.94% and 23.67%. The maximum and minimum change in the average difference between the classification accuracies is only 1.81% and 0.56%. This shows that SEA can largely maintain incremental learning performance while greatly reducing the samples being transmitted to the server and the training time on the server.

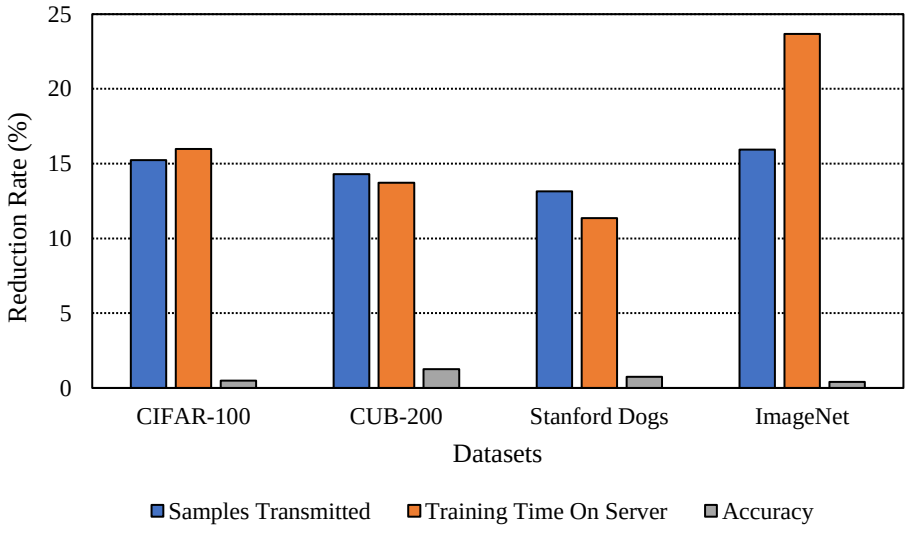


Figure 3.4. Overall change in the number of samples transmitted to the server, the training time on the server, and the classification accuracies after applying SEA.

It is also important to test the performance of data sampling when the same dataset is evaluated on different model architectures, different learning hyperparameters, and learning a different number of classes at every incremental training round. Two datasets are chosen for this experiment, the first is a balanced dataset with a high number of samples per class (CIFAR-

100 [3]) and the second dataset is an imbalanced one with a very small number of samples per class (CUB-200 [5]). Each dataset is evaluated using the following models: SqueezeNet [6] and ShuffleNet V2 [9]. For the classification layers, a single SoftMax classification layer is used in both the cases whereby new neurons are added to this layer and the weights associated with the newly added neurons are initialized randomly.

For CIFAR-100 [3], the pre-trained convolutional layers of both SqueezeNet [6] and ShuffleNet V2 [9] are used. The Adam Optimizer [206] is used together with the categorical cross-entropy loss function. All the layers except the SoftMax classification layer use the ReLU activation function, the fully connected layers use dropout with a probability of 0.25, and a batch size of 128. However, for SqueezeNet [6], 70 epochs of training are used per incremental training round with a learning rate of 0.0001 and for ShuffleNet V2 [9], 25 epochs of training are used per incremental training round with a learning rate of 0.001. For CUB-200 [5], the pre-trained convolutional layers of both SqueezeNet [6] and ShuffleNet V2 [9] are used. The Adam Optimizer [206] is used together with the categorical cross-entropy loss function. All the layers except the SoftMax classification layer use the ReLU activation function, the fully connected layers use dropout with a probability of 0.25, and a batch size of 128. The CUB-200 [5] dataset contains images with different image resolutions hence these images are resized to a standard resolution of 90 x 90 x 3. However, for SqueezeNet [6], 25 epochs of training are used per incremental training round with a learning rate of 0.0015 whereas, for ShuffleNet V2 [9], 30 epochs of training are used per incremental training round with a learning rate of 0.002. All the accuracies reported are top-5 accuracies.

For comparing SEA with other data sampling techniques, four baselines are chosen which are as follows: MT [203], WRST [204], Entropy Sampling (ES) [179], and Least Confidence Sampling (LCS) [179]. MT [203] and WRST [204] have been explained in the previous experiments, however, ES [179] and LCS [179] are active learning queries for performing data sampling whereby ES [179] eliminates the samples with the lowest entropies because samples with low entropies are not deemed 'hard' by the neural network being trained hence, such samples are not needed for further training. LCS [179] eliminates samples that have a very high classification probability because this means such samples can already be understood by a neural network and hence will not contribute much towards the model training. For statistical significance, the p-value for MT [203] and WRST [204] is set to 5%. When applying the SEA algorithm to LCS [179], the loss values in the SEA algorithm are replaced with the SoftMax probabilities of a sample at its given label index, τ^c samples with the highest SoftMax probabilities are discarded. When applying the SEA algorithm to ES [179], τ^c samples with the lowest entropies are discarded. The reason SEA is extended to LCS [179] and ES [179] is because LCS [179] and ES [179] do not have the functionality to count the number of samples to be discarded. Therefore, the purpose of these experiments is to check if applying SEA to LCS [179] and ES [179] leads to better performances as compared to just applying SEA. Table 3.7 and Table 3.8 show the incremental learning classification accuracy when learning 10 classes at every incremental training round when evaluating CIFAR-100 [3] using SqueezeNet [6] and ShuffleNet V2 [9], respectively.

Table 3.7. Incrementally learning 10 classes at a time from CIFAR-100 [3] using SqueezeNet [6].

	Classification Accuracy (%)					
Total Classes Trained	**NS**	**MT** [203]	**WRST** [204]	**SEA (Proposed)**	**ES** [179]	**LCS** [179]
10	95.40	94.60	95.40	94.20	95.70	94.60
20	83.70	83.45	83.25	82.65	82.60	81.30
30	78.43	78.03	78.13	77.13	76.93	75.80
40	71.45	71.45	71.33	70.75	69.50	69.63
50	66.60	66.56	66.80	66.10	66.02	65.16
60	64.85	64.08	64.72	64.20	63.23	62.93
70	62.56	62.71	63.09	62.11	61.04	60.96
80	60.56	60.24	59.85	60.28	59.34	59.09
90	58.73	58.33	58.09	57.62	57.29	56.74
100	56.72	56.67	56.60	56.30	55.28	55.24

Table 3.8. Incrementally learning 10 classes at a time from CIFAR-100 [3] using ShuffleNet V2 [9].

	Classification Accuracy (%)					
Total Classes Trained	**NS**	**MT** [203]	**WRST** [204]	**SEA (Proposed)**	**ES** [179]	**LCS** [179]
10	94.70	94.40	94.70	94.50	94.60	93.90
20	85.20	84.30	84.85	84.65	83.85	83.05
30	80.17	79.97	80.30	79.87	79.13	78.70
40	75.70	75.10	75.58	74.60	73.70	74.10
50	71.86	71.42	71.76	71.20	70.40	70.54
60	69.50	68.53	69.07	69.07	67.70	67.79
70	68.14	67.53	67.70	67.43	66.49	66.76
80	66.35	65.50	65.67	65.36	64.60	65.06
90	64.73	63.94	64.10	64.03	63.19	63.24
100	62.78	62.41	62.33	62.55	61.61	61.56

Table 3.9 and Table 3.10 show the incremental learning classification accuracy when learning 20 classes at every incremental training round when evaluating CIFAR-100 [3] using SqueezeNet [6] and ShuffleNet V2 [9], respectively.

Table 3.9. Incrementally learning 20 classes at a time from CIFAR-100 [3] using SqueezeNet [6].

	Classification Accuracy (%)					
Total Classes Trained	**NS**	**MT** [203]	**WRST** [204]	**SEA (Proposed)**	**ES** [179]	**LCS** [179]
20	83.80	83.05	83.45	83.25	83.50	81.20
40	71.67	71.35	71.65	71.25	71.15	68.98
60	65.23	64.22	64.63	64.47	63.33	62.82

80	60.99	60.32	60.79	60.45	59.04	58.43
100	56.94	56.64	56.63	56.39	55.26	54.74

Table 3.10. Incrementally learning 20 classes at a time from CIFAR-100 [3] using ShuffleNet V2 [9].

	Classification Accuracy (%)					
Total Classes Trained	**NS**	**MT** [203]	**WRST** [204]	**SEA (Proposed)**	**ES** [179]	**LCS** [179]
20	84.60	83.85	84.35	84.15	83.55	83.45
40	74.98	74.20	74.65	74.08	73.38	73.40
60	69.03	68.38	68.73	68.47	67.25	67.18
80	65.85	65.30	65.48	65.09	64.30	64.13
100	63.06	62.32	62.65	62.27	61.30	61.40

To test SEA, the model is trained by learning a different number of classes incrementally i.e. training 10 and 20 classes incrementally on CIFAR-100 [3]. By applying the SEA algorithm to ES [179], and LCS [179], it can be seen from Table 3.9 and Table 3.10 that for each incremental training round, the classification accuracies obtained is less than 3% irrespective of the data sampling method with respect to the accuracies obtained without any data sampling. This shows that the SEA algorithm can be successfully integrated with various data sampling techniques, resulting in the transmission of fewer samples to the cloud and still retaining the model performance with respect to no data sampling. Figure 3.5 shows the other performances of incremental learning under various data sampling techniques such as the number of samples transmitted to the cloud, the training time on the cloud, the number of useful weights extracted from the classifier, and the number of iterations needed to find the useful weights.

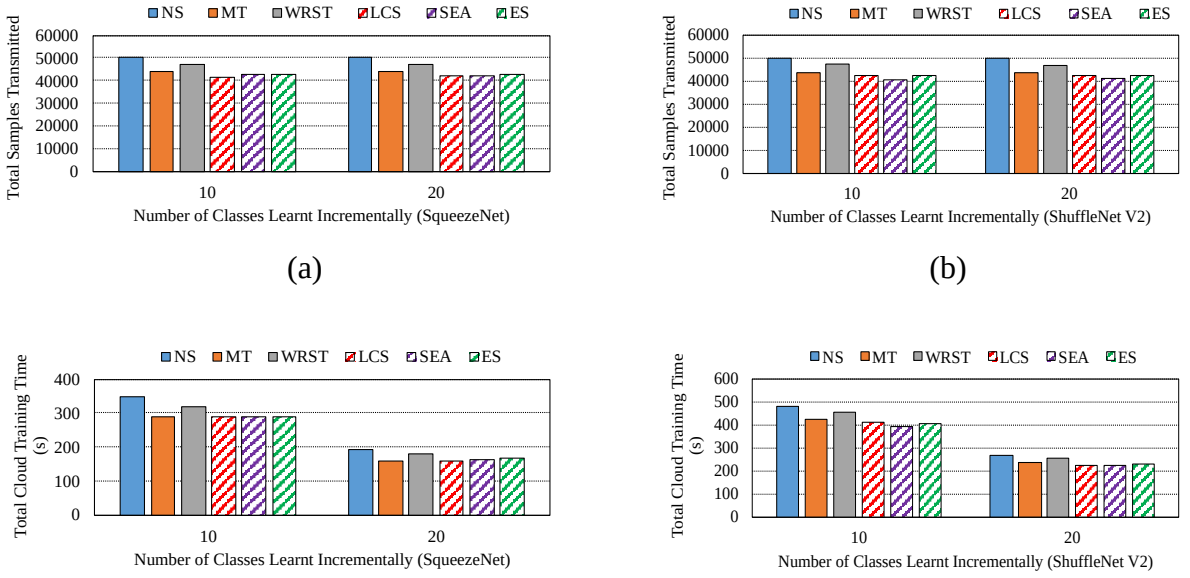


Figure 3.5. Total samples transmitted to the cloud (a) using SqueezeNet [6], (b) using ShuffleNet V2 [9]. Training time on cloud (c) using features from SqueezeNet [6], (d) using

features from ShuffleNet V2 [9] during evaluation on CIFAR-100 [3] dataset.

From Figure 3.5, all data sampling methods result in a smaller number of samples being transmitted to the cloud and a faster training time on the cloud as compared to NS. The WRST [204] method appears to be sending more samples to the cloud as compared to other data sampling techniques indicating that this method is able to quickly detect a statistically significant difference between the overall loss distribution and the distribution of the losses whose associated samples are to be transmitted to the cloud. Such quick detection of statistical significance difference is undesirable in such cases because it can clearly be seen that using other data sampling methods results in fewer samples being transmitted to the cloud as compared to WRST [204] while the model performance is similar to that of the accuracies obtained using NS.

It can be noted that learning 10 classes at a time takes more time than learning 20 classes at a time on the cloud because the fewer the number of classes to be learned at an incremental training round, the more the number of rehearsals needed for incremental learning thus the overall training time on the cloud increases. As Figure 3.5c and Figure 3.5d show, the cloud training time is faster after data sampling is performed at the IoT edge device because not all samples have been transmitted to the cloud. The classifier residing on the cloud is trained on lesser data samples when data sampling is applied at the IoT edge device.

Table 3.11 and Table 3.12 show the incremental learning classification accuracy when learning 40 classes at every incremental training round when evaluating CUB-200 [5] using SqueezeNet [6] and ShuffleNet V2 [9], respectively. Table 3.13 and Table 3.14 show the incremental learning classification accuracy when learning 50 classes at every incremental training round when evaluating CUB-200 [5] using SqueezeNet [6] and ShuffleNet V2 [9], respectively.

Table 3.11. Incrementally learning 40 classes at a time from CUB-200 [5] using SqueezeNet [6].

	Classification Accuracy (%)					
Total Classes Trained	**NS**	**MT** [203]	**WRST** [204]	**SEA (Proposed)**	**ES** [179]	**LCS** [179]
40	58.49	51.15	56.84	57.39	57.85	57.66
80	43.85	37.81	42.53	42.71	43.01	42.88
120	36.68	29.44	33.59	36.53	35.19	34.87
160	31.53	23.59	29.90	29.00	30.03	30.49
200	27.77	20.90	25.60	27.53	28.30	27.29

Table 3.12. Incrementally learning 40 classes at a time from CUB-200 [5] using ShuffleNet V2 [9].

	Classification Accuracy (%)					
Total Classes Trained	**NS**	**MT** [203]	**WRST** [204]	**SEA (Proposed)**	**ES** [179]	**LCS** [179]
40	71.99	63.73	69.15	71.63	69.77	69.97

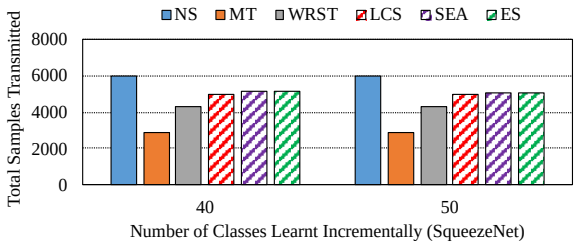
80	59.94	49.85	55.58	57.68	57.21	56.59
120	49.26	39.16	43.95	47.33	46.75	46.75
160	41.98	32.85	37.45	40.29	39.42	39.18
200	37.15	28.46	33.26	35.30	35.15	34.82

Table 3.13. Incrementally learning 50 classes at a time from CUB-200 [5] using SqueezeNet [6].

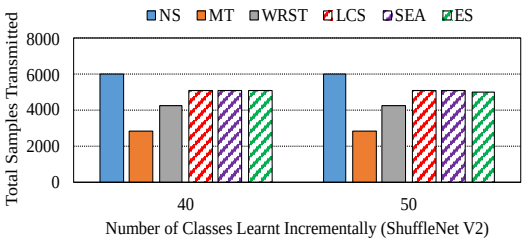
	Classification Accuracy (%)					
Total Classes Trained	**NS**	**MT** [203]	**WRST** [204]	**SEA (Proposed)**	**ES** [179]	**LCS** [179]
50	53.42	47.08	52.05	55.72	54.64	51.48
100	42.81	33.98	38.95	40.24	41.23	37.93
150	34.15	27.21	29.73	32.01	31. 28	29.06
200	27.48	21.36	25.47	27.53	26.96	27.89

Table 3.14. Incrementally learning 50 classes at a time from CUB-200 [5] using ShuffleNet V2 [9].

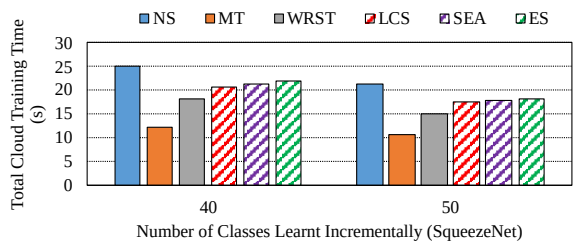
	Classification Accuracy (%)					
Total Classes Trained	**NS**	**MT** [203]	**WRST** [204]	**SEA (Proposed)**	**ES** [179]	**LCS** [179]
50	68.97	61.12	65.51	67.39	67.10	66.16
100	56.60	46.48	51.45	54.12	53.12	53.10
150	44.52	34.94	40.21	42.27	41.65	42.00
200	38.43	28.67	34.25	36.79	36.03	35.75



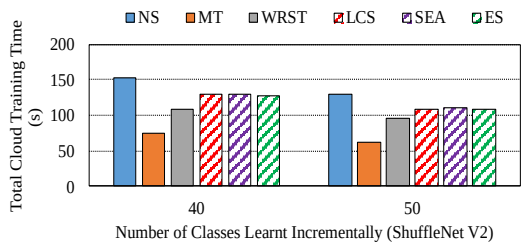
(a)



(b)



(c)



(d)

Figure 3.6. Total samples transmitted to the cloud (a) using SqueezeNet [6], (b) using ShuffleNet V2 [9]. Training time on cloud (c) using features from SqueezeNet [6], (d) using

features from ShuffleNet V2 [9] during evaluation on CUB-200 [5] dataset.

It can be seen from Table 3.11, Table 3.12, Table 3.13, and Table 3.14 that using MT [203] and WRST [204] results in a severe model performance degradation in terms of the classification accuracies at every incremental training round. This implies that when using such non-parametric data sampling techniques, a lot of images per class are discarded at the IoT edge device as evident from Figure 3.6a and Figure 3.6b, leading to very few samples being transmitted to the cloud which also results in a smaller cloud training time as compared to other data sampling algorithms. When using the MT [203] method, by the time this test detects a significant difference between the means of the selected samples with respect to the total number of samples per class, a lot of samples have already been discarded. The same applies for WRST [204] in which case a lot of samples already get discarded by the time the test finds a significant difference between the filtered samples and the overall dataset. This is very well because of the small number of images per class. Given the fact that the classification accuracies obtained when using the MT [203] method are extremely low, it means that WRST [204] is faster at detecting statistical differences between the entropies of samples to be transmitted as compared to the entropies of all the samples. However, both of these non-parametric data sampling methods fail to retain classification accuracies with respect to NS. This shows that a very high data sampling rate at the IoT edge device can lead to a very fast training time but at the expense of severe model performance degradation.

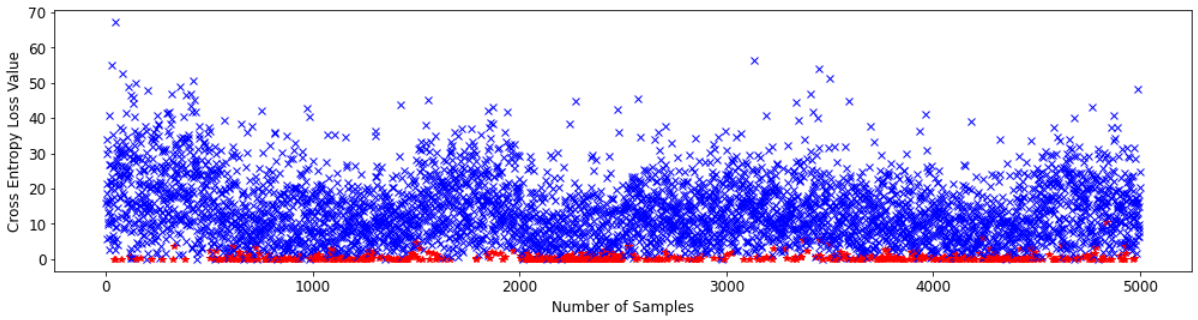
On the contrary, when the SEA algorithm is extended to ES [179] and LCS [179] data sampling methods, the training time of the classifier on the cloud is nearly the same because the number of samples transmitted to the cloud is also very similar for each of these methods as evident from Figure 3.6a, Figure 3.6b, Figure 3.6c, and Figure 3.6d. Table 3.11, Table 3.12, Table 3.13, and Table 3.14 show that the accuracy difference between NS and SEA is within 3% at every incremental training round with the exception of LCS [179] (SEA). When using LCS [179] (SEA), at certain incremental training rounds, the classification accuracies are less than 3%. LCS [179] uses SoftMax probabilities for data sampling, however, the results show that SoftMax probabilities of novel samples cannot be used as a basis for data sampling as it leads to a larger amount of catastrophic forgetting with respect to NS as compared to SEA and ES [179] (SEA).

3.5. Discussion

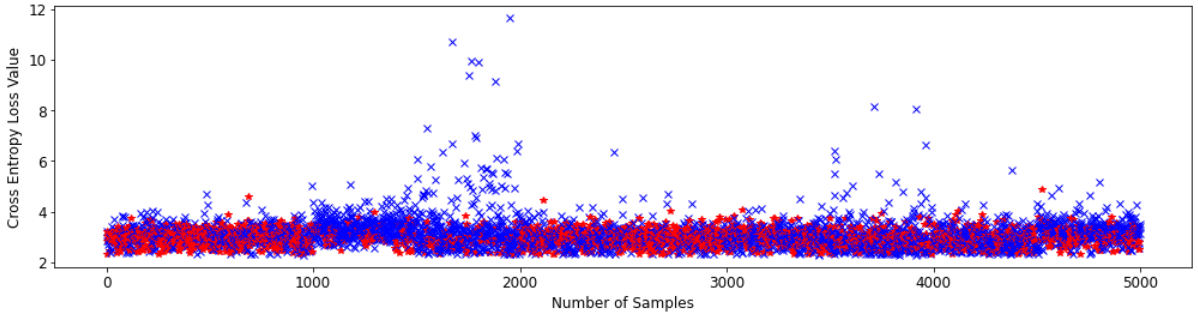
From all of the results obtained, WRST [204], MT [203], and KWT [202] are not suitable general data sampling algorithms because they lead to an extremely poor classification accuracy performance when evaluated on CUB-200 [5] which indicates that such methods are inducing a lot of catastrophic forgetting in incremental learning as compared to SEA. Furthermore, SEA leads to the highest classification accuracies obtained at the majority of the incremental training rounds for all the datasets combined.

The biggest advantage offered by SEA is in the reduction of the training samples followed by the reduction in the training time of the classifier on the server. In the context of fog computing, this leads to a big reduction in the communication cost between the fog device and the server along with an accelerated training time on the server, all of which is achieved by maintaining the model performances. In terms of the classification accuracies, what matters is the performance of the model after incrementally learning all the classes, in which case even after

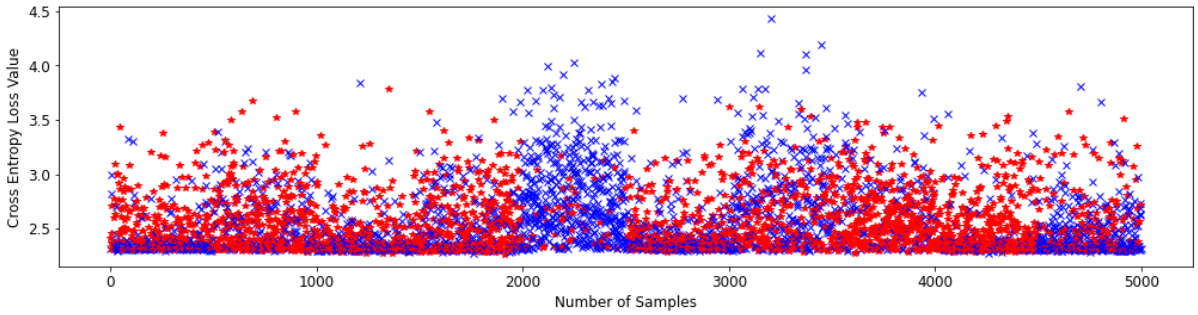
applying SEA, all the classification accuracies after the final incremental training round are less than 3% compared to classification accuracies obtained where no data sampling is used. In SEA, τ^c samples are discarded whereby τ^c samples in each class have entropies whose distance from the median of the entropy distribution is smaller than the standard deviation of the entropy distribution of that class. Due to this phenomenon, τ^c samples are not needed for training due to their extremely low variance with respect to the entropy distribution of the class as such samples will not aid the training of the neural network on the server. Thus, the results obtained in Table 3.6 support the theory of the data sampling algorithm (SEA).



(a) Initial cross entropy loss values of all samples with labels between 0-9 inclusive.



(b) Initial cross entropy loss values of all samples with labels between 50-59.



(c) Initial cross entropy loss values of all samples with labels between 90-99.

Figure 3.7. Initial cross entropy loss values of samples from CIFAR-100 [3] dataset on SqueezeNet [6] when learning 10 classes at a time using the same learning settings as shown in Table 3.1 when evaluating CIFAR-100 [3]. Red points represent samples whose entropy values increased after training and blue points represent samples whose entropy values decreased after training.

Applying SEA on small-scale datasets is challenging as there is not enough room for data sampling and even if data sampling does take place, the number of samples to be discarded should be chosen very carefully as a huge impact can be made on incremental learning in such

cases. For e.g., discarding one image from a total of hundred images results in only a 1% data discard rate but if the same one image is discarded from a total of ten images, then this results in a 10% data discard rate. Despite this, the proposed SEA is able to retain incremental learning performance for small datasets such as CUB-200 [5].

The reason why SEA works so well in terms of retaining incremental learning performance is simply because of the unpredictability of the novel classes themselves. When neural networks face novel classes and a cross-entropy loss is calculated for each image, the neural network can never guarantee that the loss values of all samples will decrease after training. On the contrary, it is possible that after training, the loss values of some samples might increase, this is especially true in the case of class incremental learning as seen in Figure 3.7.

Figure 3.7a shows the initial loss values of samples belonging to classes with labels: 0-9 inclusive of CIFAR-100 [3], Figure 3.7b shows the initial loss values of samples belonging to classes with labels: 50-59 of CIFAR-100 [3], and Figure 3.7c shows the initial loss values of samples belonging to the classes with labels: 90-99 inclusive of CIFAR-100 [3]. The points in red denote the loss values of samples that increased after an incremental training round, the points in blue denote the loss values of samples that decreased after an incremental training round. From Figure 3.7a it can be deduced that samples that have very low loss values are the ones that exhibit higher loss values after training. However, as incremental learning progresses, this no longer is the case. As incremental learning progresses, the pattern at which the samples in every incremental training round that exhibit higher loss values (red marks) after training become increasingly randomized.

Since the occurrence of entropy values that increase after training (red points) becomes increasingly randomized as incremental learning progresses, a certain number of samples are discarded in a randomized manner as this increases the probability of discarding a sample that might not turn out useful for incremental learning. Hence, SEA is the most consistent performing data sampling algorithm after being integrated with SEA as observed in the experiments that are conducted.

3.6. Summary

This chapter proposes a novel data sampling technique called Sample Elimination Algorithm (SEA) which can dynamically discard several samples from a training class and train only the remaining training samples to the cloud for further neural network training. This algorithm tackles the key issue of high transmission cost between the edge device and the cloud in the case where high dimensional data such as images need to be transmitted to the cloud for training. SEA is compared with three non-parametric statistical tests namely: Wilcoxon Rank-Sum Test (WRST) [204], Median Test (MT) [203], and Kruskal-Wallis Test (KWT) [202]. It is shown that SEA is a more generalized data sampling algorithm as compared to the other non-parametric statistical tests for data sampling. Additionally, the data sampling algorithms are also capable of reducing the neural network training time on the cloud since there are fewer samples to train. To test the proposed solutions, the data sampling algorithms are applied to various CNN architectures being trained on different datasets with randomly chosen hyperparameters. All these algorithms are executed on the edge device before training even starts on the cloud.

Out of all the experiments carried out, a statistical significance test could also have been performed where the test would show that there is no statistical difference between the accuracy distributions with and without data sampling but there is a statistical difference between the distribution of the number of samples transmitted to the cloud as this would show that the model performance is maintained while largely reducing the number of samples transmitted to the cloud. However, this test was not conducted because the number of data points are quite small to carry out such tests as the number of incremental training rounds are not many. To perform a statistical significance test, a lot more data points are needed (minimum of 30) because if the sample size is too small then disproportionate number of individuals may end up being included which are outliers and anomalies which may skew the results. However, Figure 3.4. does highlight the performance of SEA by showing the amount of accuracy change with respect to the amount of transmission cost reduction.

A limitation of the sample elimination algorithm is that the change in the model performance due to the sampling cannot be predicted in advance. Hence, the hyperparameter ω^c in Equation 3.4 needs to be finetuned depending on how much change in the model performance can be tolerated. Another limitation of this algorithm is that ω^c must be set manually and cannot automatically be deduced based on the acceptable model performance change.

From the results obtained, it can be concluded that SEA can adapt to input datasets of various sizes, different training hyperparameters, a different number of classes being learned incrementally, and still maintain incremental learning performance despite using fewer training images as compared to the total available samples. SEA can also reduce the overall time taken for incremental learning. This is for the case when working with the convolutional layers of CNNs for feature extraction of images and using Artificial Neural Networks (ANN) as classifiers for training. When training samples have been transmitted to the cloud after applying SEA, and after the classifier has been trained on the cloud, the trained neural network may also need to be transmitted back to the edge device. Neural networks have a large number of parameters, hence, transmitting all the parameters back to the edge device also leads to a high transmission cost. Chapter 4 provides details into how this problem can be solved.

Chapter 4

Parameter Extraction of Artificial Neural Networks in Class Incremental Learning

4.1. Background

As seen in Chapter 3 that it is possible to discard novel data samples at the IoT edge device before training even begins which leads to a huge transmission cost reduction and training time reduction on the cloud with a very small effect on the model's classification accuracy. Once an artificial neural network-based classifier is trained on the cloud, its trained parameters must be transmitted back to the IoT edge device to update the model residing at the IoT edge device. However, in a given CNN, a large number of parameters reside in the fully connected layers i.e., the ANN-based classifier. Hence, transmitting all of the classifier parameters from the cloud to the IoT edge device can lead to a huge transmission cost. This is true especially in the case where the fully connected layers of an ANN contain millions of parameters which means the cost of transmitting these many weights back to the IoT edge device can be very high. Hence, a solution must exist on the cloud to quickly extract important parameters from the neural network for them to be transmitted back to the edge devices without losing the model performance.

The main objective of this chapter is to select only a subset of the parameters of a trained neural network to reduce the transmission cost from the cloud to the edge devices which basically translates to transmitting fewer parameters belonging to a neural network and still retaining the model performance. It can be argued that if the objective is to reduce the transmission cost from the cloud to the edge device, techniques such as Knowledge Distillation (KD) [16], and Pruning [17], [18], [20], [21], [39] can be used to reduce the size of the neural networks which can then be transmitted back to the edge device. However, the problem with KD [16] and Pruning [17], [18], [20], [21], [39] is that they only focus on reducing the size of a neural network and that too via re-training of the model. This chapter argues that even if the ANN size can be reduced, the following question still remains: are all of the parameters in the neural network still useful?. This is a very valid and important question because no matter how much the size of the model has reduced, there is no meaning in transmitting weights of a neural network to an edge device that are not useful at all. This is why a weight extraction algorithm must be in place to select only the useful parameters of a neural network regardless of its size.

This chapter is largely based on the work of [10] that develops a method to extract useful weights from a neural network. This chapter provides an algorithm that does achieve the same results as in [10] but drastically reduces the computational time taken to find the useful weights. The weight extraction algorithm proposed in this chapter can find a certain number of useful weights without the need of re-training the model thus leading to a faster computational time.

The following is the nomenclature that will be used throughout this chapter.

Nomenclature	
C^{l}_{old}	Parameters of layer l of the classifier on the cloud before training
C^{l}_{updated}	Parameters of layer l of the classifier on the cloud after training

C^l_{diff}	Differences of values of parameters between C^l_{updated} and C^l_{old}
$\text{sorted}C^l_{\text{diff}}$	Sorted values of C^l_{diff} in ascending order
$\text{Accept}_{\text{loss}}$	Acceptable training accuracy loss
A_{ori}	Training accuracy of the trained classifier on the cloud
q	Number of quantiles
T^l	List of thresholds of layer l of the classifier on the cloud
$C^l_{\text{temporary}}$	Parameters of layer l of the temporary model
$\text{Clist}^l_{\text{temporary}}$	Parameters of layer l of the temporary classifier in a list holding every temporary model
A_{temp}	Training accuracy of $C_{\text{temporary}}$
W^l_{best}	Useful weights of layer l of the trained classifier on the cloud
W_{best}	Set of useful weights of each layer of trained classifier on the cloud
I^l_{best}	Indices corresponding to W^l_{best}
I_{best}	Set of indices corresponding to W_{best}
M_{ec}	String containing feature maps and labels transmitted from the IoT edge device to the cloud
M_{ce}	String containing W_{best} and I_{best} transmitted from cloud to the IoT edge device
P	Number of partitions in a neural network's weight difference distribution

4.2. Neural Network Weight Extraction Algorithm

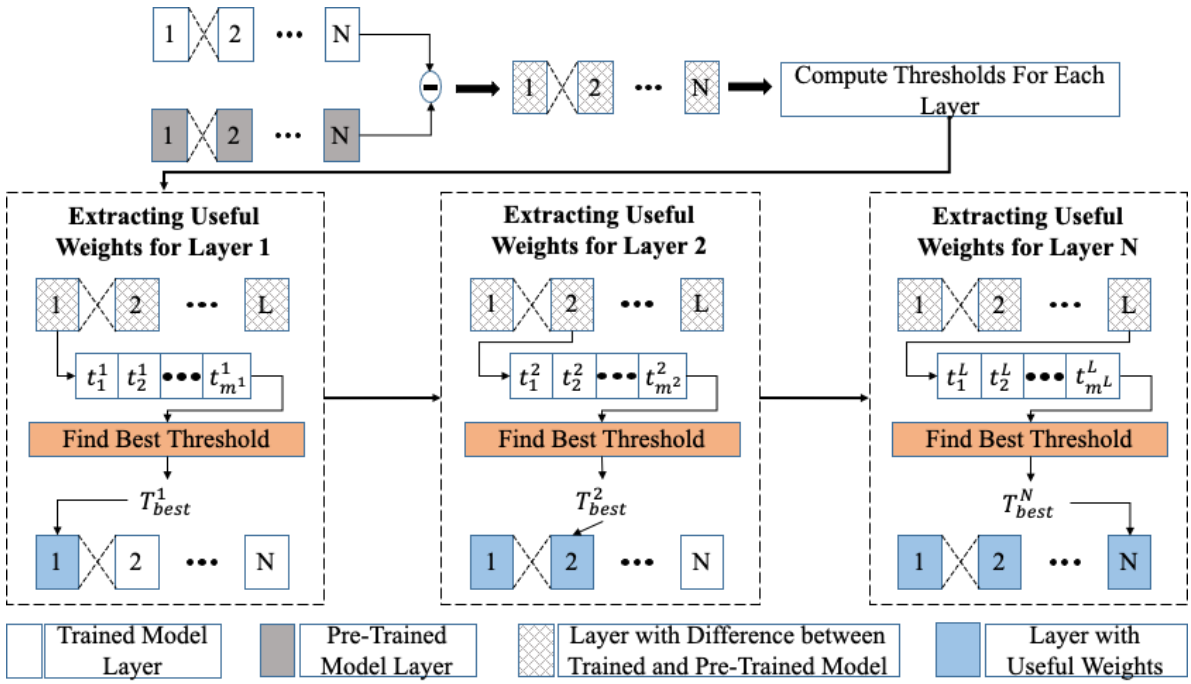
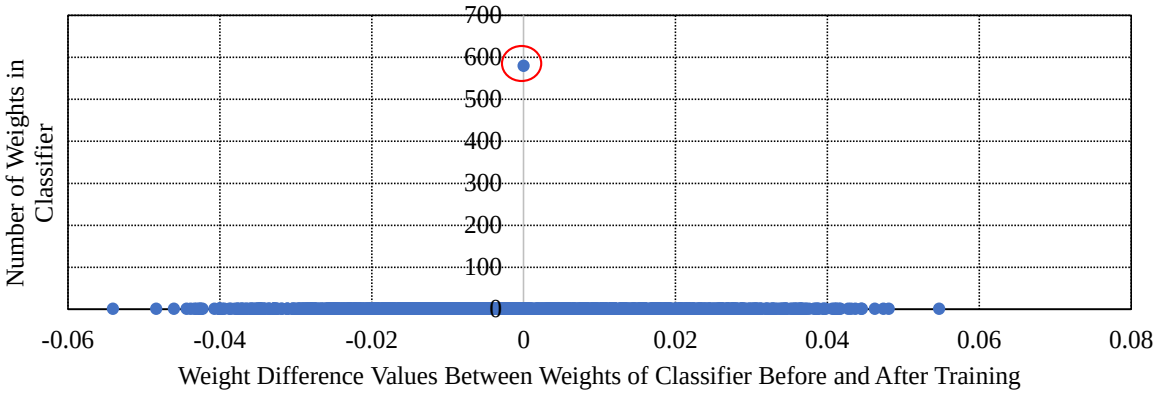


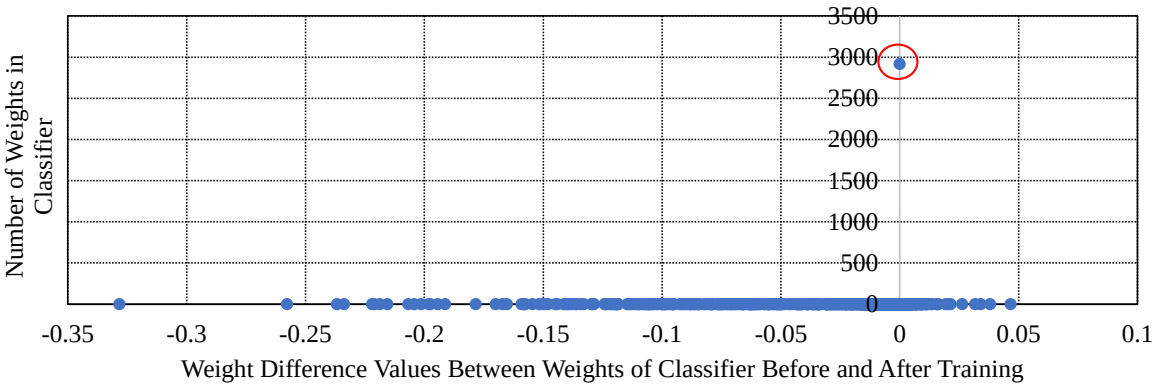
Figure 4.1. Overview of the weight extraction procedure.

Figure 4.1 shows the diagram of the weight extraction algorithm. The basic idea of this algorithm is to extract the useful weights of a neural network layer by layer. The key element here is to obtain a weight difference between the trained model and the pre-trained model. The weight differences can then act as threshold values whereby the threshold values can be compared against the weight differences based on which it can be determined whether the

weight value at a given index of the trained model is useful or not.



(a)



(b)

Figure 4.2. Distribution of the weight difference between the classifier on the cloud before and after training when learning (a) the first 10 classes (b) last 10 classes of the CIFAR-100 [3] dataset with rehearsal (training all samples accumulates so far) using the SqueezeNet [6] model.

After training the classifier on the cloud, the updated parameters of the classifier need to be transmitted back to the IoT edge device to keep the model on the IoT edge device up to date. The work in [10] shows that it is not mandatory for all the weights of a trained model on the cloud to be transmitted back to the IoT edge device as there are a number of parameters in a model that do not change after the training i.e. the difference between the weight value of a connection in the trained classifier and the pre-trained classifier is the same. The strategy proposed in FitCNN [10] performs useful weight extraction layer by layer. For each layer, the FitCNN [10] algorithm takes the difference between the weights of a layer of the trained and the pre-trained model and divides the weight difference distribution into 30 quantiles to obtain a threshold list. For each threshold value, if the absolute weight difference between a classifier's post-trained weight and pre-trained weight is greater than the threshold value, then for that particular connection, the post-trained weight value will be considered useful, or else the pre-trained weight value will be considered useful. A temporary model is then formed that has the same architecture as the trained model which has a mixture of useful and non-useful weights. The training accuracy is then computed on this temporary model. For every threshold

value, the constructed temporary model is different in terms of the number of useful parameters. Therefore, for every threshold value, the training accuracy, the number of useful weights, and the indices of the useful weights of the temporary model are stored. For a given threshold, as soon as the difference between the training accuracy of the temporary model and the trained model is greater than the pre-defined acceptable training accuracy loss, the useful weights and indices associated with the previous threshold value are considered useful weights for the layer of the model being processed. Next, the layer of the trained model being processed is frozen, and the next layer of this model undergoes the same weight extraction process. This is how the useful weights of a trained model are extracted in FitCNN [10].

However, in the case of class incremental learning, the number of negative weight differences can increase with respect to the training rounds due to catastrophic forgetting. Figure 4.2 shows the frequency distribution of the weight differences where the *x*-axis represents the value of the weight difference and the *y*-axis represents the frequency i.e., the number of times a specific weight difference occurs. For example, the points highlighted in the red circles in Figure 4.2 show the number of weight differences whose values are zero. This means the value of such weights before and after training remain the same. So, it is not necessary to send such weights back to the IoT edge device. Algorithm 4.1 shows the improved weight extraction algorithm as compared to [10].

Algorithm 4.1: Improved Neural Network Weight Extraction Algorithm (FitCNN+)

Input : C_{updated}, C_{old}, $\text{Accept}_{\text{loss}}$, A_{ori}, q
Output : W_{best} and I_{best}
for l in L **do**
  $C^{l}_{\text{diff}} = C^{l}_{\text{updated}} - C^{l}_{\text{old}}$
  sorted$C^{l}_{\text{diff}} \leftarrow$ sort (C^{l}_{diff})
  Divider $\leftarrow$ length of sortedC^{l}_{diff}/q
  for w_{diff} in sortedC^{l}_{diff} **do**
    if $|w_{\text{diff}}| > 0$ **then**
      $[T^{l}] \leftarrow w_{\text{diff}}$ at every divider
    end if
  end for
  $C_{\text{temporary}} \leftarrow C_{\text{updated}}$
  for threshold in T^{l} **do**
    for w_{diff} in C^{l}_{diff} **do**
      if $|w_{diff}| <$ threshold **then**
        $C^{l}_{\text{temporary}} \leftarrow C^{l}_{\text{old}}$ at respective index
      end if
    end for
    $[\text{Clist}_{\text{temporary}}] \leftarrow C_{\text{temporary}}$
    $A_{\text{temp}} \leftarrow$ Get $C_{\text{temporary}}$ accuracy on training data
    if $(A_{\text{ori}} - A_{\text{temp}}) \geq \text{Accept}_{\text{loss}}$ **then**

20 : W^l_{best}, $I^l_{\text{best}} \leftarrow \text{Clist}^l_{\text{temporary}}$ at previous index of list, indices of W^l_{best}
21 : **break**
22 : **else**
23 : W^l_{best}, $I^l_{\text{best}} \leftarrow \text{Clist}^l_{\text{temporary}}$ at current index of list, indices of W^l_{best}
24 : **end if**
25 : **end for**
26 : $[W_{\text{best}}] \leftarrow W^l_{\text{best}}$
27 : $[I_{\text{best}}] \leftarrow I^l_{\text{best}}$
28 : **end for**

As shown in Figure 4.2b, if the FitCNN [10] strategy is used for class incremental learning, the threshold list will contain more negative threshold values which will simply yield the same number of useful weights. This is because according to the FitCNN [10] algorithm if the absolute value of a weight difference is greater than a threshold, the corresponding weight of the post-trained model will be considered useful. However, the absolute value of any weight difference value will always have positive values. Hence, the condition in line 13 of Algorithm 4.1 will not be satisfied if the threshold is negative thus all the weights of the trained model will be considered useful. Choosing negative threshold values, therefore, yields a lot of redundancy and is also time-consuming, especially for class incremental learning. In class incremental learning, the size of the training dataset can be very large as new tasks arrive which is why there is a clear need to extract useful weights using as few thresholds as possible from the weight difference distribution list.

This chapter proposes to sort the weight difference distribution of each layer in ascending order and use only non-negative threshold values (line 6 of Algorithm 4.1) in order to speed up the process of finding the minimum useful weights of a trained classifier with negligible training accuracy loss. Algorithm 4.1 shows the improved version of the original FitCNN [10] algorithm which is named as FitCNN+ [28] in this chapter. In Algorithm 4.1, all the weight matrices are converted into 1-D vectors to make it easier to work with indexing. For each layer in the model, the weight differences are computed between the weights of the layers before and after training which the weight differences are then sorted in ascending order. Just like the FitCNN [10] algorithm, the weight difference distribution of each layer is divided into 30 quantiles. However, the weight difference value at every quantile is stored as a threshold only if the weight difference value is non-negative. Next, a temporary model with the same architecture as the classifier on the cloud is created which has the same parameters as the newly trained model but if the weight difference of a parameter of the trained model at a specific index is less than the threshold, the weight of the pre-trained model will be inserted at the given index into the temporary model. The accuracy of the temporary model is computed using the data used to train the model at the respective incremental training round. If the difference between the accuracy of the temporary model and the newly trained model is within A_{ori}, then the process of finding the useful weights continues using a larger weight difference threshold value i.e., the next threshold value in the threshold list. The larger the threshold value, the higher the chances of the condition in line 13 being satisfied thus the number of useful weights of a layer decreases, and as a result, reduces the communication costs when transmitting only the useful weights of the trained model back to the IoT edge device.

Once the difference between the accuracy of the temporary model and the original trained model is greater than A_{ori}, then the weights of the latest temporary model are considered to be the final set of useful weights and the iterative process will stop. The proposed FitCNN+ [28] algorithm reduces the number of iterations needed to find the useful weights as compared to the original FitCNN [10] algorithm by simply using positive threshold values of the weight difference distribution. At the same time, this algorithm ensures that the accuracy of the temporary model is similar to the accuracy of the original trained model. Once the useful weights are determined for a specific layer in the temporary model, the parameters of that layer are frozen in order to find the useful weights in the next layer of the temporary model that undergoes the same weight extraction process.

Apart from the useful weights, it is also mandatory to send the indices of the weights of the trained model back to the IoT edge device. This is to let the IoT edge device know which specific connections of a layer of the model must be updated. Indices are chosen depending on the number of useful weights. If the number of useful weights are greater than the number of non-useful weights, then the number of indices that indicate the position of the useful weights will be larger. However, to make the communication more efficient, if the number of useful weights is greater than the number of non-useful weights then the indices of the non-useful weights will be sent back to the IoT edge device. The message format of sending the weights and indices back to the IoT edge device is expressed in Equation 4.1 below.

$$M_{\text{ce}} = < W_{\text{best}} >, i, < I_{\text{best}} >, b, < B_{\text{trained}} >, < \text{act} > ! \tag{4.1}$$

W_{best} is converted into a 1-D vector and as shown in Equation 4.1, this flattened weight vector is converted into a comma-separated string. The character $'i'$ denotes that the next sequence of comma-separated strings is the indices. The character $'i'$ denotes the next sequence of comma-separated strings are the biases. The term "act" is either the character $'y'$ or $'n'$. Character $'y'$ indicates that the indices present in the string are exactly where the weights need to be assigned in the weight matrix on the IoT edge device and the character $'n'$ indicates that the indices present in the string are where the updates are not needed.

4.3. FitCNN++

Algorithm 4.1 already shows how the process of finding the useful weights in a neural network can be accelerated as compared to the original FitCNN [10] algorithm. However, one drawback of this approach is that this algorithm still needs to find the useful weights of a neural network using all the positive threshold values in an ascending order until the least number of weights to be transmitted back to the edge device can be found. To improve this process, the aim is to reduce the iterations needed to find the best threshold. To further accelerate this process, the FitCNN++ algorithm is proposed in this section. This algorithm is the same as FitCNN+ [28] however, the only difference is that only a subsample of the positive thresholds needs to be used to find the useful weights. To solve this problem, since the threshold values are sorted in an ascending order, this distribution can be split into P sections. P refers to the number of sections that the threshold distribution can be broken down into. For illustration purposes in Figure 4.3, P is set to 4. A visual representation is shown in Figure 4.3.

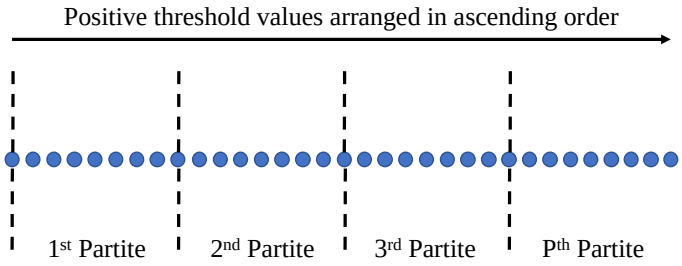


Figure 4.3. Partitioning the weight difference between the post-trained and pre-trained neural network into multiple sections.

In Figure 4.3, the blue dots represent the weight differences, and the vertical dashed lines represent the weight difference values where the weight difference distribution is partitioned. By segmenting the thresholds into four segments, there are 4 sections, i.e., the 1^{st} section lies at the 0^{th} percentile index, the 2^{nd} section lies at the 25^{th} percentile index, the 3^{rd} section lies at the 50^{th} percentile index, and the 4^{th} section lies at the 75^{th} percentile index. The thresholds in each of the sections are obtained through which the useful weights of the classifier are extracted starting from the very first section. The idea behind this algorithm is that as soon as a threshold value at a given percentile index leads to a training accuracy less which is beyond the acceptable training loss then the optimum threshold value lies between the previous percentile index and the current percentile index. To clarify once again, the optimum threshold here refers to the threshold that leads to the extraction of the minimum number of useful weights in the classifier with respect to the acceptable training loss.

As a simple example, if the useful weights are extracted using the threshold at the 0^{th} index then these useful weights are assigned to a temporary classifier which is then used to compute the training accuracy of the model. If the difference between the training accuracy the baseline training accuracy is less than the acceptable training loss, then the threshold at the 25^{th} percentile index is used to extract and assign the useful weights to a temporary classifier. If the training accuracy obtained is still within the acceptable training loss, then the algorithm knows that all the thresholds between the 0^{th} index and the 25^{th} percentile index are not needed to obtain the useful weights. The algorithm then uses the threshold at the 50^{th} percentile index to obtain the training accuracy and if the training accuracy is still within the acceptable training loss, then the algorithm will not use any of the thresholds that have indices that lie between the 25^{th} percentile index and the 50^{th} percentile index. In such a scenario, there is a guarantee that the threshold that will minimize the number of useful weights with respect to the acceptable training loss will have an index value that is greater than the 50^{th} percentile index value. However, if the training accuracy obtained due to the threshold value at the 75^{th} percentile index value is beyond with acceptable training loss, then it is guaranteed that the optimum threshold value lies between the 50^{th} and the 75^{th} percentile index. This way, the time and the iterations needed to extract the least amount of useful weights are reduced even further as compared to the FitCNN+ [28].

It should be noted that there are no theoretical guarantees that setting $P = 4$ will lead to the fastest weight extraction time since P is a hyperparameter that must be set manually. However, it is clear that dividing the thresholds into sections and checking the training accuracy at every section can help determine which specific section holds the optimum threshold value which can lead to a faster weight extraction process.

4.4. Experiments

For all of the experiments below, five weight extraction algorithms are tested which are as follows: FitCNN [10], FitCNN+ [28], FitCNN++, random weight extraction, and magnitude weight extraction. For random and magnitude-based weight extraction, the number of parameters to be pruned away is increased by 1% after computing the model training accuracy. This is repeated as long as the training accuracy of the model is within 0.5%. For random neural network weight extraction, the weights of every layer of an ANN are pruned randomly as long as the training accuracy is within the acceptable limits. For magnitude-based weight extraction, the weights with the lowest absolute L1 values are pruned away, the weights of each layer are pruned away as long as the training accuracy is within the acceptable limits. The acceptable training accuracy loss ($\text{Accept}_{\text{loss}}$) is set to 0.5%. The number of quantiles q is set to 30. All of the experiments are performed after SEA has been applied on the edge device to make sure that neural network weight extraction algorithm works with SEA as well. For these experiments, the same datasets, models, and hyperparameters are used as the ones used in chapter 3. The testing accuracies reported in this chapter are the top-5 classification accuracies after applying SEA.

Four different CNN models are chosen for the experiments and each of the CNN model is used to evaluate each one of the 4 datasets. The reason for choosing multiple CNNs is that the size of the feature extractor output of each CNN is different, hence the size of the classifier is also different, which is why the weight extraction algorithms must be tested on various classifier sizes. The reason why multiple datasets are chosen is that different datasets tend to fit neural networks differently during the training process for e.g., depending on the dataset, neural networks may overfit, underfit, or generalize well. This is why multiple datasets must also be used to test the weight extraction algorithms to see how they perform when neural networks fit differently after training. The CNN that will be used to evaluate a given dataset has been chosen randomly since the objective of the weight extraction algorithm is to be able to extract the useful weights of an ANN irrespective of the CNN feature extractor and the dataset. The purpose of these experiments is to observe the weight extraction rate and the time taken for performing weight extraction on an artificial neural network depending on the weight extraction algorithm used.

4.5. Evaluation on CIFAR-100 using MnasNet

Table 4.1-Table 4.5 show the time taken for weight extraction, the number of useful weights extracted, and the classification accuracy at every incremental training round after performing various weight extraction algorithms when evaluating CIFAR-100 [3] using MnasNet [4].

Table 4.1. Useful weights extracted and the time taken via FitCNN++ (Evaluating CIFAR-100 [3] using MnasNet [4]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
20	954000	736503	77.20	17.62
40	960000	504003	52.50	41.58
60	966000	489303	50.65	61.22
80	972000	510003	52.47	80.17
100	978000	551703	56.41	91.56

Total	**4830000**	**2791515**	**57.80**	**292.14**

Table 4.2. Useful weights extracted and the time taken via FitCNN+ [28] (Evaluating CIFAR-100 [3] using MnasNet [4]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
20	954000	590403	61.89	18.32
40	960000	502203	52.31	73.99
60	966000	473703	49.04	80.48
80	972000	496803	51.11	154.59
100	978000	611703	62.55	230.55
Total	**4830000**	**2674815**	**55.38**	**557.93**

Table 4.3. Useful weights extracted and the time taken via FitCNN [10] (Evaluating CIFAR-100 [3] using MnasNet [4]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
20	954000	679603	71.24	52.03
40	960000	458803	47.79	80.52
60	966000	496403	51.39	161.50
80	972000	507603	52.22	214.72
100	978000	479003	48.98	196.99
Total	**4830000**	**2621415**	**54.27**	**705.76**

Table 4.4. Useful weights extracted and the time taken via random weight extraction (Evaluating CIFAR-100 [3] using MnasNet [4]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
20	954000	430476	45.12	12.63
40	960000	336394	35.04	25.53
60	966000	239894	24.83	34.90
80	972000	110313	11.35	36.13
100	978000	193315	19.77	56.64
Total	**4830000**	**1310392**	**27.13**	**165.83**

Table 4.5. Useful weights extracted and the time taken via magnitude weight extraction (Evaluating CIFAR-100 [3] using MnasNet [4]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
20	954000	38709	4.06	7.07
40	960000	23194	2.42	12.70

60	966000	23194	2.40	20.15
80	972000	23194	2.39	27.81
100	978000	23194	2.37	35.69
Total	**4830000**	**131485**	**2.72**	**103.42**

It can be observed that random weight extraction and magnitude weight extraction take the least amount of time to extract the useful weights as compared to the other algorithms. The reason for this is because when the weights of the neural network are pruned, the architecture of the neural network changes which can immediately cause a loss in performance as observed in Table 4.1-Table 4.5. This is the reason why the number of useful weights extracted is nearly the same as the original parameters of the neural network which cannot reduce the transmission cost from the cloud to the edge device by a big margin. However, the FitCNN [10] weight extraction algorithm takes a lot more time to find the useful weights. This is because a lot more forward passes are executed to calculate the training accuracy. The higher the threshold, the lower the classification accuracy, however, if the time taken to extract the useful weights is larger, it means that a larger threshold still leads to a training accuracy that is similar to the original model. This directly implies that the number of useful weights is less hence which can ease the transmission cost from the cloud to the edge device. The results obtained after applying the FitCNN [10] weight extraction algorithm also indicate that at least 50% of the neural network parameters after training are very similar to the parameters of the neural network before training when CIFAR-100 [3] is evaluated.

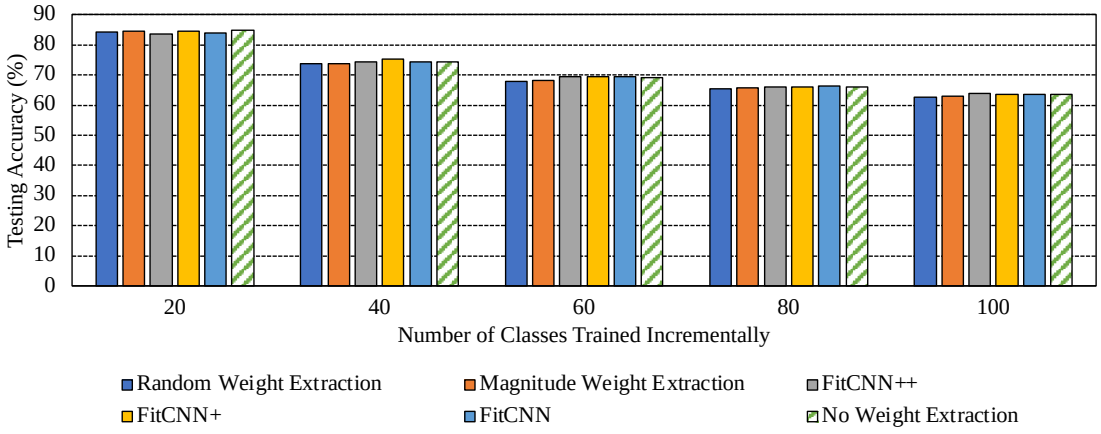


Figure 4.4. Testing accuracy of the neural network at every incremental training round when evaluating CIFAR-100 [3] using various weight extraction algorithms.

Figure 4.4 shows the testing accuracy of the neural network at every incremental training round when evaluating CIFAR-100 [3]. The bar with diagonal green lines denotes the classification accuracies obtained without performing any weight extraction after training the classifier. It can be noted that regardless of the weight extraction algorithm, all of the classification accuracies are very similar to the ones obtained without any weight extraction algorithm. In such a scenario, FitCNN++ is ideal for performing the weight extraction as it is able to retain the performance of the neural network and is also able to extract the least amount of weight from a classifier in a very short time.

4.6. Evaluation on Stanford Dogs using MobileNetV2

Table 4.6 - Table 4.10 show the time taken for weight extraction, the number of useful weights extracted, and the classification accuracy at every incremental training round after performing various weight extraction algorithms when evaluating Stanford Dogs [7] using MobileNetV2 [8].

Table 4.6. Useful weights extracted and the time taken via FitCNN++ (Evaluating Stanford Dogs [7] using MobileNetV2 [8]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
40	460800	391681	85.00	4.52
80	921600	783361	85.00	8.84
120	1382400	1175041	85.00	13.03
Total	**2764800**	**2350083**	**85.00**	**26.38**

Table 4.7. Useful weights extracted and the time taken via FitCNN+ [28] (Evaluating Stanford Dogs [7] using MobileNetV2 [8]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
40	460800	368641	80.00	10.65
80	921600	783361	85.00	22.36
120	1382400	1175041	85.00	33.23
Total	**2764800**	**2327043**	**84.17**	**66.25**

Table 4.8. Useful weights extracted and the time taken via FitCNN [10] (Evaluating Stanford Dogs [7] using MobileNetV2 [8]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
40	460800	384001	83.33	15.55
80	921600	798721	86.67	32.49
120	1382400	1198081	86.67	48.02
Total	**2764800**	**2380803**	**86.11**	**96.06**

Table 4.9. Useful weights extracted and the time taken via random weight extraction (Evaluating Stanford Dogs [7] using MobileNetV2 [8]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
40	460800	353385	76.69	11.11
80	921600	619323	67.20	19.36
120	1382400	775020	56.06	24.94
Total	**2764800**	**1747728**	**63.21**	**55.40**

Table 4.10. Useful weights extracted and the time taken via magnitude weight extraction (Evaluating Stanford Dogs [7] using MobileNetV2 [8]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
40	460800	37189	8.07	5.84
80	921600	46451	5.04	7.94
120	1382400	69677	5.04	12.10
Total	**2764800**	**153317**	**5.55**	**25.87**

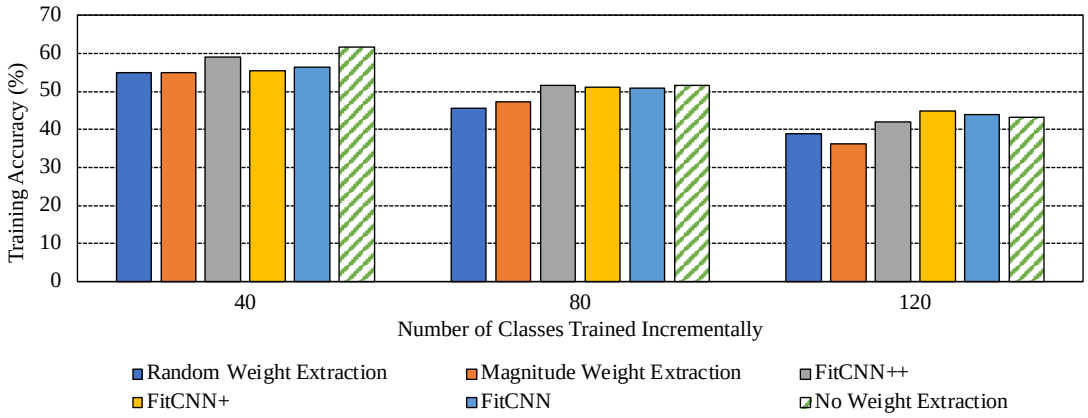


Figure 4.5. Testing accuracy of the neural network at every incremental training round when evaluating Stanford Dogs [7] using MobileNetV2 [8] using various weight extraction algorithms.

From Figure 4.5 it can be seen that random and magnitude weight extraction always lead to a testing accuracy much lower than that obtained without any weight extraction. After learning 40 classes, the accuracies obtained via FitCNN [10] and FitCNN+ [28] are also similar to the testing accuracies obtained by random and magnitude weight extraction. However, as incremental learning progresses and the number of connections in the classifier starts to increase, the performance of FitCNN [10] and FitCNN+ [28] in terms of retaining the classifier performance is the same as compared to without any weight extraction at all. After learning the

full 120 classes, FitCNN++ is able to best retain the performance of the classifier as compared to the other weight extraction algorithms.

4.7. Evaluation on ImageNet using ShuffleNetV2

Table 4.11-Table 4.15 show the time taken for useful weight extraction, the number of useful weights extracted, and the classification accuracy at every incremental training round after performing various weight extraction algorithms when evaluating ImageNet [1] using ShuffleNetV2 [9].

Table 4.11. Useful weights extracted and the time taken via FitCNN++ (Evaluating ImageNet [1] using ShuffleNetV2 [9]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
2	16386000	7592313	46.33	4.32
4	16388000	6363302	38.83	8.95
6	16390000	3270732	19.96	12.19
Total	**49164000**	**17226347**	**35.04**	**25.46**

Table 4.12. Useful weights extracted and the time taken via FitCNN+ [28] (Evaluating ImageNet [1] using ShuffleNetV2 [9]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
2	16386000	8287622	50.58	7.84
4	16388000	7985641	48.73	20.03
6	16390000	5694748	34.75	26.91
Total	**49164000**	**21968011**	**44.68**	**54.78**

Table 4.13. Useful weights extracted and the time taken via FitCNN [10] (Evaluating ImageNet [1] using ShuffleNetV2 [9]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
2	16386000	8286054	50.57	14.09
4	16388000	3898153	23.79	19.71
6	16390000	2973569	18.14	26.64
Total	**49164000**	**15157776**	**30.83**	**60.45**

Table 4.14. Useful weights extracted and the time taken via random weight extraction (Evaluating ImageNet [1] using ShuffleNetV2 [9]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
2	16386000	15938807	97.27	87.84
4	16388000	14199361	86.64	57.01
6	16390000	15516365	94.67	88.59
Total	**49164000**	**45654533**	**92.86**	**233.44**

Table 4.15. Useful weights extracted and the time taken via magnitude weight extraction (Evaluating ImageNet [1] using ShuffleNetV2 [9]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
2	16386000	329318	2.01	2.27
4	16388000	825817	5.04	6.50
6	16390000	164187	1.00	4.93
Total	**49164000**	**1319322**	**2.68**	**13.70**

The most surprising result from these experiments is that random weight extraction is able to extract very few useful parameters thus massively saving the transmission cost. Furthermore, the classification accuracy of the model does not get affected. This is a confirmation that the majority of the weights are not useful for transmitting back to the edge device. However, from a deep learning perspective, this means that throughout the training process of the classifier, the weights of the neural network did not change largely. This also means that the entire model was able to learn the patterns in this dataset in the very early training stages i.e., in the very first few epochs. This is why the weights of the neural network did not change largely after the initial training phase. The most likely reason for this behaviour is that the ImageNet [1] dataset was used for this experiment and ShuffleNet V2 [9] has been pre-trained on this same dataset as well. This is why for this incremental learning experiment, the features that are extracted from the convolutional layers of ShuffleNet V2 [9] are very discriminatory in nature hence the fully connected layers are so quick to learn the patterns in the dataset.

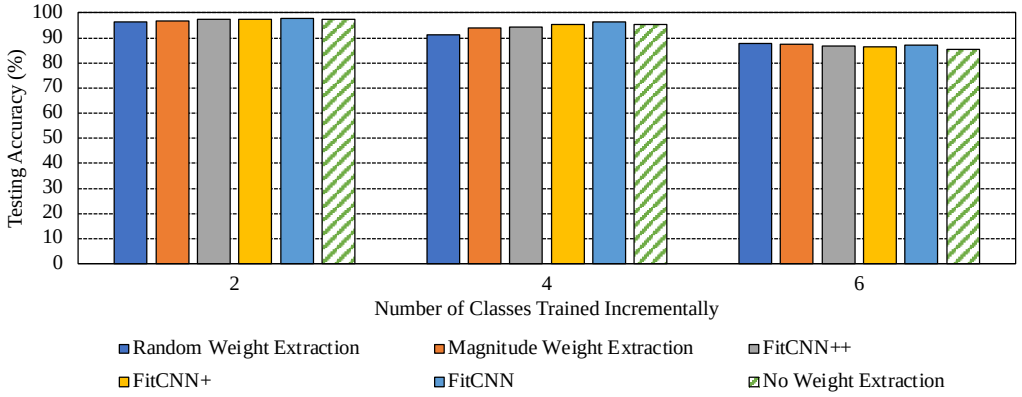


Figure 4.6. Testing accuracy of the neural network at every incremental training round when evaluating ImageNet [1] using ShuffleNetV2 [9] using various weight extraction algorithms.

It can be noted that random weight extraction takes the highest amount of time to find the useful parameters. This is because the rate of sparsity increment is 1% and at every iteration, the sparsity leads to a training accuracy that is very close to the baseline accuracy hence, it takes a lot of iterations to sparsify the network enough whose performance starts to deteriorate. This is what leads to a larger time to find the useful parameters.

From Figure 4.6, it can be seen that during the first 2 incremental training rounds, random and magnitude weight extraction algorithms lead to testing classification accuracies that are slightly lower than the other weight extraction algorithms. All 3 variants of the FitCNN [10] algorithm are largely able to retain the performance of the neural network as compared to the accuracies obtained without weight extraction. In this case, magnitude weight extraction is not needed as its weight selection ability is much lower than the other algorithms. Moreover, other weight extraction algorithms can also largely retain the model performance in terms of the classification accuracy. All 3 FitCNN [10] variants retain the model performance, however, the random weight extraction algorithm is also nearly as good as these 3 FitCNN [10] variants and the amount of transmission cost reduction back to the IoT edge device is extremely large as compared to any other weight extraction algorithms.

4.8. Evaluation on CUB-200 using SqueezeNet

Table 4.16-Table 4.20 show the time taken for weight extraction, the number of useful weights extracted, and the classification accuracy at every incremental training round after performing various weight extraction algorithms when evaluating CUB-200 [5] using SqueezeNet [6].

Table 4.16. Useful weights extracted and the time taken via FitCNN++ (Evaluating CUB-200 [5] using SqueezeNet [6]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
50	640000	353978	55.31	1.38
100	1280000	711739	55.60	2.73
150	1920000	977891	50.93	4.79
200	2560000	1308081	51.10	7.69
Total	**6400000**	**3351689**	**52.37**	**16.58**

Table 4.17. Useful weights extracted and the time taken via FitCNN+ [28] (Evaluating CUB-200 [5] using SqueezeNet [6]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
50	640000	307725	48.08	2.54
100	1280000	783903	61.24	7.46
150	1920000	1048991	54.63	11.93
200	2560000	1353310	52.86	18.15
Total	**6400000**	**3493929**	**54.59**	**40.10**

Table 4.18. Useful weights extracted and the time taken via FitCNN [10] (Evaluating CUB-200 [5] using SqueezeNet [6]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
50	640000	197838	30.91	6.69
100	1280000	582525	45.51	14.06
150	1920000	1099231	57.25	22.57
200	2560000	1413168	55.20	30.69
Total	**6400000**	**3292762**	**51.45**	**74.01**

Table 4.19. Useful weights extracted and the time taken via random weight extraction (Evaluating CUB-200 [5] using SqueezeNet [6]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
50	640000	91169	14.24	1.96
100	1280000	135315	10.57	2.78
150	1920000	218887	11.40	4.24
200	2560000	559263	21.85	9.23
Total	**6400000**	**1004634**	**15.70**	**18.21**

Table 4.20. Useful weights extracted and the time taken via magnitude weight extraction (Evaluating CUB-200 [5] using SqueezeNet [6]).

Classes Learnt	Total Parameters in Classifier	Parameters not being transmitted to the edge device	Reduction in Transmission Cost (%)	Time taken to find useful parameters (s)
50	640000	6400	1.00	0.85
100	1280000	12843	1.00	1.44
150	1920000	19512	1.02	2.10
200	2560000	26188	1.02	2.84
Total	**6400000**	**64943**	**1.01**	**7.23**

In this experiment, magnitude weight extraction is the fastest in terms of weight extraction but is the worst performing algorithm when it comes to selecting the useful weights to reduce the transmission cost from the cloud to the edge device. This behaviour is followed by the random weight extraction algorithm that is able to reduce the transmission cost further but also takes more processing time. All 3 variants of the FitCNN [10] algorithm are able to achieve nearly the same transmission cost reduction however, the FitCNN++ algorithm is the fastest out of

the three variants. FitCNN++ requires less time for weight extraction as compared to magnitude weight extraction and is also able to extract fewer important weights.

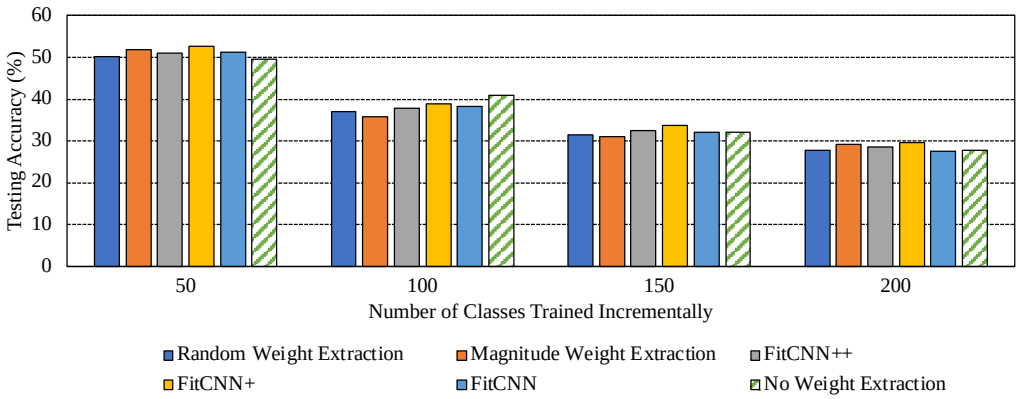


Figure 4.7. Testing accuracy of the neural network at every incremental training round when evaluating CUB-200 [5] using SqueezeNet [6] using various weight extraction algorithms.

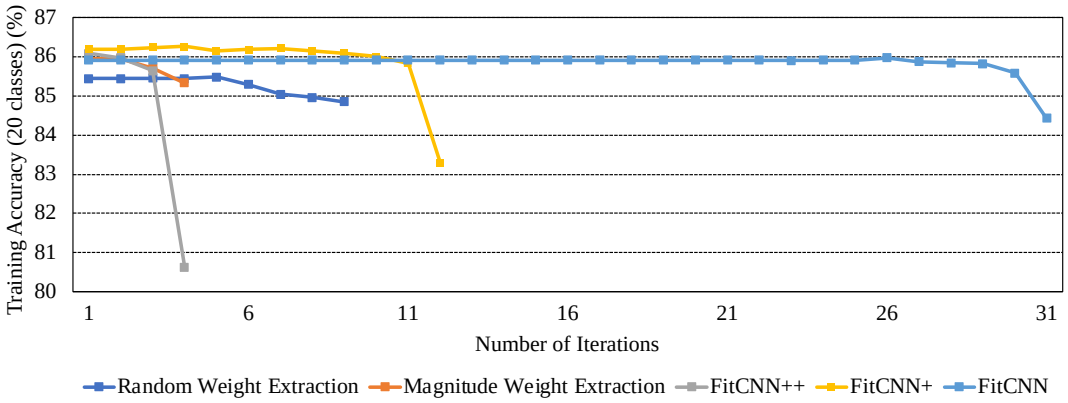
It can be seen from Figure 4.7 that out of the 3 FitCNN [10] variants, at least one of them always outperforms random and magnitude weight extraction algorithms at every incremental training round. Only at the 2nd incremental training round i.e., after learning 100 classes, the classification accuracies obtained are lower than the ones obtained without weight extraction. Apart from this incremental training round, all the 3 variants of the FitCNN [10] variants are able to retain the classification performance of the neural network which is on-par or in some cases even better than the model performance obtained without weight extraction.

4.9. Effect of Weight Extraction on Training Accuracies

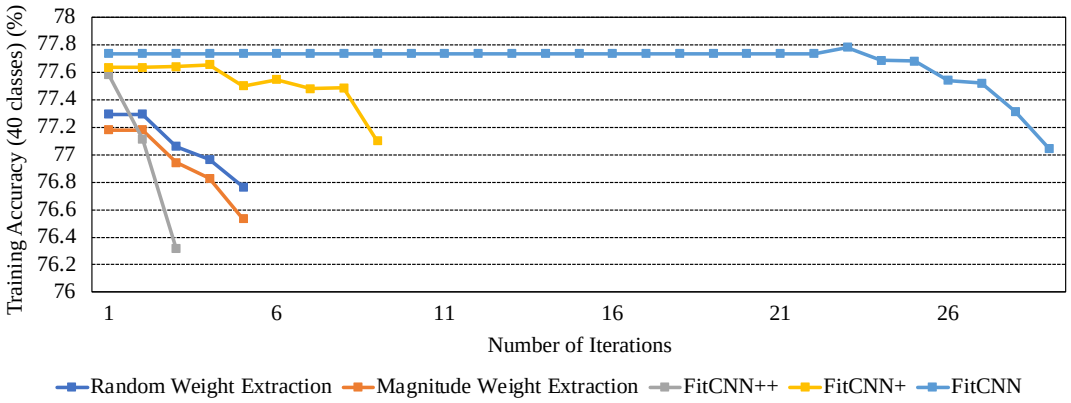
In this section, the rate of training accuracy loss with respect to the number of useful weights extracted is observed at every incremental training round. This gives insights into the number of iterations required to complete the weight extraction process depending on the weight extraction algorithm. The number of iterations refers to the number of forward passes made on the classifier to obtain the training accuracy of the classes being trained. As mentioned that this training accuracy is used to determine whether to continue the weight extraction process or not. The training accuracy values will be observed up until the point where the difference between the baseline training accuracy and the training accuracy after weight extraction is beyond the acceptable training loss.

4.9.1. Model Training Accuracy During Evaluation of CIFAR-100 Using MnasNet

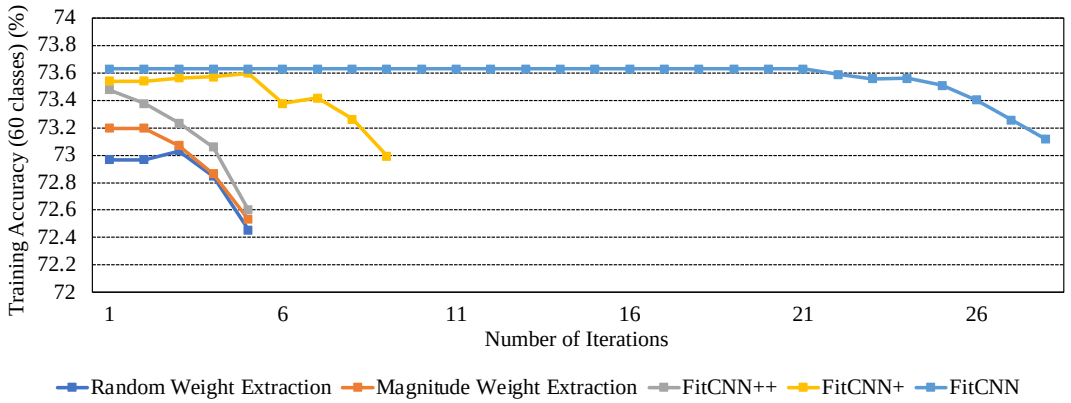
Figure 4.8 shows the training accuracy of the classifier with respect to the number of iterations when incrementally training a classifier on the classes of CIFAR-100 [3] dataset using the feature extractor of the pre-trained MnasNet [4] architecture for extracting features.



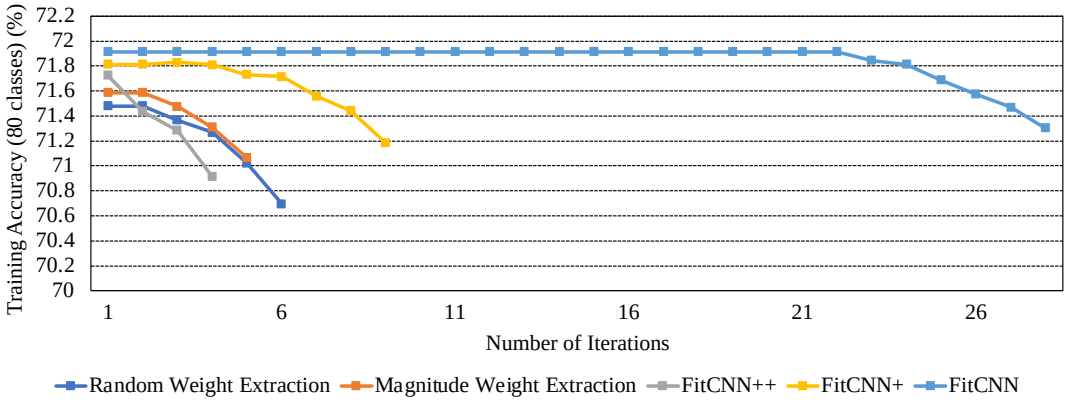
(a)



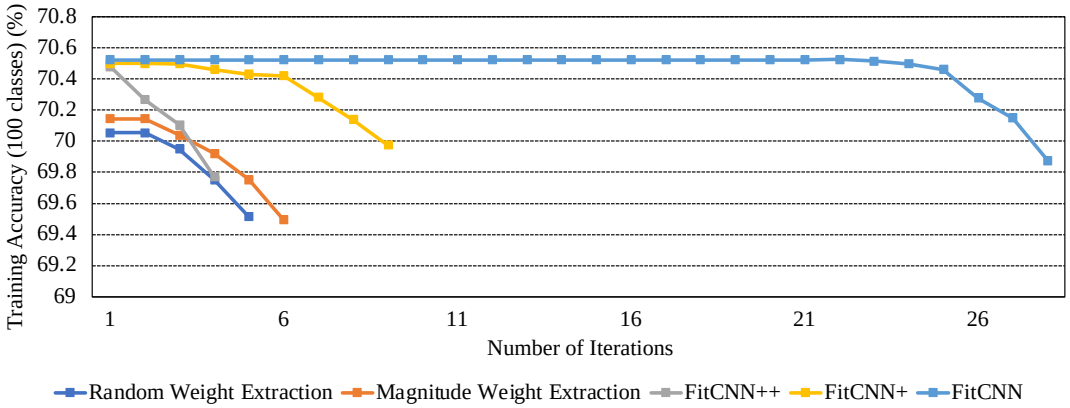
(b)



(c)



(d)



(e)

Figure 4.8. Relation between the training accuracy of the classifier with respect to the number of iterations when evaluating CIFAR-100 [3] using MnasNet [4] after learning (a) 20 classes, (b) 40 classes, (c) 60 classes, (d) 80 classes, (e) 100 classes.

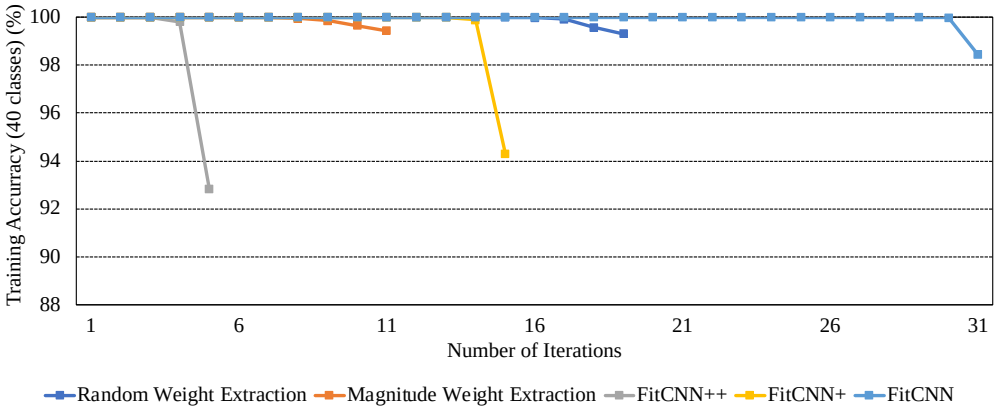
Figure 4.8 shows the training accuracy of the classifier during the weight extraction process at various incremental training rounds. This figure also includes the baseline training accuracy of the classifier (training accuracy of the classifier before performing weight extraction – the 1st marker point from the left for each of the weight extraction algorithms), which is why in some cases, the x-axis of the figures may show values greater than 30 which is the number of thresholds needed to perform weight extraction using the FitCNN [10]. When evaluating the CIFAR-100 [3] dataset, 20 classes are learned incrementally at a time which equates to 5 incremental training rounds since this dataset has 100 classes. At every incremental training round when weight extraction is performed, FitCNN [10] takes the longest time to perform weight extraction as it always uses 30 thresholds to compute the training accuracy of the classifier. FitCNN+ [28] manages to largely reduce the number of iterations as compared to the original FitCNN [10]. This is made possible by using only the positive threshold values as explained in Algorithm 4.1 in this chapter. FitCNN++ is able to complete the weight extraction process much faster than the original FitCNN [10] and the FitCNN+ [28] as well. The magnitude and random-based weight extraction algorithms are faster than the original FitCNN [10] and the FitCNN+ [28] at every incremental training round, however, these two weight

extraction algorithms are not intelligence based hence the rate of accuracy loss is fast and the number of useful weights extracted is very high hence the transmission cost reduction is very low as seen in Table 4.1 – Table 4.5. FitCNN++ requires the least number of iterations when it comes to performing weight extraction when evaluating the CIFAR-100 [3] dataset, the only exception can be seen when performing weight extraction is after learning 60 classes whereby the number of iterations needed to extract the useful weights is the same when using random weight extraction, magnitude weight extraction, and FitCNN++.

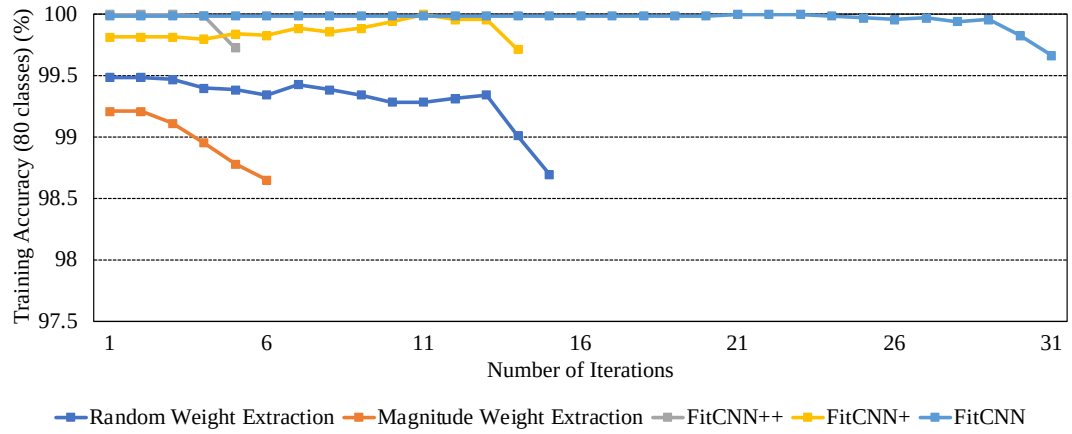
It is evident that the positive weight difference values are needed for performing a faster weight extraction process, however, choosing a threshold value at every quantile can still be slow and there is a scope of improving the execution speed further. Instead of checking the model accuracy, at every positive threshold value. The reason why FitCNN++ is faster is because it applies the same threshold search method as FitCNN [10] and FitCNN++ but only on a subset of positive threshold values. FitCNN++ divides the positive weight distribution into small partitions and starts searching from the largest threshold value to see if the threshold accuracy is still within the acceptable accuracy loss. If this is true then the weight extraction process becomes very fast, if this is not the case, then the threshold at the previous partitions are checked. When the threshold at a partition point is within the acceptable accuracy difference and the next threshold is beyond the acceptable accuracy difference, then all weights differences between the 2 partitions points are chosen are the subset. The same threshold technique as used in FitCNN [10] and FitCNN+ is then applied only on this subset. This is what leads to a faster weight extraction time.

4.9.2. Model Training Accuracy During Evaluation of Stanford Dogs Using MobileNetV2

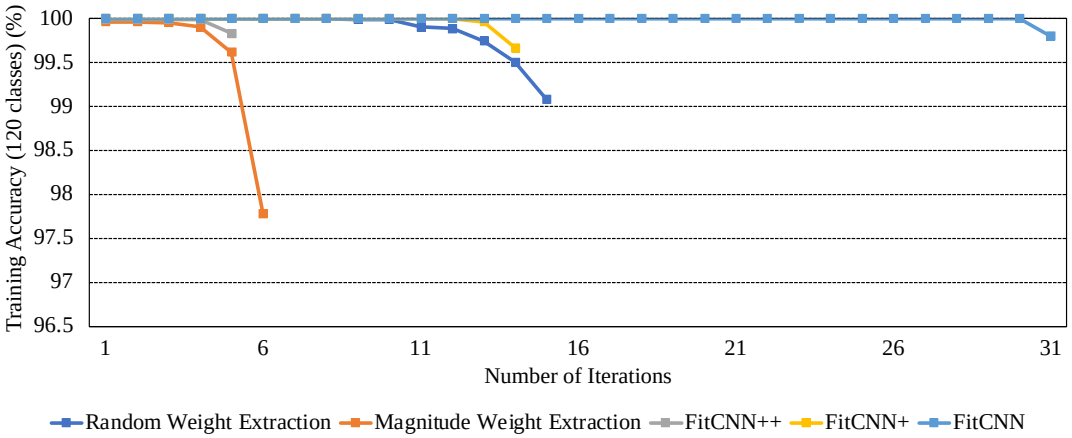
Figure 4.9 shows the training accuracy of the classifier with respect to the number of iterations when incrementally training a classifier on the classes of Stanford Dogs [7] dataset using the feature extractor of the pre-trained MobileNetV2 [8] architecture for extracting features.



(a)



(b)



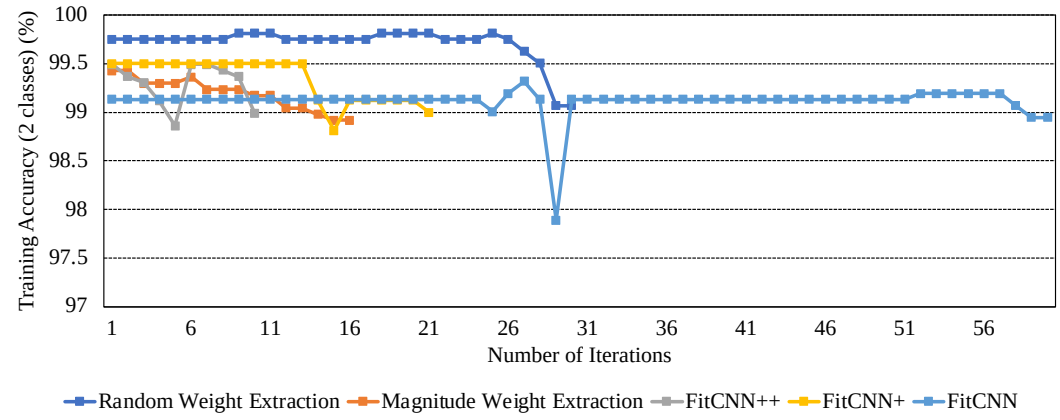
(c)

Figure 4.9. Relation between the training accuracy of the classifier with respect to the number of iterations when evaluating Stanford Dogs [7] using MobileNetV2 [8] after learning (a) 40 classes, (b) 80 classes, (c) 120 classes.

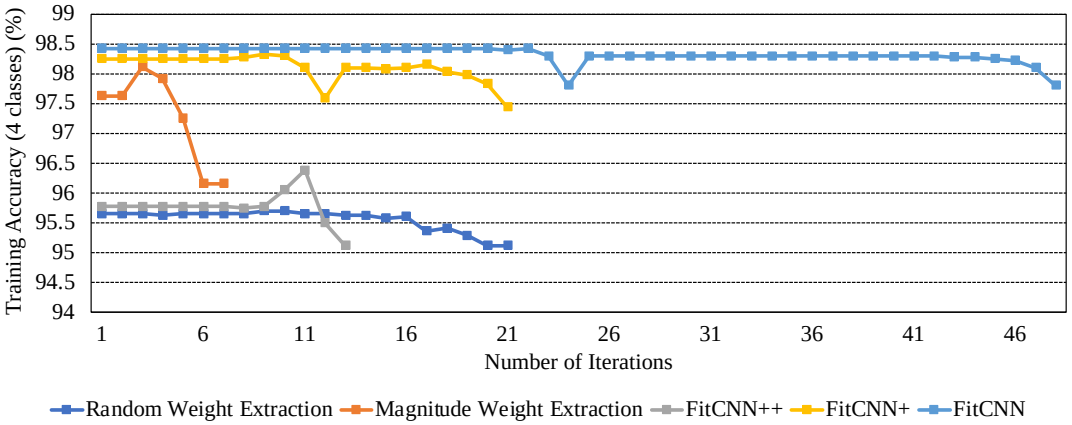
From the high training accuracies in Figure 4.9 and the low test accuracies as shown in Table 4.6-Table 4.10, it is evident that the classifier is overfitting when evaluating Stanford Dogs [7]. This is because when the model is overfitting, the training accuracy is very high due to a bias towards the training dataset which causes errors in the testing dataset and hence a lower testing classification accuracy. However, despite the overfitting phenomenon, FitCNN++ is able to retain the classifier performance as compared to the classification accuracies obtained without any weight extraction. Only after learning 40 classes are the accuracies due to weight extraction algorithms lower than the ones compared to without weight extraction, however, after the next 2 incremental training rounds, the performance of all 3 FitCNN [10] variants is on par or even better than the classification accuracy without weight extraction. The classification accuracies obtained via random and magnitude weight extraction is lower than all 3 FitCNN [10] variants.

4.9.3. Model Training Accuracy During Evaluation of ImageNet Using ShuffleNetV2

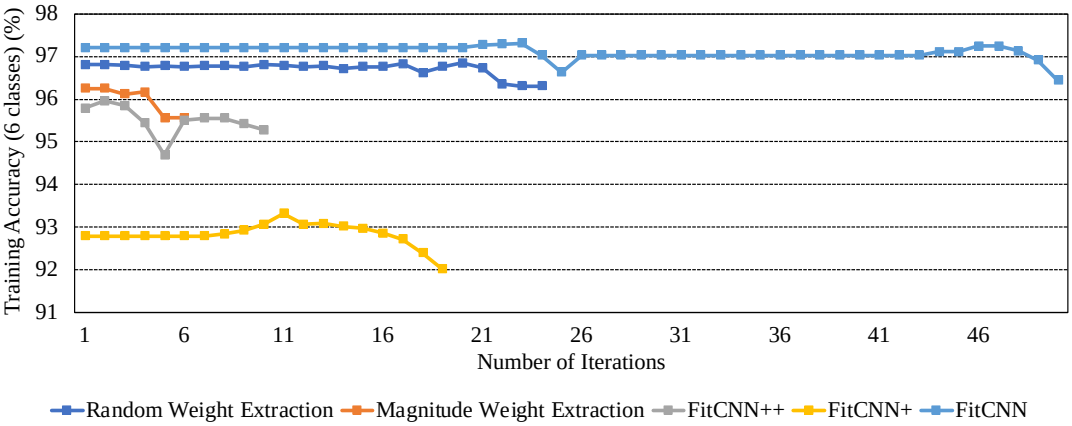
Figure 4.10 shows the training accuracy of the classifier with respect to the number of iterations when incrementally training a classifier on the classes of ImageNet [1] dataset using the feature extractor of the pre-trained ShuffleNetV2 [9] architecture for extracting features.



(a)



(b)



(c)

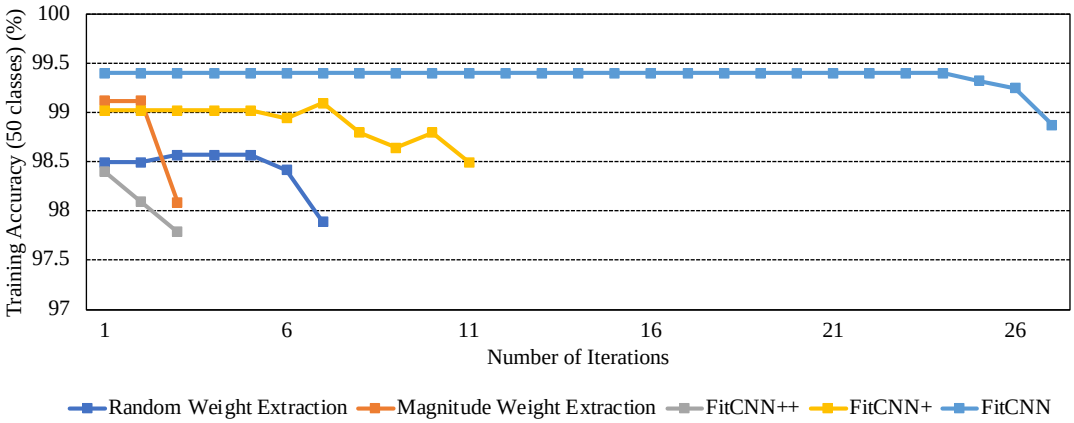
Figure 4.10. Relation between the training accuracy of the classifier with respect to the number of iterations when evaluating ImageNet [1] using ShuffleNetV2 [9] after learning (a) 2 classes, (b) 4 classes, (c) 6 classes.

From Figure 4.10 it appears that the training accuracy is very high at every incremental training round during the weight extraction process similar to what is observed in Figure 4.9, however, the testing accuracy when evaluating ImageNet [1] is also very high and only slightly lower than the training accuracy of the classifier. This shows that the model has undergone a good fit. The reason why the number of iterations needed to find the useful weights is much larger when evaluating ImageNet [1] as compared to other datasets is that 2 fully connected layers are implemented in the classifier when evaluating this dataset and the weight extraction algorithm works by performing weight extraction layer by layer. At the very first incremental training round i.e., after learning 2 classes, FitCNN++ requires the least number of iterations to extract the useful weights, however, during the next two incremental training rounds, the magnitude weight extraction algorithm requires the least amount of iterations needed to find the useful weights. These two incremental training rounds (after learning 4 and 6 classes) are the only stages where FitCNN++ is not able to outperform the other weight extraction algorithms in terms of needing the least number of iterations. However, as compared to the magnitude weight extraction algorithm, the FitCNN++ leads to a much larger communication reduction cost when transmitting weights back to the edge device.

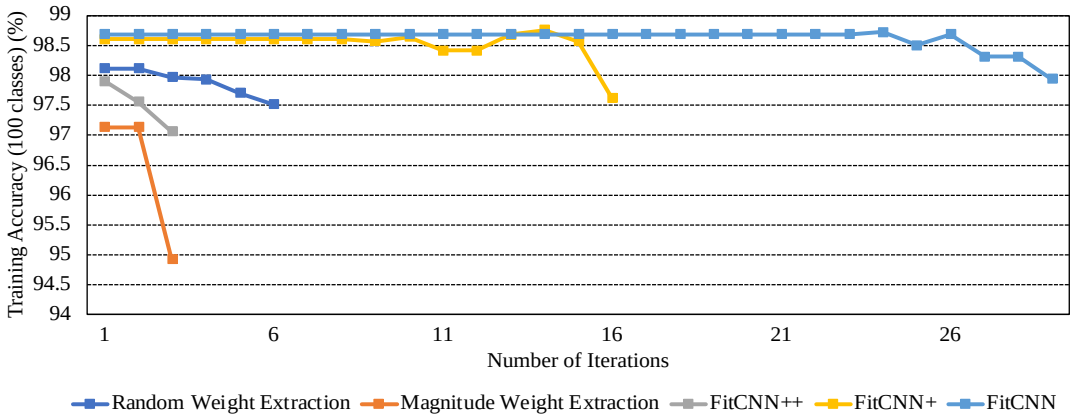
FitCNN [10] takes the most iterations to find the useful weights, however, the random weight extraction takes the longest time to extract the useful weights as shown in Table 4.11-Table 4.15. The primary reason for this is the number of parameters in the classifier when evaluating ImageNet [1]. In the experiments provided in this chapter, the smallest initial number of SoftMax units in the SoftMax classification layer is the lowest when evaluating ImageNet [1] i.e., only 2 classes are trained incrementally. Despite this, the number of parameters in the classifier for this experiment is larger than the number of parameters when evaluating other datasets in this chapter. Furthermore, in random weight extraction, once the useful weights are extracted and the updated classifier leads to an accuracy that is still similar to the baseline accuracy of the classifier, then this updated classifier needs to be discarded and the trained classifier again needs to undergo random weight extraction from scratch, this time with a higher number of parameters being deemed not useful. Hence, this experiment gives 2 main insights i.e., the number of iterations is not directly proportional to the time taken for completing the weight extraction process. Secondly, the performance of weight extraction algorithms is not only dependent on the mathematical nature of the algorithm itself but also on the number of parameters present in a neural network.

4.9.4. Model Training Accuracy During Evaluation of CUB-200 Using SqueezeNet

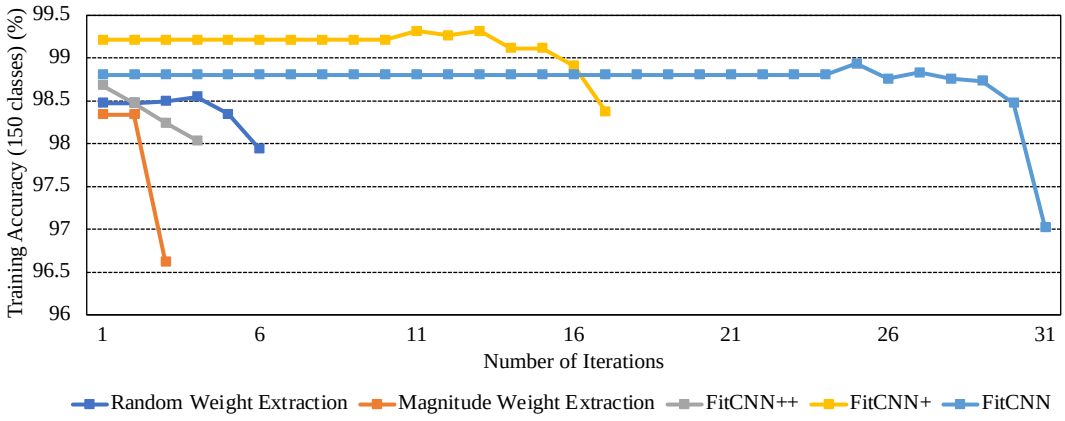
Figure 4.11 shows the training accuracy of the classifier with respect to the number of iterations when incrementally training a classifier on the classes of the CUB-200 [5] dataset using the feature extractor of the pre-trained SqueezeNet [6] architecture for extracting features.


Training Accuracy (50 classes) (%)
100
99.5
99
98.5
98
97.5
97
1
6
11
16
21
26
Number of Iterations
Random Weight Extraction
Magnitude Weight Extraction
FitCNN++
FitCNN+
FitCNN

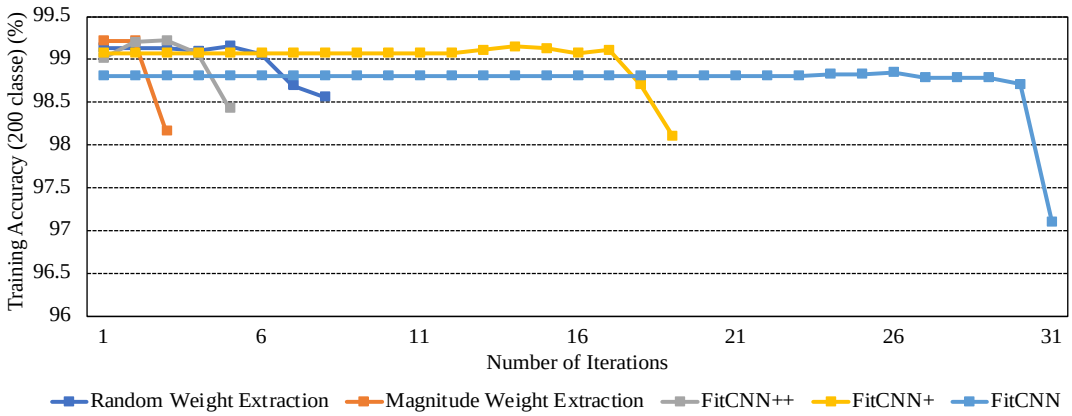
(a)


Training Accuracy (100 classes) (%)
99
98.5
98
97.5
97
96.5
96
95.5
95
94.5
94
1
6
11
16
21
26
Number of Iterations
Random Weight Extraction
Magnitude Weight Extraction
FitCNN++
FitCNN+
FitCNN

(b)


Training Accuracy (150 classes) (%)
99.5
99
98.5
98
97.5
97
96.5
96
1
6
11
16
21
26
31
Number of Iterations
Random Weight Extraction
Magnitude Weight Extraction
FitCNN++
FitCNN+
FitCNN

(c)



(d)

Figure 4.11. Relation between the training accuracy of the classifier with respect to the number of iterations when evaluating CUB-200 [5] using SqueezeNet [6] after learning (a) 50 classes, (b) 100 classes, (c) 150 classes, (d) 200 classes.

From Figure 4.11, it can be seen that during the first 2 incremental training rounds (after learning 50 and 100 classes incrementally), the number of iterations required by FitCNN++ and magnitude weight extraction is the same, and during the next 2 incremental training rounds, magnitude weight extraction takes the least amount of iterations. Furthermore, the weight extraction time due to magnitude weight extraction is the smallest and this also leads to the least transmission cost reduction as shown in Table 4.16-Table 4.20. FitCNN+ [28] and FitCNN [10] always require more iterations as compared to random and magnitude weight extraction. Though FitCNN+ [28] and FitCNN [10] are able to extract fewer useful weights as compared to random and magnitude weight extraction, the number of iterations FitCNN++ takes is similar to random and magnitude weight extraction and FitCNN++ is also able to extract much fewer weights from the neural network.

4.10. Effect of Partition Size on The Weight Extraction Time

Different number of partitions may have different effects on the overall weight extraction time. in this experiment, different values of *P* are input into the weight extraction algorithm and the overall weight extraction time is observed.

Table 4.21. Weight extraction times with respect to different partitions of the weight difference distribution.

Architecture	**Dataset**	**Total Weight Extraction Time (s)**			
		P = 2	**P = 4**	**P = 8**	**P = 30**
ShuffleNetV2 [9]	ImageNet [1]	33.16	29.54	35.30	60.45
SqueezeNet [6]	CUB-200 [5]	34.57	19.91	34.57	74.01
MobileNetV2 [8]	Stanford Dogs [7]	30.17	26.37	33.74	96.06
MnasNet [4]	CIFAR-100 [3]	491.92	293.41	480.87	705.76

It can be observed that when *P* is set to 2 and when *P* is set to 8, there is not a very big difference in the weight extraction time. This is because when *P* is very small, the sub group of the weight difference distribution list is big, hence the time taken to find the correct group can be very fast

but since the sub group can be big, the time taken to find the optimum threshold may increase. When P is set to a slightly larger value, the sub group of the weight difference distribution list is small hence the time to find the right sub group can also increase. Setting P to a very large value is leading to the slowest weight extraction time in all cases. It should be noted that these results do not prove the setting $P = 4$ is a universal solution that will yield the fastest weight extraction time because though the aim of the weight extraction algorithm is to find the largest threshold that still keeps the model performance within tolerance, however, depending on the neural network and the dataset, it is possible for the optimum threshold value to be anywhere in the weight difference distribution list. The only fact this chapter tries to stress is not to use too many threshold values to find the optimum threshold value when it is possible to do the same using fewer thresholds via partitioning the weight difference distribution list.

4.11. Summary

The key issue this chapter highlights is the transmission cost that occurs when transmitting all of the trained neural network parameters back to the edge device. The reason this is an issue is due to the typically large number of parameters present in deep neural networks. This chapter proposes a solution that can select only the useful parameters in the trained neural network and only these useful weights are then transmitted back to the edge device in an effort to reduce the transmission cost between the edge device and the cloud. The proposed solutions are tested by testing them on different CNN architectures being trained on different datasets using randomly chosen hyperparameter values. Finding the optimum threshold value can be viewed as a search/optimization problem and hence this problem can be open to various search methods. However, as seen in all figures in section 4.9 that as a larger threshold is chosen in an effort to obtain lesser important weights, the amount of accuracy change can vary. This is due to the stochasticity of a neural network, hence, various threshold values have to be tested one after the other to reduce the number of useful weights without largely affecting the model performance.

As mentioned in chapter 3, out of all the experiments carried out, a statistical significance test could also have been performed in this chapter as well where the test would show that there is no statistical difference between the accuracy distributions with and without weight extraction but there is a statistical difference between the distribution of the number of useful weights extracted as this would show that the model performance is maintained while largely reducing the number of useful weights transmitted to the edge device. However, this test was not conducted because the number of data points are quite small to carry out such tests as the number of incremental training rounds are not many. To perform a statistical significance test, a lot more data points are needed (minimum of 30) because if the sample size is too small then disproportionate number of individuals may end up being included which are outliers and anomalies which may skew the results.

Overall, it can be seen that the time taken to find the useful parameters depends on the size of the training dataset size and the complexity of the classifier both in terms of the number of layers and the number of neurons in each layer. Two variants of the FitCNN [10] algorithm are proposed in this chapter. The key idea in the original version of FitCNN [10] is that a weight difference is calculated between the trained and the pre-trained parameters of a neural network. This weight difference distribution is divided into a certain number of quantiles and the weight difference values at the position of every divider are chosen as the thresholds. These threshold values are then used to determine the usefulness of a parameter of a neural network. FitCNN

[10] takes the highest amount of time to find the useful weights as compared to FitCNN+ [28] and FitCNN++ because the original version uses 30 thresholds that contain both positive and negative values. These 30 quantiles are divided uniformly across the weight difference distribution list which can contain both positive and negative values. FitCNN+ [28] is proposed in this chapter that uses only positive thresholds which leads to a faster time to find the useful weights. FitCNN++ is also proposed in this chapter which is an improvement to FitCNN+ [28], and the reason why FitCNN++ is the fastest out of all the 3 versions is that this approach does not perform weight extraction using all of the positive thresholds but only a subset of the positive thresholds.

It can also be concluded that the number of useful weights extracted is inversely proportional to the time taken to do so. Magnitude weight extraction leads to the fastest weight extraction time because this method is able to eliminate only a few weights of the classifier. This also shows that it is not true that the weights with the least magnitude are not useful because if that was the case then the weight extraction rate will be higher, however that is not the case. This behaviour also seems to suggest why the random weight extraction process is able to eliminate more weights, this tells that weights with large magnitudes can also prove to be not useful hence, can be eliminated from not being transmitted back to the edge device.

A limitation of this algorithm is that a few forward passes are needed to compute the training accuracy for determining which parameters are useful. This can be computationally expensive especially in the case of incremental learning whereby the size of the neural network grows as more tasks are learnt. This computational cost is also dependent on the size of the training set that is used to calculate the training accuracy.

With the contribution of this chapter, this research achieves the objectives of reducing the data transmission cost between the edge device and the cloud in terms of reducing the number of features of images being transmitted to the cloud and the number of trained parameters of a neural network being transmitted from the cloud back to the edge device. So far only one edge device has been used, however, the next chapter mentions how multiple edge devices can be used to carry out class incremental learning and the challenges associated with that.

Chapter 5

Effects of Distributed Learning for Multi-Incremental-Task Learning on Energy Consumption

5.1. Background

In this chapter, a framework is provided that highlights how multi-task learning can be performed when there are multiple edge devices collecting data and are connected to the cloud. The constraint placed is that data on an edge device will not be shared with other devices or the cloud. Hence, the challenge is to carry out multi-task learning in the presence of heterogeneous edge devices without ever sharing the data stored on edge devices. Out of the different types of incremental learning methods discussed in Chapter 2, this chapter shows that using the concept of dynamic neural networks and preserving the weights associated with old tasks distributed multi-task learning can be carried out without sharing any old training data.

For rehearsal-based class incremental learning [35], [132], both the new and old samples must be present for neural network training to take place. Regularization-based multi-task learning focuses on constraining the weights of the learned classes in order to avoid catastrophic forgetting i.e., the weights corresponding to the old tasks are prevented from changing by a large margin when learning new classes [40], [207]. By doing so, there is no need to store samples from the old classes. This is a favorable condition for edge devices because re-training with old classes is not required when learning new classes on new edge devices. This form of learning relies on the multi-head approach as there are multiple classification heads (classification network) in the overall neural network that are responsible for classifying dedicated tasks. Progressive Neural Networks [37] completely eliminate catastrophic forgetting by instantiating new classification heads as new tasks are learnt, while doing so, the parameters of the previous classification heads are frozen. Dynamically Expandable Networks [207] also incrementally add new classification heads as new tasks are learnt while selectively re-training the old networks in a regularized manner and adding new classification heads when necessary.

To carry out rehearsal-based class incremental learning in the case of multiple edge devices, the training data from all of the edge devices must be transmitted to the cloud whereby the cloud carries out rehearsal. However, this involves data sharing and thus cannot be used in cases where local data of edge devices cannot be shared. If the data cannot be shared, the neural network must learn only from the current data samples because edge devices do not have enough resources to continuously store data from all of the observed classes. Therefore, the only possible way to carry out multi-task learning scenarios in such cases is to do it via expanding neural network dynamically and preserving parameters of old tasks [208].

Once this objective is realized, a key issue that arises is that if training is being carried out on one of the edge devices, then the remaining edge devices must wait to receive the neural network. In the case when heterogeneous devices are participating, the combined overall energy consumption across all edge devices may vary largely depending on the order in which the edge devices participate to contribute towards distributed multi-task learning. This is explained in more detail in the following paragraph.

The main objective of this chapter is to study the energy consumption of edge devices with respect to the order in which the training is carried out. In dynamic multi-task learning, training must take place only on one device after which the trained neural network is sent to the next edge device, where the parameters of the neural network are regularized (preserved) on this new edge device when new classes are learnt. In multi-task learning, it is guaranteed that the SoftMax layer of a neural network will grow, therefore, the computational cost of the neural network and the memory footprint consumption must be considered in the context of the edge devices. Depending on factors such as the hyperparameters, the number of classes being learned at a time, the number of samples in one incremental training round, dimensionality of the data, available RAM on the edge devices, the type of the processors (CPU, GPU, TPU), the clock frequency of the device, edge device power consumption, edge device temperature, the overall energy consumption across the edge devices may vary, for e.g., if there are two edge devices: edge device A and edge device B, then carrying out training on edge device A followed by edge device B can lead to a smaller overall energy consumption as compared to starting training on edge device B followed by edge device A depending on the neural network and hardware properties mentioned above.

The main problem this chapter tries to look at is how to select the order of edge devices in which the training needs to be performed i.e., which edge device should start and end the training. Hence, the objectives of this chapter are listed as follows:

- Provide information on how to carry out multi-task learning across various edge devices without sharing the raw data on any of the edge devices.
- Study the effect of edge device training order selection on the end-to-end energy consumption across all edge devices.

5.2. Distributed Multi-Task Learning System

This section describes the framework of how class incremental learning can be performed across multiple edge devices without ever having edge devices share the training data on them with one another. The terms class incremental learning and multi-task incremental learning are used interchangeably. Firstly, it is important to understand how dynamic class incremental learning takes place. Neural networks with dynamic architectures grow in size as new tasks are learned [35]. However, as the neural network grows in size, the neural network's knowledge on the old tasks must also be retained and this can be achieved via regularization [207] where the idea is to prevent the weights associated with the old tasks from changing by a large value when learning new tasks. This chapter focuses on multi-task learning in which new classifiers with new SoftMax layers are used to classify new tasks. Each task denotes a collection of multiple classes being represented together, also each of these SoftMax functions can be learned incrementally on separated/different edge devices. Once the neural network has been trained on any edge device, its parameters are frozen and never trained again when learning new tasks on new edge devices. This way, the neural network's knowledge of the old tasks can be retained. This methodology is inspired by [37]. Figure 5.1 shows the basic methodology of how the neural network evolves using such a method.

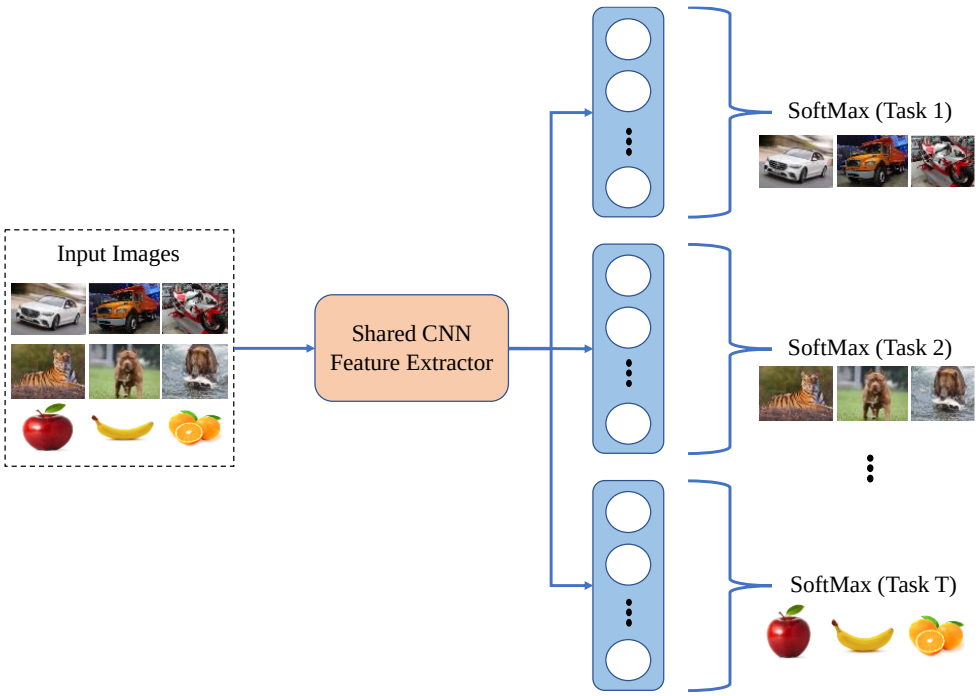


Figure 5.1. Basic high-level diagram of multi-task learning.

In Figure 5.1, the shared CNN feature extractor refers to a pre-trained CNN feature extractor. It can also be seen that each classifier is responsible for classifying a few classes which form one given task. Each classifier is responsible only to classify the samples it has been trained on. Therefore, in the context of distributed edge computing, each edge device is responsible for learning a task. Figure 5.2 visually shows how multi-task learning can be performed across various edge devices [37].

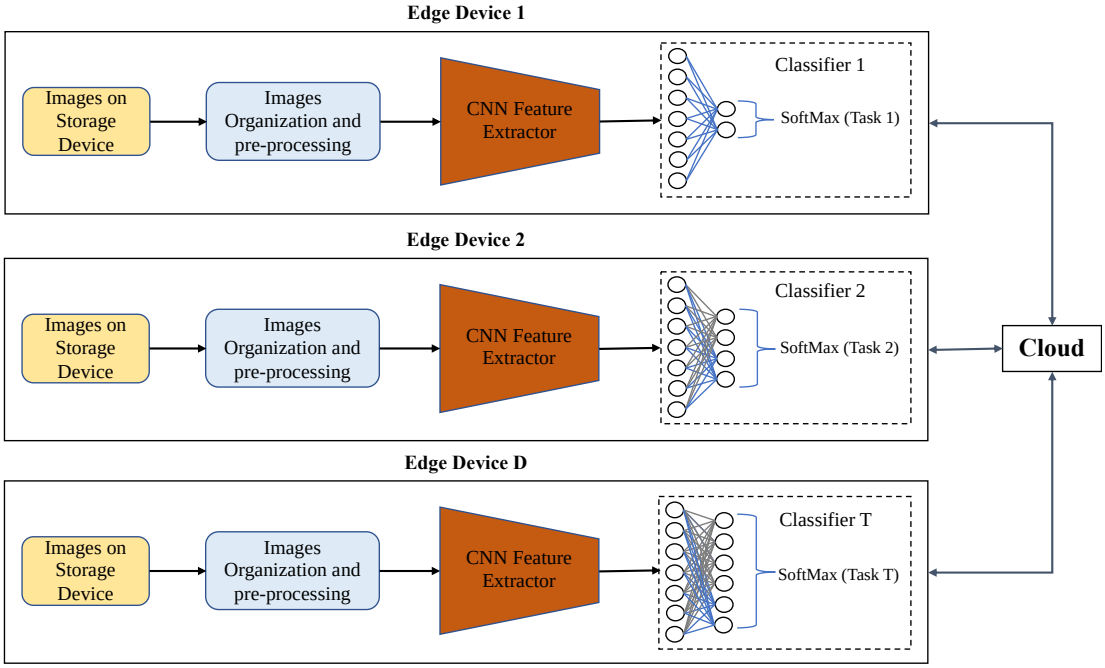


Figure 5.2. Overview of multi-incremental task learning taking places across multiple edge devices.

Each edge device has a storage device where all of the images to be trained are stored, for e.g., the storage device can be a hard drive or an external drive (if the edge device does not have enough space to store the images). Once the images are read from the storage device, these images are then pre-processed and organized into appropriate data structures. After the image pre-processing and organization process is completed, the images are forward propagated through the pre-trained feature extractor to obtain feature maps of the images which are then used to train the classifier. The CNN feature extractor on each edge device is the same in terms of the architecture and the parameter values. The classifier training needs to be carried out on one of the edge devices before transmitting the trained neural network to the other edge device for training. In Figure 5.2, the training starts on edge device 1 followed by edge device 2 and all the way up to edge device D. On edge device 2 and edge device 3, the grey colored weights of the fully connected layers denote the frozen weights, for e.g., once the classifier training takes place on edge device 1, the classifier is then transmitted to the cloud after which the cloud transmits the classifier to the next edge device which is edge device 2 in Figure 5.2. It can be noted that the weights connected to the first two neurons of the SoftMax layer are frozen (grey colored wrights) and only the remaining weights are trained to learn the patterns from the new images stored only on edge device 2. The function of the frozen weights is to preserve the knowledge of the old classes. The same can be seen for the classifier on edge device D whereby only the weights connected to the last two neurons are trained while the remaining weights remain frozen.

The key in this learning setting is that previously observed data is not needed for training, and the neural network being trained on an edge device is only trained with the data stored on the edge device itself while keeping the weights belonging to the old classes unchanged. Therefore, once a neural network is expanded and trained incrementally on an edge device. The parameters of this neural network are transmitted to the cloud via TCP/IP and the cloud then transmits these parameters to the next edge device for training. When these parameters arrive at the new edge device, these parameters are kept frozen while new weights are added to these parameters forming a larger neural network whereby the new weights are trained using the data present on the new edge device. This process repeats until all of the edge devices have participated in the neural network training step.

Using this framework, multi-task learning can be carried out across multiple edge devices using any weight regularization approach without having to share the local data on the edge device with either the cloud or other edge devices. The reason why a sequential training approach is explored in this chapter is because not all edge devices can be active at the same time and may have different schedulers to be active at different times, therefore, there must be a way to carry out incremental learning sequentially without sharing the data on edge devices with each other.

5.3. On Device Neural Network Training for Larger than Memory Training Samples

One of the biggest challenges of carrying out training on a resource-constrained edge device is the available RAM on the edge devices. In an image classification task, the problem arises if even one of the classes contains a certain number of images that are greater than the edge device can hold in its RAM. Generally, when neural networks are trained, at least one or more classes are trained together but if a device cannot hold all samples belonging to a class in its memory, then the whole distribution of the class cannot be captured before training begins.

One solution to tackle this problem is to read samples belonging to these classes from the storage device in the form of mini-batches. For neural network training in general, all samples from different classes are present in a computer system's RAM after which the samples are shuffled at every epoch and during an epoch, mini-batches of samples are processed from this shuffled dataset. Similarly, if the number of samples present in a class is too large for an edge device to handle, then a small mini-batch of samples can be taken from this class along with the mini-batches from other classes. These collected mini-batches can then be shuffled and forward and backward propagated through the neural network.

As compared to the standard deep neural network training, the only difference with this modified form of learning is that the edge devices need to continuously read data from the storage device with an assumption that a powerful server may accommodate all of these samples in its RAM.

$$F = [f_0, f_1, \dots f_{H-1}] \tag{5.1}$$

Equation 5.1 shows that a list is created that contains the location (file path) of all the images on the storage device that needs to be trained, and the number of these images is such that they cannot be stored together at one time on the edge device. f_0 denotes the 1st image file location on the storage device, f_{H-1} also denotes the H^{th} image file location on the storage device where H is the total number of images that must be trained on the edge device in a batch-wise manner. Once this list is obtained, MF_d samples are read from the edge device with respect to F in an ascending order after which MF_d samples are trained on the edge device. It should be noted that MF_d denotes the mini-batch size in this case. Once all files have been read from F and training has been carried out, the neural network is said to have been trained on one epoch. To continue training for multiple epochs, the file ordering in F is randomly shuffled in order to mimic the shuffling of the training images for standard deep neural network training. After F has been randomly shuffled at the beginning of every epoch, the same procedure of reading the file locations and training ensues.

However, if the edge device can store the features of the total number of samples to be trained, then all of the samples can undergo feature extraction, and all features can be stored in the memory, and the mini-batches from these features can be used one by one to carry out neural network training.

5.4. Edge Device Training Order Selection Algorithm

This section describes the details of the algorithm that can be used to choose the order in which the training can be carried out on the edge devices to minimize the end-to-end distributed energy consumption [209]. Mathematically, this is considered a permutation problem. Let D be the number of devices that are participating in the distributed multi-task learning process. Given D devices, the training can be carried out using $D!$ ways given that all of the edge devices are being considered for permutations. For e.g., consider four devices namely A, B, C, and D, Table 5.1 shows the permutation examples.

Table 5.1. Possible ways of training a neural network incrementally across various devices.

	Order of choosing the edge device			
Permutation Number	**1st**	**2nd**	**3rd**	**4th**
1	A	B	C	D
2	A	B	D	C
3	A	C	B	D
4	A	C	D	B
5	A	D	B	C
6	A	D	C	B
7	B	A	C	D
8	B	A	D	C
9	B	C	A	D
10	B	C	D	A
11	B	D	A	C
12	B	D	C	A
13	C	A	B	D
14	C	A	D	B
15	C	B	A	D
16	C	B	D	A
17	C	D	A	B
18	C	D	B	A
19	D	A	B	C
20	D	A	C	B
21	D	B	A	C
22	D	B	C	A
23	D	C	A	B
24	D	C	B	A

In the case of four devices, there are 24 ways of training the neural network incrementally. So, for the edge device training order selection algorithm, the aim is to find the permutation that leads to the fastest end-to-end incremental training time or the least end-to-end energy consumption. However, this chapter focuses on reducing energy consumption.

The modification to the classifier to enable class incremental learning in terms of the neural network size and the neural network hyperparameters are the main reason why an edge device training order selection algorithm needs to be in place. This is due to the heterogeneity and the diversity of the edge devices. This work assumes that the edge devices participating in the distributed training process can have different processors such as CPUs or GPUs. which is what defines the heterogeneity. The diversity comes from the fact that the number of devices that have CPUs as their processors can have varying hardware parameters such as the available RAM, clock frequency, etc. Due to the fact that the neural network architecture changes, different edge devices with different hardware parameters can lead to varying neural network training times. For e.g., a CPU may be able to train a single layer neural network with 10 neurons faster than a GPU under certain learning hyperparameters, however, for the same learning hyperparameters, training a single layer neural network with 100 neurons can be executed faster on a GPU rather than a CPU. Due to such hardware differences, different

devices will have different training times due to learning settings such as the image resolution, learning rate, epochs, and other learning settings.

Due to a large range of available edge devices and an infinite number of hyperparameter combinations, all of the participating edge devices need to carry out a form of analysis before starting the distributed training. Certainly, if the objective is to minimize the end-to-end training time or energy consumption, then for each of the permutations, the neural network training time must be computed on every edge device by varying the neural network architecture. Every possible architecture of the neural network must be trained on every edge device under the given hyperparameters. This way, the training time or energy consumption of all the possible neural network architectures on all the devices can be calculated after which the training order can be selected.

Carrying out neural network training to compute the training time of all possible combinations of the neural network architectures on each of the edge devices can be a very time-consuming process and is not the ideal solution to find out the best edge device order especially due to a large number of epochs or a low learning rate, etc. In order to speed up this process, the training time needs to be predicted using machine learning models on each edge device whereby the input should be the hyperparameters and the output should be the predicted training time for the given classifier. Figure 5.3 below shows the proposed edge device selection algorithm.

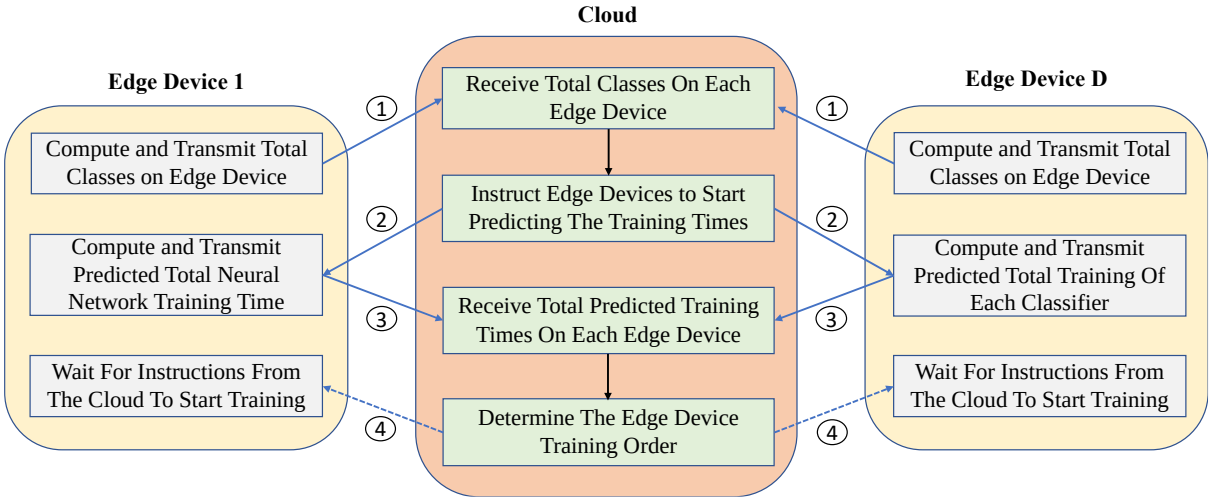


Figure 5.3. Major steps involved in the complete edge device order selection algorithm.

There are a total of 3 main steps involved in making the complete edge device order selection algorithm work. Firstly, each edge device will have a neural network to train, hence, each edge device must predict the overall neural network training time and transmit this information to the cloud. Depending on the power consumption of an edge device, the cloud can then predict the energy consumption on the edge device, this is explained in detail in section 5.4.1. Once the cloud receives the predicted training time on each edge device, the cloud then needs to determine which specific training order will result in the least energy consumption across all edge devices, along with the predicted training time, the power consumption on the edge devices is also utilized, this is explained in detail in section 5.4.2. Once the training order is determined, training needs to be carried out on an edge device followed by another, this requires the neural network to be transmitted from one edge device to the next, this is explained in detail in section 5.4.3.

Figure 5.3 shows the overview of the edge device training order selection algorithm. The numbers shown in Figure 5.3 represent the total communications between the edge device and the cloud done so far. Firstly, each device counts the number of classes it has and transmits this information to the cloud. Once the cloud has received the number of classes present on each device, it computes the possible combinations of the SoftMax units in the classifier that can be run on each edge device. This combination is computed with respect to each edge device, the SoftMax unit combination belonging to each edge device is then transmitted to the respective edge devices. Each edge device then instantiates a classifier whereby the number of SoftMax units in the classifier is equivalent to one of the combinations of the SoftMax units on the respective edge device. Next, for each of these instantiated classifiers, the overall training time is predicted with respect to the classifier architecture and the hyperparameters whereby the total number of predicted time values is equivalent to the number of instantiated classifiers. Each edge device then sends these predicted training time values to the cloud. Using the predicted training time values from all edge devices, the cloud then computes in which order should the multi-task increment learning be carried for e.g., which edge device should start the training process and on which device shall the training stop. The cloud then sends a signal to the edge device that is supposed to start the training and waits for the training to complete, once the training completes on the edge device, the parameters of the classifier are sent to the cloud from this edge device, and the cloud then sends this trained classifier to the next edge device. This process repeats and stops only when the last edge device has carried out the training. This entire process is discussed in greater detail below.

Each device has access to the total number of classes and samples in each class. This is the very first piece of information that gets the edge device selection algorithm started. The cloud has the role of coordinating actions between the participating edge devices and to enable this functionality, all of the edge devices transmit a message to the cloud. Firstly, each edge device transmits a message in the form: $E!$ whereby the character $'E'$ lets the cloud know that edge devices are initiating a connection with the cloud i.e., when the cloud detects the character $'E'$, it knows that a new edge device has connected. Once all of the edge devices have completed transmitting an initial message to the cloud, the cloud is aware of how many edge devices there are.

Once an edge device receives all the possible number of neurons that can be present in the classifier, the edge device calculates the maximum number of features of images it can store at a time in the memory. To do so, a mini-batch of tensors with the same dimensions as the training images are forward propagated through the feature extractor of a CNN. The time taken for this feature extraction process is recorded and these features are also saved in memory. This process continues repeating and as more features are extracted, they are concatenated. When the time taken to extract features of the first mini-batch is less than τ_a seconds as compared to processing subsequent mini-batches, it indicates that the RAM is starting to get consumed and there will be a slowdown in the further neural network training process if more features are stored. τ_a denotes the acceptable time difference. Once this event occurs, the total features saved so far are considered the maximum number of features that can be stored on the edge device's RAM. τ_a needs to be set manually, however, the main idea behind τ_a is to ensure that a high RAM consumption does not slow down the processing speed of an edge device.

5.4.1. Total Neural Network Prediction Time on Edge Devices

For the cloud to determine the training order which will result in the least overall energy consumption across all devices, it must have 2 main pieces of information: the processing power of an edge device and the time the edge device will take to train a neural network with respect to the hyperparameters. Here, the setup of a CNN is the same as what is mentioned in Chapter 3 and Chapter 4 i.e., the feature extractor of a CNN is frozen (no backpropagation takes place on the feature extractor of the CNN) and only the fully connected layers are trained. Therefore, to predict the overall training time of a CNN on an edge device, the total feature extraction time must be predicted and the total training time of only the fully connected layers must be predicted as well. To do so, random tensors are instantiated that have the same dimension as the training images. The hyperparameters and the neural network architecture is also the same as what are used during the actual distributed training. The random tensors are propagated through the neural network to predict the overall training time on the edge device first to predict the overall training time.

If the number of samples to be trained is greater than the number of samples that can be stored on an edge device, then the training must be carried out by reading batches of images from the storage device of the full dataset and repeating this process for the total number of epochs. Equation 5.2 shows how the overall CNN training time can be predicted when the number of training samples is greater than the maximum number of samples that can be stored on an edge device d.

$$\gamma_d \approx \left(ti_{r_s}^d * e\right) + \left(\left\lceil \frac{s_d}{b} \right\rceil * e * \left(\left(ti_{i_r}^d + ti_{i_e}^d + ti_{i_s}^d\right) + \left(ti_{f_p}^d\right)\right)\right) \tag{5.2}$$

For the case where the number of samples to be trained is greater than the number of features that can be held by the edge device in its memory, the time taken to read and transform the image is recorded, the time taken to extract the features is recorded, the time taken for a full pass of a mini-batch of features is recorded, and the time taken to shuffle F is recorded. In Equation 5.2, γ_d denotes the total predicted training time on edge device d, $ti_{r_s}^d$ denotes the time taken to randomly shuffle F, however, since this process has to be repeated for e epochs, $ti_{r_s}^d$ is multiple by e. s_d denotes the total samples to be trained on edge device d. The terms $ti_{i_r}^d$, $ti_{i_e}^d$, and $ti_{i_s}^d$ denote the time taken to read, the time taken to extract features, and the time taken to save one mini batch. The term $ti_{f_p}^d$ denotes the time taken to perform one full pass of the b features. A full pass here denotes both forward and backward propagation. Image reading, image extraction, and image saving all need to take place for all s_d samples and must be repeated for e epochs. However, these tasks only take place for b iterations after which the memory storing all the features so far must be cleared. $\lceil s_d/b \rceil$ denotes the number of full passes required when training b images at a time, furthermore, this repeats for e epochs which is why a multiplication by e is done. The same logic applies to $ti_{f_p}^d$ because the full pass only takes place once b features have been stored.

$$\alpha = \delta \cdot s_d^t \tag{5.3}$$

In Equation 5.3, s_d^t denotes the number of samples at an incremental training round t on edge device d and δ denotes a percentage level. δ decides what the value of α should be, hence, δ

enables lesser time needed to obtain the accumulated training time points which can then be fit into a regression model.

For the case where the number of samples to be trained is less than the number of features that can be held by the edge device in its memory, features from all images must be extracted and saved. For the time prediction, not all features can be extracted due to the possible large time consumption. Therefore, instead of s_d^t iterations for extracting all features, only α iterations are used to extract features of images using Equation 5.3. For each iteration, the time taken to read the images is recorded and accumulated, this list which has a length of α and contains the accumulated feature extraction time is then used to predict the total feature extraction time. However, as features are extracted and saved continuously, it is important to observe the behavior of the accumulated time list before applying any form of regression models. A simple experiment is carried out whereby 10 classes from the CIFAR-100 [3] dataset are trained, and before training, features of all images of these 10 classes are extracted and saved one by one along with the accumulated time.

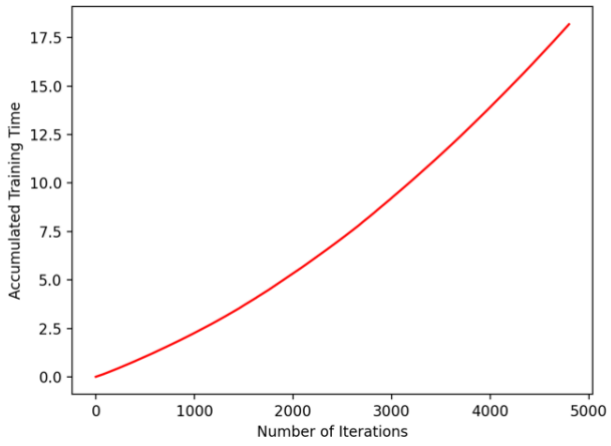


Figure 5.4. Accumulated training time when features of images are saved while training.

From Figure 5.4, it can be seen that as more features are stored, the cumulative time of feature extraction becomes polynomial in nature, therefore, the cumulative timing list that has a length of α must be fed into a polynomial regression algorithm and the input to the trained algorithm should be s_d^t for which the trained polynomial regression model would output the predicted total time that will be taken to extract all of the features of samples that are to be trained. The reason why the feature extraction time is polynomial in nature is that as more features are being saved, the larger the RAM consumption which in turn slows down the processing speed of an edge device. Equation 5.4 shows how the overall CNN training time can be predicted when incremental training is carried out in the case where the number of training samples is less than the maximum number of samples that can be stored on an edge device.

$$\gamma_d \approx \sum_t \left(\psi^{d,t}(\zeta^{d,t}, \rho)(s_d^t) + \left(e * \left(ti_{s_p}^{d,t} \right) \right) \right) \tag{5.4}$$

In Equation 5.4, at the incremental training round t, $ti_{s_p}^{d,t}$ denotes the time taken to train the classifier for only one epoch, $\psi^{d,t}(\zeta^{d,t}, \rho)$ denotes the trained polynomial regression model whereby $\zeta^{d,t}$ denotes the accumulated feature extraction time list and ρ denotes the degree

order of the polynomial regression model, s_d^t is the number of samples being trained on edge device d at incremental training round t. s_d^t is then input to the trained polynomial regression model ($\psi^{d,t}(\zeta^{d,t}, \rho)$) which outputs the predicted total feature extraction time at incremental training round t on edge device d. Following this, for predicting the neural network training time, all of the features have already been stored in the edge device memory. In this case, the time taken to perform a full pass for one single epoch is computed which is then multiplied by e i.e., the actual number of epochs for which the neural network will be trained later.

$$A = \begin{pmatrix} a_0 \\ \vdots \\ a_{D-1} \end{pmatrix} \tag{5.5}$$

Each device transmits the predicted training time to the cloud after which the cloud determines in which order should the distributed training be carried out based on the predicted training times obtained on the edge devices. Each row of vector A in Equation 5.5 corresponds to predicted training times on an edge device d. Vector A shown in Equation 5.5 is used to determine the edge device training order. Vector A is formed on the cloud by appending the predicted training times in the form of rows i.e., each edge device transmits the predicted neural network training time, and the cloud appends in this data into a vector. The edge device training order selection algorithm is performed on the cloud which is mentioned in the next section.

5.4.2. Algorithm for selecting Edge device training order to minimize energy consumption

If the objective is to minimize the total end-to-end energy consumption, then the algorithm for achieving that is different from the algorithm for minimizing the end-to-end training time. This is because when heterogenous devices take part in distributed learning, some of the devices can be a lot more powerful than the other edge devices. Powerful devices can have a fast-processing time but at the same time can also lead to high power consumption. For e.g., if training starts on computationally weaker devices, training needs to be completed on those devices first before the trained neural networks are transferred to the other edge devices, however, till the trained neural networks do not arrive on the powerful edge devices, these edge devices need to continuously stay active and keep listening to the cloud which adds to the overall power consumption. Algorithm 5.1 shows how the overall end-to-end energy consumption minimization can be achieved.

Algorithm 5.1: Algorithm for end-to-end energy consumption reduction

Input : $A, device_{id}, P_{idle}, P_{train}$
Output : Z
$P \leftarrow permutations(device_{id})$
$W \leftarrow []$
for x in P **do**
  $rows \leftarrow x$
  $temp_{list} \leftarrow []$
  $row_waiting_time_list \leftarrow []$
  $row_wait_time \leftarrow 0$
  for col in $rows$ **do**
    if $col == 0$ **then**
      $[row_waiting_time_list] \leftarrow t_{wait}^d$

```
11 :         else
12 :             row_wait_time ← row_wait_time + A[rows[col − 1]]
13 :             [row_waiting_time_list] ← row_wait_time
14 :     [W] ← row_waiting_time_list
15 : E ← []
16 : for w_col in W do
17 :     row_waiting_time ← W[w_col]
18 :     total_energy ← 0
19 :     for p_col in P do
20 :         p_idle ← P_idle[P[p_col]] * row_waiting_time[p_col]
21 :         p_train ← P_train[P[p_col]] * A[p_col]
22 :         p_tot ← p_idle + p_train
23 :         t_tot ← row_waiting_time[p_col] + A[P[p_col]]
24 :         total_energy ← total_energy + (p_tot * t_tot)
25 :     [E] ← total_energy
26 : min_energy ← E[0]
27 : Z ← P[0]
28 : for i in E do
29 :     if E[i] < min_energy then
30 :         min_energy ← E[i]
31 :         Z ← P[i]
```

After the edge device selection algorithm takes place, all edge devices need to perform just two tasks: waiting to receive a message from the cloud and to carry out the training, hence, the energy consumption needs to be predicted only for these two tasks. The total amount of time an edge device has to wait to receive the trained neural network is approximately equal to the sum of the total training times on all of the edge devices that have carried out training thus far. This is because an edge device does not receive the trained neural network until the previous edge device has completed the training. Due to this fact, vector A can again be used to determine the edge device training order that can minimize the overall energy consumption. Equation 5.6 shows the formula for computing energy consumption.

$$Energy = Power * Time \tag{5.6}$$

Vector A already denotes the predicted training times on edge devices, however, the power consumption of an edge device must also be known to predict the energy consumption on the edge device. This is why P_{idle} and P_{train} also act as inputs to the algorithm whereby P_{idle} denotes the power consumption of edge devices while waiting to receive a message from the cloud and P_{train} denotes the power consumption of edge devices during neural network training. Using these inputs, the energy consumption of edge devices can be predicted during neural network training and while waiting to receive a message from the cloud.

In line 1 of Algorithm 5.1, the term P denotes all possible edge device training arrangement orders i.e., deciding which edge device will start the training and which device will be the last to perform the training. Out of all of the possible training order combinations in P, one of the combinations leads to the smallest end-to-end energy consumption. The edge device that first starts training will have the smallest waiting time because that edge device does not need to wait for any of the edge devices to complete the training. However, for other edge devices that are not the ones to start the training must wait for the other edge devices to finish training.

The energy consumption during neural network training on edge devices can be predicted using P_{train} and A. However, to compute the energy of edge devices while they wait to receive messages from the cloud needs some more steps. The matrix W in line 14 denotes the waiting time on each edge device. Each row of matrix W denotes one of the possible ways in which training can be carried out across multiple edge devices. In line 10, t_{wait}^{d} denotes the time that edge device d needs to wait to receive a message from the cloud. Line 10 is only executed for the first edge device to carry out the training. If an edge device is not the first device to carry out the training, then as mentioned, the waiting time on an edge device is approximately equal to the sum of the training times on the other devices that have already carried out the training which can be seen in line 12.

Once W has been obtained then for each of the possible training orders, the energy consumption is computed for each edge device. As shown in line 20-21, for each edge device, the waiting time and the training time is multiplied by the power consumption during listening to the cloud and training, respectively. The total predicted power is then accumulated along with the total processing time on the edge devices and the predicted energy consumption is then computed as seen in lines 22-24. After the total energy computed across all of the edge devices has been computed then this value is entered into a list denoted by E in line 25.

The row in vector E that contains the smallest energy consumption is the row that denotes the training order which will lead to the smallest end-to-end energy consumption, this is shown in lines 27-31 and is denoted by the term Z.

5.4.3. Communication Between Edge Devices

Once training has taken place on an edge device, the weights of the trained model must be transmitted to the next edge device to continue the training. This process repeats until the parameters of the model have been transmitted to the final edge device. Each edge device must convert the parameters into a string format so they can be transmitted to the cloud via TCP/IP.

$$D_{msg}^{d} = n_{o}^{l,d}, n_{i}^{l,d}, n_{o}^{l+1,d}, n_{i}^{l+1,d}, \dots n_{o}^{L,d}, n_{i}^{L,d} \tag{5.7}$$

In Equation 5.7, the dimensions of the weights of the neural network are converted into a string. For each device d, $n_{o}^{l,d}$ and $n_{i}^{l,d}$ denote the number of output and input neurons associated with layer l of the neural network. L denotes the total number of layers in the neural network.

$$W_{msg}^{d} = w_{0}^{l,d}, w_{1}^{l,d}, \dots, w_{J}^{l,d}, l, w_{0}^{l+1,d}, w_{1}^{l+1,d}, \dots, w_{J}^{l+1,d}, l, w_{0}^{L,d}, w_{1}^{L,d}, \dots, w_{J}^{L,d} \tag{5.8}$$

In Equation 5.8, the weights of the neural network are converted into a string. $w_{0}^{l,d}$ represents the weight value at index 0 of the flattened weight tensor at layer l. J is the length of this flattened tensor. The character $'l'$ denotes a separation between the weight values between different layers.

$$B_{msg}^{d} = b_{0}^{l,d}, b_{1}^{l,d}, \dots, b_{J}^{l,d}, k, b_{0}^{l+1,d}, b_{1}^{l+1,d}, \dots, b_{J}^{l+1,d}, k, b_{0}^{L,d}, b_{1}^{L,d}, \dots, b_{J}^{L,d} \tag{5.9}$$

In Equation 5.9, the biases of the neural network are converted into a string. b_{0}^{l} represents the weight value at index 0 of the flattened weight tensor at layer l. J is the length of this flattened

tensor. The character k denotes a separation between the weight values between different layers.

$$Final_{msg}^{d} = < D_{msg}^{d} >, s, < W_{msg}^{d} >, *, < B_{msg}^{d} > ! \quad (5.10)$$

In Equation 5.10, the overall string is formed containing the details of the classifier. The character $'s'$ lets the cloud know that the entire message before $'s'$ contains the dimensions of the neural network at every layer. The characters $'s'$ and $'*'$ let the cloud know that the message between these characters denote the weight tensors at every layer of the neural network.

The edge device that first transmits data to the cloud is assigned an ID of 0 and the last edge device to transmit data to the cloud is assigned an ID of $D - 1$. Due to the fact that the parameters of a neural network are being transmitted to the cloud, the dimensions of the matrices must also be transmitted, this dimension helps in letting the next edge device know exactly how the parameters need to be organized in a tensor before neural network training continues on the new edge device.

5.5. Experiment Setup for Performing Distributed Multi-Task Learning

Due to the lack of availability of edge devices, a total of two devices are used as edge devices in all of the experiments described below. A MacBook Pro and a Jetson Nano are used as the edge devices which enables a heterogeneous edge device system since the MacBook Pro contains a CPU as its main processor and the Jetson Nano contains both the CPU and GPU as the processors. A Linux-based virtual machine is setup on the Google Cloud Platform as the cloud for all of the experiments. The overall purpose of the experiments is to observe the end-to-end energy consumption across the distributed edge devices with respect to edge device training order, datasets, and hyperparameters.

For all of the experiments mentioned below, the datasets to be trained are stored on the hard disks of both edge devices. Before the training procedure starts on either device, all the applications, folders, and documents are closed on all edge devices. As the SoftMax layer grows in size, only the weights associated with the newly added neurons in the SoftMax layer are trained.

A total of two datasets are used for all of the experiments: CIFAR-100 [3] and Stanford Dogs [7]. In each of these datasets, 80% of the samples in a class as used as the training set, and the remaining 20% are used as the testing set. As for data augmentation, the input images are flipped horizontally and normalized. All images are resized to 32 x 32 x 3. The pre-trained SqueezeNet [6] architecture is used to extract features from the input images. A classifier is trained from scratch that consists of a single SoftMax layer with a dropout [150] probability of 0.25. The binary cross-entropy loss function is used together with the Adam [206] optimizer for stochastic gradient descent. The learning rate, number of epochs, and batch size are set differently for the different sets of experiments and are mentioned below. The polynomial degree ρ is set to 2 and δ is set to 30%.

For each experiment, the number of samples to be trained are partitioned in different ways i.e., both the edge devices store different numbers of images on them. The purpose of this partition

is to study the end-to-end training time and the energy consumption when the order in which the training is carried out varies with respect to the size of the dataset on each edge device. The details of the dataset partition are explained as experiments are described.

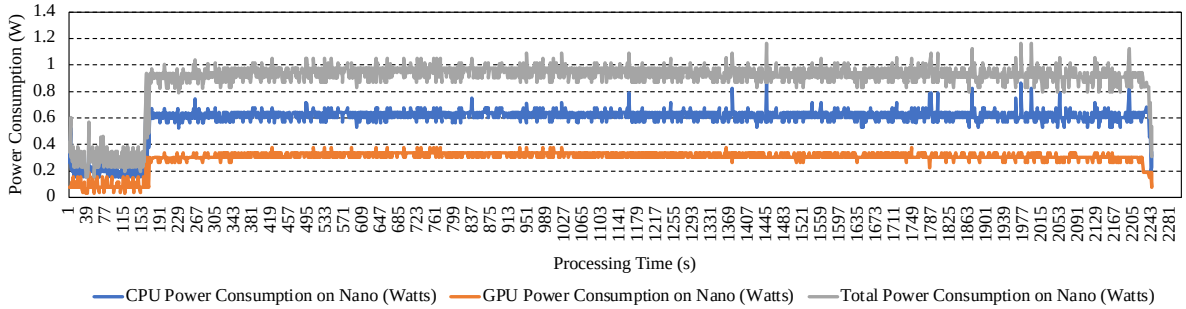
To compute the energy consumed due to the training on edge devices, two parameters are needed: power consumption and time taken for the training process to complete. The amount of time taken for an edge device to carry out training can be computed using certain software libraries. The process of measuring the power consumption is different on the Jetson Nano and the MacBook Pro. On the Jetson Nano, the *tegrastats* utility is used to obtain the CPU and GPU power consumption, and on the MacBook Pro, the *powermetrics* utility is used to obtain the CPU power consumption. The outputs of these commands are stored every second. The *powermetrics* utility outputs the power consumption in watts (W) and the *tegrastats* outputs the power consumption in milliwatts, therefore, the power consumption readings from the MacBook Pro are converted to watts.

5.6. Evaluation on a Dataset with a High Number of Classes

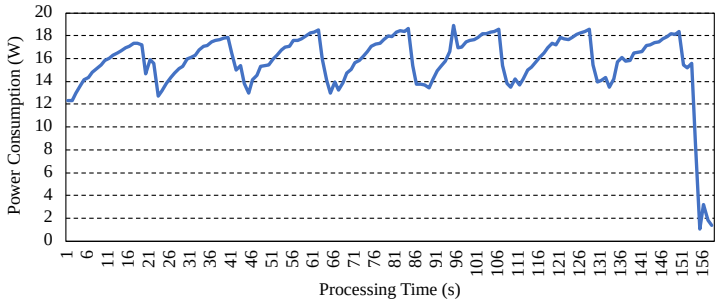
In these set of experiments, only the CIFAR-100 [3] dataset is partitioned among the two edge devices. This dataset is chosen randomly to observe the end-to-end energy consumption across all participating edge devices. The learning rate, number of epochs, and batch size are set to 0.0001, 5, and 128 respectively for the neural network training across both edge devices.

5.6.1. 70-30 Split

In this experiment, the CIFAR-100 [3] dataset is split in a way such that 70 classes are placed on the MacBook Pro and the other 30 classes are placed on the Jetson Nano. For this dataset partition, two main sets of experiments are carried out. The 70 classes present on the MacBook Pro are trained first after which the parameters of the trained neural network are sent to the Jetson Nano via the cloud. After the Jetson Nano, receives the parameters of the trained neural network, the remaining 30 classes are trained on the Jetson Nano. In the second experiment, the training first takes place on the Jetson Nano followed by the MacBook Pro.



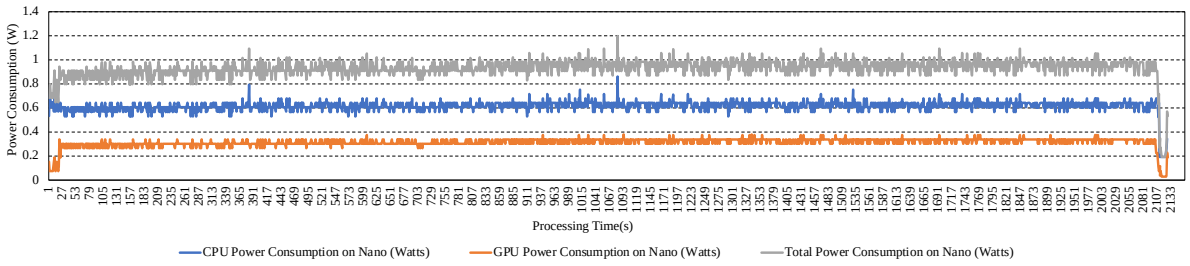
a) Power consumption and processing time on the Jetson Nano.



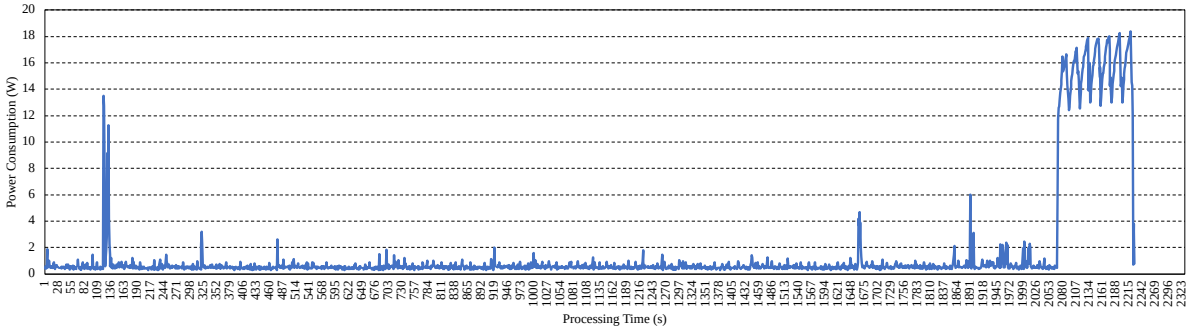
b) Power consumption and processing time on the MacBook Pro.

Figure 5.5. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 70 classes of CIFAR-100 [3] are first learnt on the MacBook Pro and the remaining 30 classes of CIFAR-100 [3] are then learnt on the Jetson Nano.

In Figure 5.5, the MacBook Pro is the first edge device to start the training, therefore, the time it needs to wait to receive a message from the cloud is extremely small after which the power consumption increases as the training starts on the MacBook Pro. As soon as the training is done on the MacBook Pro, the power consumption on it immediately reduces and the trained neural network is then sent to the Jetson Nano via the cloud. The Jetson Nano must wait for the training on the MacBook Pro to be finished which is why initially, the power consumption on the Jetson Nano is very small as it just needs to listen to the cloud to receive the trained neural network as it can be seen in Figure 5.5a). As soon as the MacBook Pro completes the training and sends the trained neural network to the Jetson Nano, the power consumption on the Jetson Nano increases and roughly stays at that level till the training is not complete.



a) Power consumption and processing time on the Jetson Nano.



b) Power consumption and processing time on the MacBook Pro.

Figure 5.6. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 30 classes of CIFAR-100 [3] are first learnt on the Jetson Nano and the remaining 70 classes of CIFAR-100 [3] are then learnt on the MacBook Pro.

In Figure 5.6, the Jetson Nano is the first edge device to start the training, therefore, it can be seen in Figure 5.5a) that the initial power consumption on the Jetson Nano is low for a very short time after which the power consumption increases as the training starts. The MacBook Pro now has to wait for the Jetson Nano to finish training, hence the power consumption on the MacBook Pro remains small while it listens to the cloud to receive the trained neural network as it can be seen in Figure 5.6b). As soon as the Jetson Nano completes the training and sends the trained neural network to the MacBook Pro, the power consumption on the Jetson Nano drops and the power consumption on the MacBook Pro increases as the training is being carried out.

Table 5.2. Power, time, and energy consumption across edge devices during distributed multi-task learning. The MacBook Pro having 70 classes and the Jetson Nano having the remaining 30 classes of CIFAR-100 [3] dataset.

Training Order	**MacBook Pro**	**Jetson Nano**	
	Power Consumption (Watts)		**Total Power Consumption (Watts)**
MacBook Pro then Jetson Nano	2482.03	1989.85	4471.88
Jetson Nano then MacBook Pro	3622.07	1977.65	5599.72
	Execution time (s)		**Total Execution Time (s)**
MacBook Pro then Jetson Nano	155.53	2246.00	2401.53
Jetson Nano then MacBook Pro	2266.88	2106.89	4373.77
	Energy Consumption (kWh)		**Total Energy Consumption (kWh)**
MacBook Pro then Jetson Nano	0.11	1.24	1.35
Jetson Nano then MacBook Pro	2.28	1.16	3.44

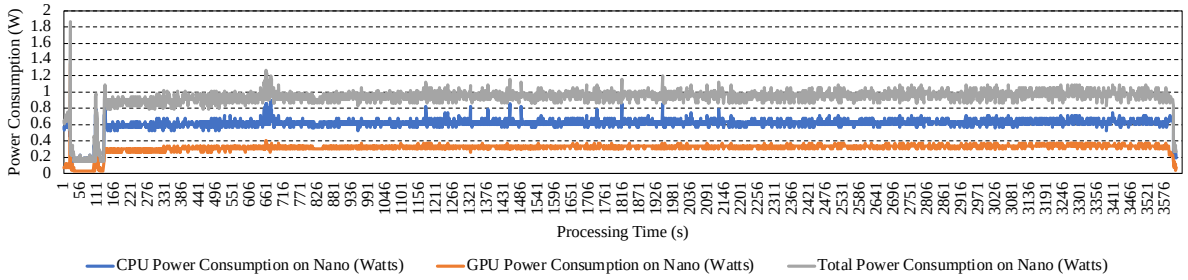
Table 5.2 shows the power, time, and energy consumption on each edge device with different training orders. When training 70 classes on the MacBook Pro followed by training 30 classes on the Jetson Nano, the total power consumption of both the devices combined is 4471.88W. However, if the training is first carried out on the Jetson Nano and then followed by the MacBook Pro, then the total power consumption of both the devices combined is 5599.72W which is a 13.39% increase in the overall power consumption.

The explanation for such a power consumption difference lies in the total processing time on each edge device. When the training first takes place on the MacBook Pro followed by the Jetson Nano, then the processing time on the MacBook Pro and the Jetson Nano is 155.53s and 2246.00s which amounts to a total execution time across both devices of 2401.53s. However, if the training order is reversed, then the processing time on the MacBook Pro and the Jetson Nano is 2266.88s and 2106.89s whereby the total execution time of both devices combined is 4373.77s. When the training starts on the Jetson Nano, the MacBook Pro must wait for the training to complete on the Jetson Nano to receive the trained classifier only after which the training can resume on the MacBook Pro. Due to the fact that the Jetson Nano is a much less powerful device as compared to the MacBook Pro, the time taken to complete the training takes a long time on the Jetson Nano and therefore, the MacBook Pro needs to keep waiting and listening to the cloud for incoming neural network parameters. Due to waiting, the power consumption on the MacBook Pro keeps accumulating, and hence, the overall power consumption is much higher when the training is first carried out on the Jetson Nano followed

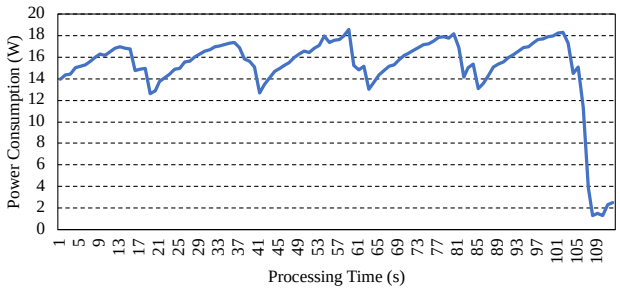
by the MacBook Pro. Results show that the energy consumption increases by 2.55x if the training is first carried out on the Jetson Nano followed by the MacBook Pro as compared to the other way around.

5.6.2. 50-50 Split

In this experiment, 50% of the CIFAR-100 [3] dataset is present on the MacBook Pro and the remaining 50% of the dataset is placed on the Jetson Nano.



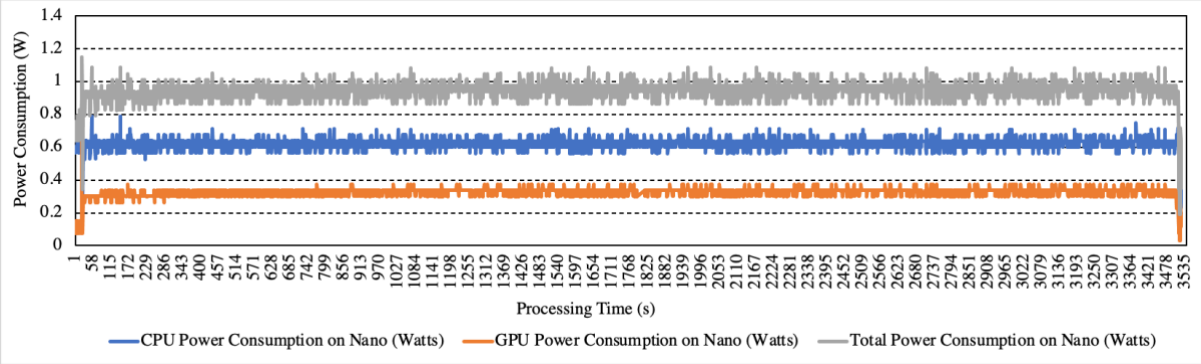
a) Power consumption and processing time on the Jetson Nano.



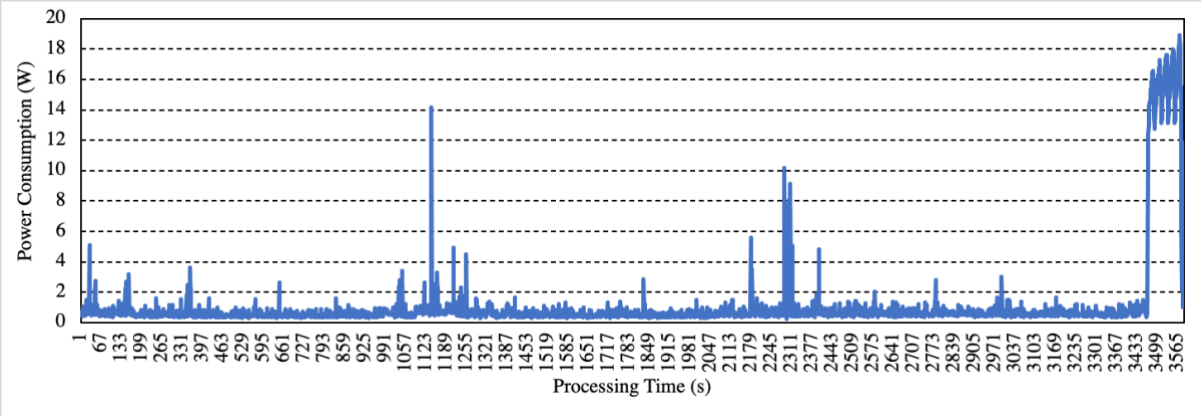
b) Power consumption and processing time on the MacBook Pro.

Figure 5.7. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 50 classes of CIFAR-100 [3] are first learnt on the MacBook Pro and the remaining 50 classes of CIFAR-100 [3] are then learnt on the Jetson Nano.

The results presented in Figure 5.7 exhibit the same pattern as what is shown in section 5.6.1 in terms of the power consumption on the MacBook Pro and the Jetson Nano. The only difference in Figure 5.7 as compared to Figure 5.6 is that the training time on the Jetson Nano has increased due to a larger number of classes to be learnt. Along with this, the training time on the MacBook Pro has also reduced because the MacBook Pro has fewer classes to learn from.



a) Power consumption and processing time on the Jetson Nano.



b) Power consumption and processing time on the MacBook Pro.

Figure 5.8. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 50 classes of CIFAR-100 [3] are first learnt on the Jetson Nano and the remaining 50 classes of CIFAR-100 [3] are then learnt on the MacBook Pro.

Table 5.3. Power, time, and energy consumption across edge devices during distributed multi-task learning. The MacBook Pro having 50 classes and the Jetson Nano having the remaining 50 classes of CIFAR-100 [3] dataset.

Training Order	**MacBook Pro**	**Jetson Nano**	
	Power Consumption (Watts)		**Total Power Consumption (Watts)**
MacBook Pro then Jetson Nano	1693.94	3358.23	5052.17
Jetson Nano then MacBook Pro	3979.74	3340.27	7320.01
	Execution time (s)		**Total Execution Time (s)**
MacBook Pro then Jetson Nano	111.43	3615.44	3726.87
Jetson Nano then MacBook Pro	3657.1	3537.99	7195.09
	Energy Consumption (kWh)		**Total Energy Consumption (kWh)**
MacBook Pro then Jetson Nano	0.05	3.37	3.42
Jetson Nano then MacBook Pro	4.04	3.28	7.32

The pattern of the power consumption and the processing time as seen in Figure 5.8 is similar to the ones observed in Figure 5.6. The only difference in Figure 5.8 is that the processing time on both edge devices have increased because the training first starts on the Jetson Nano which first has to train 50 classes of CIFAR-100 [3] first only after which the training on the MacBook Pro can be carried out.

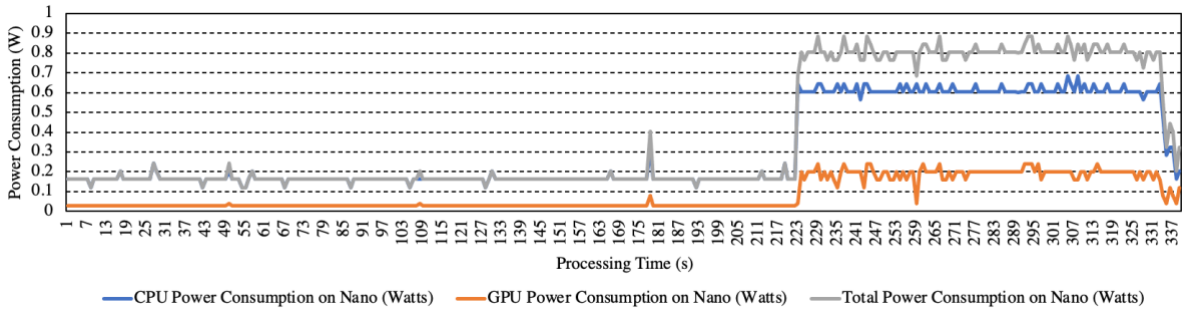
Table 5.3 shows the power, time, and energy consumption on each edge device with different training orders. In this experiment, each edge device trains an equal number of samples. As compared to section 5.6.1, the only difference in the results obtained for this experiment is that the overall power consumed due to the training process on each device has increased. This is mainly because the Jetson Nano has more samples to train, therefore, the power consumption of the Jetson Nano increases, and also the waiting time on the MacBook Pro increases if the Jetson Nano is the first device to carry out the training. It can be observed from Table 5.3 that the overall energy consumption increases by 2.14x if the training is first carried out on the Jetson Nano followed by the MacBook Pro as compared to the other way around. These results show that if a participating edge device is a lot less powerful than another participating edge device then the overall energy consumption increases if the less powerful edge device is the one to carry out the training first because the powerful edge device that is not the first edge device to carry out the training must wait to receive the neural network which eventually leads to a higher power consumption.

5.7. Evaluation on a Hybrid Dataset

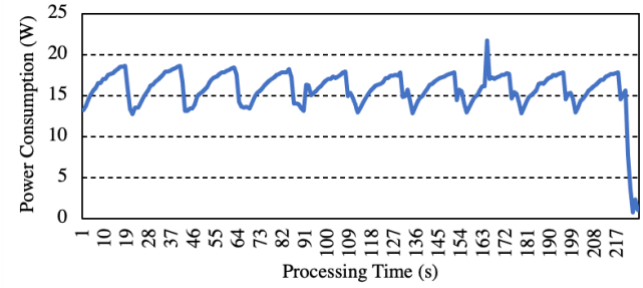
In these sets of experiments, a certain number of classes from the Stanford Dogs [7] dataset are placed on the Jetson Nano and all of the 100 classes of the CIFAR-100 [3] dataset are placed on the MacBook Pro. On the Jetson Nano, the learning rate, number of epochs, and batch size are set to 0.0001, 3, and 128 for the neural network training and on the MacBook Pro, the learning rate, number of epochs, and batch size are set to 0.0001, 10, and 128 for the neural network training. The purpose of these experiments is to observe the overall energy consumption across both devices when the number of samples to be trained on the less powerful device is small and the number of samples to be trained on the powerful device is high. Hence, for this reason, the Stanford Dogs [7] dataset is chosen randomly as well since this dataset has a smaller number of samples per class as compared to CIFAR-100 [3].

In this experiment, five classes from the Stanford Dogs [7] dataset are placed on the Jetson Nano and all of the 100 classes of CIFAR-100 [3] are placed on the MacBook Pro. Training is first started on the MacBook Pro where 100 classes of CIFAR-100 [3] are trained whereby 10 classes are trained incrementally at a time. Followed by this, the neural network is then sent to the Jetson Nano which has the Stanford Dogs [7] dataset where five classes are trained.

In this case, all of the samples in CIFAR-100 [3] are being trained on the MacBook Pro, hence the long training time on the MacBook Pro results in the long waiting time on the Jetson Nano as seen in Figure 5.9 a). On the Jetson Nano, only 5 classes belonging to the Stanford Dogs [7] dataset are being learnt. Hence, the training time on the Jetson Nano is faster as compared to the training time on the MacBook Pro.

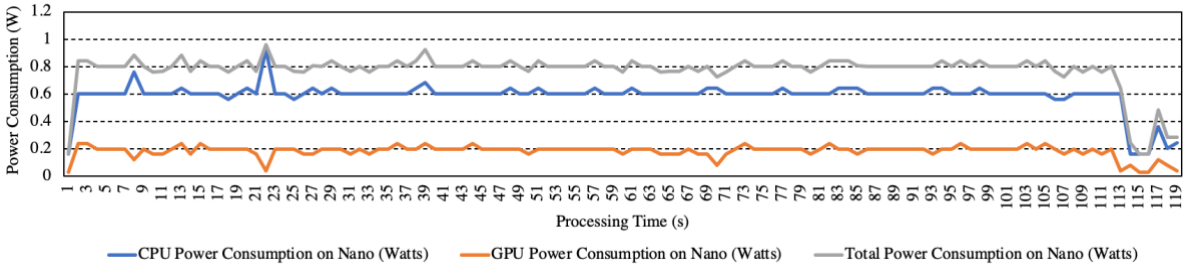


a) Power consumption and processing time on the Jetson Nano.

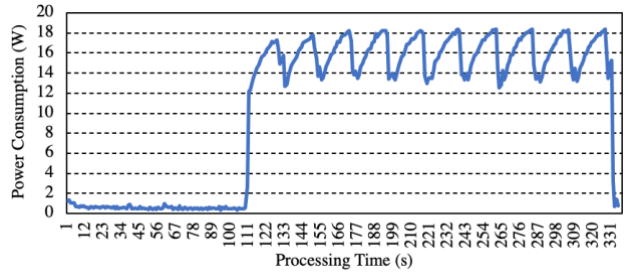


b) Power consumption and processing time on the MacBook Pro.

Figure 5.9. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 100 classes of CIFAR-100 [3] are first learnt on the MacBook Pro and 5 classes of Stanford Dogs [3] are then learnt on the Jetson Nano.



a) Power consumption and processing time on the Jetson Nano.



b) Power consumption and processing time on the MacBook Pro.

Figure 5.10. Power consumption and processing time across the MacBook Pro and the Jetson Nano where 5 classes of Stanford Dogs [7] are first learnt on the Jetson Nano and 100 classes of CIFAR-100 [3] are then learnt on the MacBook Pro.

Similarly, it can be seen in Figure 5.10 that the MacBook Pro has to wait for the Jetson Nano to complete the training if the Jetson Nano is the first device to start training. However, since only 5 classes of the Stanford Dogs [7] dataset are being learnt on the Jetson Nano, the training time is not very large and hence the waiting time on the MacBook Pro is also not very large.

Table 5.4. Power, time, and energy consumption across edge devices during distributed multi-task learning. The MacBook Pro having 100 classes of CIFAR-100 [3] and the Jetson Nano having 10 classes of Stanford Dogs [7] dataset.

Training Order	**MacBook Pro**	**Jetson Nano**	
	Power Consumption (Watts)		**Total Power Consumption (Watts)**
MacBook Pro then Jetson Nano	3556.39	128.11	3684.5
Jetson Nano then MacBook Pro	3967.11	92.07	4059.18
	Execution time (s)		**Total Execution Time (s)**
MacBook Pro then Jetson Nano	224.12	335.82	559.94
Jetson Nano then MacBook Pro	336.59	111.86	448.45
	Energy Consumption (kWh)		**Total Energy Consumption (kWh)**
MacBook Pro then Jetson Nano	0.22	0.01	0.23
Jetson Nano then MacBook Pro	0.37	0.002	0.37

From Table 5.4, it can be observed that the overall time consumed across both the edge devices combined is much smaller when training is first started on the Jetson Nano as compared to starting the training on the MacBook Pro. For this experiment, the number of epochs and the number of samples to be trained on the MacBook Pro are increased which is why if the training starts on the MacBook Pro, then the training time also increases, and hence, the Jetson Nano must wait longer for the MacBook Pro to complete the training. This is because when the training is started on the Jetson Nano, the time taken to receive the training order from the cloud is a lot lesser as compared to the waiting time if the training starts on the MacBook Pro.

However, if the training first starts on the MacBook Pro, then the overall energy consumption is smaller as compared to if the training first starts on the Jetson Nano. Even though the overall processing time is smaller across both edge devices when the Jetson Nano starts the training, the MacBook Pro has to wait for the Jetson Nano to complete training, and the power consumption of the MacBook Pro is a lot higher than the power consumption of the Jetson Nano. Hence, this alone leads to a higher energy consumption if the training starts on the Jetson Nano first. This experiment proves that the minimization of the overall execution time across all edge devices does not necessarily lead to the minimized energy consumption across all edge devices due to the power consumption differences across the edge devices.

5.8. Edge Device Training Order Selection Algorithm Results

This section presents the results of the edge device training order selection algorithm. Table 5.5 shows the predicted training time on each edge device along with the actual training time. The MacBook Pro is denoted by the number 0 and the Jetson Nano is denoted by the number

1 so for e.g., the term 0 → 1 denotes that training is first carried out on the MacBook Pro followed by the Jetson Nano.

Table 5.5. The actual and predicted neural network training times on edge devices.

	70-30 Split (CIFAR-100 [3]**)**	
	MacBook Pro	**Jetson Nano**
Actual Training Time (s)	155.40	2089.09
Predicted Training Time (s)	163.05	2100.6
Difference (%)	**4.80**	**0.55**
	50-50 Split (CIFAR-100 [3]**)**	
	MacBook Pro	**Jetson Nano**
Actual Training Time (s)	111.30	3502.56
Predicted Training Time (s)	109.99	3364.67
Difference (%)	**1.18**	**4.01**
	Stanford Dogs [7] **- 5 classes, CIFAR-100** [3] **- 100 classes**	
	MacBook Pro	**Jetson Nano**
Actual Training Time (s)	224.08	111.73
Predicted Training Time (s)	234.27	107.61
Difference (%)	**4.45**	**3.76**

From Table 5.5, it can be seen that all of the predicted training times on the edge devices are within 5% of the actual training times. The fact that some of the predicted training times are very close to the actual training times as compared to others suggests that during training, the time taken for carrying out a single full pass (forward and backward propagation of a mini-batch of features) is not the same at every epoch. To enable a regression model to be able to predict the training time that is very close to the actual training time, mini-batches of features must be made to go through a full pass multiple times and the execution time must be recorded at every iteration. However, after using the regression model mentioned in section 5.4.1., the model is still able to predict the training time that is within 5% of the actual training time.

Table 5.6. The actual and predicted energy consumption on edge devices.

	70-30 Split (CIFAR-100 [3]**)**			
Training Order	**Actual Total Energy Consumption (kWh)**	**Predicted Total Energy Consumption (kWh)**	**Difference (%)**	**Recommendation**
0 → 1	1.35	1.17	14.29	0 → 1
1 → 0	3.44	3.19	7.54	
	50-50 Split (CIFAR-100 [3]**)**			
Training Order	**Actual Total Energy Consumption (kWh)**	**Predicted Total Energy Consumption (kWh)**	**Difference (%)**	**Recommendation**
0 → 1	3.42	2.88	17.14	0 → 1
1 → 0	7.32	6.32	14.66	
	Stanford Dogs [7] **- 5 classes, CIFAR-100** [3] **- 100 classes**			

Training Order	Actual Total Energy Consumption (kWh)	Predicted Total Energy Consumption (kWh)	Difference (%)	Recommendation
0 → 1	0.23	0.25	8.33	0 → 1
1 → 0	0.35	0.37	5.56	

From Table 5.6, it can be seen that for the case when 5 classes from the Stanford Dogs [7] dataset are placed on the Jetson Nano and 100 classes from the CIFAR-100 [3] dataset are placed on the MacBook Pro, then the percentage difference between the actual and the predicted energy consumption is within 9%. However, for the remaining experiments, the percentage difference between the actual and the predicted energy consumption ranges between 7-18%. A reason for this could be that when the cloud predicts the energy consumption on the edge device, it uses a single power consumption value to predict the energy consumption on edge devices. However, as seen in section 5.6-section 5.7, the power consumption of edge devices keeps varying slightly throughout the training process. This is the reason why there is a difference between the actual and predicted energy consumption. Nonetheless, the edge device selection algorithm presented in Algorithm 5.1 is able to correctly predict the edge device training order that can minimize the energy consumption across both edge devices.

5.9. Effect of Edge Device Training Order on The Model Accuracy

As seen that the order in which edge devices participate in the training has a big effect on the overall energy consumption across all edge devices. However, it is also important to note that the effect of the edge device training order should not affect the model accuracy. In multi-task learning, the accuracy of the model is computed using only the SoftMax layer which is used to learn the respective tasks. Table 5.7 shows the top – 5 classification accuracies across both edge devices under different edge device training orders when 50 classes of the CIFAR-100 [3] dataset are placed on the Jetson Nano and the remaining 50 classes are placed on the MacBook Pro.

Table 5.7. Multi-task learning classification accuracies across multiple edge devices with respect to different edge device training orders. 50 classes each of the CIFAR-100 [3] dataset are placed on the MacBook Pro and the Jetson Nano.

	Top – 5 Accuracy for Number of Classes Learnt (%)									
	Jetson Nano (1)					**MacBook Pro (0)**				
Training Order	60	70	80	90	100	10	20	30	40	50
0 → 1	91.00	91.75	91.83	91.25	93.25	85.83	84.00	80.92	85.58	87.00
	Top – 5 Accuracy for Number of Classes Learnt (%)									
Training Order	10	20	30	40	50	60	70	80	90	100
1 → 0	89.75	93.42	90.00	93.83	94.00	83.33	85.67	81.67	87.50	84.83

Table 5.8 shows the top – 5 classification accuracies across both edge devices under different edge device training orders when 70 classes of the CIFAR-100 [3] dataset are placed on the Jetson Nano and the remaining 30 classes are placed on the MacBook Pro.

Table 5.8. Multi-task learning classification accuracies across multiple edge devices with respect to different edge device training orders. 70 classes of the CIFAR-100 [3] dataset are placed on the MacBook Pro and the remaining 30 classes placed on the Jetson Nano.

	Top – 5 Accuracy for Number of Classes Learnt (%)									
	Jetson Nano (1)			**MacBook Pro (0)**						
Training Order	80	90	100	10	20	30	40	50	60	70
0 → 1	90.75	92.50	93.25	85.58	87.42	83.92	80.67	91.83	81.92	86.42
	Top – 5 Accuracy for Number of Classes Learnt (%)									
Training Order	10	20	30	40	50	60	70	80	90	100
1 → 0	92.00	91.16	91.25	84.50	87.83	85.58	80.92	90.25	80.17	85.17

From Table 5.7 and Table 5.8, it can be seen that regardless of the edge device training order, there is not much change in the classification accuracies. The accuracies are not exactly the same because every time the experiment is carried out, the parameters of the fully connected layers on all edge devices are initialized randomly. Despite this effect, all classification accuracies are within 3% of each other regardless of the edge device training order. These results show that in multi-task learning, the classification accuracies can be retained while reducing the overall end-to-end energy consumption across all edge devices.

Single-incremental-task learning is performed in Chapters 3 and 4 whereby only one SoftMax layer is used for learning and performing classification. The classification accuracies reported in Table 3.7 are obtained using the SqueezeNet [6] model and the CIFAR-100 [3] dataset under the single-incremental-task learning. In this section, the same model and dataset are used, however, the training scheme is multi-task learning instead of single-incremental-task learning. Table 5.9 shows the differences in the classification accuracies.

Table 5.9. Top – 5 classification accuracies of the SqueezeNet [6] model trained on the CIFAR-100 [3] dataset incrementally using 10 classes at a time in a single-incremental-task learning scenario and multi-task learning scenario.

	Top – 5 Classification Accuracies (%) using SqueezeNet [6] model and CIFAR-100 [3] dataset				
	Single-incremental-task learning	Multi-task learning			
Total Classes Trained		0 → 1/ 50-50 Split	1 → 0/ 50-50 Split	0 → 1/ 70-30 Split	1 → 0/ 70-30 Split
10	95.40	85.83	89.75	85.58	92.00
20	83.70	84.00	93.42	87.42	91.16
30	78.43	80.92	90.00	83.92	91.25
40	71.45	85.58	93.83	80.67	84.50
50	66.60	87.00	94.00	91.83	87.83
60	64.85	91.00	83.33	81.92	85.58
70	62.56	91.75	85.67	86.42	80.92
80	60.56	91.83	81.67	90.75	90.25
90	58.73	91.25	87.50	92.50	80.17
100	56.72	93.25	84.83	93.25	85.15

From the results obtained in Table 5.9, it is evident that when using single-incremental-task learning, the classification accuracies of the model decrease as new classes are added. This is because only a single SoftMax layer is used to do all the learning as new classes are learnt, furthermore, in this learning setting, the parameters of the SoftMax layer representing the old tasks are also modified when new tasks are learnt. However, in the multi-task learning setting, the parameters of the SoftMax layer associated with the old tasks are not changed hence the knowledge of the old classes is retained.

5.10. Discussion

Less powerful edge devices such as the Jetson Nano may be very slow to carry out neural network training because of RAM limitations. If the RAM is only able to hold a very small number of features at a time before training, then in such cases, the training samples must be read from the storage devices in mini-batches and must be loaded onto the RAM of the edge device. Once this mini-batch has gone through a full pass (forward and backward pass) in a neural network, the next set of training samples must be read from the storage device and must be loaded onto the edge device RAM. Reading file locations at every epoch from the storage device is a bottleneck due to reading samples from the storage device and loading them onto the RAM of the edge device which is what causes a large training time on such edge devices. This also results in the remaining participating edge devices in the training process to wait.

It can be noticed from the results obtained that the devices which possess faster processing times usually have a higher power consumption and the devices that have a slower processing time have a lower power consumption. Therefore, if training begins on a less powerful device, the overall power consumption of the system may increase because the more powerful devices need to wait longer for the less powerful device to complete training. Due to this fact, even though in some cases if the overall training time across both edge devices is faster when the training first starts on the Jetson Nano, the overall power consumption across both devices can be smaller if the training first starts on the MacBook Pro since it will not expend power to wait for the Jetson Nano to complete the training. Edge device training order has no effect on the classification accuracies of the model being trained. This is because, as new tasks are learned, the parameters representing the old tasks are not modified. Therefore, no catastrophic forgetting occurs, however, from a machine learning point of view a downside is that at test time, choosing the SoftMax layer for classification requires knowledge of the task label. From an edge computing point of view, another limitation is that in this form of learning, the overall neural network keeps growing in size that may not be resource-friendly for edge devices especially the ones with less storage space and RAM.

Therefore, based on these properties, the cloud only needs to utilize the predicted training times on edge devices along with the power consumption values of edge devices during training and the power consumption while waiting for the cloud. Based on these parameters, the cloud can determine the order in which the training must be carried out. However, there are two areas that need improvement. Firstly, the regression model present on edge devices must be improved since the training time during every epoch is not always constant, hence, the regression model to predict neural network training time must be robust to such variations. Secondly, the cloud only uses a static value of power consumption, however, results show that the power consumption on edge devices throughout the training also keeps fluctuating. Therefore, one possible improvement is to have an algorithm that can predict the power consumption along these fluctuations.

5.11. Summary

This chapter provides a methodology into how multi-task learning can be performed across multiple edge devices without having the edge devices share any of their training data with one another. This can be performed through dynamically expanding the neural networks and by applying regularization-based incremental learning techniques. We consider a scenario where not all edge devices can be active at the same time due to the possibility, therefore, for such cases, training cannot be carried out in parallel across the edge devices. In sequential training, certain edge devices need to wait which the other edge device carry out training which can have an effect on the end-to-end energy consumption across all devices. This is why this chapter attempts to study the effect of edge device training order selection on the end-to-end energy consumption across heterogeneous edge devices in the case of multi-task learning scenario by keeping the raw training data only on the edge devices and not sharing them. As for the neural network training, once the training is completed on an edge device, the weights are frozen on the next edge device while only the newly added weights to the neural network are trained. To predict the edge device training order, a regression model is presented in this chapter which can predict the neural network training time on an edge device in both cases where the number of features to be trained can and cannot be stored at a time on an edge device's RAM. Based on the predicted neural network training times on the edge devices and the power consumption ratings, the cloud then determines the training order in which training should be carried out to minimize the end-to-end energy consumption across all edge devices. The proposed solutions are then tested by using two different edge devices where training samples for different datasets are placed on each device in various quantities. Results show that the overall energy consumption and the overall processing time across all edge devices are highly dependent on the order in which training is carried out across edge devices. The total energy consumption across all edge devices can be reduced by a factor of up 2.73x if the correct training order can be determined with the help of Algorithm 5.1.

In this chapter, a limitation from an AI point of view is that incremental learning is performed by adding a new SoftMax classification layer every time new tasks are learnt and as mentioned, this may not be resource-friendly for resource-constrained edge devices especially when lots of tasks are needed to be learned. Furthermore, during inference, knowledge of the task is needed to be known to know which SoftMax classification layer to be used for inference [37].

Chapter 6

Conclusion and Future Work

This research tries to emphasize the fact that as the number of IoT edge devices grows exponentially, the computational load on the cloud will also increase especially in the case where these devices completely rely on the cloud for computation. Along with this, the data transmission costs between the edge devices and the cloud also increase. To mitigate this problem, intelligence must be offloaded from the cloud to the edge devices so that the cloud is not the only component performing intelligent computation. Due to the fact that the distribution of the real-world data collected by IoT edge devices always changes over time, intelligence must also be continuous in nature. Therefore, going forward, incremental learning and edge computing have to be merged together to tackle all of the mentioned challenges.

This research mainly presents edge computing techniques to reduce data transmission costs between the edge device and the cloud. Firstly, not all of the training images are transmitted to the cloud from the edge device for training while maintaining the model performance on the cloud. Secondly, after neural network training has taken place on the cloud, the 2^{nd} step of reducing the transmission cost is performed by picking only the useful weights of the trained neural network and sending them back to the edge device. By achieving these two steps of data transmission reduction, the main objectives of this research are achieved. However, an additional chapter is presented that aims to reduce the end-to-end energy consumption on multiple edge devices when multiple edge devices are connected to the cloud and participate in distributed multi-task learning.

Aim 1: In an edge-cloud system whereby the training samples arrive first at the edge and when the training is carried out on the cloud, samples must be transmitted to the cloud for training. In the case of incremental learning, models must learn continuously from the training data, hence in the context of an edge-cloud system, the edge device then needs to continuously transmit training samples to the cloud. Hence, the Sample Elimination Algorithm (SEA) is proposed to satisfy this objective that can prevent a number of images from being transmitted to the cloud for training. Experiments were carried out on various datasets and each dataset was evaluated using a randomly chosen pre-trained CNN architecture. For each of these combinations, the hyperparameters for training the fully connected layers for classification are also chosen randomly. The reason for doing so is to test if SEA can work under multiple training setups. The results showed that regardless of the hyperparameters, dataset, and CNN architecture, even after the data sampling algorithm takes place, the model performance does not get affected on the cloud. Furthermore, the training time on the cloud also gets faster since there are not many samples for the neural network to learn from. SEA eliminates training samples in a random manner from a class on an edge device because it is shown in Figure 3.7 that as incremental learning progresses, the pattern in which the number of samples that have a higher entropy value after training becomes randomized. Hence, the training samples also must be dropped in a random manner on an edge device. SEA is also coupled with a hyperparameter (ω^c) that can control the amount of data sampling that can be performed.

Aim 2: After the neural network training is completed on the cloud, the neural network needs to be sent back to the edge device to ensure that the latest trained classifier is present on the edge device as well. The contribution to satisfy this objective is largely based on an algorithm known as FitCNN [10]. This work presents an algorithm that can select the useful weights from

a neural network that can then be sent back to the edge device. This also leads to a reduction in the data transmission cost. Chapter 4 presents an algorithm that can select the same number of useful weights as [10], however, the only difference is the speed at which the weight selection is done. This chapter presents two variants of FitCNN [10] named FitCNN+ [28] and FitCNN++. To select useful weights, a difference is computed between the weights of the trained neural network and the pre-trained neural network. This weight difference is then split up into multiple sections and the weight difference value at every divider is considered as a threshold. As per FitCNN [10], a useful weight is the one where the absolute value of the difference between a post-trained weight and a pre-trained weight is greater than a threshold. After a threshold has been used to select the useful weights, the training accuracy is computed to check if the performance of the model is still intact or not. This process repeats for all thresholds up until the point where a threshold is encountered that makes the training accuracy fall outside the acceptable training loss. When this happens, the previous threshold is used to get back the useful weights. This denotes the least number of useful weights that can be transmitted back to the edge device without affecting the classifier performance.

FitCNN+ [28] speeds up the weight selection process by choosing only the positive thresholds unlike FitCNN [10] that uses all of the thresholds. This way, the training accuracy does not need to be computed for all of the thresholds which is why FitCNN+ [28] is faster than FitCNN [10]. FitCNN++ goes a step further in reducing the number of thresholds to be used by using only a subset of the positive thresholds to determine the least number of useful weights. This is done by splitting the positive weight difference distribution list into four sections, the divider at the end of each section is chosen as the threshold. This means that four threshold values are obtained and the threshold value that leads to the least number of useful weights lies in between one of these four threshold values. In FitCNN++, if one of the four threshold values leads to a training accuracy that is beyond the acceptable training loss, then the optimum threshold value lies in between the previous and the current threshold value. Therefore, the total thresholds to be used are the ones that lie only within this range. This is why FitCNN++ is the fastest algorithm to extract the useful weights.

Aim 3: For this objective, multi-task learning is performed across multiple edge devices. Since the constraint placed here is that raw training data present on each edge device cannot be shared with one another and cannot even be shared with the cloud, to satisfy this objective, chapter 5 shows that in such cases, regularized dynamic incremental learning scenarios are the only way this can be achieved. In regularized dynamic incremental learning, after a neural network has been trained and new tasks need to be learnt, the weights belonging to the previously learnt classes are prevented from changing largely or not at all when new classes are being learnt. This way, the old knowledge of the neural networks can be retained. However, the main objective of this chapter is to study the effect of the edge device training order on the overall energy consumption across all of the edge devices. Due to regularized dynamic incremental learning being employed, training can only be carried out on one edge device at a time. While that happens, the remaining edge devices need to wait to receive the trained neural network. The power consumption on the edge devices during waiting can accumulate with respect to the time the edge devices need to wait. The results presented in this chapter show that depending on the number of samples to be trained on an edge device, hyperparameters, hardware properties, and neural network architecture, the overall energy consumption across the edge devices can get impacted. Due to this phenomenon, each edge device uses a regression model that is used to predict the neural network training time on the edge device itself whereby these predicted training times are transmitted to the cloud. The cloud uses these predicted training time values along with the power consumption values during training on each device to predict

which training order will lead to the least to the smallest end-to-end energy consumption. Results show that the algorithm proposed in chapter 5 is able to correctly compute the edge device training order in which the training should be carried out to minimize the overall energy consumption across all edge devices.

Future work: Regarding pre-training data sampling on edge devices, one advancement that can be made is by coming up with a novel algorithm that can automatically cluster new incoming samples, by doing so, pre-training data sampling on edge devices can be performed in an unsupervised manner. This is vital due to the fact that edge devices are the first devices that receive large amounts of raw data and edge devices also have very little human supervision. Another advancement that can be made on the edge device is to predict the loss of neural network training accuracy with respect to the amount of data sampling that takes place on the edge device. For e.g., compared to the baseline (no data sampling), if the tolerable accuracy drop is only 0.5% after data sampling, then based on this input hyperparameter, the value of ω^c should be able to be computed that can automatically filter the number of samples. This provides more control over the model performance while reducing the transmission costs as well. This feature can let the edge device know if the number of samples to be eliminated needs to be reduced, increased, or be kept the same with respect to the predicted model performance.

Regarding the weight extraction algorithm, the training accuracy of the neural network has to be computed first before a larger threshold can be used to compute the training accuracy again. To compute the training accuracy, the training dataset has to be present, and the larger the dataset, the higher the computational cost of computing the training accuracy. As mentioned, the current process of weight extraction can be computationally expensive with respect to the number of forward passes, the size of the training set, and the size of the neural network. For a large number of training samples, sampling techniques can be employed to select a subsample that represents the original training distribution. An innovative algorithm needs to be formulated that can determine the useful weights throughout the course of the neural network training particularly towards the end of the training since that is the period when the majority of the weights of a neural network have been learnt. This can eliminate the need for computing the training accuracy for every threshold value and can thus lead to a very fast weight extraction algorithm. This feature will become even more useful when applied to the convolutional layers of a neural network since the convolutional layers can be very deep as well, hence the weight extraction must be very fast and efficient when handling different types of layers in a neural network.

As mentioned in section 5.9, a lot of improvements are needed to predict the total neural network training energy consumption on edge devices. This is because the power consumption throughout the training does not remain constant. Therefore, future work should involve the development of an innovative algorithm that can precisely predict the overall neural network training energy consumption on edge devices. Such algorithms can be very helpful in increasing the performance of edge-cloud scheduling depending on the objective to be optimized for e.g., energy consumption or execution time. For e.g., with the presence of such predictive algorithms, the cost of training on edge devices can be predicted accurately, and based on this information, if the cost of training a neural network is much higher on the edge device as compared to transmitting the neural network and data to the cloud and carrying out training there, then the training process on the edge device can be offloaded to the cloud thereby saving overall edge-cloud computational costs. In the incremental learning context, to prevent a large growth of the neural network in size, a forgetting mechanism must be in place whereby the parameters associated with the very old or irrelevant tasks can be removed. Another feature

that can be implemented is that as SoftMax layers are being trained to learn new tasks, the training of these networks should be carried out in a way such that these networks are Bayesian [183]. This is because traditional deep learning models can output very high confidence scores for samples that have never been trained before [184], however, Bayesian networks are able to tell if a sample has been encountered by a neural network before or not. This way, during inference time in multi-task learning, the test sample can be forward propagated to all the SoftMax layers, and the SoftMax layer which was trained on a similar distribution will give the highest score. This way, the entire neural network will automatically know which SoftMax layer to send the test sample to and get the classification score at inference time.

Bibliography

[1] L. Fei-Fei, J. Deng, and K. Li, “ImageNet: Constructing a large-scale image database,” *J. Vis.*, vol. 9, no. 8, pp. 1037–1037, 2010, doi: 10.1167/9.8.1037.

[2] A. Swanson, M. Kosmala, C. Lintott, R. Simpson, A. Smith, and C. Packer, “Snapshot Serengeti, high-frequency annotated camera trap images of 40 mammalian species in an African savanna,” *Sci. Data*, vol. 2, pp. 1–14, 2015, doi: 10.1038/sdata.2015.26.

[3] A. Krizhevsky, “Learning multiple layers of features from tiny images,” 2009. https://www.cs.toronto.edu/~kriz/cifar.html (accessed Jan. 10, 2020).

[4] M. Tan *et al.*, “Mnasnet: Platform-aware neural architecture search for mobile,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2019-June, pp. 2815–2823, 2019, doi: 10.1109/CVPR.2019.00293.

[5] P. Welinder, S. Branson, T. Mita, C. Wah, and F. Schroff, “Caltech-ucsd Birds 200,” *Caltech-UCSD Tech. Rep.*, vol. 200, pp. 1–15, 2010, doi: CNS-TR-2010-001.

[6] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size,” pp. 1–13, 2016, [Online]. Available: http://arxiv.org/abs/1602.07360.

[7] A. Khosla, N. Jayadevaprakash, B. Yao, and L. Fei-Fei, “Novel dataset for fine-grained image categorization,” *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, vol. 2, p. 1, 2011.

[8] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 4510–4520, 2018, doi: 10.1109/CVPR.2018.00474.

[9] N. Ma, X. Zhang, H. T. Zheng, and J. Sun, “Shufflenet V2: Practical guidelines for efficient cnn architecture design,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 11218 LNCS, pp. 122–138, 2018, doi: 10.1007/978-3-030-01264-9_8.

[10] D. Liu *et al.*, “FitCNN: A cloud-assisted and low-cost framework for updating CNNs on IoT devices,” *Futur. Gener. Comput. Syst.*, vol. 91, pp. 277–289, 2019, doi: 10.1016/j.future.2018.09.020.

[11] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778, doi: 10.1109/CVPR.2016.90.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Neural Inf. Process. Syst.*, pp. 1106–1114, 2012.

[13] M. Asif-Ur-Rahman *et al.*, “Toward a Heterogeneous Mist, Fog, and Cloud-Based Framework for the Internet of Healthcare Things,” *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4049–4062, Jun. 2019, doi: 10.1109/JIOT.2018.2876088.

[14] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, “Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence,” *IEEE Internet Things J.*, vol. 7, no. 8, pp. 7457–7469, Aug. 2020, doi: 10.1109/JIOT.2020.2984887.

[15] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017, [Online]. Available: http://ieeexplore.ieee.org/document/8114708/.

[16] G. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network,” *Adv. Neural Inf. Process. Syst.*, no. Nips, pp. 1–9, 2015, [Online]. Available: http://arxiv.org/abs/1503.02531.

[17] J. H. Luo, J. Wu, and W. Lin, “ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression,” *Proc. IEEE Int. Conf. Comput. Vis.*, vol. 2017-Octob, pp. 5068–5076, 2017, doi: 10.1109/ICCV.2017.541.

[18] E. D. Karnin, “A Simple Procedure for Pruning Back-Propagation Trained Neural Networks,” *IEEE Trans. Neural Networks*, vol. 1, no. 2, pp. 239–242, 1990, doi: 10.1109/72.80236.
[19] X. Long, Z. Ben, and Y. Liu, “A Survey of Related Research on Compression and Acceleration of Deep Neural Networks,” *J. Phys. Conf. Ser.*, vol. 1213, no. 5, 2019, doi: 10.1088/1742-6596/1213/5/052003.
[20] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, “Importance estimation for neural network pruning,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2019-June, no. 11264, pp. 11256–11264, 2019, doi: 10.1109/CVPR.2019.01152.
[21] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” *4th Int. Conf. Learn. Represent. ICLR 2016 - Conf. Track Proc.*, pp. 1–14, 2016.
[22] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients,” vol. 1, no. 1, pp. 1–13, 2016, [Online]. Available: http://arxiv.org/abs/1606.06160.
[23] C. Zhu, H. Mao, S. Han, and W. J. Dally, “Trained ternary quantization,” *5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc.*, pp. 1–10, 2019.
[24] C. Louizos, M. Reisser, T. Blankevoort, E. Gavves, and M. Welling, “Relaxed quantization for discretized neural networks,” *7th Int. Conf. Learn. Represent. ICLR 2019*, pp. 1–14, 2019.
[25] A. Paszke *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” *Adv. Neural Inf. Process. Syst.*, no. NeurIPS, 2019, [Online]. Available: http://arxiv.org/abs/1912.01703.
[26] M. Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” 2016, [Online]. Available: http://arxiv.org/abs/1603.04467.
[27] T. Lesort, V. Lomonaco, A. Stoian, D. Maltoni, D. Filliat, and N. Díaz-Rodríguez, “Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges,” *Inf. Fusion*, vol. 58, pp. 52–68, 2020, doi: 10.1016/j.inffus.2019.12.004.
[28] S. Dube, W. Y. Wan, and H. Nugroho, “A Novel Approach of IoT Stream Sampling and Model Update on the IoT Edge Device for Class Incremental Learning in an Edge-Cloud System,” *IEEE Access*, vol. 9, pp. 29180–29199, 2021, doi: 10.1109/ACCESS.2021.3059251.
[29] S. Dube, Y. W. Wong, and H. Nugroho, “Dynamic sampling of images from various categories for classification based incremental deep learning in fog computing,” *PeerJ Comput. Sci.*, vol. 7, p. e633, Jul. 2021, doi: 10.7717/peerj-cs.633.
[30] G. Alain, A. Lamb, C. Sankar, A. Courville, and Y. Bengio, “Variance Reduction in SGD by Distributed Importance Sampling,” pp. 1–18, 2015, [Online]. Available: http://arxiv.org/abs/1511.06481.
[31] I. Loshchilov and F. Hutter, “Online Batch Selection for Faster Training of Neural Networks,” no. 2015, pp. 1–20, 2015, [Online]. Available: http://arxiv.org/abs/1511.06343.
[32] S. Gopal, “Adaptive sampling for SGD by exploiting side information,” *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 1, pp. 567–575, 2016.
[33] A. Katharopoulos and F. Fleuret, “Not all samples are created equal: Deep learning with importance sampling,” *35th Int. Conf. Mach. Learn. ICML 2018*, vol. 6, pp. 3936–3949, 2018.
[34] D. Needell, N. Srebro, and R. Ward, “Stochastic gradient descent, weighted sampling, and the randomized Kaczmarz algorithm,” *Math. Program.*, vol. 155, no. 1–2, pp.

549–573, 2016, doi: 10.1007/s10107-015-0864-7.
[35] S. A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, “iCaRL: Incremental classifier and representation learning,” *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 5533–5542, 2017, doi: 10.1109/CVPR.2017.587.
[36] R. Kemker and C. Kanan, “FearNet: Brain-inspired model for incremental learning,” *6th Int. Conf. Learn. Represent. ICLR 2018 - Conf. Track Proc.*, pp. 1–16, 2018.
[37] A. A. Rusu *et al.*, “Progressive Neural Networks,” 2016, [Online]. Available: http://arxiv.org/abs/1606.04671.
[38] J. Kirkpatrick *et al.*, “Overcoming catastrophic forgetting in neural networks,” *Proc. Natl. Acad. Sci. U. S. A.*, vol. 114, no. 13, pp. 3521–3526, 2017, doi: 10.1073/pnas.1611835114.
[39] A. Mallya, D. Davis, and S. Lazebnik, “Piggyback: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 11208 LNCS, pp. 72–88, 2018, doi: 10.1007/978-3-030-01225-0_5.
[40] A. Mallya and S. Lazebnik, “PackNet: Adding Multiple Tasks to a Single Network by Iterative Pruning,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 7765–7773, 2018, doi: 10.1109/CVPR.2018.00810.
[41] S. A. Alex and J. J. V. Nayahi, “Deep Incremental Learning for Big Data Stream Analytics,” in *Proceeding of the International Conference on Computer Networks, Big Data and IoT (ICCBI - 2018). ICCBI 2018. Lecture Notes on Data Engineering and Communications Technologies, vol 31.*, Springer, 2020, pp. 600–614.
[42] M. McCloskey and N. J. Cohen, “Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem,” *Psychol. Learn. Motiv. - Adv. Res. Theory*, vol. 24, no. C, pp. 109–165, 1989, doi: 10.1016/S0079-7421(08)60536-8.
[43] R. M. French, “Catastrophic forgetting in connectionist networks,” *Trends Cogn. Sci.*, vol. 3, no. 4, pp. 128–135, 1999, doi: 10.1016/S1364-6613(99)01294-2.
[44] C. Szegedy *et al.*, “Going deeper with convolutions,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 07-12-June, pp. 1–9, 2015, doi: 10.1109/CVPR.2015.7298594.
[45] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks For Large-Scale Image Recognition,” *Int. Conf. Learn. Represent.*, 2015.
[46] J. Chen and X. Ran, “Deep Learning With Edge Computing: A Review,” *Proc. IEEE*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019, doi: 10.1109/JPROC.2019.2921977.
[47] W. Dai, H. Nishi, V. Vyatkin, V. Huang, Y. Shi, and X. Guan, “Industrial Edge Computing: Enabling Embedded Intelligence,” *IEEE Ind. Electron. Mag.*, vol. 13, no. 4, pp. 48–56, Dec. 2019, doi: 10.1109/MIE.2019.2943283.
[48] J.-H. Huh and Y.-S. Seo, “Understanding Edge Computing: Engineering Evolution With Artificial Intelligence,” *IEEE Access*, vol. 7, pp. 164229–164245, 2019, doi: 10.1109/ACCESS.2019.2945338.
[49] O. Salman, L. Chaddad, I. H. Elhajj, A. Chehab, and A. Kayssi, “Pushing intelligence to the network edge,” in *2018 Fifth International Conference on Software Defined Systems (SDS)*, Apr. 2018, pp. 87–92, doi: 10.1109/SDS.2018.8370427.
[50] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, “Edge intelligence: Paving the last mile of artificial intelligence with edge computing,” *Proc. IEEE*, vol. 107, no. 8, 2019.
[51] F. Samie, L. Bauer, and J. Henkel, “From Cloud Down to Things: An Overview of Machine Learning in Internet of Things,” *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4921–4934, Jun. 2019, doi: 10.1109/JIOT.2019.2893866.

[52] A. Sobecki, J. Szymański, D. Gil, and H. Mora, “Deep learning in the fog,” *Int. J. Distrib. Sens. Networks*, vol. 15, no. 8, p. 155014771986707, Aug. 2019, doi: 10.1177/1550147719867072.
[53] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth, “Machine learning for internet of things data analysis: a survey,” *Digit. Commun. Networks*, vol. 4, no. 3, pp. 161–175, 2018, doi: 10.1016/j.dcan.2017.10.002.
[54] C. González García, E. Núñez-Valdez, V. García-Díaz, C. Pelayo G-Bustelo, and J. M. Cueva-Lovelle, “A Review of Artificial Intelligence in the Internet of Things,” *Int. J. Interact. Multimed. Artif. Intell.*, vol. 5, no. 4, p. 9, 2019, doi: 10.9781/ijimai.2018.03.004.
[55] C. Tsai, P. Tsai, M. Chiang, and C. Yang, “Data analytics for internet of things: A review,” *WIREs Data Min. Knowl. Discov.*, vol. 8, no. 5, Sep. 2018, doi: 10.1002/widm.1261.
[56] R. Reddy and A. V. L. N. Sujith, “A Comprehensive Literature Review on Data Analytics in IIoT (Industrial Internet of Things),” *HELIX*, vol. 8, no. 1, pp. 2757–2764, Jan. 2018, doi: 10.29042/2018-2757-2764.
[57] G. Plastiras, M. Terzi, C. Kyrkou, and T. Theocharidcs, “Edge Intelligence: Challenges and Opportunities of Near-Sensor Machine Learning Applications,” in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul. 2018, pp. 1–7, doi: 10.1109/ASAP.2018.8445118.
[58] A. Keshavarzi and W. van den Hoek, “Edge Intelligence—On the Challenging Road to a Trillion Smart Connected IoT Devices,” *IEEE Des. Test*, vol. 36, no. 2, pp. 41–64, Apr. 2019, doi: 10.1109/MDAT.2019.2899075.
[59] M. Merenda, C. Porcaro, and D. Iero, “Edge Machine Learning for AI-Enabled IoT Devices: A Review,” *Sensors*, vol. 20, no. 9, p. 2533, Apr. 2020, doi: 10.3390/s20092533.
[60] Y. Liu, M. Peng, G. Shou, Y. Chen, and S. Chen, “Toward Edge Intelligence: Multiaccess Edge Computing for 5G and Internet of Things,” *IEEE Internet Things J.*, vol. 7, no. 8, pp. 6722–6747, Aug. 2020, doi: 10.1109/JIOT.2020.3004500.
[61] J. Asharf, N. Moustafa, H. Khurshid, E. Debie, W. Haider, and A. Wahab, “A Review of Intrusion Detection Systems Using Machine and Deep Learning in Internet of Things: Challenges, Solutions and Future Directions,” *Electronics*, vol. 9, no. 7, p. 1177, Jul. 2020, doi: 10.3390/electronics9071177.
[62] J. Wanner, C. Wissuchek, and C. Janiescha, “Machine Learning and Complex Event Processing A Review of Real-time Data Analytics for the Industrial Internet of Things,” *Enterp. Model. Inf. Syst. Archit. Int. J.*, vol. 15, pp. 1:1-27, 2020, doi: 10.18417/emisa.15.1.
[63] G. Mohindru, K. Mondal, and H. Banka, “Internet of Things and data analytics: A current review,” *WIREs Data Min. Knowl. Discov.*, vol. 10, no. 3, May 2020, doi: 10.1002/widm.1341.
[64] S. Sepasgozar *et al.*, “A Systematic Content Review of Artificial Intelligence and the Internet of Things Applications in Smart Home,” *Appl. Sci.*, vol. 10, no. 9, p. 3074, Apr. 2020, doi: 10.3390/app10093074.
[65] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, “A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective,” *Comput. Networks*, vol. 182, p. 107496, Dec. 2020, doi: 10.1016/j.comnet.2020.107496.
[66] T. Zhang, Z. Shen, J. Jin, X. Zheng, A. Tagami, and X. Cao, “Achieving Democracy in Edge Intelligence: A Fog-Based Collaborative Learning Scheme,” *IEEE Internet*

Things J., vol. 8, no. 4, pp. 2751–2761, Feb. 2021, doi: 10.1109/JIOT.2020.3020911.
[67] G. Carvalho, B. Cabral, V. Pereira, and J. Bernardino, "Computation offloading in Edge Computing environments using Artificial Intelligence techniques," *Eng. Appl. Artif. Intell.*, vol. 95, p. 103840, Oct. 2020, doi: 10.1016/j.engappai.2020.103840.
[68] H. Yang, A. Alphones, Z. Xiong, D. Niyato, J. Zhao, and K. Wu, "Artificial-Intelligence-Enabled Intelligent 6G Networks," *IEEE Netw.*, vol. 34, no. 6, pp. 272–280, Nov. 2020, doi: 10.1109/MNET.011.2000195.
[69] L. Sun, X. Jiang, H. Ren, and Y. Guo, "Edge-Cloud Computing and Artificial Intelligence in Internet of Medical Things: Architecture, Technology and Application," *IEEE Access*, vol. 8, pp. 101079–101092, 2020, doi: 10.1109/ACCESS.2020.2997831.
[70] S. Lins *et al.*, "Artificial Intelligence for Enhanced Mobility and 5G Connectivity in UAV-Based Critical Missions," *IEEE Access*, vol. 9, pp. 111792–111801, 2021, doi: 10.1109/ACCESS.2021.3103041.
[71] M. Asim, Y. Wang, K. Wang, and P.-Q. Huang, "A Review on Computational Intelligence Techniques in Cloud and Edge Computing," *IEEE Trans. Emerg. Top. Comput. Intell.*, vol. 4, no. 6, pp. 742–763, Dec. 2020, doi: 10.1109/TETCI.2020.3007905.
[72] A. Munir, E. Blasch, J. Kwon, J. Kong, and A. Aved, "Artificial Intelligence and Data Fusion at the Edge," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 36, no. 7, pp. 62–78, Jul. 2021, doi: 10.1109/MAES.2020.3043072.
[73] H. Ji, O. Alfarraj, and A. Tolba, "Artificial Intelligence-Empowered Edge of Vehicles: Architecture, Enabling Technologies, and Applications," *IEEE Access*, vol. 8, pp. 61020–61034, 2020, doi: 10.1109/ACCESS.2020.2983609.
[74] F. Liang, W. Yu, X. Liu, D. Griffith, and N. Golmie, "Toward Edge-Based Deep Learning in Industrial Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4329–4341, May 2020, doi: 10.1109/JIOT.2019.2963635.
[75] Y. Du, Z. Wang, and V. C. M. Leung, "Blockchain-Enabled Edge Intelligence for IoT: Background, Emerging Trends and Open Issues," *Futur. Internet*, vol. 13, no. 2, p. 48, Feb. 2021, doi: 10.3390/fi13020048.
[76] A. Al-Ansi, A. M. Al-Ansi, A. Muthanna, I. A. Elgendy, and A. Koucheryavy, "Survey on Intelligence Edge Computing in 6G: Characteristics, Challenges, Potential Use Cases, and Market Drivers," *Futur. Internet*, vol. 13, no. 5, p. 118, Apr. 2021, doi: 10.3390/fi13050118.
[77] Z. Xu, W. Liu, J. Huang, C. Yang, J. Lu, and H. Tan, "Artificial Intelligence for Securing IoT Services in Edge Computing: A Survey," *Secur. Commun. Networks*, vol. 2020, pp. 1–13, Sep. 2020, doi: 10.1155/2020/8872586.
[78] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "Moving Deep Learning to the Edge," *Algorithms*, vol. 13, no. 5, p. 125, May 2020, doi: 10.3390/a13050125.
[79] J. Zhang and D. Tao, "Empowering Things With Intelligence: A Survey of the Progress, Challenges, and Opportunities in Artificial Intelligence of Things," *IEEE Internet Things J.*, vol. 8, no. 10, pp. 7789–7817, May 2021, doi: 10.1109/JIOT.2020.3039359.
[80] S. Ben Atitallah, M. Driss, W. Boulila, and H. Ben Ghézala, "Leveraging Deep Learning and IoT big data analytics to support the smart cities development: Review and future directions," *Comput. Sci. Rev.*, vol. 38, p. 100303, Nov. 2020, doi: 10.1016/j.cosrev.2020.100303.
[81] Imran, Z. Ghaffar, A. Alshahrani, M. Fayaz, A. M. Alghamdi, and J. Gwak, "A Topical Review on Machine Learning, Software Defined Networking, Internet of Things Applications: Research Limitations and Challenges," *Electronics*, vol. 10, no. 8, p. 880, Apr. 2021, doi: 10.3390/electronics10080880.

[82] Y. Liu, J. Wang, J. Li, S. Niu, and H. Song, “Class-Incremental Learning for Wireless Device Identification in IoT,” *IEEE Internet Things J.*, vol. 8, no. 23, pp. 17227–35, 2021, doi: 10.1109/JIOT.2021.3078407.
[83] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 8689 LNCS, no. PART 1, pp. 818–833, 2014, doi: 10.1007/978-3-319-10590-1_53.
[84] G. Venkatesh, E. Nurvitadhi, and D. Marr, “Accelerating Deep Convolutional Networks using low-precision and sparsity,” *ICASSP, IEEE Int. Conf. Acoust. Speech Signal Process. - Proc.*, pp. 2861–2865, 2017, doi: 10.1109/ICASSP.2017.7952679.
[85] M. Courbariaux, Y. Bengio, and J. P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” *Adv. Neural Inf. Process. Syst.*, vol. 2015-Janua, pp. 3123–3131, 2015.
[86] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-net: Imagenet classification using binary convolutional neural networks,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9908 LNCS, pp. 525–542, 2016, doi: 10.1007/978-3-319-46493-0_32.
[87] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” 2016, [Online]. Available: http://arxiv.org/abs/1602.02830.
[88] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, “Quantized convolutional neural networks for mobile devices,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2016-Decem, pp. 4820–4828, 2016, doi: 10.1109/CVPR.2016.521.
[89] A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, “WRPN: Wide reduced-precision networks,” *6th Int. Conf. Learn. Represent. ICLR 2018 - Conf. Track Proc.*, pp. 1–11, 2018.
[90] Y. Gong, L. Liu, M. Yang, and L. Bourdev, “Compressing Deep Convolutional Networks using Vector Quantization,” pp. 1–10, 2014, [Online]. Available: http://arxiv.org/abs/1412.6115.
[91] S. Teerapittayanon, B. McDanel, and H. T. Kung, “BranchyNet: Fast inference via early exiting from deep neural networks,” *Proc. - Int. Conf. Pattern Recognit.*, vol. 0, pp. 2464–2469, 2016, doi: 10.1109/ICPR.2016.7900006.
[92] T. Cao, E. P. Lim, Z. H. Zhou, T. B. Ho, D. Cheung, and H. Motoda, “Advances in knowledge discovery and data mining: 19th pacific-asia conference, PAKDD 2015 Ho Chi Minh City, Vietnam, May 19-22, 2015 proceedings, part I,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9077, no. 2, pp. 550–562, 2015, doi: 10.1007/978-3-319-18038-0.
[93] M. D. Zeiler, “ADADELTA: An Adaptive Learning Rate Method,” 2012, [Online]. Available: http://arxiv.org/abs/1212.5701.
[94] L. N. Smith, “Cyclical learning rates for training neural networks,” *Proc. - 2017 IEEE Winter Conf. Appl. Comput. Vision, WACV 2017*, no. April, pp. 464–472, 2017, doi: 10.1109/WACV.2017.58.
[95] B. Chandra and R. K. Sharma, “Deep learning with adaptive learning rate using laplacian score,” *Expert Syst. Appl.*, vol. 63, pp. 1–7, 2016, doi: 10.1016/j.eswa.2016.05.022.
[96] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting linear structure within convolutional networks for efficient evaluation,” *Adv. Neural Inf. Process. Syst.*, vol. 2, no. January, pp. 1269–1277, 2014.
[97] M. Mathieu, M. Henaff, and Y. LeCun, “Fast training of convolutional networks through FFTS,” *2nd Int. Conf. Learn. Represent. ICLR 2014 - Conf. Track Proc.*, pp.

1–9, 2014.
[98] M. Jaderberg, A. Vedaldi, and A. Zisserman, “Speeding up convolutional neural networks with low rank expansions,” *BMVC 2014 - Proc. Br. Mach. Vis. Conf. 2014*, 2014, doi: 10.5244/c.28.88.
[99] A. G. Howard *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” 2017, [Online]. Available: http://arxiv.org/abs/1704.04861.
[100] X. Zhang, X. Zhou, M. Lin, and J. Sun, “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 6848–6856, 2018, doi: 10.1109/CVPR.2018.00716.
[101] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, 2010, doi: 10.1561/2200000016.
[102] K. Bonawitz *et al.*, “Towards Federated Learning at Scale: System Design,” 2019, [Online]. Available: http://arxiv.org/abs/1902.01046.
[103] W. Wen *et al.*, “TernGrad: Ternary gradients to reduce communication in distributed deep learning,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, no. Nips, pp. 1510–1520, 2017.
[104] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang, “GPU asynchronous stochastic gradient descent to speed up neural network training,” *2nd Int. Conf. Learn. Represent. ICLR 2014 - Work. Track Proc.*, pp. 1–6, 2014.
[105] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *ACM Comput. Surv.*, vol. 52, no. 4, 2019, doi: 10.1145/3320060.
[106] J. Zhang, H. Yu, and I. S. Dhillon, “AutoAssist: A Framework to Accelerate Training of Deep Neural Networks,” no. Cl, 2019, [Online]. Available: http://arxiv.org/abs/1905.03381.
[107] M. Song *et al.*, “In-Situ AI: Towards Autonomous and Incremental Deep Learning for IoT Systems,” *Proc. - Int. Symp. High-Performance Comput. Archit.*, vol. 2018-Febru, pp. 92–103, 2018, doi: 10.1109/HPCA.2018.00018.
[108] J. Azar, A. Makhoul, M. Barhamgi, and R. Couturier, “An energy efficient IoT data compression approach for edge machine learning,” *Futur. Gener. Comput. Syst.*, vol. 96, pp. 168–175, 2019, doi: 10.1016/j.future.2019.02.005.
[109] F. Xhafa, B. Kilic, and P. Krause, “Evaluation of IoT stream processing at edge computing layer for semantic data enrichment,” *Futur. Gener. Comput. Syst.*, vol. 105, pp. 730–736, 2020, doi: 10.1016/j.future.2019.12.031.
[110] Y. Li, A. C. Orgerie, I. Rodero, B. L. Amersho, M. Parashar, and J. M. Menaud, “End-to-end energy models for Edge Cloud-based IoT platforms: Application to data stream analysis in IoT,” *Futur. Gener. Comput. Syst.*, vol. 87, pp. 667–678, 2018, doi: 10.1016/j.future.2017.12.048.
[111] M. S. Hossain and G. Muhammad, “Emotion recognition using secure edge and cloud computing,” *Inf. Sci. (Ny).*, vol. 504, pp. 589–601, 2019, doi: 10.1016/j.ins.2019.07.040.
[112] Y. D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, “Compression of deep convolutional neural networks for fast and low power mobile applications,” *4th Int. Conf. Learn. Represent. ICLR 2016 - Conf. Track Proc.*, pp. 1–16, 2016.
[113] E. Baccarelli *et al.*, “Fall detection in older adults with mobile IoT devices and machine learning in the cloud and on the edge,” *Inf. Sci. (Ny).*, vol. 530, pp. 148–163, 2020, doi: 10.1016/j.ins.2018.04.065.

[114] P. Punithavathi, S. Geetha, M. Karuppiah, S. H. Islam, M. M. Hassan, and K. K. R. Choo, “A lightweight machine learning-based authentication framework for smart IoT devices,” *Inf. Sci. (Ny).*, vol. 484, pp. 255–268, 2019, doi: 10.1016/j.ins.2019.01.073.
[115] K. Kawaguchi and H. Lu, “Ordered SGD: A New Stochastic Optimization Framework for Empirical Risk Minimization,” vol. 108, 2019, [Online]. Available: http://arxiv.org/abs/1907.04371.
[116] Y. Fan, S. Lyu, Y. Ying, and B. G. Hu, “Learning with average top-K loss,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, pp. 498–506, 2017.
[117] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2016-Decem, pp. 770–778, 2016, doi: 10.1109/CVPR.2016.90.
[118] Y. Gao *et al.*, “End-to-End Evaluation of Federated Learning and Split Learning for Internet of Things,” 2020, [Online]. Available: http://arxiv.org/abs/2003.13376.
[119] A. Sergeev and M. Del Balso, “Horovod: fast and easy distributed deep learning in TensorFlow,” no. September, 2018, [Online]. Available: http://arxiv.org/abs/1802.05799.
[120] R. Ratcliff, “Connectionist Models of Recognition Memory: Constraints Imposed by Learning and Forgetting Functions,” *Psychol. Rev.*, vol. 97, no. 2, pp. 285–308, 1990, doi: 10.1037/0033-295X.97.2.285.
[121] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, “DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics,” *Proc. - IEEE INFOCOM*, vol. 2018-April, pp. 1421–1429, 2018, doi: 10.1109/INFOCOM.2018.8485905.
[122] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, “MCDNN: An approximation-based execution framework for deep stream processing under resource constraints,” *MobiSys 2016 - Proc. 14th Annu. Int. Conf. Mob. Syst. Appl. Serv.*, pp. 123–136, 2016, doi: 10.1145/2906388.2906396.
[123] T. S. Sethi and M. Kantardzic, “On the reliable detection of concept drift from streaming unlabeled data,” *Expert Syst. Appl.*, vol. 82, pp. 77–99, 2017, doi: 10.1016/j.eswa.2017.04.008.
[124] J. de Andrade Silva, E. R. Hruschka, and J. Gama, “An evolutionary algorithm for clustering data streams with a variable number of clusters,” *Expert Syst. Appl.*, vol. 67, pp. 228–238, 2017, doi: 10.1016/j.eswa.2016.09.020.
[125] M. Cejnek and I. Bukovsky, “Concept drift robust adaptive novelty detection for data streams,” *Neurocomputing*, vol. 309, pp. 46–53, 2018, doi: 10.1016/j.neucom.2018.04.069.
[126] O. Aydogdu and M. Ekinci, “A new approach for data stream classification: unsupervised feature representational online sequential extreme learning machine,” *Multimed. Tools Appl.*, vol. 79, no. 37–38, pp. 27205–27227, 2020, doi: 10.1007/s11042-020-09300-y.
[127] J. A. Olvera-López, J. A. Carrasco-Ochoa, J. F. Martínez-Trinidad, and J. Kittler, “A review of instance selection methods,” *Artif. Intell. Rev.*, vol. 34, no. 2, pp. 133–143, 2010, doi: 10.1007/s10462-010-9165-y.
[128] A. Srisawat, T. Phienthrakul, and B. Kijsirikul, “SV-kNNC: An Algorithm for Improving the Efficiency of k-Nearest Neighbor,” 2006, pp. 975–979.
[129] S. Saha, P. S. Sarker, A. Al Saud, S. Shatabda, and M. A. Hakim Newton, “Cluster-oriented instance selection for classification problems,” *Inf. Sci. (Ny).*, vol. 602, pp. 143–158, Jul. 2022, doi: 10.1016/j.ins.2022.04.036.
[130] A. Lumini and L. Nanni, “A clustering method for automatic biometric template selection,” *Pattern Recognit.*, vol. 39, no. 3, pp. 495–497, Mar. 2006, doi: 10.1016/j.patcog.2005.11.004.

[131] R. A. Mollineda, F. J. Ferri, and E. Vidal, “An efficient prototype merging strategy for the condensed 1-NN rule through class-conditional hierarchical clustering,” *Pattern Recognit.*, vol. 35, no. 12, pp. 2771–2782, Dec. 2002, doi: 10.1016/S0031-3203(01)00208-4.
[132] Z. Li and D. Hoiem, “Learning without Forgetting,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 12, pp. 2935–2947, 2018, doi: 10.1109/TPAMI.2017.2773081.
[133] D. Lopez-Paz and M. Ranzato, “Gradient episodic memory for continual learning,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, no. Nips, pp. 6468–6477, 2017.
[134] D. Maltoni and V. Lomonaco, “Continuous learning in single-incremental-task scenarios,” *Neural Networks*, vol. 116, pp. 56–73, 2019, doi: 10.1016/j.neunet.2019.03.010.
[135] H. Shin, J. K. Lee, J. Kim, and J. Kim, “Continual learning with deep generative replay,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, no. Nips, pp. 2991–3000, 2017.
[136] A. Rannen, R. Aljundi, M. B. Blaschko, and T. Tuytelaars, “Encoder Based Lifelong Learning,” *Proc. IEEE Int. Conf. Comput. Vis.*, vol. 2017-Octob, pp. 1329–1337, 2017, doi: 10.1109/ICCV.2017.148.
[137] M. A. Hashmani, S. M. Jameel, H. Alhussain, M. Rehman, and A. Budiman, “Accuracy performance degradation in image classification models due to concept drift,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 10, no. 5, pp. 422–425, 2019, doi: 10.14569/ijacsa.2019.0100552.
[138] F. Bellas, R. J. Duro, A. Faiña, and D. Souto, “Multilevel darwinist brain (MDB): Artificial evolution in a cognitive architecture for real robots,” *IEEE Trans. Auton. Ment. Dev.*, vol. 2, no. 4, pp. 340–354, 2010, doi: 10.1109/TAMD.2010.2086453.
[139] N. Bredeche, E. Haasdijk, and A. Prieto, “Embodied evolution in collective robotics: A review,” *Front. Robot. AI*, vol. 5, no. FEB, 2018, doi: 10.3389/frobt.2018.00012.
[140] R. Traoré *et al.*, “DisCoRL: Continual Reinforcement Learning via Policy Distillation,” no. Figure 1, pp. 1–15, 2019, [Online]. Available: http://arxiv.org/abs/1907.05855.
[141] R. Jonschkowski and O. Brock, “Learning state representations with robotic priors,” *Auton. Robots*, vol. 39, no. 3, pp. 407–428, 2015, doi: 10.1007/s10514-015-9459-7.
[142] H. Van Hoof, N. Chen, M. Karl, P. Van Der Smagt, and J. Peters, “Stable reinforcement learning with autoencoders for tactile and visual data,” *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2016-Novem, pp. 3928–3934, 2016, doi: 10.1109/IROS.2016.7759578.
[143] R. R, “Catastrophic models of recognition memory: Constraints imposed by learning and forgetting functions,” *Psychol. Rev.*, vol. 97, no. 2, pp. 285–308, 1990.
[144] R. M. French, “Semi-distributed Representations and Catastrophic Forgetting in Connectionist Networks,” *Conn. Sci.*, vol. 4, no. 3–4, pp. 365–377, 1992, doi: 10.1080/09540099208946624.
[145] G. I. Parisi, J. Tani, C. Weber, and S. Wermter, “Lifelong learning of spatiotemporal representations with dual-memory recurrent self-organization,” *Front. Neurorobot.*, vol. 12, no. November, pp. 1–19, 2018, doi: 10.3389/fnbot.2018.00078.
[146] A. Gepperth and C. Karaoguz, “A Bio-Inspired Incremental Learning Architecture for Applied Perceptual Problems,” *Cognit. Comput.*, vol. 8, no. 5, pp. 924–934, 2016, doi: 10.1007/s12559-016-9389-5.
[147] N. Kamra, U. Gupta, and Y. Liu, “Deep Generative Dual Memory Network for Continual Learning,” 2017, [Online]. Available: http://arxiv.org/abs/1710.10368.
[148] P. Sprechmann *et al.*, “Memory-based parameter adaptation,” *6th Int. Conf. Learn. Represent. ICLR 2018 - Conf. Track Proc.*, pp. 1–16, 2018.

[149] D. Das *et al.*, “Distributed Deep Learning Using Synchronous Stochastic Gradient Descent,” 2016, [Online]. Available: http://arxiv.org/abs/1602.06709.
[150] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *J. Mach. Learn. Res.*, 2014.
[151] C. Cortes, G. Research, and N. York, “L 2 Regularization for Learning Kernels,” *Proc. Twenty-Fifth Conf. Uncertain. Artif. Intell.*, 2004.
[152] F. Zenke, B. Poole, and S. Ganguli, “Continual learning through synaptic intelligence,” *34th Int. Conf. Mach. Learn. ICML 2017*, vol. 8, pp. 6072–6082, 2017.
[153] X. He and H. Jaeger, “Overcoming catastrophic interference using conceptor-aided backpropagation,” *6th Int. Conf. Learn. Represent. ICLR 2018 - Conf. Track Proc.*, no. 2014, pp. 1–11, 2018.
[154] H. Jaeger, “Using conceptors to manage neural long-term memories for temporal patterns,” *J. Mach. Learn. Res.*, vol. 18, pp. 1–43, 2017.
[155] Y. Wu *et al.*, “Incremental Classifier Learning with Generative Adversarial Networks,” 2018, [Online]. Available: http://arxiv.org/abs/1802.00853.
[156] A. A. B and M. Jansen, *Learning and Multi-atlas Segmentation for Semi-automated Labelling of Histology*, vol. 1, no. i. Springer International Publishing, 2018.
[157] A. Radford, L. Metz, and S. Chintala, “DCGAN original paper 2016,” pp. 1–16, 2016, [Online]. Available: https://arxiv.org/pdf/1511.06434.pdf.
[158] H. Caselles-Dupré, M. Garcia-Ortiz, and D. Filliat, “S-TRIGGER: Continual State Representation Learning via Self-Triggered Generative Replay,” 2019, [Online]. Available: http://arxiv.org/abs/1902.09434.
[159] T. J. Draelos *et al.*, “Neurogenesis deep learning: Extending deep networks to accommodate new classes,” *Proc. Int. Jt. Conf. Neural Networks*, vol. 2017-May, pp. 526–533, 2017, doi: 10.1109/IJCNN.2017.7965898.
[160] W. Xu, S. Keshmiri, and G. Wang, “Adversarially Approximated Autoencoder for Image Generation and Manipulation,” *IEEE Trans. Multimed.*, vol. 21, no. 9, pp. 2387–2396, 2019, doi: 10.1109/TMM.2019.2898777.
[161] E. Choi, K. Lee, and K. Choi, “Autoencoder-Based Incremental Class Learning without Retraining on Old Data,” 2019, [Online]. Available: http://arxiv.org/abs/1907.07872.
[162] C. Ma and C. Liu, “Two dimensional hashing for visual tracking,” *Comput. Vis. Image Underst.*, vol. 135, pp. 83–94, 2015, doi: 10.1016/j.cviu.2015.01.003.
[163] X. Bai, P. Ren, H. Zhang, and J. Zhou, “An incremental structured part model for object recognition,” *Neurocomputing*, vol. 154, pp. 189–199, 2015, doi: 10.1016/j.neucom.2014.12.004.
[164] J. Dou, J. Li, Q. Qin, and Z. Tu, “Moving object detection based on incremental learning low rank representation and spatial constraint,” *Neurocomputing*, vol. 168, pp. 382–400, 2015, doi: 10.1016/j.neucom.2015.05.088.
[165] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: Synthetic Minority Over-sampling Technique,” *J. Artif. Intell. Res.*, vol. 16, no. 2, pp. 321–357, Jun. 2002, doi: 10.1613/jair.953.
[166] T. Furlanello, J. Zhao, A. M. Saxe, L. Itti, and B. S. Tjan, “Active Long Term Memory Networks,” pp. 1–10, 2016, [Online]. Available: http://arxiv.org/abs/1606.02355.
[167] W. Fang, X. Yin, Y. An, N. Xiong, Q. Guo, and J. Li, “Optimal scheduling for data transmission between mobile devices and cloud,” *Inf. Sci. (Ny).*, vol. 301, no. January 2015, pp. 169–180, 2015, doi: 10.1016/j.ins.2014.12.059.
[168] W. Fang, Y. Li, H. Zhang, N. Xiong, J. Lai, and A. V. Vasilakos, “On the throughput-energy tradeoff for data transmission between cloud and mobile devices,” *Inf. Sci.*

(Ny)., vol. 283, pp. 79–93, 2014, doi: 10.1016/j.ins.2014.06.022.
[169] G. Carvalho, B. Cabral, V. Pereira, and J. Bernardino, "Computation offloading in Edge Computing environments using Artificial Intelligence techniques," *Eng. Appl. Artif. Intell.*, vol. 95, no. April, p. 103840, 2020, doi: 10.1016/j.engappai.2020.103840.
[170] M. G. R. Alam, M. M. Hassan, M. Zi. Uddin, A. Almogren, and G. Fortino, "Autonomic computation offloading in mobile edge for IoT applications," *Futur. Gener. Comput. Syst.*, vol. 90, pp. 149–157, 2019, doi: 10.1016/j.future.2018.07.050.
[171] X. Xu *et al.*, "A computation offloading method over big data for IoT-enabled cloud-edge computing," *Futur. Gener. Comput. Syst.*, vol. 95, pp. 522–533, 2019, doi: 10.1016/j.future.2018.12.055.
[172] H. Lu, C. Gu, F. Luo, W. Ding, and X. Liu, "Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning," *Futur. Gener. Comput. Syst.*, vol. 102, pp. 847–861, 2020, doi: 10.1016/j.future.2019.07.019.
[173] X. Zhao *et al.*, "Deep learning based mobile data offloading in mobile edge computing systems," *Futur. Gener. Comput. Syst.*, vol. 99, pp. 346–355, 2019, doi: 10.1016/j.future.2019.04.039.
[174] F. Lu, L. Gu, L. T. Yang, L. Shao, and H. Jin, "Mildip: An energy efficient code offloading framework in mobile cloudlets," *Inf. Sci. (Ny).*, vol. 513, no. xxxx, pp. 84–97, 2020, doi: 10.1016/j.ins.2019.10.008.
[175] Z. Tong, X. Deng, F. Ye, S. Basodi, X. Xiao, and Y. Pan, "Adaptive computation offloading and resource allocation strategy in a mobile edge computing environment," *Inf. Sci. (Ny).*, vol. 537, pp. 116–131, 2020, doi: 10.1016/j.ins.2020.05.057.
[176] A. Cangelosi and M. Schlesinger, "From Babies to Robots: The Contribution of Developmental Robotics to Developmental Psychology," *Child Dev. Perspect.*, vol. 12, no. 3, pp. 183–188, 2018, doi: 10.1111/cdep.12282.
[177] E. Shelhamer, P. Mahmoudieh, M. Argus, and T. Darrell, "Loss is its own reward: Self-supervision for reinforcement learning," *5th Int. Conf. Learn. Represent. ICLR 2017 - Work. Track Proc.*, 2019.
[178] J. M. Wong, "Towards Lifelong Self-Supervision: A Deep Learning Direction for Robotics," 2016, [Online]. Available: http://arxiv.org/abs/1611.00201.
[179] B. Settles, "Active Learning Literature Survey," *Comput. Sci. Tech. Rep.*, no. 1648, 2009.
[180] P. Sarkar and W. Q. Meeker, "A Bayesian on-line change detection algorithm with process monitoring applications," *Qual. Eng.*, vol. 10, no. 3, pp. 539–549, 1998, doi: 10.1080/08982119808919168.
[181] Y. Sun, F. Gomez, and J. Schmidhuber, "Planning to be surprised: Optimal Bayesian exploration in dynamic environments," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6830 LNAI, pp. 41–51, 2011, doi: 10.1007/978-3-642-22887-2_5.
[182] B. Krawczyk and A. Cano, "Online ensemble learning with abstaining classifiers for drifting and noisy data streams," *Appl. Soft Comput. J.*, vol. 68, no. September, pp. 677–692, 2018, doi: 10.1016/j.asoc.2017.12.008.
[183] Y. Gal, "Dropout as a Bayesian Approximation: Representing," vol. 48, 2016.
[184] A. Kendall and Y. Gal, "What uncertainties do we need in Bayesian deep learning for computer vision?," *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, no. Nips, pp. 5575–5585, 2017.
[185] W. Maddox, T. Garipov, P. Izmailov, D. Vetrov, and A. G. Wilson, "A Simple Baseline for Bayesian Uncertainty in Deep Learning," no. NeurIPS, pp. 1–12, 2019, [Online]. Available: http://arxiv.org/abs/1902.02476.

[186] J. Postels, F. Ferroni, H. Coskun, N. Navab, and F. Tombari, “Sampling-free epistemic uncertainty estimation using approximated variance propagation,” *Proc. IEEE Int. Conf. Comput. Vis.*, vol. 2019-Octob, pp. 2931–2940, 2019, doi: 10.1109/ICCV.2019.00302.
[187] A. G. Wilson, “The Case for Bayesian Deep Learning,” pp. 1–7, 2020, [Online]. Available: http://arxiv.org/abs/2001.10995.
[188] F. Tian, B. Gao, Q. Cui, E. Chen, and T. Y. Liu, “Learning deep representations for graph clustering,” *Proc. Natl. Conf. Artif. Intell.*, vol. 2, pp. 1293–1299, 2014.
[189] J. R. Hershey, Z. Chen, J. Le Roux, and S. Watanabe, “DEEP CLUSTERING : DISCRIMINATIVE EMBEDDINGS FOR SEGMENTATION AND SEPARATION Mitsubishi Electric Research Laboratories (MERL), Cambridge , MA 02139 , USA,” *Icassp 2016*, pp. 31–35, 2016.
[190] E. Aljalbout, V. Golkov, Y. Siddiqui, M. Strobel, and D. Cremers, “Clustering with Deep Learning: Taxonomy and New Methods,” pp. 1–12, 2018, [Online]. Available: http://arxiv.org/abs/1801.07648.
[191] B. Yang, X. Fu, N. D. Sidiropoulos, and M. Hong, “Learning and Clustering,” *Proc. 34th Int. Conf. Mach. Learn. {ICML} 2017, Sydney, NSW, Aust. 6-11 August 2017*, vol. 70, pp. 3861–3870, 2017, [Online]. Available: http://proceedings.mlr.press/v70/yang17b.html.
[192] J. Xie, U. ofWashington R. Girshick, F. A. R. (FAIR) A. Farhadi, and U. OfWashington, “Unsupervised Deep Embedding for Clustering Analysis arXiv:1511.06335v2,” *AFIPS Conf. Proc.*, vol. 50, pp. 401–405, 2016.
[193] S. Pidhorskyi, R. Almohsen, D. A. Adjeroh, and G. Doretto, “Generative probabilistic novelty detection with adversarial autoencoders,” *Adv. Neural Inf. Process. Syst.*, vol. 2018-Decem, no. Nips, pp. 6822–6833, 2018.
[194] T. Amarbayasgalan, B. Jargalsaikhan, and K. H. Ryu, “Unsupervised novelty detection using deep autoencoders with density based clustering,” *Appl. Sci.*, vol. 8, no. 9, 2018, doi: 10.3390/app8091468.
[195] E. Principi, F. Vesperini, S. Squartini, and F. Piazza, “Acoustic novelty detection with adversarial autoencoders,” *Proc. Int. Jt. Conf. Neural Networks*, vol. 2017-May, pp. 3324–3330, 2017, doi: 10.1109/IJCNN.2017.7966273.
[196] M. Kliger and S. Fleishman, “Novelty Detection with GAN,” 2018, [Online]. Available: http://arxiv.org/abs/1802.10560.
[197] C. Zhou and R. C. Paffenroth, “Anomaly detection with robust deep autoencoders,” *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, vol. Part F1296, pp. 665–674, 2017, doi: 10.1145/3097983.3098052.
[198] M. A. A. Dewan, E. Granger, G. L. Marcialis, R. Sabourin, and F. Roli, “Adaptive appearance model tracking for still-to-video face recognition,” *Pattern Recognit.*, vol. 49, pp. 129–151, 2016, doi: 10.1016/j.patcog.2015.08.002.
[199] S. Zhang, J. Wang, J. Tong, J. Zhang, and M. Zhang, “Cloud-Edge Fusion Based Abnormal Object Detection of Power Transmission Lines Using Incremental Learning,” *IEEE Access*, vol. 8, pp. 218694–218701, 2020, doi: 10.1109/ACCESS.2020.3037172.
[200] S.-H. Kim, C. Lee, and C.-H. Youn, “An Accelerated Edge Cloud System for Energy Data Stream Processing Based on Adaptive Incremental Deep Learning Scheme,” *IEEE Access*, vol. 8, pp. 195341–195358, 2020, doi: 10.1109/access.2020.3033771.
[201] D. Nallaperuma *et al.*, “Online Incremental Machine Learning Platform for Big Data-Driven Smart Traffic Management,” *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 12, pp. 4679–4690, 2019, doi: 10.1109/TITS.2019.2924883.
[202] W. H. Kruskal and W. A. Wallis, “Use of Ranks in One-Criterion Variance Analysis,”

J. Am. Stat. Assoc., vol. 47, no. 260, p. 583, Dec. 1952, doi: 10.2307/2280779.
[203] A. M. Mood, "On the Asymptotic Efficiency of Certain Nonparametric Two-Sample Tests," *Ann. Math. Stat.*, vol. 25, no. 3, pp. 514–522, Sep. 1954, doi: 10.1214/aoms/1177728719.
[204] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *Ann. Math. Stat.*, vol. 18, no. 1, pp. 50–60, Mar. 1947, doi: 10.1214/aoms/1177730491.
[205] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, Jun. 2002, doi: 10.1613/jair.953.
[206] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.*, pp. 1–15, 2015.
[207] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," *6th Int. Conf. Learn. Represent. ICLR 2018 - Conf. Track Proc.*, pp. 1–11, 2018.
[208] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113, pp. 54–71, May 2019, doi: 10.1016/j.neunet.2019.01.012.
[209] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN: An Efficient Training and Inference Engine for Intelligent Mobile Cloud Computing Services," *IEEE Trans. Mob. Comput.*, vol. 20, no. 2, pp. 565–576, 2021, doi: 10.1109/TMC.2019.2947893.