CONVOLUTIONAL ENSEMBLE LEARNING FOR EDGE INTELLIGENCE

by

İlkay Sıkdokur

B.S., Mathematics Engineering, Istanbul Technical University, 2018

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computational Science and Engineering

Boğaziçi University

2023

ACKNOWLEDGEMENTS

First and foremost, I am utterly grateful to my supervisors, Prof. Arda Yurdakul and Assist. Prof. İnci Meliha Baytaş for their unceasing support, precious advice, and patience during my MSc study. In addition, I would like to thank the members of the jury of my thesis, Prof. Necati Aras, Assoc. Prof. Özlem Durmaz İncel and Prof. İlker Hamzaoğlu for their precious time, detailed review and valuable suggestions for the improvement of my thesis. I would also like to thank my managers, Anıl and Celal, colleagues, and friends for their support and belief during my study. Finally, I would like to express my gratitude to my dear parents, brother Eray, and lovely wife Seçil. Without their tremendous understanding and encouragement over the past few years, it would be impossible for me to complete my study.

ABSTRACT

CONVOLUTIONAL ENSEMBLE LEARNING FOR EDGE INTELLIGENCE

Deep Edge Intelligence targets the deployment of deep learning algorithms in the edge network. While training deep networks requires computational resources, edge devices frequently lack high computational power. Decentralized learning methods such as federated learning provide a solution for gathering limited information from edge devices and collectively improving prediction performance. However, a drawback of such methods is that they often require multiple rounds of network communication, which increases communication time and the risk of communication errors. Another drawback is that the same model architecture is often used on all edge devices, which makes it mandatory to work with devices above a level of computational capacity.

This thesis proposes a hybrid learning approach that employs ensemble learning with a convolutional scheme for different edge model architectures, except for a selected fully connected layer of the same dimensionality. Initially, shallow neural networks are trained on edge devices until a certain level of performance is achieved. Next, the feature representations obtained by the shallow models are transferred to an ensemble model. Subsequently, the proposed convolutional ensemble model is trained to boost the prediction performance. This method facilitates the completion of the system training with a one-way data transfer between edge devices and the server. Variational auto-encoders are also utilized to generate feature vectors in case transferring the required representations from the edge devices fails. Extensive experiments demonstrate that the suggested method outperforms state-of-the-art techniques in terms of accuracy while requiring fewer communications and a lower amount of data in various training scenarios.

ÖZET

KENAR ZEKA İÇİN EVRİŞİMSEL TOPLULUK ÖĞRENMESİ

Derin Kenar Zeka, derin öğrenme algoritmalarının kenar bilişimde kullanımını amaçlar. Derin ağ eğitimi, güçlü hesaplama kaynakları gerektirirken kenar cihazlar genelde bu kapasitede değildir. Federe öğrenme gibi dağıtık öğrenme yöntemleri, kenar cihazlardan sınırlı bilgiyi alarak işbirliği ile tahmin başarımını arttırmayı hedefler. Genellikle ağ iletişim seferlerinin fazlalığından dolayı iletişim zamanı artmaktadır. Aynı zamanda iletişimde olası hataların artmasına da yol açmaktadır. Bir diğer dezavantaj ise tüm uç cihazlarda genelde aynı model mimarisinin kullanılmasından kenarda belli bir hesaplama kapasitesinin üstündeki cihazların kullanımına izin verilmesidir.

Bu tezde, aynı boyutlu seçilmiş bir tam bağlı katman hariç tamamen farklı model mimarilerini destekleyen ve evrişimsel özelliğe sahip melez bir topluluk öğrenmesi çerçevesi önerilmektedir. Öncelikle, sığ sinir ağları kenar cihazlarda belli bir performansa kadar eğitilir. Kenar cihazlardaki her ağda, belli boyutta bir tam bağlı katman bulunması esastır. Bu katmana ait özellikler, modellerin eğitiminden sonra sunucudaki topluluk modeline giriş olarak aktarılır. Önerilen evrişimsel topluluk modeli sistemin kestirim başarımını arttırmak için eğitilir. Bu yöntem, sistemin öğrenmesinin sadece cihazdan sunucuya tek-yönlü iletişim ile yapılmasını sağlamaktadır. Gerekli öznitelik temsillerinin kenar cihazlardan sunucuya iletilememesi halinde, sunucudaki varyasyonel otomatik kodlayıcılar istatistiksel bir şekilde uygun vektörleri topluluk modeline sunmaktadır. Kapsamlı deneyler, önerilen yöntemin kestirim başarımının güncel yöntemlerden üstünlüğünü göstermektedir. Ayrıca, bazı öğrenme senaryolarında daha az iletişim ve veri aktarımı sağlanmaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

BS	Minibatch size
Bit_{com}	The bit-size of the feature vector transferred from edge to server
c	Total number of classes
C	Number of channels
D_{train}^{Ens}	The training data built for the ensemble model
D_{train}	The full train data
$D_{train}^{EM_i}$	The train data available for the i^{th} edge model
EM_i	The i^{th} edge model
Ens	The ensemble model on the server
Ep^{EM}	The training epoch for an edge model
Ep_d^{Ens}	How many epochs a transferred minibatch is to be used in repeating training on the ensemble model (for Scenario 2)
Ep^{Ens}	How many epochs the ensemble model is to be trained (for Scenario 3)
Ep^{VAE}	The training epoch of the VAE model
$F_{inference}^{EM_i}$	The feature maps out of the FCLs with the same size from the inference of the inference data of the i^{th} edge model
$F_{train}^{EM_i}$	The feature maps out of the FCLs with the same size from the inference of the training data of the i^{th} edge model
H	Height of an image
K_{test_i}	The indices of the images whose class label is i in the whole test data
K_{test}	The indices of the images in the whole test data
K_{train_i}	The indices of the images whose class label is i in the whole train data
K_{train}	The indices of the images in the whole train data

$K_{train}^{EM_i}$	The indices of the images in the train data of the i^{th} edge model
KH	Height of a kernel filter
KL	Kullback-Leibler divergence
KW	Width of a kernel filter
L	Length of a fully connected layer
L_{com}	The dimensionality of the fully connected layers with the same size of edge models
M_{memory}^{EM}	Memory requirement on an edge device (bit)
M_{memory}^{server}	Memory requirement on the server (bit)
$M_{transfer}^{cum}$	Cumulative transferred data size combined all edge models (bit)
$M_{transfer}^{EM}$	Total transferred data size from an edge model in a server communication (bit)
$MaxDisc\%$	Maximum discrepancy percentage threshold for selecting the subsets of the whole test data
$Min\%$	Minimum percentage threshold for selecting the subsets of the whole training data
N	The number of edge models
$\mathcal{N}$	Normal distribution
$\mathcal{U}$	Uniform random distribution
P_{train}^{EM}	The percentage of the whole training data used for an edge model
$\mathbb{R}^d$	The domain of the latent space of VAE models in real numbers with dimensionality d
$R_{transfer}^{EM}$	Data transfer rate from an edge model to the server (Mbps)
S_m^{EM}	The number of samples in a minibatch for an edge model
S_m^{Ens}	The number of samples in a minibatch for the ensemble model
S_{train}	The number of samples in whole training data
$T_{inference_m}^{EM}$	The inference time of a minibatch for an edge model (sec)

$T_{inference}^{EM}$	The inference time of an edge model (sec)
$T_{inference}^{VAE}$	The average inference time of the VAE model associated with the edge models for whole training data (sec)
$T_{train_m}^{EM}$	The training time of a minibatch for an edge model (sec)
T_{train}^{EM}	The training time of an edge model (sec)
$T_{train}^{Ens_{1Ep}}$	Training time of the ensemble model for an epoch (sec)
T_{train}^{Ens}	Total training time of the ensemble model (sec)
$T_{train}^{VAE_{all}}$	The training time of a VAE model with whole training data for one epoch (sec)
$T_{train}^{VAE_{EM}}$	The training time of the VAE model associated with an edge model (sec)
$T_{transfer}^{EM}$	Total transfer time of data from an edge model to server (sec)
VAE_i	The i^{th} VAE model
$VAE_i^{dec}(z)$	The decoding obtained from the i^{th} VAE model for a random vector z from latent space
W	Width of an image
y_i	Ground truth label for the i^{th} class
$\hat{y}_i$	Predicted value for the i^{th} class
μ_{enc}	Mean term of the encoding
σ_{enc}	Standard deviation term of the encoding

LIST OF ACRONYMS/ABBREVIATIONS

BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
DNN	Deep Neural Network
DSP	Digital Signal Processor
FC	Fully Connected
FCL	Fully Connected Layer
FF	Flip-Flop
FIFO	First-In-First-Out
FL	Federated Learning
FPGA	Field-Programmable Gate Array
HLS	High-Level Synthesis
i.i.d.	Independent and Identically Distributed
IO	Input/Output
IoT	Internet of Things
KL	Kullback-Leibler
LUT	Lookup Table
MB	Megabytes
Mbps	Megabits Per Second
ms	Millisecond
sec	Second
t-SNE	t-Distributed Stochastic Neighbour Embedding
VAE	Variational Auto-Encoder
WAN	Wide Area Network

1. INTRODUCTION

Edge computing has become an integral component of next-generation solutions for various industries such as automotive [1], oil and gas [2], agriculture [3], and cyber security [4]. In particular, edge intelligence offers innovative frameworks for Internet of Things (IoT) applications. For example, sensors are predominantly used to collect data for safety applications for automobiles [1], effective business decisions [2], and early detection of crop and animal diseases [3]. Therefore, developing efficient learning techniques for edge intelligence has turned out to be inevitable [5]. Deep neural networks are also preferred for edge intelligence applications due to their accurate results [4]. Deep Edge Intelligence is a type of edge intelligence where Deep Neural Networks (DNN) are harnessed in the edge network. An illustration of the deep edge intelligence application covered in this study is presented in Figure 1.1. In the figure, the devices on end nodes are called *edge devices*, and the models deployed on the devices are called *edge models*. The same naming convention is used throughout this thesis.

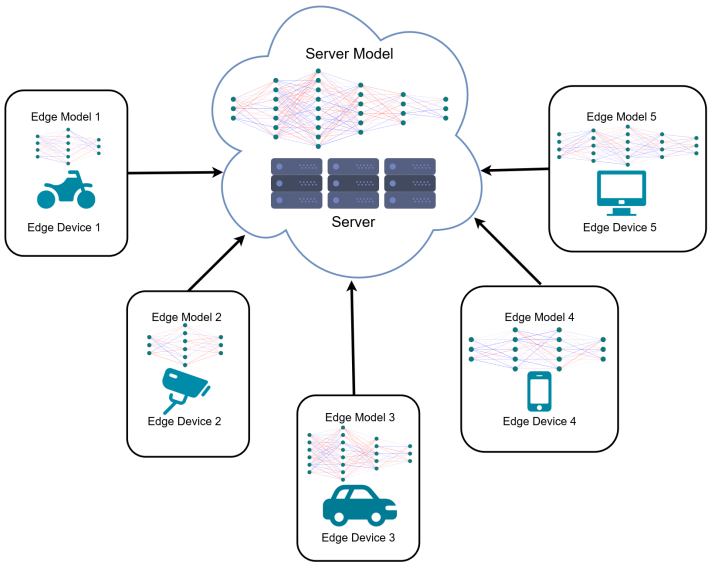


Figure 1.1. Deep edge intelligence setting covered in this study.

DNNs are predominantly employed in many data-driven applications, such as image classification [6–8], speech recognition [9–11], and natural language processing [12–14], due to their ability to attain state-of-the-art performances. The training and deployment of DNNs require extensive time [15,16], memory [17,18], and energy [19,20] consumption. Using DNNs in IoT applications leads to a bottleneck since the edge devices often have limited capacity [21, 22].

Table 1.1. Number of parameters and accuracy values of example models on CIFAR-10.

Model	**#Parameters**	**Accuracy (%)**
CCT-6 [23]	3M	95.3
EfficientNet [24]	3M	98.1
DenseNet [25]	5M	96.5
VGG-19 [26]	20M	95.1
ResNet-50 [27]	25M	98.3
Edge Model Type 1 (this work)	**7K**	**36.7**

The deep models that produce state-of-the-art accuracy results often consist of millions of parameters [23–27] for a centralized setting for the CIFAR-10 [28] dataset. Table 1.1 shows that the state-of-the-art models contain a large number of parameters for acceptable accuracy, but they are too big to be deployed in resource-poor edge devices. The last row of the table shows an example model used in this study, detailed in Table 5.2. It can be seen that acceptable accuracy values can be obtained by using models with an excessive number of parameters, but shallow models do not produce accuracy values at that level. Besides, in edge intelligence applications, the data are usually distributed. Gathering all the raw data to a central location takes a considerable amount of time along with data privacy concerns [29].

Designing new methods for deploying and accelerating deep models in resource-constrained edge devices is crucial for edge intelligence. The common goal is to develop models that can fit edge devices without sacrificing accuracy. A state-of-the-art solution to this problem is the efficient implementation of inference algorithms. This approach requires offline training followed by techniques such as model pruning [30, 31], model compression [32, 33] and quantization [34, 35] before deploying on edge devices. On the other hand, proposals exist where resource-constrained devices can contribute to the learning process [36, 37].

Federated learning is one of the popular decentralized learning methods used for training models of IoT solutions [38, 39]. However, there are several drawbacks of federated learning in applications on edge devices. For example, communication bottlenecks caused by frequent parameter updates lead to increasing data transfer. Another example is the same edge model architecture has to be used on all edge devices [38].

Edge intelligence methods may face disconnection problems during data transfer [40]. Such applications are mainly built on the data acquired by edge nodes. In real-world cases, every edge node may not acquire the data with the same occurrence or transfer all of the acquired data to the central server due to connection problems. As edge intelligence models rely on data acquired by edge devices, this problem severely affects the training and inference of edge intelligence applications.

This thesis proposes a convolutional ensemble learning method for edge intelligence applications. This method aims to fuse the information sent from the edge devices on the server to reach a final decision with boosted accuracy. The information sent from the edge devices is not a product of a feature engineering process, as the proposed method directly learns from the data, whereas feature engineering applications require domain knowledge. The standpoint of the proposed method differs from federated learning on approaching the edge intelligence problems. Federated learning targets to enhance the model that every edge device deploys, which makes it focused on developing

personalized models for the devices. However, the proposed method is focused on reaching a global and accurate decision based on the experience of each edge model. The goal is not to enhance the models on edge devices. When the edge devices lack abundant hardware resources, deployed models are not generally deep enough to result in state-of-the-art accuracy. The proposed method is devised as a solution to this problem. It does not require long training phases for edge models since long training phases do not improve the accuracy of shallow models due to their small capacity. The proposed method targets improving the prediction accuracy of the model on the server rather than the models on edge devices.

In the real world, the proposed method might be useful for problems in which local decisions are not quite important, but the local ideas and the server makes the ultimate decision based on all local ideas. For example, cameras located at different points on the highway might not be quite strong devices, and they might only deploy shallow models. All the cameras may send their local ideas about the traffic on the highway, and the server may decide whether there is an incident or traffic congestion based on the local ideas.

In this thesis, three major contributions are proposed to tackle these problems:

- The models on IoT end nodes are implemented on Field-Programmable Gate Array (FPGA) devices in this study. A flexible acceleration method is proposed for training the edge models on FPGAs. Matrix multiplications are accelerated by partial parallelism. The system is built on a task-level streaming approach with pipelining to increase acceleration and decrease resource utilisation. This approach allows limited memory storage.
- Since edge devices lack abundant computational resources, they are expected to train shallow neural networks. A convolutional ensemble learning scheme is proposed for boosting the prediction performance of shallow models trained on heterogeneous edge models. As opposed to standard federated learning frameworks, weak models do not receive global updates from the server. The architecture

of the ensemble model is a Convolutional Neural Network (CNN) that enables learning to fuse the underlying information in the intermediate layers.

- Variational Auto-Encoder (VAE) models are trained on the server for all edge models. VAE models learn to generate feature vectors which are statistically similar to feature vectors of the edge models. These models aim to generate feature vectors on the server in case the transfer of required feature representations from the edge devices fails.

As experiments point out, the proposed task-level streaming method for training deep networks decreases the training time while fitting into the target FPGA device. Moreover, feature generation via VAE models boosts the ensemble model accuracy compared to basic filling methods for missing values. In addition, the proposed method tops numerous federated learning and basic ensemble methods in accuracy for different benchmark datasets. Also, different IoT setups bring different efficiencies in communication and memory usage with certain trade-offs.

The thesis is organized as follows. Selected previous works in edge intelligence, FPGA acceleration in neural networks, federated learning and ensemble learning are analyzed in the next chapter. Chapter 3 presents details of the proposed design and its components. Chapter 4 elaborates on three training scenarios with different IoT setups for the proposed design. Chapter 5 contains the experimental results, their interpretations and comparative analysis of earlier works. The last chapter concludes the thesis with a review of the achievements of the study and future research ideas.

2. RELATED WORKS

In this chapter, related works in literature are examined and compared with this study. In Section 2.1, this study and recent edge intelligence works are categorized based on data offloading. In Section 2.2, the details of FPGA deployment in deep neural networks are compared with recent works. In Section 2.3, recent federated learning works are examined and the differences with this study are exhibited. A similar investigation is conducted for recent ensemble learning works in Section 2.4.

2.1. Edge Intelligence

Edge intelligence applications could be designed in several ways. For example, training and inference of the model can ultimately be in a cloud server, or the model may be trained in a cloud server, and inference may be made on edge. In [41], a data offloading scale of seven levels based on the design choice of edge intelligence applications is proposed. The summarized descriptions of the levels are given as follows:

(1) All in cloud: Training and inference of the model are completely made on the cloud server.
(2) Training in cloud and co-inference in cloud-edge: Training of the models is done on the cloud server, and inference is made in collaboration with edge devices and the cloud server by data offloading between the cloud and edge.
(3) Training in cloud and co-inference in edge: Training of the models is done on the cloud server, and inference is made in the network of edge devices by data offloading between edge devices.
(4) Training in cloud and inference on device: Training of the models is done on the cloud server, and inference is made in the edge devices. There exists no data offloading between the edge devices.
(5) Cloud-edge co-training and co-inference: Training and inference are collaboratively made by the edge and cloud devices.

(6) All in edge: Inference and training are conducted within the network of edge devices.

(7) All on device: Inference and training are conducted on individual devices.

Figure 2.1. Edge intelligence levels by data offloading [41].

The level hierarchy by data offloading is illustrated in Figure 2.1. The volume and path length of data unloading decrease as the level increases. The edge intelligence solution proposed in this work can be placed between Level 6 and Level 7 because the models on edge are independently trained, and their inference is also independently done on individual edge devices. However, the inference data obtained from the edge devices are required on the server device for making the final prediction. According to the study in [41], data offloading quantity and path length decrease with level. This also causes less transmission latency of data offloading, higher data privacy, and lower communication bandwidth.

Edge offloading is also categorized as device-to-cloud (D2C), device-to-edge (D2E), device-to-device (D2D) and hybrid in [42]. The data offloading process of the proposed

work can be categorized as D2E with the exception that edge models do not receive any information back from the server for training and inference. Moreover, the training of the edge models is done on the device. Works like [43, 44] are categorized in the D2E category in [42]. However, the proposed method in this thesis differs from [43, 44] in several perspectives. For example, the method in [43] trains a Branchynet [45] model, and the network is optimally partitioned into two parts yielding the least latency between server and edge devices by data exchange between both ends. The inference starts on the server, and the intermediate features are transferred to the edge to continue the inference. It can be seen that the method has a two-way data offload between the server and the edge. The method given in [44] divides a deep neural network into two halves: one part is allocated on the edge device, and the server is assigned to the other. For training and inference of those two works, a data offload between device and server is required. Such a data offload increases the number of communication and data transfer times. The suggested approach in this thesis does edge model training on device so data offload decreases between device and server.

2.2. Field-Programmable Gate Array

In this study, the edge devices are chosen as Field-Programmable Gate Array (FPGA) devices in which the training of shallow models is implemented and accelerated. Recently, FPGA devices are in use in edge intelligence applications [46–48]. The studies assert that FPGA devices are helpful for edge computing and superior to GPU devices in some cases because FPGA devices are insensitive to workload for throughput adjustment, can be adapted to spatial and temporal parallelism at the fine granularity and offer improved energy efficiency and thermal stability. The studies argue that GPU and FPGA devices can be complementarily utilized for edge computations [49, 50]. FPGA and GPU units are frequently used in heterogeneous systems, not only in edge intelligence contexts but also in some other computational task contexts for quite a while for improved energy efficiency and acceleration rate [51–54]. The proposed method is implemented on an FPGA-GPU hybrid system in this thesis. In addition, a partial acceleration method is proposed in this thesis. The proposed acceleration method of

training neural networks on FPGA supports the assertions of the suitability of FPGA devices for improved parallelism in calculations.

Deep neural network acceleration on FPGA is a crucial area of research. Current works in hardware acceleration of neural networks offer various techniques and designs for implementing trained models on FPGA. In [55], the weight matrices are reduced into systolic arrays by pruning various nodes of weight arrays. Hence, they achieve a faster inference in fewer cycles. In [56], matrix multiplier blocks of systolic arrays are added on FPGA to accelerate the process of machine learning applications. They obtain almost four times faster results than reference design. Accelerating the training process is a crucial research topic because training deep neural networks often require much execution time. Binarized Neural Network speeds up training and inference which consist of only -1 and 1 values for weights [57]. In Quantized Neural Networks, weights are limited in an interval for better accuracy [58]. Another training acceleration method is proposed in [59] through the use of smaller data types for the weights. However, these methods accelerate the training while decreasing accuracy due to low numerical precision. In this work, training acceleration is not done by quantising or decreasing the size of the data types used. Therefore, an accuracy-acceleration trade-off does not exist. The training is accelerated by parallelizing the matrix multiplications with adjustable factors using a task-level streaming approach.

In the proposed method of this thesis, changing the acceleration factors affects resource utilization. Thus, resource management can be achieved by adjusting the acceleration factors to reach the desired resource utilization. This approach also enables applicability to various target devices with different hardware resources to maximize the acceleration within the resource limits of the devices. Numerous studies propose methods for matrix multiplication acceleration in deep neural networks. For example, the training of CNN is accelerated by accelerating matrix multiplication of fully connected layer calculations in [60]. The acceleration is done by partitioning the arrays into 4×4 pieces and multiplying the pieces in parallel. In [61], the convolutional operation is accelerated by flattening three-dimensional arrays into two-dimensional arrays and partitioning the

flattened arrays by given factors to make parallel convolution calculations. The method given in [62] accelerates training by doing the calculations in parallel for elements of the minibatch. These works focus on the parallelization of the calculations by certain levels only, such as fully connected layer calculations [60], convolution layer calculations [61], and parallelization on minibatch level [62]. The proposed method in this study parallelizes the calculation of all layer operations by chosen array dimensions, including minibatch level, channel and filter level for convolutional operations, and chosen partition level for fully connected layer operations.

2.3. Federated Learning

Decentralized learning methods on edge are widely studied. The importance of this research area increases as IoT solutions on edge become widespread. Federated learning (FL) is the mainstream decentralized learning method for IoT applications. There are numerous federated learning-based decentralized learning methods to tackle the problem of training on edge. A recent survey on FL [38] lists several methods that differ in several aspects for more efficient learning. However, the main problem that most of the methods try to solve is to reduce communication rounds caused by the frequent transfer of model weights. Strategies that include compression and restricting learning space are investigated to alleviate the communication bottleneck in FL [32, 63]. The second drawback of most FL algorithms is that all clients and the central server try to learn the same model architecture. This property dictates a minimum resource requirement for each edge device for training the model. Otherwise, federated learning cannot boost the overall accuracy when the distributed models are shallow enough to fit in the devices with limited resources. Recent federated learning algorithms rely on transferring the models with the same architecture back and forth from the edge to the server [64–70]. The communication bottleneck caused by frequent model parameter exchange may be decreased by lowering the exchange frequency; nevertheless, decreasing the model parameter exchange usually results in lower accuracy, as stated in these studies.

The proposed method in this thesis relies on ensemble learning which discards the exchange of model parameters that cause a communication bottleneck. There is only a one-way transfer from edge devices to the server. This notion frees each edge device to deploy any model suitable for its resources, provided that they can yield the exact size of the input feature map to the output layer. Thanks to the ensemble learner at the server device, the proposed method in this study boosts the overall system performance using edge models. It should be noted that this method does not aim to improve the accuracy of the individual edge models as opposed to federated learning. The method seeks to produce a final accurate prediction. The results of the experiments demonstrate that this method also tops cutting-edge federated learning techniques in terms of prediction accuracy.

The research of developing FL algorithms working with heterogeneous models on edge with fewer communication rounds is also crucial. For example, a system is proposed where the edge devices share the class-based mean of their embedding vectors to the server in [71]. The server then aggregates all class-based mean embedding vectors and sends them back to the edge devices for further edge training. It can be seen that model heterogeneity is preserved as done in this thesis; however, the system proposed in this thesis does not send any information back to the edge devices. In [72], the heterogeneous model weights and biases are aggregated on the server in a one-way transfer. There is no information transfer back to the edge models from the server. However, the reported accuracy values are remarkably less than in this thesis. In [73], the logit output distributions are transferred to the server device to calculate confidence weights for the heterogeneous edge models. It differs from the proposed method in this thesis by constantly sending the outputs of the edge models. In this thesis, arbitrarily selected intermediate outputs of the edge models are sent to the server as long as the size of the outputs is the same for all edge models. The number of layers before or after the chosen layers is irrelevant to the proposed method in this thesis. In [74], both feature maps and logit outputs are transferred from the edge to the server. Then the model on the server transfers a logit output back to the edge using the data transferred from the edge. Their method does not require using the same model architecture;

nevertheless, it requires multiple communication rounds, whereas the proposed method in this thesis works with only a one-way network communication.

2.4. Ensemble Learning

Ensemble methods are broadly used in deep learning studies. In [75], ensemble methods leverage the training of CNN models to predict Li-ion battery capacity. Measurements like the voltage, current and charge capacity are formed to be used in several CNN models, and their outputs are averaged to reach the final prediction. Ensemble methods are also used for dynamic multi-objective evolutionary algorithms [76]. In the given work, two base models initially build the ensemble model. The number of base models and layers of each base model iteratively change until an optimal accuracy is reached. The ensemble operation is again built by averaging the output of base models. Recent studies also use ensemble learning for IoT applications [77, 78]. The method proposed in [77] uses an implementation of Dynamic AdaBoost.NC [79] method to train models for machine anomaly detection in manufacturing sites. In [78], a method is proposed where packet transmissions are received from devices connected to a network and four individual predictions are obtained from four different ensemble models: boosted tree [80], bagged tree [81], subspace discriminant [82], and RUS boosted tree [83] to make network attack prediction by voting. Although these works use ensemble methods for edge intelligence applications, they use basic ensemble methods such as bagging [84] and boosting [85]. In addition, recent works in edge intelligence exist using ensemble methods with knowledge distillation [86] approaches. Recent studies utilize an ensemble of the logit outputs and gradients of larger models to train smaller models [87, 88].

This thesis presents a convolution-based ensemble method to increase the system's overall accuracy. The method studied in this thesis proposes a feature fusion method by convolution operation over the feature vectors. Experiments demonstrate that the suggested method tops fundamental ensemble methods and knowledge-distillation-based methods.

3. INTELLIGENT EDGE DESIGN

The proposed system consists of a server and edge devices with various hardware capacities. Edge devices are assumed to have medium computational power. Therefore, in this thesis, shallow neural network models, named *edge models*, are trained for image classification and regression tasks on edge devices, and their performance is poor. The server's objective is to boost overall prediction performance. In the proposed system, any computational hardware restrictions are not imposed on the server. Thus, a complex model can be trained on the server. The server leverages ensemble learning where the server model, named the *ensemble model*, collects the embeddings learned by the edge models and learns how to fuse them. The learned ensemble model makes the final predictions; thus, the server does not broadcast the ensemble model back to the edge devices.

The inference workflow of the system can be summarized in four steps as shown in Figure 3.1 for the image classification task. The study describes the methods based on the image classification task; nonetheless, all of the steps are the same for the regression task as well. Since the inputs and the outputs are in different forms, different loss functions are used in edge models for both tasks. However, it does not affect the way of working of the proposed methods.

Firstly, the edge models take images as inputs and output their embeddings in inference. A fully connected layer with the dimensionality L_{com} appears in all edge models. After the edge inference, feature vectors obtained from the L_{com}-size fully connected layer are transferred to the server. Some edge models might not deliver their feature vectors for several reasons, such as data transfer errors or not getting input images to create feature vectors. Such cases are handled by filling in the missing information using a generative approach based on Variational Auto-Encoder (VAE). Then the ensemble model takes those feature vectors as input and produces the final prediction.

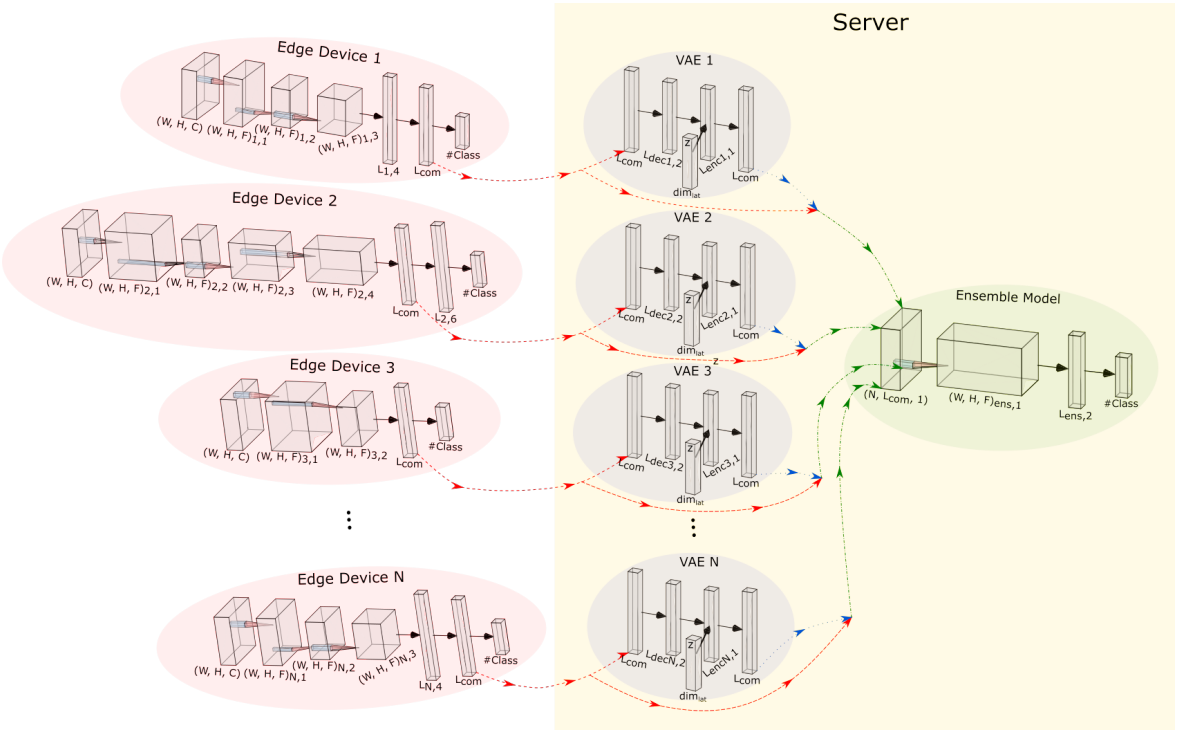


Figure 3.1. Illustration of the system for the image classification task. Red dashed lines denote the successful transfer of edge features. Missing features are filled by VAE outputs (blue dashed lines). Green dashed lines represent the complete feature set to train the CNN-based ensemble model.

The training of the overall system also follows similar steps. Initially, the edge models are independently trained with the training data that are available for them. Training data for each edge model are assumed as a subset of the training data of the classification problem consisting of independent and identically distributed (i.i.d.) samples. Training sets on edge devices might have both overlapping and different images from a training dataset. After the training of the edge models, they produce the corresponding feature vectors using their training data and transfer them to the server. On the server, a Variational Auto-Encoder (VAE) model is deployed for each edge model for feature vector generation. The VAE models are trained via the successfully transferred feature vectors obtained from the corresponding edge devices. After training VAE models, feature vectors are generated for missing vectors from the edge devices by corresponding VAE models. After the generation of feature vectors, the training data

of the ensemble model are prepared by taking either the original feature vector or the generated feature vector for all images in the full training data. The original feature vector for an image is taken if it is successfully transferred from the edge device to the server. Otherwise, the generated feature vector is taken for that image. Finally, the ensemble model is trained via that ensemble training data.

The system components are presented in the rest of this chapter. The next section presents the details of the training of edge models and their FPGA implementation in the system. In Section 3.2, the training of VAE models is described. The last section details the training of the convolutional ensemble model.

3.1. Edge Models

The edge models need not share the same model structure. However, the designer of the system has to ensure every edge model has a fully connected layer with the same dimension, L_{com} as shown in Figure 3.1. As can be seen, each edge model may have a varied number of layers. The system can be built using any edge model that fits in the edge device if all have a fully connected layer of L_{com} size.

The training of the edge models is done independently from each other and the ensemble model. Let the training data set be represented with D_{train}. The edge model EM_i is trained with a subset of D_{train}, that is named $D_{train}^{EM_i}$. After the edge models are independently trained, $D_{train}^{EM_i}$ is reused in the forward pass of the edge model EM_i to obtain the required outputs from the layer with size L_{com}. Let $F_{train}^{EM_i}$ denote feature outputs of the edge model EM_i.

The feature vectors obtained as $F_{train}^{EM_i}$ are transferred to the server device. The transfer is conducted in parallel for all edge models if the server device has a connection port for all edge devices. Otherwise, the transfer is sequentially done for some of the edge devices, which is bound to the number of available ports of the server. The transfer process is illustrated with red lines in Figure 3.1.

The transferred feature vectors, $F_{train}^{EM_i}$, are aimed to be used in the ensemble model deployed on the server device. The ensemble model expects to receive a feature vector from each edge model to realize the ensemble operation for all edge models. However, there may be several problems in acquiring a feature vector from an edge model in real-world applications, such as connection problems from edge models to the server and data accessibility of edge models. Hence, every edge model may not be able to produce a feature vector to transfer to the server for the same input data. More specifically, for the training phase of the system, $n(F_{train}^{EM_i}) \neq n(F_{train}^{EM_j})$ for some $i, j \in [1, N] \subset \mathbb{N}$ where the system is built with N edge models and $n(A)$ denotes the cardinality of a set A.

Edge intelligence applications comprise numerous devices on the system's edge side connected to a server-side device or a group of devices. In real-world applications, edge-side devices are generally not abundant in computational resources. The lack of computational resources causes a long training time for models. Therefore, accelerating the model training on edge-side devices is crucial for effective edge intelligence solutions. In this work, training of edge models is implemented on Xilinx FPGAs using high-level synthesis (HLS) [89].

Task-level parallelism is a method in which the computations of sequential modules can be concurrently executed in parallel processes. As a lucrative computational acceleration method, task-level parallelism brings its load to the hardware resources. While reaching maximum acceleration, the task-level parallelism may exceed the capacity of the edge devices. Thus, devising novel methods for handling the hardware load of task-level parallelism for computational acceleration is an important research topic. In this work, a partial task-level parallelism approach for the acceleration of matrix multiplication is proposed. With this approach, the level of acceleration can be flexibly chosen to make sure that targeted hardware utilization is met.

The proposed method transfers data using data stream methodology [90]. In data stream methodology, the data flow from the source to the destination module via FIFO (First-In-First-Out) buffers. FIFO buffers allow the consumer module to use the

data when the source module starts to send the data. That provides acceleration for the calculations. In addition, the data interface is chosen as AXI4-Stream [91] for all modules. AXI4-Stream interface allows the input data to be consumed as it streams until it is completely read.

Another paradigm that is used in the proposed method is pipelining. Pipelining is a processing paradigm in which parts of the computation are executed in parallel. One part may start without waiting for the end of the execution of the part before. The proposed method calculates matrix multiplication using a streaming data flow with pipelined task-level parallelism.

An example of a program flow that may take leverage from the proposed method can be seen in Figure 3.2. In the example, the matrix I_1 is the program's input, whereas O_1 and O_2 matrices are the program's outputs. Input and output matrices are read and written through the AXI4-Stream interface. In the example program, T_1 is obtained as the output of func1. Assume that there exists a function called func2, which uses the matrix T_1 twice. The process of func2 can be achieved in two parallel processes, namely func2_1 and func2_2 functions, on the process of the matrix T_1. However, the streaming approach requires the matrices to be read and written once. To comply with the requirement, the matrix T_1 needs to be duplicated, and both parts are used in two parallel processes separately. T_1 matrix is duplicated as T_{1_1} and T_{1_2} by duplicate function, and they are used as inputs to func2_1 and func2_2 functions, respectively. The output matrices T_{2_1} and T_{2_2} are combined into the matrix T_2, which is the input to the func3 function. Task-level parallelism can be applied to func2_1 and func2_2 functions. In this case, the calculations of those functions are concurrently done, and all matrices are stored in FIFO buffers. For example, when pipelining is applied, func2_1 and func2_2 functions may start their processes without waiting for the process of the duplicate function to be completed. In general, the next module can start its process as soon as the current module starts to write the outputs to the input of the next module.

Input: Matrix I_1

Output: Matrices O_1, O_2

#pragma HLS INTERFACE axis port=I_1

#pragma HLS INTERFACE axis port=O_1

#pragma HLS INTERFACE axis port=O_2

#pragma HLS STREAM variable=T_1

#pragma HLS STREAM variable=T_{1_1}

#pragma HLS STREAM variable=T_{1_2}

#pragma HLS STREAM variable=T_{2_1}

#pragma HLS STREAM variable=T_{2_2}

#pragma HLS STREAM variable=T_2

#pragma HLS DATAFLOW

func1(I_1, T_1)

duplicate(T_1, T_{1_1}, T_{1_2})

func2_1(T_{1_1}, T_{2_1})

func2_2(T_{1_2}, T_{2_2})

aggregate(T_{2_1}, T_{2_2}, T_2)

func3(T_2, O_1, O_2)

Figure 3.2. An example of a program that uses partial task-level parallelism on FPGA.

The comparison of the execution paradigm between sequential and task-level streaming with pipelining approaches presented in Figure 3.2 is illustrated in Figure 3.3. In the figure, t_0 represents the initial time of the execution of processes for both sequential and task-level streaming with pipelining approaches. The process of task-level streaming with pipelining approach ends at time t_1 whereas the process of sequential approach ends at t_2.

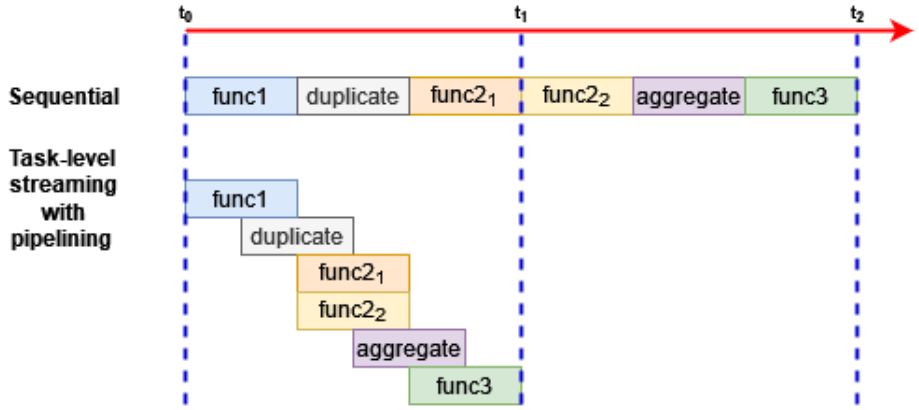


Figure 3.3. The comparison of the execution paradigm of Figure 3.2 between sequential and task-level streaming with pipelining approaches.

The parallelization and pipelining paradigms can be applied using pragma commands in Xilinx HLS [92]. For example, the AXI4-Stream interface is given using the "#pragma HLS INTERFACE axis" command. The interface ports need to be specified by adding the "port=" parameter as the input and output arrays. In addition, arrays can be streamed through the process flow as implemented in FIFO buffers by using the "#pragma HLS STREAM" command. The streamed array can be specified by adding the "variable=" parameter as the array's name. Finally, task-level pipelining can be achieved by using the "#pragma HLS DATAFLOW" command. The usage of the pragma commands is also given in Figure 3.2.

The task-level streaming method has its constraints, such as reading and writing an array element only once, in addition to the constraint of in-order access to the memory. Used in a task-level streaming module, every bit of the data is used for only one read-write operation through the execution of the module with in-order memory access. Valid and invalid task-level streaming module examples can be seen in Figure 3.4. The input and output arrays are stored in FIFO buffers in this example due to the AXI4-Stream interface pragma. Module 1 is valid because every element of I_1, I_2, and O_1 is accessed in order and only once. Module 2 is invalid because the elements of I_2 are accessed X times. Module 3 is matrix multiplication and invalid because I_1, I_3, and O_2 are accessed more than once. However, it can be transformed into Module 4 for

task-level parallelism compatibility. I_1 and I_3 are read once into local arrays which are not stored in FIFO buffers, and their multiplication is summed into a local variable *sum*. The local variable *sum* is then written into O_2 only once.

Task-level streaming provides an overall acceleration of the whole calculation without using other parallelism methods; nonetheless, the other parallelism methods need to be utilized to reach the maximum acceleration of computation. Operator-level parallelism is another parallelism method in which parallelism is used on the instruction level of operator processes. Similar to task-level parallelism, pipelining can also be used for operator-level parallelism; hence, the operations need not stall until all the operations are completed in sequential order. The acceleration of computation can be remarkably increased when task-level streaming and operator-level parallelism are applied with pipelining. However, operator-level parallelism can be applied to the arrays used in the modules of task-level streaming if and only if the arrays are completely partitioned in all dimensions. When an array's dimension is completely partitioned, the array is divided into independent memory parts by the dimension. Completely partitioning the arrays may bring the need for excessive use of hardware resources that are scarce on small edge devices. Hence, the parallelism of the computations needs to be done with a method in which all dimensions need not be completely partitioned when the data are streamed.

In the proposed method, the arrays are manually tiled in preprocessing phase by the factors that the system designer decides. Then, the arrays are completely partitioned into tiles during synthesis by the factorized part of the chosen dimension. The factors are decided as the system fits into the chosen device for the edge. Assume that the matrices $M_1[X][Y]$ and $M_2[Y][Z]$ are aimed to be multiplied into the result matrix $M_3[X][Z]$. It can be seen that the Y dimension is the common dimension of multiplication. The elements of the matrix M_3 are calculated as

$$M_{3_{i,j}} = \sum_{k=1}^{Y} M_1[i][k] * M_2[k][j]. \tag{3.1}$$

Input: Matrices $I_1[X][Y], I_2[X], I_3[Y][Z]$

Output: Matrices $O_1[X][Y], O_2[X][Z]$

#pragma HLS INTERFACE axis port=I_1

#pragma HLS INTERFACE axis port=I_2

#pragma HLS INTERFACE axis port=I_3

#pragma HLS INTERFACE axis port=O_1

#pragma HLS INTERFACE axis port=O_2

for $i < X$ **do** ▷ Module 1: Valid

  $val = I_2[i]$

  for $j < Y$ **do**

    $O_1[i][j] = I_1[i][j] * val$

for $i < X$ **do** ▷ Module 2: Invalid

  for $j < Y$ **do**

    $O_1[i][j] = I_1[i][j] * I_2[i]$

for $i < X$ **do** ▷ Module 3: Invalid

  for $j < Z$ **do**

    for $k < Y$ **do**

      $O_2[i][j] += I_1[i][k] * I_3[k][j]$

$I_{1_{temp}} = I_1$, $I_{3_{temp}} = I_3$ ▷ Module 4: Valid

for $i < X$ **do**

  for $j < Z$ **do**

    $sum = 0$

    for $k < Y$ **do**

      $sum += I_{1_{temp}}[i][k] * I_{3_{temp}}[k][j]$

    $O_2[i][j] = sum$

Figure 3.4. Examples of valid and invalid task-level streaming modules.

Assume that $Y = f * y$ where f and y are positive integers. Then, the matrix multiplication can be rewritten as

$$M_{3_{i,j}} = \sum_{k=1}^{f} \sum_{l=1}^{\frac{Y}{f}} M_1[i][k][l] * M_2[k][l][j]. \tag{3.2}$$

In this form, the matrices M_1 and M_2 are manually tiled into the f factor on the Y dimension. After this modification, the arrays can be completely partitioned by the new factor dimension f to apply parallelism to the matrix multiplication. Then, the completely partitioned independent memory parts are streamed through the system in FIFO buffers.

Applying this factorized matrix multiplication with constraints of task-level streaming brings another challenge in training neural networks. Unlike the above matrix multiplication example, the number of manually factorized dimensions of the matrices is more than one in neural network applications. Hence, there are uncommon factors between two matrices when applying matrix multiplication.

For example, when a batch of input images is used for the training of a CNN model, the input matrix can be denoted as $I[BS][C][W][H]$ where BS, C, W, H stand for mini-batch size, number of channels, width and height respectively. This matrix can be factorized in BS and C dimensions. Assume $BS = BS_f * BS_p$ and $C = C_f * C_p$ in which BS_f and C_f denote the regarding factors of the dimensions. CNN models also have kernel matrices, and a kernel matrix can be denoted as $K[F][C][KW][KH]$ in which F, C, KW, KH stand for the number of filters, the number of channels, kernel width, and kernel height respectively. This matrix can be factorized in F and C dimensions. Assume $F = F_f * F_p$ and $C = C_f * C_p$ in which F_f and C_f denote the regarding factors of the dimensions.

It can be noticed that dimension size C is common in both matrices; however, the dimension size BS and F differ. In the calculations, the matrices need to be reused multiple times for dimensions with uncommon sizes.

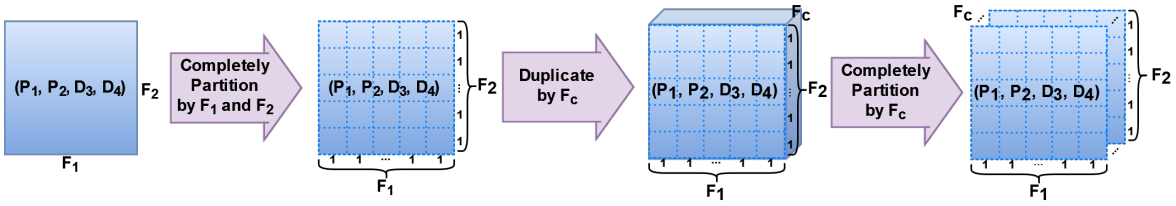


Figure 3.5. Illustration of duplication and tiling of arrays in dataflow parallelization.

Since reusing the same data is prohibited in data streaming, the matrices must be duplicated temporarily for task-level compliance. An illustration of the generalized method of duplication can be seen in Figure 3.5. Assume that $D_n = F_n \times P_n$ where D_n denotes dimension size, F_n denotes the manual tiling factor, and P_n denotes the remaining part after manual tiling of the n^{th} dimension. In the figure, it can be seen that D_3 and D_4 are not manually tiled. Assume that $X_1 \times X_2$ operation is to be calculated, and the image shows the duplication of the matrix X_1. It means that the matrices X_1 and X_2 share a matching dimension, and both have a non-matching dimension. Assume that F_c denotes the manual tiling factor of the dimension of X_2 that does not match with X_1. Therefore, X_1 is duplicated by F_c for compliance with task-level streaming constraints.

In the proposed method, calculations of neural network training are done in three steps. These three methods can be coined briefly as duplication, calculation in parallel, and aggregation. An example of a convolution operation of a CNN model can be seen in Figure 3.6. In this example, I, K, O represent the input array, the kernel array and the output array respectively. In addition, $BS, C, F, W, H, KW, KH, W_O, H_O$ stand for mini-batch size, number of channels, number of kernel filters, image width, image height, kernel width, kernel height, the width and height of the output in the same regard. Also, BS_f, C_f, F_f stand for the factorizations of the BS, C, F in which $BS = BS_f * BS_p$, $C = C_f * C_p$ and $F = F_f * F_p$. I, K, O arrays are also completely partitioned by their factorization dimension. In this example, CALCULATE function is equivalent to the convolution operation of CNN models for a mini-batch of images. The algorithm can be modified and used for fully connected layer calculations with appropriate inputs,

outputs and CALCULATE function. Assume that $W^{'}[L1][L2]$ is the weight matrix of an FCL in which $L1$ denotes the length of the FCL and $L2$ denotes the length of the input feature map. The matrix $W^{'}$ can be factorized by the dimension $L1$ as $L1 = L1_f * L1_p$ and $L2$ as $L2 = L2_f * L2_p$ where $L1_f$ and $L2_f$ are the factors of the matrix $W^{'}$.

In the algorithm, completely partitioning the manual tiles is done with the "#pragma HLS ARRAY_PARTITION complete" command where the "variable=" parameter is used to specify the array and the "dim=" parameter is used for specifying the dimensions to partition. The arrays are also streamed inside the module, and pipelining is applied by dataflow pragma. In addition, the task-level parallelism on calculation is applied by using the "#pragma HLS UNROLL" command.

Calculations of deep neural networks are mainly based on matrix multiplication. Weight matrices of every two connected layers share a common dimension. For example, assume an image of three channels is taken as an input to convolution operation using eight filters of a three-channel kernel. The output of this operation consists of eight channels. If another convolution operation is applied to the output, the kernel contains eight channels, equal to the previous kernel's number of filters. When manual tiling is applied to the first convolution kernel, our tool automatically sets the tiling factor of the next convolution kernel on the corresponding dimension. It also works for the backpropagation phase of the training as well. Using this fact, an automation process of the proposed method can be seen in Figure 3.8. In the algorithm, the dimensions of the convolution kernels are given with the dimensional notation $[F_f^i, C_f^i][F_p^i, C_p^i][KW^i][KH^i]$ for $F^i = F_f^i \times F_p^i$ number of filters and for $C^i = C_f^i \times C_p^i$ number of channels. Fully connected layer weights are given with the dimensional notation $[L1_f^i, L2_f^i][L1_p^i, L2_p^i]$ for the lengths of $L1^i = L1_f^i \times L1_p^i$ and $L2^i = L2_f^i \times L2_p^i$. The outputs of the convolutional operations are given with the dimensional notation $[BS_f, C_f^i][BS_p, C_p^i][W^i][H^i]$ where $BS = BS_f \times BS_p$ is the minibatch size. The outputs of the fully connected layers are given with dimensional notation $[BS_f, L2_f^i][BS_p, L2_p^i]$. In the notations, i denotes the index of the layer in the model. The dimensional notations are the same notation with given examples of tiling factorizations.

Input: Batch of images: $I[BS_f, C_f][BS_p, C_p][W][H]$,
Kernel: $K[F_f, C_f][F_p, C_p][KW][KH]$
Output: $O[BS_f, F_f][BS_p, F_p][W_O][H_O]$
function TaskLevelConvolution(I, K, O)
Define $I_{temp}[BS_f, C_f, F_f][BS_p, C_p][W][H]$
Define $K_{temp}[F_f, C_f, BS_f][F_p, C_p][KW][KH]$
Define $O_{temp}[BS_f, F_f, C_f][BS_p, F_p][W_O][H_O]$

#pragma HLS ARRAY_PARTITION variable=I_{temp} complete dim=1,2,3
#pragma HLS ARRAY_PARTITION variable=K_{temp} complete dim=1,2,3
#pragma HLS ARRAY_PARTITION variable=O_{temp} complete dim=1,2,3

#pragma HLS STREAM variable=I_{temp}
#pragma HLS STREAM variable=K_{temp}
#pragma HLS STREAM variable=O_{temp}

#pragma HLS DATAFLOW

DUPLICATE(I, K, I_{temp}, K_{temp})
for $i < BS_f$ **do** ▷ #pragma HLS UNROLL
for $j < C_f$ **do** ▷ #pragma HLS UNROLL
for $k < F_f$ **do** ▷ #pragma HLS UNROLL
CALCULATE($I_{temp}[i, j, k], K_{temp}[k, j, i],$
$O_{temp}[i, k, j]$)
AGGREGATE(O_{temp}, O)

Figure 3.6. Example algorithm of a convolution operation using the proposed method on FPGA.

```
function DUPLICATE(I, K, I_temp, K_temp)
    for i < BS_f do                                   ▷ #pragma HLS UNROLL
        for j < C_f do                                ▷ #pragma HLS UNROLL
            val = I[i, j]
            for k < F_f do                            ▷ #pragma HLS UNROLL
                I_temp[i, j, k] = val
    for i < F_f do                                    ▷ #pragma HLS UNROLL
        for j < C_f do                                ▷ #pragma HLS UNROLL
            val = F[i, j]
            for k < BS_f do                           ▷ #pragma HLS UNROLL
                F_temp[i, j, k] = val

function AGGREGATE(O_temp, O)
    for i < BS_f do                                   ▷ #pragma HLS UNROLL
        for j < F_f do                                ▷ #pragma HLS UNROLL
            val = 0
            for k < C_f do                            ▷ #pragma HLS UNROLL
                val+ = O_temp[i, j, k]
            O[i, j] = val
```

Figure 3.7. Example algorithm of a convolution operation using the proposed method on FPGA. (cont.)

In Figure 3.8, $layers(EM)$ denotes the layers, $weights(EM)$ and $outputs(EM)$ denote the weights and the outputs of the corresponding layers. Similarly, $dim(\partial\mathcal{L}(EM)$ stands for the losses sent back from the next layer and $gradient(EM)$ represents the corresponding gradients of the layers. Here, $dim()$ gives the dimensions of the corresponding array and $n()$ gives the cardinality of an array. Also, f is the manually selected tiling factorizations.

Input: Batch of images: $I[BS_f, C_f][BS_p, C_p][W][H]$,
Edge model: EM
Manual factorizations of layers: f
for $i <= n(layers(EM))$ **do** ▷ Forward Pass
if $layers(EM)[i] =$ Convolution **then**
if $i = 1$ **then**
$dim(weights(EM)[i]) = [f[i], C_f][\frac{F^i}{f[i]}, \frac{C^i}{C_f}][KW^i][KH^i]$
else if $i > 1$ **then**
$dim(weights(EM)[i]) = [f[i], f[i-1]][\frac{F^i}{f[i]}, \frac{F^{i-1}}{f[i]}][KW^i][KH^i]$
$dim(outputs(EM)[i]) = [BS_f, f[i]][BS_p, \frac{F^i}{f[i]}][W^i][H^i]$
else if $layers(EM)[i] =$ Fully Connected Layer **then**
if $i = 1$ **then**
$dim(weights(EM)[i]) = [C_f, f[i]][\frac{L1^i}{C_f}, \frac{L2^i}{f[i]}]$
else if $i > 1$ **then**
$dim(weights(EM)[i]) = [f[i-1], f[i]][\frac{L1^i}{f[i-1]}, \frac{L2^i}{f[i]}]$
$dim(outputs(EM)[i]) = [BS_f, f[i]][BS_p, \frac{L2^i}{f[i]}]$

for $i <= n(layers(EM))$ **do** ▷ Backward Pass
if $layers(EM)[i] =$ Fully Connected Layer **then**
$dim(\partial\mathcal{L}(EM)[i]) = [BS_f, f[i]][BS_p, \frac{L2^i}{f[i]}]$
$dim(gradient(EM)[i]) = [f-1[i], f[i]][\frac{L1^i}{f[i-1]}, \frac{L2^i}{f[i]}]$
else if $layers(EM)[i] =$ Convolution **then**
$dim(\partial\mathcal{L}(EM)[i]) = [BS_f, f[i]][BS_p, \frac{L2^i}{f[i]}]$
$dim(gradient(EM)[i]) = [f[i], f[i-1]][\frac{F^i}{f[i]}, \frac{F^{i-1}}{f[i]}][KW^i][KH^i]$

Figure 3.8. Automation process of the tiling.

The designer chooses a manual tiling factor for the applicable dimension of all layer weights, and the other factorizable dimension is automatically factorized based on the manual factorization of the weight of the previous layer as shown in the algorithm. In

this case, the minibatch size and the channel of the input, kernel filter of the convolution kernels and length of the fully connected layers need to be manually factorized into tiles. As the result of the algorithm with the application of the dataflow approach, the Xilinx HLS tool creates a structure in which data streams from the input to the output between calculation modules. The illustration of this structure is given in Figure 3.9. The input images I_0 and I_1, their ground-truth labels y_0 and y_1, and packed weight and bias arrays are taken as inputs, shown as weight_in and bias_in. Data flow through the forward and backward propagation calculation as shown with pink arrows between modules. Forward propagation starts with conv2d_1 module and ends with dense_5 module. Backward propagation starts with dL5_func module and ends with dF1_func module. The predictions POUT_0 and POUT_1, and the calculated gradients are given as the output, shown as weight_out and bias_out.

The edge models are trained independently from each other and the models that are on the server. For the image classification task, the cross-entropy loss function is the loss function applied to the edge models during training, given as

$$Loss_{EM}(y, \hat{y}) = -\sum_{i}^{c} y_i log(\hat{y}_i), \tag{3.3}$$

where y_i denotes the ground truth label for the i^{th} class and $\hat{y}_i$ denotes the predicted value for the i^{th} class. For the regression task, mean squared error (MSE) loss is used as the loss function during edge model training, given as

$$Loss_{EM}(y, \hat{y}) = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2, \tag{3.4}$$

where n denotes the sample size, y_i denotes the ground truth value for the i^{th} observation in the sample and $\hat{y}_i$ denotes the predicted value for the i^{th} observation.

3.2. Variational Auto-Encoder (VAE) Models

In the conventional ensemble learning approach, each edge model sends its feature vector, and the final decision is made upon all of those feature vectors via the ensemble model on the server side. Nevertheless, receiving a feature vector from all edge devices is not always granted in real-world applications.

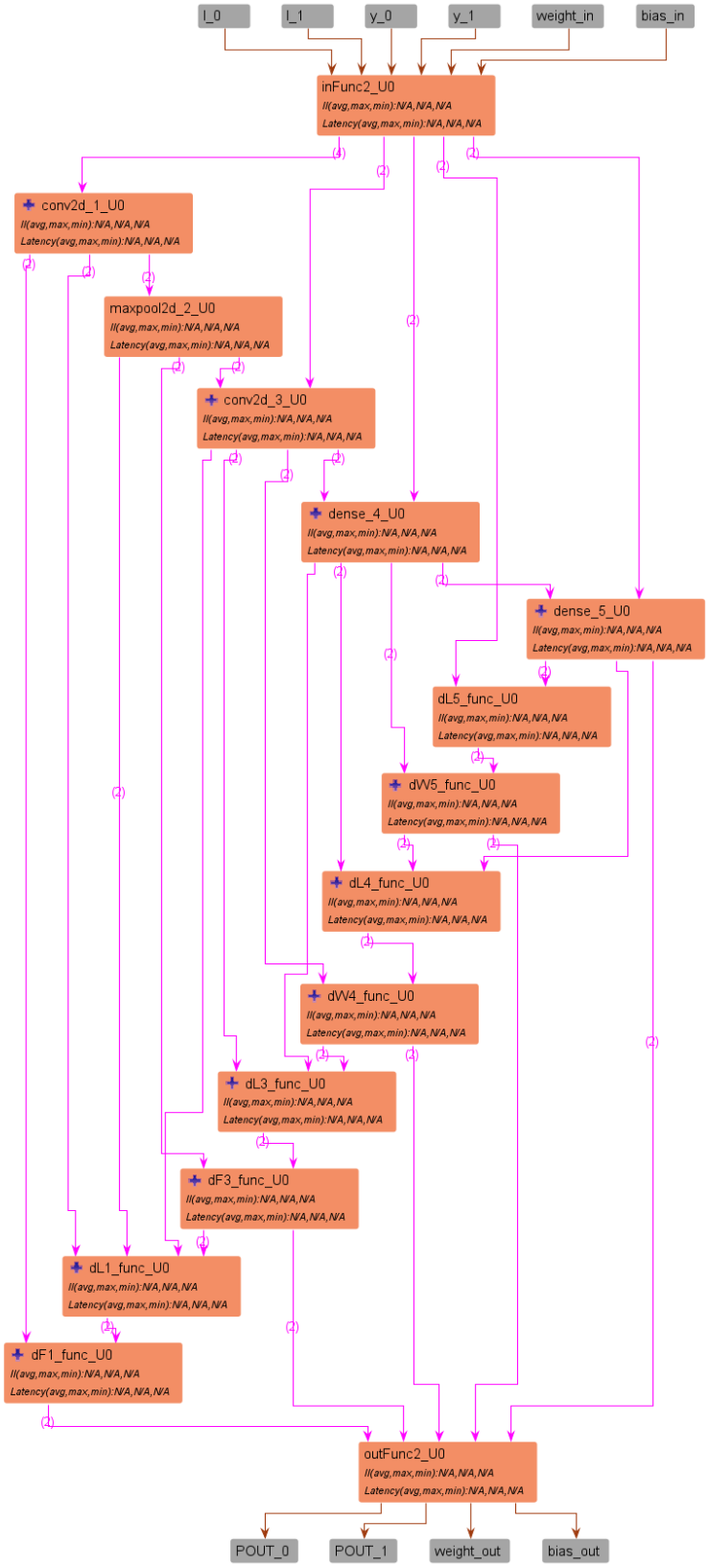


Figure 3.9. The dataflow structure created by the Xilinx HLS tool.

In real-world applications, it is possible to receive feature vectors from only a subset of the edge devices for the same occasion due to several possible hindrances, such as connection problems to the server or edge models not being able to process some inputs due to transient faults. However, the proposed ensemble learning model will still need a feature vector from all edge devices. In such cases, this work proposes to replace the missing feature vectors with relevant values to obtain the ensemble model's final decision on the server. As the distributions of the feature vectors transferred from the edge models are expected to vary, these missing features need to be filled in a specific way for each edge model.

The feature vector that cannot be gathered from an edge model EM_i needs to be replaced with a feature vector with the same dimension. Moreover, the feature vector used for the replacement needs to be statistically meaningful; that is, the replacement should not cause redundancy in the ensemble data. Therefore, the proposed system leverages VAE models for feature vector generation for the replacement data. N number of VAE models exists exclusive for all N edge models. The VAE model VAE_i is independently trained with $F_{train}^{EM_i}$ for i from 1 to N. The goal of the VAE models is to mimic the feature vectors that the edge models may produce in the sense of following a similar statistical distribution. In Figure 3.1, the red line from the edge devices to VAE models represents the transfer of the $F_{train}^{EM_i}$ from the inference of the edge model EM_i to the VAE model, VAE_i.

The VAE models in this approach receive the feature vectors from the edge models, encoded as a distribution over latent space. Then, a sample chosen from the latent space is decoded as the output feature vectors of VAE models whose reconstruction errors are used in backpropagation to train VAE models. The loss function used in the training of VAE models is given as

$$Loss_{VAE_i} = \left\| F_{train}^{EM_i} - VAE_i^{dec}(z) \right\|^2 + KL[\mathcal{N}(\mu_{enc}, \sigma_{enc}), \mathcal{N}(0,1)], \tag{3.5}$$

where z denotes a random sample vector from the latent space with the distribution $\mathcal{N}(\mu_{enc}, \sigma_{enc})$, μ_{enc} and σ_{enc} denote mean and standard deviation terms of the encoding, KL denotes the Kullback-Leibler (KL) divergence [93] and $\mathcal{N}$ denotes normal distri-

bution. Reconstruction loss, $\left\|F_{train}^{EM_i} - VAE_i^{dec}(z)\right\|^2$, calculates the Euclidean distance between the actual and generated feature vectors. KL divergence is a type of statistical distance that gives the distance between two statistical distributions. For discrete probability distributions, it can be defined as

$$KL[P,Q] = \sum_{x \in \mathcal{X}} P(x) log(\frac{P(x)}{Q(x)}), \tag{3.6}$$

where P and Q are probability distributions defined on discrete probability space $\mathcal{X}$. For continuous probability distributions, KL divergence is defined as

$$KL[P,Q] = \int_{-\infty}^{\infty} p(x) log(\frac{p(x)}{q(x)}) dx, \tag{3.7}$$

where P and Q are probability distributions defined on continuous probability space $\mathcal{X}$. p and q are probability density functions of P and Q.

After the training of VAE models, they generate artificial feature vectors that come from a similar distribution to the feature vectors obtained from the edge model for training. The feature generation is done by selecting a random variable from the latent space as z and decoding it as $VAE_i^{dec}(z)$. Also, it should be noted that the VAE models are deployed on the server device.

3.3. Ensemble Model

The edge devices used in this thesis are not strong enough to deploy state-of-the-art model architectures, so they cannot reach high accuracy values. To make a final prediction with a high accuracy value, an ensemble model is deployed on the server, which can perform computationally heavy calculations with abundant resources. The ensemble model aims to fuse the information gathered from the edge devices and the VAE models in case an error occurs during information transfer from the edge devices. This thesis proposes a convolutional ensemble method to create such a fusion to make a final prediction. In the convolutional ensemble method, a convolution operation is applied to the stacked feature vectors gathered from the edge and VAE models.

After the VAE models are trained, the ensemble data are ready to be gathered to train the ensemble model on the server device. Assume K_{train} contains the indices of the images that are elements of full training data set D_{train}. Also, let $K_{train}^{EM_i}$ contain the indices of the images that are elements of the training data used by the edge model EM_i, $D_{train}^{EM_i}$. Since $D_{train}^{EM_i} \subseteq D_{train}$, it can be deduced that $K_{train}^{EM_i} \subseteq K_{train}$.

Assume that $k \in K$ and $k \notin K_{train}^{EM_i}$. The ensemble model denoted by Ens needs its training data, D_{train}^{Ens}, of size $n(D_{train}) \times N \times L_{com}$ where $n(D_{train})$ is the cardinality of the full train set D_{train}, N is the number of the edge models and L_{com} is the dimensionality of the fully connected layer that is chosen to be the same for all edge models. Feature vector $D_{train}^{Ens}[k,i]$ is a vector generated by the VAE_i when EM_i cannot deliver the regarding feature representation for k^{th} image. Therefore, D_{train}^{Ens} is built according to the formula for all $k \in K_{train}$ given as

$$D_{train}^{Ens}[k,i] = \begin{cases} F_{train}^{EM_i} & if\ k \in K_{train}^{EM_i} \\ VAE_i^{dec}(z) & if\ k \notin K_{train}^{EM_i} \end{cases}. \tag{3.8}$$

In the formula, $F_{train}^{EM_i}$ denotes the feature vector array obtained from the edge model EM_i for its related training set, $VAE_i^{dec}()$ function denotes the decoder part of the VAE model VAE_i and $z \in \mathbb{R}^d$ denotes a random vector from the latent space of the VAE_i model. In Figure 3.1, the blue-dotted line represents the $VAE_i^{dec}(z)$ part, which is the feature replacing the part in Equation 3.8. In addition, the green-dashed-dotted line shows D_{train}^{Ens}, which denotes the data for the ensemble model. The training of the ensemble model runs on the server side.

After the data preparation, D_{train}^{Ens} is used as input for training with the proposed convolutional ensemble operation. An illustration of the convolutional ensemble operation is shown in Figure 3.10. Here, the feature vectors gathered from the edge model, F^{EM_i}, and VAE models when features from the edge devices are not obtained, $VAE_i^{dec}(z)$, are stacked in the edge device order. After that, a convolution filter is passed through the stacked feature vectors. The goal of using a convolution operation for ensemble learning is to extract valuable relationships between the elements of feature vectors of the edge models. In this work, the ensemble model is a CNN model consisting

of a convolutional layer and a hidden fully-connected layer. In addition, the kernel size for the ensemble model is taken as $(\frac{N}{2}, \frac{L_{com}}{2})$.

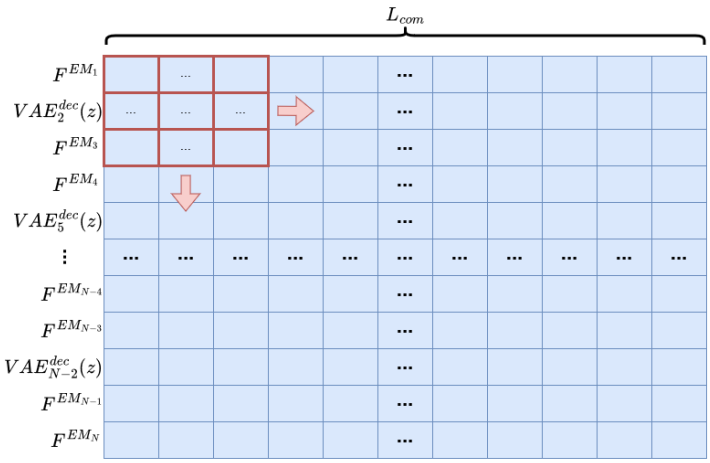


Figure 3.10. Illustration of the convolutional ensemble operation for a single image. F^{EM_i}: the feature vectors transferred from the i^{th} edge model. $VAE_i^{dec}(z)$: the feature vectors generated by i^{th} VAE model for the same image.

Similar to edge models, the ensemble model is trained using the cross-entropy loss function for the image classification task, given as

$$Loss_{Ens}(y, \hat{y}) = -\sum_{i}^{c} y_i log(\hat{y}_i), \tag{3.9}$$

where y_i denotes the ground truth label for the i^{th} class and $\hat{y}_i$ denotes the prediction for the i^{th} class. For the regression task, mean squared error (MSE) error is used for ensemble training, given as

$$Loss_{Ens}(y, \hat{y}) = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2, \tag{3.10}$$

where n denotes the sample size, y_i denotes the ground truth value for the i^{th} observation in the sample and $\hat{y}_i$ denotes the prediction for the i^{th} observation.

4. TRAINING SCENARIOS

The training of the proposed system can be done in three different scenarios based on the data transfer scheme. These three scenarios bring advantages and disadvantages to the training process and stand on a trade-off between training time, memory requirement, and overall accuracy. The appropriate scenario can be chosen according to the device choice and attributes of the data transfer medium.

In all scenarios, Ep^{EM}, Ep^{VAE} and Ep^{Ens} denote the training epoch of edge models, the training epoch of VAE models, and the training epoch of the ensemble model, respectively.

4.1. Training Scenario 1 - Abundant Memory on the Server

The first scenario is considered when all feature vectors obtained from the inference of training data are transferred from the edge models to the server at once. Therefore, VAE training and ensemble learning can start without any interruption on the server side once the data transfer is completed. In this scenario, the memory requirements on the server side are the highest among the three scenarios since all the data must be stored on the server device. On the other hand, the first scenario achieves the highest accuracy among all the scenarios discussed in this study.

The first scenario is preferable if the server can store the ensemble data all at once and the data transfer is fast. Its algorithmic representation of training with Scenario 1 can be seen in Figure 4.1. Firstly, edge models are trained independently using their training data for Ep^{EM} epochs. After their training, feature vectors, $F_{train}^{EM_i}$, with the common length are obtained by inference. After the inference, the feature vectors are transferred to the server in only one transfer. When the transfer is done, the VAE model of each edge model denoted as VAE_i is trained for Ep^{VAE} epochs. Then, missing feature vectors are replaced via these trained VAE models for each edge model. As the

features are obtained from the same input for edge models, the missing features are filled using the same latent vector denoted as z. In the end, the final ensemble model is trained using all feature vectors received and replaced.

Input: Edge models: EM_i,
Training data of edge models: $D_{train}^{EM_i}$
Output: Ensemble model: Ens
for $i < N$ **do**
TrainEdgeModel($EM_i, D_{train}^{EM_i}, Ep^{EM}$)
$F_{train}^{EM_i}$ =InferenceEdgeModel($EM_i, D_{train}^{EM_i}$)
TransferToServer($F_{train}^{EM_i}$)
VAE_i =TrainVAEModel($F_{train}^{EM_i}, Ep^{VAE}$)
ReplaceFeature($F_{train}^{EM_i}, VAE_i, z$)
TrainEnsembleModel($Ens, F_{train}^{EM_1}, ..., F_{train}^{EM_N}, Ep^{Ens}$)

Figure 4.1. Algorithm of system training in Scenario 1.

4.2. Training Scenario 2 - Limited Memory on the Server

The second scenario is when only a minibatch of ensemble training data is sent to ensemble learning. This scenario remarkably reduces the memory requirement on the server side as the server-side device needs to store only a minibatch of all training data rather than all of it. In the second scenario, the total ensemble training epoch Ep^{Ens} is divided by a factor Ep_d^{Ens}. In one communication, the stored minibatch is repeatedly used in training for $\frac{Ep^{Ens}}{Ep_d^{Ens}}$ epoch. In the next communication, another minibatch is sent from the edge devices to the server for ensemble training. Every minibatch is transferred to the server for Ep_d^{Ens} times. The second scenario requires less data per transfer but more communication. Its algorithmic representation is presented in Figure 4.2. The disadvantage of this scenario is that training the ensemble learner using the same minibatch repeatedly causes bias in training hence, degrading the accuracy. However, the degradation in accuracy caused by bias can be alleviated to a certain level by

decreasing the repeated use of a minibatch in training. For example, increasing the number of Ep_d^{Ens} alleviates such bias in training. By that, the same minibatch is used less repeatedly in the training of the ensemble model, precisely Ep^{Ens}/Ep_d^{Ens} epoch at once. In other words, increasing Ep_d^{Ens} decreases bias in training. Nevertheless, it also increases the number of transfers.

Input: Edge models: EM_i,
  Training data of edge models: $D_{train}^{EM_i}$,
  Divisor of ensemble learning epochs: Ep_d^{Ens}
Output: Ensemble model: Ens
  for $i < N$ **do**
    TrainEdgeModel($EM_i, D_{train}^{EM_i}, Ep^{EM}$)
    $F_{train}^{EM_i}$ =InferenceEdgeModel($EM_i, D_{train}^{EM_i}$)
  for $j < Ep_d^{Ens}$ **do**
    for $i < N$ **do**
      for $F_{train}^{EM_i,j}$ in $F_{train}^{EM_i}$ **do** ▷ Store on server
        TransferToServer($F_{train}^{EM_i,j}$)
        VAE_i =TrainVAEModel($F_{train}^{EM_i,j}, Ep^{VAE}$)
        ReplaceFeature($F_{train}^{EM_i,j}, VAE_i, z$)
    TrainEnsembleModel($Ens, F_{train}^{EM_1,j}, ..., F_{train}^{EM_N,j}, Ep^{Ens}/Ep_d^{Ens}$)

Figure 4.2. Algorithm of system training in Scenario 2.

The training of VAE models is also deteriorated by low Ep_d^{Ens}. It yields lower accuracy in the feature generation of VAE models than Scenario 1. This scenario can be selected when the server-side device has a lower memory capacity and the transfer medium is fast enough to make up the repeated one-way communications. Scenario 2 differs from the Scenario 1 algorithm by repeatedly using the same minibatch of feature vectors transferred from the edge models to the server. The same minibatch of training data is used in training for Ep^{Ens}/Ep_d^{Ens} epochs at once. After then, another minibatch is transferred and used in training for the same number of epochs. The minibatches

are stored on the server until they are used in training for Ep^{Ens}/Ep_d^{Ens} epochs. They are transferred Ep_d^{Ens} times in a random communication from the edge models to the server. In the end, all minibatches are used in training for the same number of epochs, Ep^{Ens}, in total.

4.3. Training Scenario 3 - Extremely Low Memory on the Server

In the third scenario, it is assumed that there is no memory space reserved in the server-side device for storing the training data. In this scenario, a minibatch of ensemble training data is randomly sent to the ensemble learner for each ensemble learning epoch. This method may be preferred when the server-side device has no designated memory for keeping training data and the transfer medium is fast enough to send repeatedly generated mini-batches without too much delay.

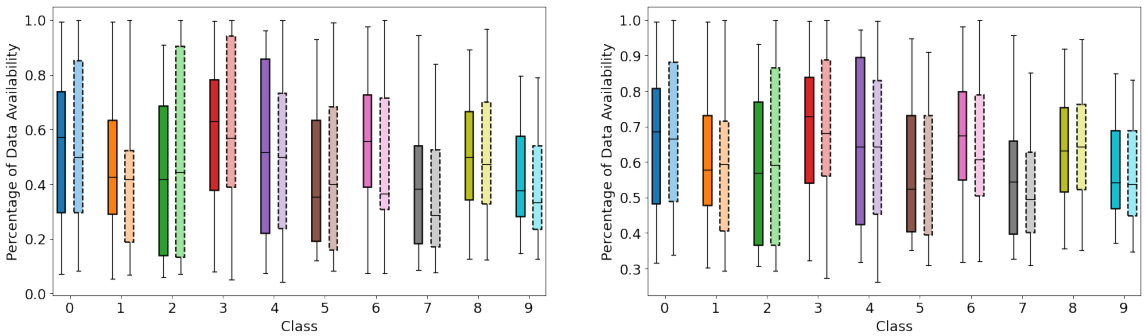
Input: Edge models: EM_i,
Training data of edge models: $D_{train}^{EM_i}$,
Divisor of ensemble learning epochs: Ep_d^{Ens}
Output: Ensemble model: Ens
for $i < N$ **do**
TrainEdgeModel($EM_i, D_{train}^{EM_i}, Ep^{EM}$)
$F_{train}^{EM_i}$ =InferenceEdgeModel($EM_i, D_{train}^{EM_i}$)
for $j < Ep^{Ens}$ **do**
for $i < N$ **do**
for $F_{train}^{EM_i,j}$ in $F_{train}^{EM_i}$ **do** ▷ Not store
TransferToServer($F_{train}^{EM_i,j}$)
VAE_i =TrainVAEModel($F_{train}^{EM_i,j}, Ep^{VAE}$)
ReplaceFeature($F_{train}^{EM_i,j}, VAE_i, z$)
TrainEnsembleModel($Ens, F_{train}^{EM_1,j}, ..., F_{train}^{EM_N,j}, 1$)

Figure 4.3. Algorithm of system training in Scenario 3.

Since each minibatch is used in training for one epoch without creating a bias, the accuracy acquired is close to the first scenario. The disadvantage of this scenario is the requirement for an excessive number of one-way communication. The algorithmic representation of training with Scenario 3 can be seen in Figure 4.3. Scenario 3 algorithm can be considered as a special case of Scenario 2 algorithm as Ep_d^{Ens} chosen as Ep^{Ens}. The main difference is that the Scenario 3 algorithm does not store a minibatch on the server. It is used in training for an epoch and discarded immediately.

5. EXPERIMENTAL RESULTS

As pointed out in Chapter 1, the edge models fed by stationary data sources are most likely to keep getting the data with a similar class percentage over time in real-world applications. For example, a camera located on a highway and targeted at the right lane is likely to record heavy and slow vehicles most of the time. In this case, the models are trained and used later for inference with the data that has a similar categorical data percentage. The experiments also focus on this assumption and examine the effect of the case when training and test data have a discrepancy in the percentage of class sample sizes of the train and test data. For example, suppose an edge model is trained with 17% of the training data of a particular class and tested with 22% of the test data of the particular class. In that case, the mentioned discrepancy rate is 5% for this class between the training and test data.



(a) (b)

Figure 5.1. Two different class sample percentage box plots of train and test sets of 20 edge devices for CIFAR-10 dataset. (a) $Min\% = 0.05$ and $MaxDisc\% : 0.5$, (b) $Min\% = 0.3$ and $MaxDisc\% : 0.2$. Boxes with straight lines are of train sets and dashed lines are of test sets.

Two main symbols are used for the size of training and test data. $Min\%$ denotes the minimum percentage of whole training data of classes used in the training edge models. For example, if $Min\% = 0.25$ and the whole training data consists of 4000

images of ten classes, each edge model is trained with at least 1000 randomly sampled images of all ten classes. In addition, $MaxDisc\%$ denotes the maximum discrepancy rate between the percentage of sampled training data and sampled test data for a class. For example, if 10% of the whole training data of class 0 is used for training an edge model and $MaxDisc\% = 0.5$ then the percentage of test data of class 0 that the edge model accesses is between 5% and 15%.

The experiments are conducted by taking random subsets of the whole train and test data. First of all, the train data that are available for each edge model are randomly selected. For example, $K_{train}^{EM_i}$ is chosen as $\bigcup_{i=0}^{c}\{k_i \subset K_{train}, \frac{|k_i|}{|K_{train_i}|} = X \sim \mathcal{U}_{[Min\%,1]}$ and k_i is chosen randomly$\}$. Thus, $Min\%$ of the train data for each class is randomly chosen. In addition, K_{test} is chosen as $\bigcup_{i=0}^{c}\{k_i \subset K_{test}, \frac{|k_i|}{|K_{test_i}|} = max\{min\{|K_{train_i}| * (1+X), 1\}, 0\}$ and $X \sim \mathcal{U}_{[-MaxDisc\%,MaxDisc\%]}\}$. Hence, the available percentage of the test data for each class may differ from the available data percentage of the same class by $MaxDisc\%$. Figure 5.1 shows that two different values of $Min\%$ and $MaxDisc\%$ remarkably change the class data percentage for train and test data of edge devices. The analysis is done with 20 edge devices. For each device, training and test sets are randomly generated according to the $Min\%$ and $MaxDisc\%$ values. Different values of $Min\%$ and $MaxDisc\%$ remarkably affect the overall prediction accuracy with timing and memory requirements.

The classification experiments are conveyed in four different image classification benchmark datasets. The datasets are as follows:

- CIFAR-10 dataset of objects [28]
- CIFAR-100 dataset of objects [28]
- The MNIST dataset of handwritten digits [94]
- Fashion-MNIST dataset of fashion products [95]

Moreover, the performance of the method is also assessed for regression problems on the following datasets:

- Boston Housing dataset of house prices [96]
- California Housing dataset of house prices [97]
- Pecan Street dataset of electricity consumption [98]

It should be highlighted that the edge and ensemble models are developed using float (32-bit) datatype. ADAM optimizer [99] is the optimizer used for the learning process of the ensemble model with 10^{-4} learning rate. For edge models, Stochastic Gradient Descent (SGD) optimizer [100] is used with 10^{-4} learning rate.

The experimental results are obtained by deciding the required variables, such as the number of edge models, how much of the whole training set is available for each edge model, and how much discrepancy the test data has compared to the training data on the class data percentage. The time and memory requirements are calculated based on the following formulae and the used values given in Table 5.1.

The total required time for the training of the edge model is calculated as

$$T_{train}^{EM} = S_{train} * P_{train}^{EM} / S_{m}^{EM} * T_{train_m}^{EM} * Ep^{EM}. \tag{5.1}$$

The total required time for the inference of the edge model is calculated as

$$T_{inference}^{EM} = S_{train} * P_{train}^{EM} / S_{m}^{EM} * T_{inference_m}^{EM}. \tag{5.2}$$

The total required time for the transfer of the feature vectors from the edge model to the server is calculated as

$$T_{transfer}^{EM} = \begin{cases} S_{train} * P_{train}^{EM} * Bit_{com} * L_{com} / R_{transfer}^{EM} & if\ Scenario1 \\ S_{train} * P_{train}^{EM} * Bit_{com} * L_{com} * Ep_{d}^{Ens} / R_{transfer}^{EM} & if\ Scenario2. \\ S_{train} * P_{train}^{EM} * Bit_{com} * L_{com} * Ep^{Ens} / R_{transfer}^{EM} & if\ Scenario3 \end{cases} \tag{5.3}$$

The total required time for the training of the VAE model is calculated as

$$T_{train}^{VAE_{EM}} = T_{train}^{VAE_{all}} * P_{train}^{EM} * Ep^{VAE}. \tag{5.4}$$

The total required time for the training of the ensemble model is calculated as

$$T_{train}^{Ens} = \begin{cases} T_{train}^{Ens_{1Ep}} * Ep_{Ens} + (1 - P_{train}^{EM}) * T_{inference}^{VAE} & if\ Scenario1 \\ T_{train}^{Ens_{1Ep}} * Ep_{Ens} + (1 - P_{train}^{EM}) * T_{inference}^{VAE} * Ep_{d}^{Ens} & if\ Scenario2. \\ T_{train}^{Ens_{1Ep}} * Ep_{Ens} + (1 - P_{train}^{EM}) * T_{inference}^{VAE} * Ep^{Ens} & if\ Scenario3 \end{cases} \tag{5.5}$$

The total number of communication rounds required for the edge model is calculated as

$$N_{com}^{EM} = \begin{cases} 1 & if\ Scenario1 \\ \lceil S_{train} * P_{train}^{EM} / S_m^{Ens} * Ep_d^{Ens} \rceil & if\ Scenario2 \\ \lceil S_{train} * P_{train}^{EM} / S_m^{Ens} * Ep^{Ens} \rceil & if\ Scenario3 \end{cases}. \tag{5.6}$$

The total cumulative transferred data size from all edge models to the server is calculated as

$$M_{transfer}^{cum} = \begin{cases} S_{train} * P_{train}^{EM} * L_{com} * Bit_{com} * N & if\ Scenario1 \\ S_{train} * P_{train}^{EM} * L_{com} * Bit_{com} * N * Ep_d^{Ens} & if\ Scenario2 \\ S_{train} * P_{train}^{EM} * L_{com} * Bit_{com} * N * Ep^{Ens} & if\ Scenario3 \end{cases}. \tag{5.7}$$

The total transferred data size from the edge model to the server in a communication round is calculated as

$$M_{transfer}^{EM} = \begin{cases} S_{train} * P_{train}^{EM} * L_{com} * Bit_{com} & if\ Scenario1 \\ S_m^{Ens} * L_{com} * Bit_{com} & if\ Scenario2 \\ S_m^{Ens} * L_{com} * Bit_{com} & if\ Scenario3 \end{cases}. \tag{5.8}$$

The total required memory on the server is calculated as

$$M_{memory}^{server} = \begin{cases} S_{train} * P_{train}^{EM} * L_{com} * Bit_{com} * N & if\ Scenario1 \\ S_m^{Ens} * L_{com} * Bit_{com} * N & if\ Scenario2 \\ 0 & if\ Scenario3 \end{cases}. \tag{5.9}$$

The total required memory on the edge model is calculated as

$$M_{memory}^{EM} = \begin{cases} S_{train} * P_{train}^{EM} * L_{com} * Bit_{com} & if\ Scenario1 \\ S_m^{Ens} * L_{com} * Bit_{com} & if\ Scenario2 \\ S_m^{Ens} * L_{com} * Bit_{com} & if\ Scenario3 \end{cases}. \tag{5.10}$$

The evaluations are done for the CIFAR-10 dataset; thus, the values given in Table 5.1 are given for the CIFAR-10 dataset. Nonetheless, the values in the table are also relevant for the other datasets, with a few exceptions. S_{train} is 60000 for MNIST and Fashion MNIST. Ep^{Ens} is 200 for CIFAR-100. Time-related values must be considered for CIFAR-10. As shown in Table 5.1, L_{com} is chosen as 64 for edge models. The VAE and the ensemble model architecture are the same for all datasets and training scenarios. VAE model consists of an FCL with 64 neurons in encoder and decoder parts and the latent space is taken as $\mathbb{R}^{32}$. The total number of parameters used for both the encoder and decoder parts of the VAE model is 260K.

Table 5.1. The values used for evaluating execution time and memory requirements.

Variable	**Value**
S_{train}	50000
L_{com}	64
Bit_{com}	32
Ep^{Ens}	100
S_m^{Ens}	128
N	20
Ep^{EM}	30
$T_{train_m}^{EM}$	0.022 sec
$T_{inference_m}^{EM}$	0.007 sec
$R_{transfer}^{EM}$	450 Mbps
T_{train}^{Ens}	120 sec
$T_{train}^{VAE_{all}}$	47 sec
$T_{inference}^{Ens}$	2 sec
$T_{inference}^{VAE}$	0.2 sec
Ep_d^{Ens}	20
P_{train}^{EM} for $Min\% = (0.05, 0.3, 0.7)$	(0.6, 0.7, 0.87)

Ensemble model consists of 64 $\frac{N}{2} \times \frac{L_{com}}{2}$ filters and an FCL with 64 neurons. The ensemble model has 182K parameters for all datasets and training scenarios. The VAE and ensemble model experiments are conducted on GeForce RTX 2080 Ti device. Also, it should be noted that the system is trained in Scenario 1 unless a different scenario is mentioned for training. Moreover, the accuracy values are given for $Min\% = 0.05$ and $MaxDisc\% = 0$ unless mentioned. Experiments on classification make use of train and test data that are randomly generated for each edge device based on $Min\%$ and $MaxDisc\%$ values. For the regression datasets, the targets are continuous values rather than categorical values. The target values are divided into ten groups in training data where all groups have the same density of data to mimic the classification behaviour. After then, test data are also grouped based on the training grouping. After creating classes for regression datasets, $Min\% = 0.05$ is chosen for the experiments.

5.1. Edge Models

In the experiments, different edge model structures are used as the proposed method allows. The layer structures and the number of total model parameters are given in Table 5.2 and Table 5.3 for the image classification and regression tasks, respectively.

Table 5.2. Edge models used in regression experiments.

Edge Model Type	**Layers**	**Number of Parameters**
1	64 FCL 1 FCL	5213
2	64 FCL 64 FCL 1 FCL	9373

The edge model structures are randomly chosen among the model structures given in Table 5.2 for the image classification task and Table 5.3 for the regression task. For the image classification task, output layers differ according to the problem set that is dealt with. For example, the main datasets chosen for the experiments have ten classes, except for CIFAR-100, which has 100 classes. The number of parameters differs for both datasets as the number of classes differs. The number of parameters and the dimensionality of the output layer are given according to the ten class datasets. For the regression task, the dimensionality of the output layer is always one. In Figure 5.2, the average feature dimensionalities are reduced to 2 by using the t-distributed stochastic neighbour embedding (t-SNE) method [101] of the feature vectors obtained from the layers of edge models for CIFAR-10 are given. t-SNE is a method used for visualizing high-dimensional data by projecting the dimensions of the data to two or three dimensions.

Table 5.3. Edge models used in image classification experiments.

Edge Model Type	**Layers**	**Number of Parameters**
1	one 5×5 filter, 2×2 Max pooling one 5×5 filter 64 FCL, 10 FCL	7216
2	two 5×5 filters, 2×2 Max pooling two 5×5 filters 64 FCL, 10 FCL	13768
3	four 5×5 filters, 2×2 Max pooling four 5×5 filters 64 FCL, 10 FCL	44238
4	one 5×5 filter, 2×2 Max pooling one 5×5 filter 64 FCL, 64 FCL, 10 FCL	11376
5	two 5×5 filters, 2×2 Max pooling two 5×5 filters 64 FCL, 64 FCL, 10 FCL	26600
6	four 5×5 filters, 2×2 Max pooling four 5×5 filters 64 FCL, 64 FCL, 10 FCL	31182

Firstly, an appropriate probability distribution is fit for high-dimensional data, which gives a higher probability for close data points and a lower probability for distant data points in terms of Euclidean distance. Secondly, a lower-dimensional probability distribution is built by minimizing the KL divergence between the high-dimension distribution and the low-dimension distribution. It can be seen that the outputs aimed to be used in the ensemble model come from different t-SNE distributions. When a representation for one of the classes is not received from an edge device, the missing representation should be filled on the server side, considering the distribution captured by the corresponding edge model. Hence, VAE models are trained and deployed on the server for each edge model to generate the values in a similar distribution space to the edge models.

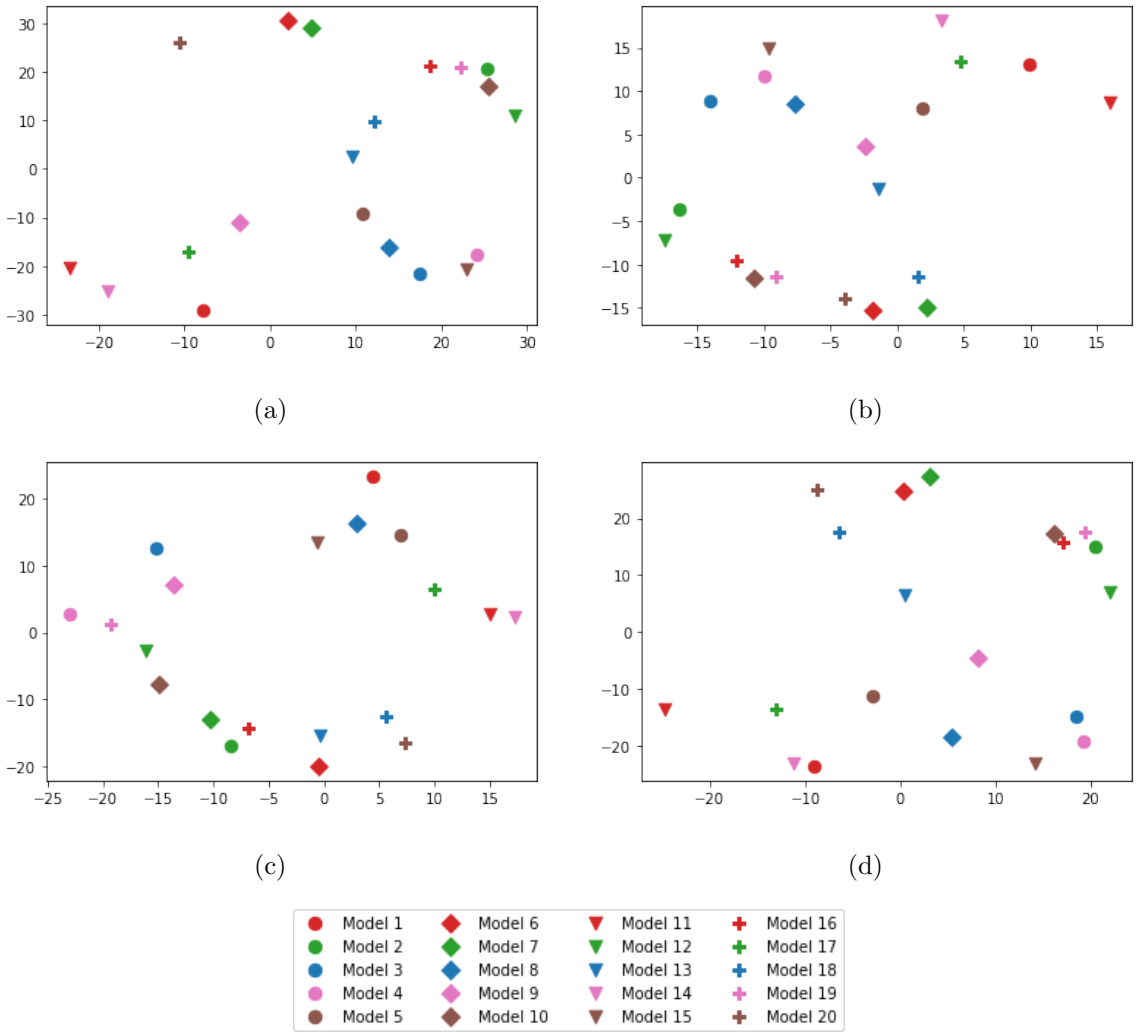


Figure 5.2. Two-dimensional t-distributed stochastic neighbour embedding (t-SNE) plots of the average feature vectors extracted from 20 edge models. The feature vectors are obtained from the training set of the CIFAR-10 dataset for four different classes such as (a) automobile, (b) bird, (c) dog, and (d) truck. Each marker represents individual models without any relation.

For experiments, 20 edge models are randomly selected among the model options given in Table 5.2. Also, the training data and training epochs are randomly selected for each edge model. The minimum, mean and maximum statistics of the training epoch, model accuracy, and the total number of model parameters are given in Table 5.4 separately for each classification dataset.

Table 5.4. Edge model training epochs, number of total parameters, and accuracy.

Dataset	**Number of Training Epochs**			**Number of Parameters (thousand)**			**Accuracy (%)**		
	Min	**Mean**	**Max**	**Min**	**Mean**	**Max**	**Min**	**Mean**	**Max**
CIFAR-10	11	32	49	7	25	44	10.36	37.46	51.37
CIFAR-100	10	25	46	13	25	50	4.53	8.43	17.23
MNIST	12	24	45	4	15	21	93.57	96.74	97.94
Fashion MNIST	10	32	47	4	18	21	71.76	79.91	90.16

It can be seen that the accuracy values of the edge models vary quite a lot because edge models differ in structure, and some edge models are trained for a few epochs. Due to their relatively small and shallow structures, the edge models do not produce state-of-the-art results independently. It should be noted that the accuracy is calculated based on the test data that the edge model can receive for all cases.

Moreover, the effectiveness of the proposed partial task-level streaming implemented on FPGA devices is presented. This experiment aims to show that partially parallelizing the training of certain layers of the deep learning model in a task-level pipelining manner accelerates the calculation while controlling resource utilization on edge.

The FPGA experiments are conducted on Xilinx Artix-7 AC701 Evaluation Platform using CIFAR 10 dataset. The synthesis and implementation of the system are done using Vitis HLS 2022.2 and Vivado 2022.2 platforms. The model is constructed as model type 2 in Table 5.2. In the experiments, different model layers are parallelized with the given parallelization factors. The training is done with minibatches of size 2. The results can be seen in Table 5.5.

Table 5.5. Latency and resource utilization effect of the proposed method.

Configuration	**Latency (ms)**	**BRAM (%)**	**DSP (%)**	**FF (%)**	**LUT (%)**	**LUTRAM (%)**	**IO (%)**
Without Optimizations	194	20	6	3	6	1	39
Proposed Method BF1-FF1,1	90	64	13	6	11	2	61
Proposed Method BF1-FF1,2	72	69	17	7	14	3	61
Proposed Method BF1-FF2,1	51	75	17	7	14	3	61
Proposed Method BF2-FF1,1	**46**	**76**	**22**	**9**	**18**	**4**	**86**

In Table 5.5, Regular Training is the training of the model without any task-level parallelism applied. In the Configuration column, BF denotes the parallelization factor of the minibatch dimension of the data, and FF denotes the parallelization factor of the filter dimension of the data. Therefore, BFx-FFy,z points to that minibatch are parallelized by factor x, the first convolutional operation is parallelized by factor y and the second convolutional operation is parallelized by factor z. The utilization values are taken from the post-implementation phase of the system. In addition, latency values are taken from the C/RTL Cosimulation results of the system. Experimental results show that task-level parallelism considerably accelerates the whole operation using any parallelization factors. However, parallelizing different parts of the operation with the same parallelization factor causes varying acceleration and resource utilization rates according to the data load of the part of the operation. The parallelization of the minibatch dimension brings the highest acceleration along with the highest resource utilization in DSP (Digital Signal Processor), FF (Flip-Flop), LUT (Lookup Table), LUTRAM, and IO (Input/Output Pins) resources. However, a similar acceleration can be achieved using different configurations while alleviating resource utilization. For example, the BF1-FF2,1 configuration results in approximately the same acceleration rate with less resource utilization. These results show that the proposed method can be configured for devices with differing hardware capacities.

5.2. VAE Models

In edge intelligence applications, edge devices may transfer missing information due to connection issues and lack of data. Therefore, the proposed method offers a solution to the missing value situation. As shown in Section 3.3, the proposed ensemble model is trained with the neural network layer outputs of the edge devices. The dimensionality of each edge output is designed to be the same. The experiments done in this study show that the outputs of the layers used in the ensemble model training have considerably different distributions.

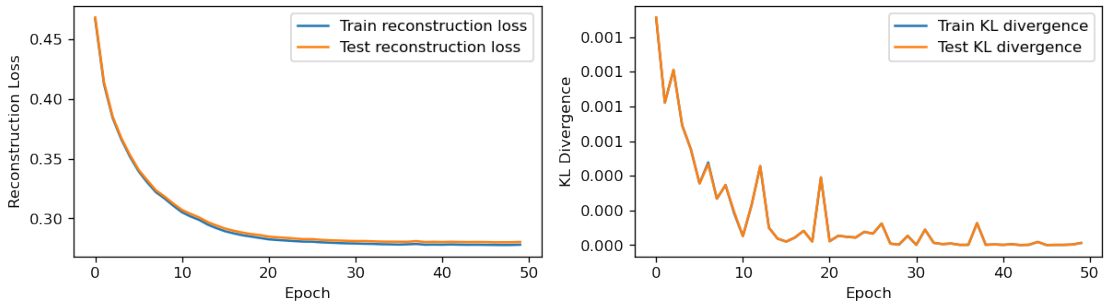
Table 5.6. The effect of replacing the missing feature vectors with different methods on accuracy on CIFAR-10.

Replacement Method	**Accuracy (%)**		
	$\boldsymbol{Min\% = 0.3}$ $\boldsymbol{MaxDisc\% = 0.2}$	$\boldsymbol{Min\% = 0.5}$ $\boldsymbol{MaxDisc\% = 0.5}$	$\boldsymbol{Min\% = 0.7}$ $\boldsymbol{MaxDisc\% = 0}$
With Zero	62	58	58
With Mean	52	54	57
With Max	35	42	52
With VAE	**82**	**73**	**63**

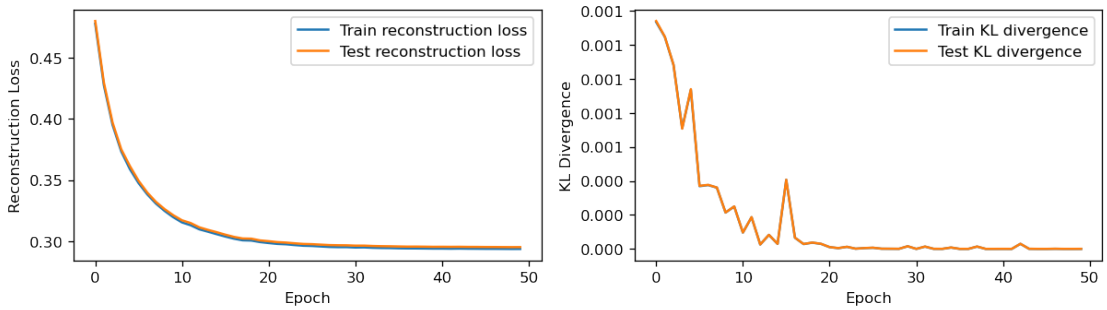
The importance of replacing the missing feature vectors with appropriate values can be also seen in Table 5.6. The results show that replacing the missing feature vectors with trivial values such as zero, the mean, and the maximum element of the successfully transferred feature vectors detriments the overall accuracy of the ensemble model. It can be seen that replacing the feature vectors with the proposed method produces remarkably more accurate results in terms of overall ensemble accuracy.

Moreover, it is crucial to assess the generation performance of the VAE model. For this purpose, the reconstruction loss and the KL divergence plot are assessed during the training of VAE models. The plots are given in Figure 5.3 for $Min\% = 0.05$ and

$MaxDisc\% = 0.5$. It can be seen that reconstruction loss and KL divergence decrease over time during training. Hence, the VAE models generate more accurate feature vectors as the training continues for training and test datasets.



(a)



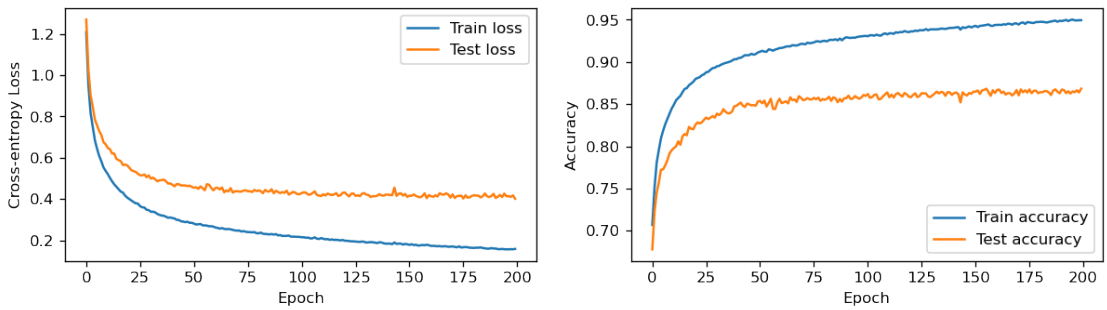
(b)

Figure 5.3. The loss plots of VAE models trained for (a) CIFAR-10 and (b) CIFAR-100 datasets. The plots overlap in the parts where both are not visible.

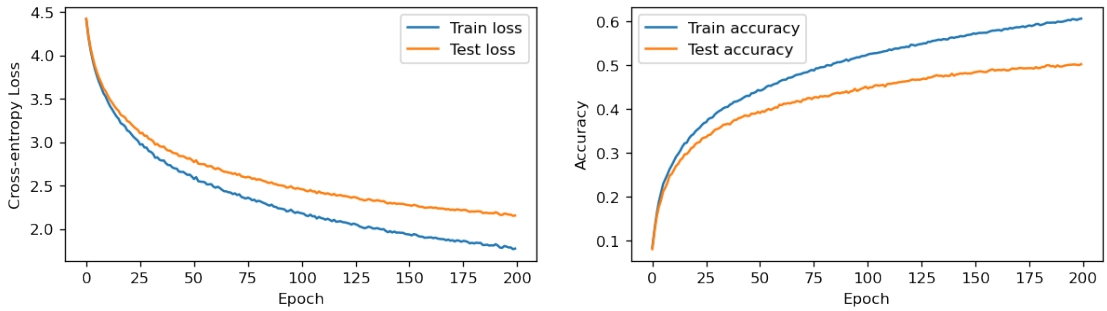
5.3. Ensemble Model

5.3.1. Experiments on Accuracy

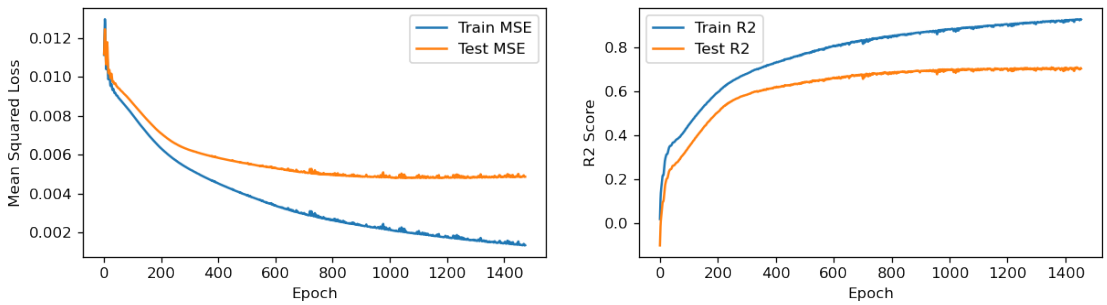
Examining the loss and accuracy plots of the training of the proposed ensemble model is essential for convergence analysis. In Figure 5.5, the accuracy of the ensemble model for different choices of $Min\%$ and $MaxDisc\%$ values and training scenarios are given for all image classification datasets. It should be noted that the accuracy is calculated based on the whole test data for all cases.



(a)



(b)



(c)

Figure 5.4. The loss plots of the ensemble model trained for (a) CIFAR-10, (b) CIFAR-100 and (c) Pecan Street datasets with $Min\% = 0.05$ and $MaxDisc\% = 0.5$.

The figure shows that the ensemble accuracy increases as the $Min\%$ value decreases and the $MaxDisc\%$ value increases. The ensemble model performs better when some edge models are trained with data containing samples of a limited number of classes. Moreover, it performs even better when the class data percentage of the test data is similar to the training data. In a real-world example, the proposed ensemble model is expected to perform better when the edge models are specifically trained with data

obtained from a specific point of a data source, for example, the different lanes of a highway. Heavier vehicles generally move in the right lane, whereas faster and lighter vehicles tend to move in the left lane. It is expected to receive a similar class data percentage for both training and live use later, which is the situation in which the proposed method performs best. In addition, Figure 5.5 shows that the accuracy of the ensemble model is limited to the edge model accuracy. Considering the average edge model accuracy given in Table 5.4, it can be deduced that the higher the edge model accuracy, the higher the ensemble model accuracy.

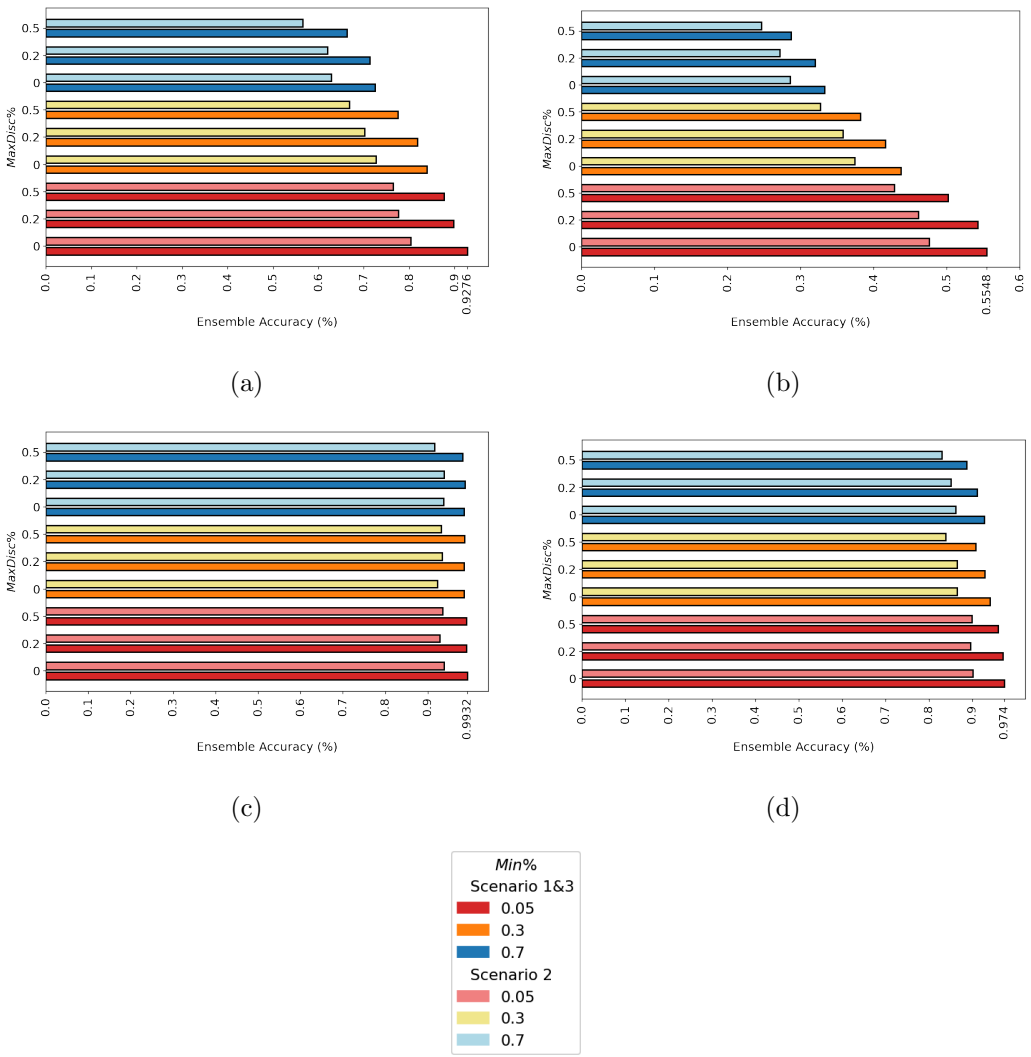


Figure 5.5. Ensemble accuracy for different training scenarios and $Min\%$, $MaxDisc\%$ values. (a) CIFAR-10, (b) CIFAR-100, (c) MNIST, (d) Fashion MNIST.

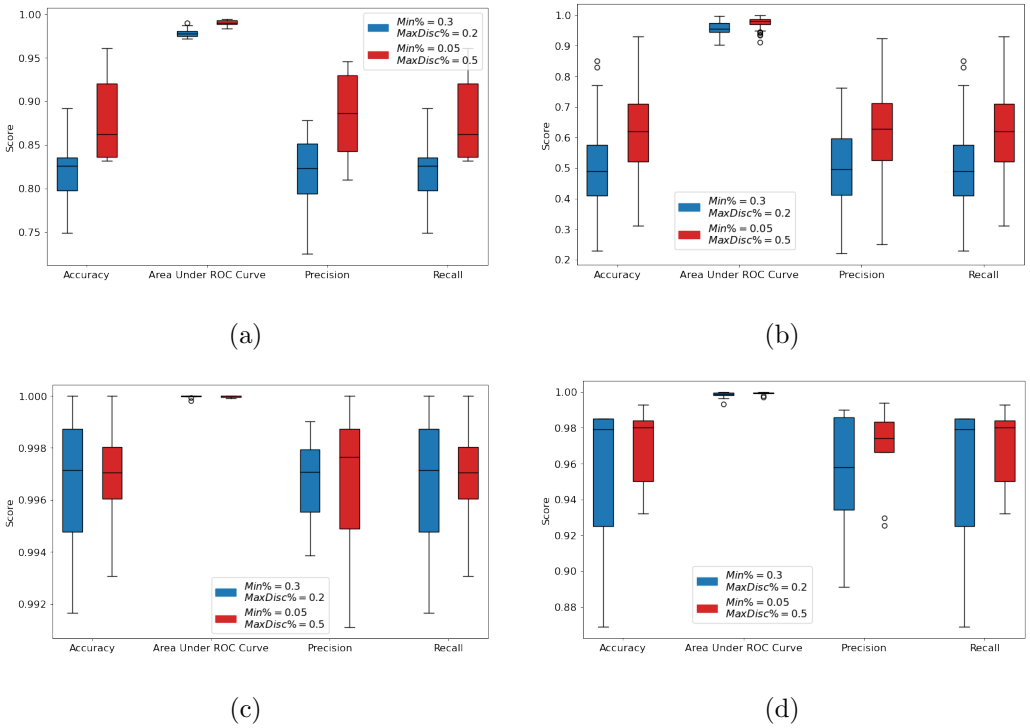


Figure 5.6. Performance metrics of the ensemble model on different classes. (a) CIFAR-10, (b) CIFAR-100, (c) MNIST, (d) Fashion MNIST.

Also, several extra performance metrics are given in Figure 5.6 for all image classification datasets. The figure shows the box plot of the performance metrics of accuracy, the area under the ROC curve, precision, and recall values calculated for each class in test data. The results seem to support the overall ensemble accuracy for given $Min\%$ and $MaxDisc\%$ values. Moreover, it can be seen that the ensemble model produces similar performance values for all classes for better-performing edge models. Even though the performances are not state-of-the-art for the edge models, the ensemble model performs at a similar level for all classes, even when the edge model does not receive the same amount of data for all classes in the training phase.

The effect of changing the number of edge models and the transferred feature vector size L_{com} is also investigated in Table 5.7. It can be seen that increasing the number of edge models also increases the final ensemble accuracy. The same effect for changing the L_{com} can be observed but the increase ceases to be material after a certain point of L_{com} value, which is after 128 for this example. For changing the number of

edge models, L_{com} is taken as 64; for changing L_{com}, the number of edge models is taken as 20 in all cases.

Table 5.7. Effect of changing the number of edge models and size of transferred feature vector on test accuracy.

Dataset	**10 Edge Models**	**20 Edge Models**	**30 Edge Models**	**50 Edge Models**
CIFAR-10	80.80%	92.76%	96.24%	98.79%
CIFAR-100	41.60%	55.48%	64.10%	75.77%
Dataset	$\boldsymbol{L_{com} = 32}$	$\boldsymbol{L_{com} = 64}$	$\boldsymbol{L_{com} = 128}$	$\boldsymbol{L_{com} = 256}$
CIFAR-10	90.89%	92.76%	93.33%	93.47%
CIFAR-100	51.77%	55.48%	67.37%	67.40%

The proposed method is also compared with recent edge intelligence studies regarding final accuracy for image classification problems. The comparison results can be seen in Table 5.8. All of the values are taken as the best-reported values from the studies. Table 5.8 shows that the proposed method gives state-of-the-art accuracy for image classification problems in an edge intelligence setting. Moreover, using less accurate edge models, the proposed method results in state-of-the-art accuracy. For example, the edge models used in [74] give results with 37.67% and 78.94% accuracy on CIFAR-100 and CIFAR-10, respectively. Table 5.4 shows that higher final accuracy is obtained using the proposed method for both datasets using edge models with an average of 8.43% and 37.46% accuracy models, respectively.

The proposed method is also compared with recent edge intelligence studies regarding certain performance metrics for regression problems. The comparison results can be seen in Table 5.9. The metric values are taken as the best-reported values from all of the works. It can be seen that the proposed method produces more accurate predictions for regression problems as well compared to recent works.

Table 5.8. Accuracy comparison with other methods for image classification datasets.

Method	**CIFAR-10**	**CIFAR-100**	**MNIST**	**Fashion MNIST**
FedAvg [64]	81.72%	-	95.04%	94.50%
FedProx [65]	83.25%	-	96.26%	94.53%
FedDyn [66]	85.19%	53.27%	-	-
SCAFFOLD [67]	85.99%	53.32%	-	-
FedGen [68]	83.91%	50.38%	-	-
FedProto [71]	84.49%	-	97.13%	97.10%
FedFTG [69]	87.34%	56.94%	-	-
FedDC [70]	86.18%	55.52%	98.45%	-
Bayesian Nonparametric [72]	45.80%	-	97.80%	-
AE-KD [87]	93.01%	72.36%	-	-
EKD [88]	92.33%	67.78%	-	-
FedGKT [74]	92.97%	69.57%	-	-
Majority Voting	42.30%	14.06%	93.12%	82.74%
Average Voting	43.26%	17.62%	93.46%	82.98%
Proposed Method	**98.79%**	**75.77%**	**99.32%**	**97.40%**

5.3.2. Evaluation in Terms of Execution Time

In Table 5.10, the time requirements for the training and inference of the edge models are given in average for different $Min\%$ values. It can be seen that the edge models are trained in a shorter time when they are trained with less amount of data. In the whole training process for the proposed method, the training of the edge models takes the most time due to the hardware constraints of the edge devices.

Table 5.9. Accuracy comparison with other methods for regression datasets.

Work	**Dataset**	**Metric**	**Reported Value**	**Proposed Method**
SAFA [102]	Boston Housing	Accuracy [102] (higher is better)	0.643	**0.728**
FedAG [103]	Boston Housing	RMSE (lower is better)	4.070	**3.479**
PrivFL [104]	Boston Housing	MAE (lower is better)	3.266	**2.788**
Tiresias [105]	California Housing	R2 Score (higher is better)	0.717	**0.726**
Sherpa [106]	California Housing	R2 Score (higher is better)	0.502	**0.868**
DER Forecast [107]	Pecan Street	RMSE (lower is better)	1.98	**0.63**

Table 5.10. Average execution time for edge model training and inference.

$\boldsymbol{Min}\%$ **Value**	$\boldsymbol{T^{EM}_{train}}$ **(minute)**	$\boldsymbol{T^{EM}_{inference}}$ **(minute)**
0.05	345	3
0.3	402	4
0.7	500	5

In Figure 5.7, the time requirements for data transfer, VAE, and ensemble model training are shown. The results are invariant with the dataset used for training. The figure shows that the total transfer time for an edge model is the most for Scenario 3 as a minibatch is repeatedly transferred to the server. The transfer time is less in Scenario 2 due to the repeated use of the same minibatch to complete the ensemble training epochs. Transfer time is the least for Scenario 1 because the data is sent to the server once. The other time requirements do not remarkably differ by the scenario;

nonetheless, the ensemble learning time slightly increases from Scenario 1 to Scenario 3 due to repeated VAE decoding operations as the same minibatch is received by the server multiple times. It can also be seen that the times of transfer and VAE training visibly decrease for decreasing $Min\%$ values as expected. The execution time increases from Scenario 1 to Scenario 3 can be verified by the total number of communications for each edge model to the server.

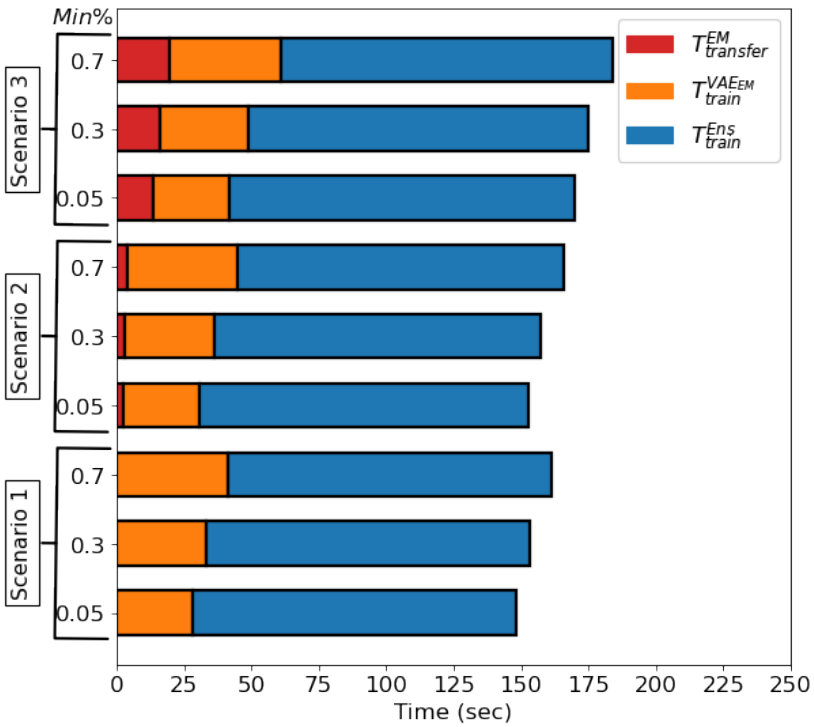


Figure 5.7. Overall latency for feature vector transfer, VAE training, and ensemble training.

The total number of communications can be seen in Table 5.11 for all training scenarios. It can be seen that Scenario 3 requires the most number of communications for each edge model. This requirement also increases the total time spent on data transfer operations. End-to-end latency is approximately 0.05 seconds for inference for all training scenarios.

Table 5.11. Total number of communications for an edge model.

Scenario	N_{com}^{EM}
Scenario 1	1
Scenario 2	7813
Scenario 3	39063

5.3.3. Evaluation in Terms of Memory Requirement

In Figure 5.8, the cumulative amount of data transferred to the server is shown. The figure shows that most data is transferred to the server for Scenario 3 due to repeated transfers. Scenario 2 seems to be placed between, and less cumulative data is transferred in Scenario 1 due to a one-time transfer.

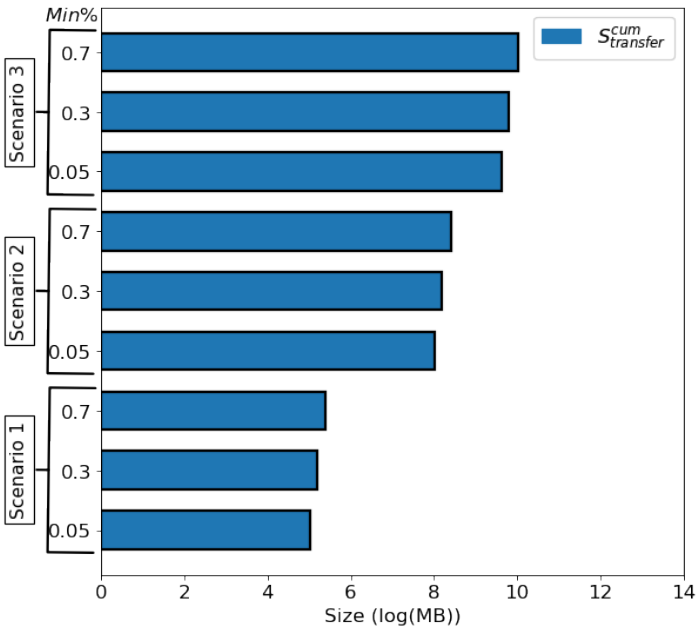


Figure 5.8. Total cumulative transferred data amount from all edge models.

Table 5.12 shows the other memory requirement values for the edge models and the server by training scenarios and $Min\%$ values. The table shows that the most amount of memory requirement is needed for Scenario 1 because the whole training data of each edge model are transferred to the server once and stored there. However,

the required memory remarkably reduces when the system is trained with Scenario 2 and Scenario 3. Scenario 3 requires no memory space allocated for the transferred data, whereas Scenario 2 requires it for only a minibatch.

Table 5.12. Memory requirements on edge devices and server.

Scenario	$Min\%$	$M^{EM}_{transfer}$ **(MB)**	M^{server}_{memory} **(MB)**	M^{EM}_{memory} **(MB)**
Scenario 1	0.05	7.7	256.0	7.7
	0.3	9.0		9.0
	0.7	11.1		11.1
Scenario 2	All	0.03	0.7	0.03
Scenario 3			0	

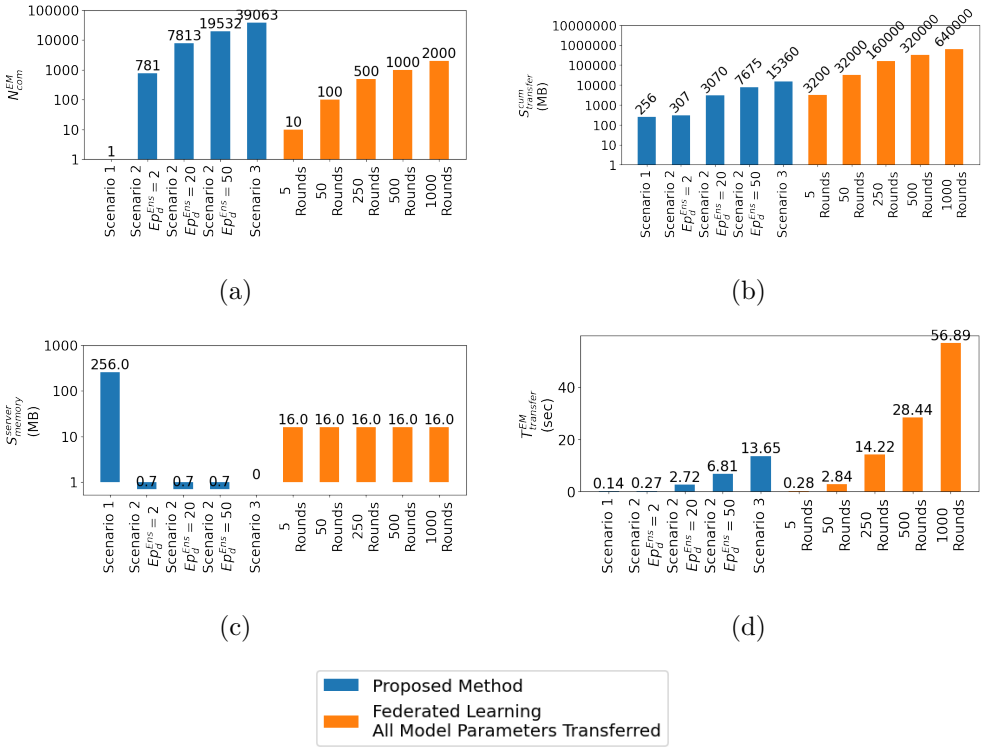


Figure 5.9. Comparison of the proposed method with an equivalent federated learning approach (a) the number of communication (N^{EM}_{com}), (b) the cumulative amount of data transfer ($S^{cum}_{transfer}$), (c) the total required amount of memory (S^{server}_{memory}) and (d) the total required time for data transfer ($T^{EM}_{transfer}$).

In Figure 5.9, the proposed method's communication, data transfer and timing requirements are compared to federated learning methods in which all model parameters are transferred. Studies such as [68, 70, 71] allocate models with approximately 200k parameters to the edge devices, so the comparisons are based on 200k parameters on the edge models. The rest of the values used in the calculations are given in Table 5.1. It can be seen that training the proposed method in Scenario 1 yields quite less communication compared to federated learning examples as the system is trained within a one-way communication in Scenario 1. However, Scenario 1 requires considerably more memory requirement on the server. Scenario 3 brings the least memory requirement on the server; nevertheless, it requires network communication more than federated learning. It can be seen that Scenario 2 requires less memory on the server, and the communication it requires is less than federated learning in different settings of the Ep_d^{Ens} parameter.

6. CONCLUSION

This thesis proposes a hybrid training method for Deep Edge Intelligence with a convolutional ensemble learning approach. The proposed method aims to boost the overall system accuracy by developing an ensemble model on the server side when edge models are used on edge devices. The experiments point out that the ensemble accuracy shows a remarkable increase which outperforms some other popular federated learning and ensemble learning methods used in distributed learning of edge intelligence.

In addition, the proposed method is implemented on Xilinx FPGA devices in the study. FPGA devices offer advantageous attributes such as energy efficiency and reconfigurability in computing. Therefore, a customizable training method for DNNs is proposed in this study. The proposed method allows the training to be parallelized with manually selected factors that change hardware requirements; hence, the training can be accelerated using the method with different parallelization factors for any target device desired.

Moreover, the proposed method offers a solution to information transfer failure problems from the edge to the main server. Information from the edge to the server may not be successfully transferred for several reasons, such as the lack of data the edge devices acquire to process and transfer to the server and transfer failures due to technical reasons. The proposed method generates the required information on the server by mimicking the successfully sent information vectors. Experiments show that the distribution of generated feature vectors converges to the distribution of the successfully sent feature vectors from the edge. Also, the generated feature vectors remarkably contribute to the overall accuracy.

Last but not least, the proposed method can be trained with three scenarios according to the system's hardware capacity. The training of the system can be done in one communication when certain requirements are met, unlike most of the

popular federated learning algorithms requiring an excessive number of communications. Nevertheless, the proposed method makes the training of the system available for devices with differing hardware capacities thanks to different learning scenarios with various tradeoffs.

The proposed method assumes that the server knows which edge model is trained with which subset of a general training dataset. Developments might enhance the method assuming that the server does not have such knowledge. Moreover, the method assumes that all edge models contain an FCL with the same dimensionality whose output is transferred to the server. The method might be generalized for any model structure that is allowed to be used for the edge models.

REFERENCES

1. Krasniqi, X. and E. Hajrizi, "Use of IoT Technology to Drive the Automotive Industry From Connected to Full Autonomous Vehicles", *IFAC-PapersOnLine*, Vol. 49, No. 29, pp. 269–274, 2016.

2. Khan, W. Z., M. Y. Aalsalem, M. K. Khan, M. S. Hossain and M. Atiquzzaman, "A Reliable Internet of Things Based Architecture for Oil and Gas Industry", *2017 19th International Conference on Advanced Communication Technology*, pp. 705–710, IEEE, 2017.

3. Misra, N., Y. Dixit, A. Al-Mallahi, M. S. Bhullar, R. Upadhyay and A. Martynenko, "IoT, Big Data and Artificial Intelligence in Agriculture and Food Industry", *IEEE Internet of Things Journal*, Vol. 9, No. 9, pp. 6305–6324, 2020.

4. Hodo, E., X. Bellekens, A. Hamilton, P.-L. Dubouilh, E. Iorkyase, C. Tachtatzis and R. Atkinson, "Threat Analysis of IoT Networks Using Artificial Neural Network Intrusion Detection System", *2016 International Symposium on Networks, Computers and Communications*, pp. 1–6, IEEE, 2016.

5. Mendez, J., K. Bierzynski, M. Cuéllar and D. P. Morales, "Edge Intelligence: Concepts, Architectures, Applications and Future Directions", *ACM Transactions on Embedded Computing Systems (TECS)*, Vol. 21, No. 5, pp. 1–41, 2022.

6. Szummer, M. and R. W. Picard, "Indoor-Outdoor Image Classification", *Proceedings 1998 IEEE International Workshop on Content-Based Access of Image and Video Database*, pp. 42–51, IEEE, 1998.

7. Wang, F., M. Jiang, C. Qian, S. Yang, C. Li, H. Zhang, X. Wang and X. Tang, "Residual Attention Network for Image Classification", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3156–3164, 2017.

8. Chen, C.-F. R., Q. Fan and R. Panda, "Crossvit: Cross-Attention Multi-Scale Vision Transformer for Image Classification", *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 357–366, 2021.

9. Deng, L., G. Hinton and B. Kingsbury, "New Types of Deep Neural Network Learning for Speech Recognition and Related Applications: An Overview", *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8599–8603, IEEE, 2013.

10. Dahl, G. E., D. Yu, L. Deng and A. Acero, "Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition", *IEEE Transactions on Audio, Speech, and Language Processing*, Vol. 20, No. 1, pp. 30–42, 2011.

11. Hinton, G., L. Deng, D. Yu, G. E. Dahl, A.-R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen and T. N. Sainath, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups", *IEEE Signal Processing Magazine*, Vol. 29, No. 6, pp. 82–97, 2012.

12. Collobert, R. and J. Weston, "A Unified Architecture for Natural Language Processing: Deep Neural Networks With Multitask Learning", *Proceedings of the 25th International Conference on Machine Learning*, pp. 160–167, 2008.

13. Goldberg, Y., "Neural Network Methods for Natural Language Processing", *Synthesis Lectures on Human Language Technologies*, Vol. 10, No. 1, pp. 1–309, 2017.

14. Young, T., D. Hazarika, S. Poria and E. Cambria, "Recent Trends in Deep Learning Based Natural Language Processing", *IEEE Computational Intelligence Magazine*, Vol. 13, No. 3, pp. 55–75, 2018.

15. Ambrogio, S., P. Narayanan, H. Tsai, R. M. Shelby, I. Boybat, C. Di Nolfo, S. Sidler, M. Giordano, M. Bodini and N. C. Farinha, "Equivalent-Accuracy

Accelerated Neural-Network Training Using Analogue Memory", *Nature*, Vol. 558, No. 7708, pp. 60–67, 2018.

16. Iandola, F. N., M. W. Moskewicz, K. Ashraf and K. Keutzer, "Firecaffe: Near-Linear Acceleration of Deep Neural Network Training on Compute Clusters", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2592–2600, 2016.

17. Sohoni, N. S., C. R. Aberger, M. Leszczynski, J. Zhang and C. Ré, "Low-Memory Neural Network Training: A Technical Report", ArXiv:1904.10631, 2019.

18. Liu, J., H. Zhao, M. A. Ogleari, D. Li and J. Zhao, "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach", *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 655–668, IEEE, 2018.

19. Kumar, R., R. Aggarwal and J. Sharma, "Energy Analysis of a Building Using Artificial Neural Network: A Review", *Energy and Buildings*, Vol. 65, pp. 352–358, 2013.

20. Patterson, D., J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier and J. Dean, "Carbon Emissions and Large Neural Network Training", ArXiv:2104.10350, 2021.

21. Ren, J., G. Yu, Y. He and G. Y. Li, "Collaborative Cloud and Edge Computing for Latency Minimization", *IEEE Transactions on Vehicular Technology*, Vol. 68, No. 5, pp. 5031–5044, 2019.

22. Kiani, A. and N. Ansari, "Edge Computing Aware NOMA for 5G Networks", *IEEE Internet of Things Journal*, Vol. 5, No. 2, pp. 1299–1306, 2018.

23. Hassani, A., S. Walton, N. Shah, A. Abuduweili, J. Li and H. Shi, "Escaping the Big Data Paradigm With Compact Transformers", ArXiv:2104.05704, 2021.

24. Tan, M. and Q. Le, "Efficientnet: Rethinking Model Scaling for Convolutional Neural Networks", *International Conference on Machine Learning*, pp. 6105–6114, PMLR, 2019.

25. Huang, G., Z. Liu, L. Van Der Maaten and K. Q. Weinberger, "Densely Connected Convolutional Networks", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4700–4708, 2017.

26. Zhu, C., R. Ni, Z. Xu, K. Kong, W. R. Huang and T. Goldstein, "Gradinit: Learning to Initialize Neural Networks for Stable and Efficient Training", *Advances in Neural Information Processing Systems*, Vol. 34, pp. 16410–16422, 2021.

27. Wightman, R., H. Touvron and H. Jégou, "Resnet Strikes Back: An Improved Training Procedure in TIMM", ArXiv:2110.00476, 2021.

28. Krizhevsky, A., V. Nair and G. Hinton, "The CIFAR-10 Dataset", 2014, `http://www.cs.toronto.edu/kriz/cifar.html`, accessed on November 21, 2021.

29. Nguyen, D. C., M. Ding, Q.-V. Pham, P. N. Pathirana, L. B. Le, A. Seneviratne, J. Li, D. Niyato and H. V. Poor, "Federated Learning Meets Blockchain in Edge Computing: Opportunities and Challenges", *IEEE Internet of Things Journal*, Vol. 8, No. 16, pp. 12806–12825, 2021.

30. Jiang, Y., S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung and L. Tassiulas, "Model Pruning Enables Efficient Federated Learning on Edge Devices", ArXiv:1909.12326, 2019.

31. Vahidian, S., M. Morafah and B. Lin, "Personalized Federated Learning by Structured and Unstructured Pruning Under Data Heterogeneity", *2021 IEEE 41st International Conference on Distributed Computing Systems Workshops*, pp. 27–34, IEEE, 2021.

32. Sattler, F., S. Wiedemann, K.-R. Müller and W. Samek, "Robust and Communication-Efficient Federated Learning From Non-IID Data", *IEEE Transactions on Neural Networks and Learning Systems*, Vol. 31, No. 9, pp. 3400–3413, 2019.

33. Haddadpour, F., M. M. Kamani, A. Mokhtari and M. Mahdavi, "Federated Learning With Compression: Unified Analysis and Sharp Guarantees", *International Conference on Artificial Intelligence and Statistics*, pp. 2350–2358, PMLR, 2021.

34. Reisizadeh, A., A. Mokhtari, H. Hassani, A. Jadbabaie and R. Pedarsani, "Fedpaq: A Communication-Efficient Federated Learning Method With Periodic Averaging and Quantization", *International Conference on Artificial Intelligence and Statistics*, pp. 2021–2031, PMLR, 2020.

35. Shlezinger, N., M. Chen, Y. C. Eldar, H. V. Poor and S. Cui, "UVeQFed: Universal Vector Quantization for Federated Learning", *IEEE Transactions on Signal Processing*, Vol. 69, pp. 500–514, 2020.

36. Wang, S., T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He and K. Chan, "Adaptive Federated Learning in Resource Constrained Edge Computing Systems", *IEEE Journal on Selected Areas in Communications*, Vol. 37, No. 6, pp. 1205–1221, 2019.

37. Yang, Z., M. Chen, W. Saad, C. S. Hong and M. Shikh-Bahaei, "Energy Efficient Federated Learning Over Wireless Communication Networks", *IEEE Transactions on Wireless Communications*, Vol. 20, No. 3, pp. 1935–1949, 2020.

38. Li, T., A. K. Sahu, A. Talwalkar and V. Smith, "Federated Learning: Challenges, Methods, and Future Directions", *IEEE Signal Processing Magazine*, Vol. 37, No. 3, pp. 50–60, 2020.

39. Bonawitz, K., H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov,

C. Kiddon, J. Konečnỳ, S. Mazzocchi and B. McMahan, "Towards Federated Learning at Scale: System Design", *Proceedings of Machine Learning and Systems*, Vol. 1, pp. 374–388, 2019.

40. Li, Q., Z. Wen, Z. Wu, S. Hu, N. Wang, Y. Li, X. Liu and B. He, "A Survey on Federated Learning Systems: Vision, Hype and Reality for Data Privacy and Protection", ArXiv:1907.09693, 2019.

41. Zhou, Z., X. Chen, E. Li, L. Zeng, K. Luo and J. Zhang, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing", *Proceedings of the IEEE*, Vol. 107, No. 8, pp. 1738–1762, 2019.

42. Xu, D., T. Li, Y. Li, X. Su, S. Tarkoma, T. Jiang, J. Crowcroft and P. Hui, "Edge Intelligence: Architectures, Challenges, and Applications", ArXiv:2003.12172, 2020.

43. Li, E., Z. Zhou and X. Chen, "Edge Intelligence: On-Demand Deep Learning Model Co-Inference With Device-Edge Synergy", *Proceedings of the 2018 Workshop on Mobile Edge Communications*, pp. 31–36, 2018.

44. Ko, J. H., T. Na, M. F. Amir and S. Mukhopadhyay, "Edge-Host Partitioning of Deep Neural Networks With Feature Space Encoding for Resource-Constrained Internet-of-Things Platforms", *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance*, pp. 1–6, IEEE, 2018.

45. Teerapittayanon, S., B. McDanel and H.-T. Kung, "Branchynet: Fast Inference via Early Exiting From Deep Neural Networks", *2016 23rd International Conference on Pattern Recognition*, pp. 2464–2469, IEEE, 2016.

46. Biookaghazadeh, S., M. Zhao and F. Ren, "Are FPGAs Suitable for Edge Computing?", ArXiv:1804.06404, 2018.

47. Hao, C., X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu and

D. Chen, "FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge", *2019 56th ACM/IEEE Design Automation Conference*, pp. 1–6, IEEE, 2019.

48. Xia, M., Z. Huang, L. Tian, H. Wang, V. Chang, Y. Zhu and S. Feng, "SparkNoC: An Energy-Efficiency FPGA-based Accelerator Using Optimized Lightweight CNN for Edge Computing", *Journal of Systems Architecture*, Vol. 115, p. 101991, 2021.

49. Liu, X., J. Yang, C. Zou, Q. Chen, X. Yan, Y. Chen and C. Cai, "Collaborative Edge Computing With FPGA-based CNN Accelerators for Energy-Efficient and Time-Aware Face Tracking System", *IEEE Transactions on Computational Social Systems*, Vol. 9, No. 1, pp. 252–266, 2021.

50. Zhang, X., Y. Wang, S. Lu, L. Liu and W. Shi, "OpenEI: An Open Framework for Edge Intelligence", *2019 IEEE 39th International Conference on Distributed Computing Systems*, pp. 1840–1851, IEEE, 2019.

51. Tsoi, K. H. and W. Luk, "Axel: A Heterogeneous Cluster With FPGAs and GPUs", *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 115–124, 2010.

52. Carballo-Hernández, W., M. Pelcat and F. Berry, "Why Is FPGA-GPU Heterogeneity the Best Option for Embedded Deep Neural Networks?", ArXiv:2102.01343, 2021.

53. Tu, Y., S. Sadiq, Y. Tao, M.-L. Shyu and S.-C. Chen, "A Power Efficient Neural Network Implementation on Heterogeneous FPGA and GPU Devices", *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science*, pp. 193–199, IEEE, 2019.

54. Liu, X., J. Yang, C. Zou, Q. Chen, X. Yan, Y. Chen and C. Cai, "Collaborative Edge Computing With FPGA-based CNN Accelerators for Energy-Efficient and

Time-Aware Face Tracking System", *IEEE Transactions on Computational Social Systems*, Vol. 9, No. 1, pp. 252–266, 2021.

55. Chitty-Venkata, K. T. and A. K. Somani, "Array Aware Training/Pruning: Methods for Efficient Forward Propagation on Array-Based Neural Network Accelerators", *2020 IEEE 31st International Conference on Application-Specific Systems, Architectures and Processors*, pp. 37–44, IEEE, 2020.

56. Arora, A., Z. Wei and L. K. John, "Hamamu: Specializing FPGAs for ML Applications by Adding Hard Matrix Multiplier Blocks", *2020 IEEE 31st International Conference on Application-Specific Systems, Architectures and Processors*, pp. 53–60, IEEE, 2020.

57. Courbariaux, M., 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", ArXiv:1602.02830, 2016.

58. Blott, M., T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser and K. Vissers, "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks", *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, Vol. 11, No. 3, pp. 1–23, 2018.

59. Zhu, F., R. Gong, F. Yu, X. Liu, Y. Wang, Z. Li, X. Yang and J. Yan, "Towards Unified Int8 Training for Convolutional Neural Network", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1969–1979, 2020.

60. Sikdokur, I., I. Baytas and A. Yurdakul, "Image Classification on Accelerated Neural Networks", ArXiv:2203.11081, 2022.

61. Ahmad, A. and M. A. Pasha, "Optimizing Hardware Accelerated General Matrix-Matrix Multiplication for CNNs on FPGAs", *IEEE Transactions on Circuits and*

Systems II: Express Briefs, Vol. 67, No. 11, pp. 2692–2696, 2020.

62. Luo, C., M.-K. Sit, H. Fan, S. Liu, W. Luk and C. Guo, "Towards Efficient Deep Neural Network Training by FPGA-based Batch-Level Parallelism", *Journal of Semiconductors*, Vol. 41, No. 2, p. 022403, 2020.

63. Konečnỳ, J., H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh and D. Bacon, "Federated Learning: Strategies for Improving Communication Efficiency", ArXiv:1610.05492, 2016.

64. McMahan, B., E. Moore, D. Ramage, S. Hampson and B. A. Arcas, "Communication-Efficient Learning of Deep Networks From Decentralized Data", *Artificial Intelligence and Statistics*, pp. 1273–1282, PMLR, 2017.

65. Li, T., A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar and V. Smith, "Federated Optimization in Heterogeneous Networks", *Proceedings of Machine Learning and Systems*, Vol. 2, pp. 429–450, 2020.

66. Acar, D. A. E., Y. Zhao, R. M. Navarro, M. Mattina, P. N. Whatmough and V. Saligrama, "Federated Learning Based on Dynamic Regularization", ArXiv:2111.04263, 2021.

67. Karimireddy, S. P., S. Kale, M. Mohri, S. Reddi, S. Stich and A. T. Suresh, "Scaffold: Stochastic Controlled Averaging for Federated Learning", *International Conference on Machine Learning*, pp. 5132–5143, PMLR, 2020.

68. Zhu, Z., J. Hong and J. Zhou, "Data-Free Knowledge Distillation for Heterogeneous Federated Learning", ArXiv:2105.10056, 2021.

69. Zhang, L., L. Shen, L. Ding, D. Tao and L.-Y. Duan, "Fine-Tuning Global Model via Data-Free Knowledge Distillation for Non-IID Federated Learning", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10174–10183, 2022.

70. Gao, L., H. Fu, L. Li, Y. Chen, M. Xu and C.-Z. Xu, "FedDC: Federated Learning With Non-IID Data via Local Drift Decoupling and Correction", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10112–10121, 2022.

71. Tan, Y., G. Long, L. Liu, T. Zhou, Q. Lu, J. Jiang and C. Zhang, "Fedproto: Federated Prototype Learning Across Heterogeneous Clients", *AAAI Conference on Artificial Intelligence*, Vol. 1, p. 3, 2022.

72. Yurochkin, M., M. Agarwal, S. Ghosh, K. Greenewald, N. Hoang and Y. Khazaeni, "Bayesian Nonparametric Federated Learning of Neural Networks", *International Conference on Machine Learning*, pp. 7252–7261, PMLR, 2019.

73. Fang, X. and M. Ye, "Robust Federated Learning With Noisy and Heterogeneous Clients", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10072–10081, 2022.

74. He, C., M. Annavaram and S. Avestimehr, "Group Knowledge Transfer: Federated Learning of Large CNNS at the Edge", *Advances in Neural Information Processing Systems*, Vol. 33, pp. 14068–14080, 2020.

75. Shen, S., M. Sadoughi, M. Li, Z. Wang and C. Hu, "Deep Convolutional Neural Networks With Ensemble Learning and Transfer Learning for Capacity Estimation of Lithium-Ion Batteries", *Applied Energy*, Vol. 260, p. 114296, 2020.

76. Alam, K. M., N. Siddique and H. Adeli, "A Dynamic Ensemble Learning Algorithm for Neural Networks", *Neural Computing and Applications*, Vol. 32, No. 12, pp. 8675–8690, 2020.

77. Lin, C.-C., D.-J. Deng, C.-H. Kuo and L. Chen, "Concept Drift Detection and Adaption in Big Imbalance Industrial IoT Data Using an Ensemble Learning Method of Offline Classifiers", *IEEE Access*, Vol. 7, pp. 56198–56207, 2019.

78. Verma, A. and V. Ranga, "ELNIDS: Ensemble Learning Based Network Intrusion Detection System for RPL Based Internet of Things", *2019 4th International Conference on Internet of Things: Smart Innovation and Usages*, pp. 1–6, IEEE, 2019.

79. Wang, S. and X. Yao, "Using Class Imbalance Learning for Software Defect Prediction", *IEEE Transactions on Reliability*, Vol. 62, No. 2, pp. 434–443, 2013.

80. Niculescu-Mizil, A. and R. Caruana, "Predicting Good Probabilities With Supervised Learning", *Proceedings of the 22nd International Conference on Machine Learning*, pp. 625–632, 2005.

81. Caruana, R. and A. Niculescu-Mizil, "An Empirical Comparison of Supervised Learning Algorithms", *Proceedings of the 23rd International Conference on Machine Learning*, pp. 161–168, 2006.

82. Hamm, J. and D. D. Lee, "Grassmann Discriminant Analysis: A Unifying View on Subspace-Based Learning", *Proceedings of the 25th International Conference on Machine Learning*, pp. 376–383, 2008.

83. Seiffert, C., T. M. Khoshgoftaar, J. Van Hulse and A. Napolitano, "RUSBoost: A Hybrid Approach to Alleviating Class Imbalance", *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, Vol. 40, No. 1, pp. 185–197, 2009.

84. Breiman, L., "Bagging Predictors", *Machine Learning*, Vol. 24, No. 2, pp. 123–140, 1996.

85. Schapire, R. E., "A Brief Introduction to Boosting", *International Joint Conference on Artificial Intelligence*, Vol. 99, pp. 1401–1406, Citeseer, 1999.

86. Hinton, G., O. Vinyals and J. Dean, "Distilling the Knowledge in a Neural Network", ArXiv:1503.02531, 2015.

87. Du, S., S. You, X. Li, J. Wu, F. Wang, C. Qian and C. Zhang, "Agree to Disagree: Adaptive Ensemble Knowledge Distillation in Gradient Space", *Advances in Neural Information Processing Systems*, Vol. 33, pp. 12345–12355, 2020.

88. Asif, U., J. Tang and S. Harrer, "Ensemble Knowledge Distillation for Learning Improved and Efficient Networks", ArXiv:1909.08097, 2019.

89. McFarland, M. C., A. C. Parker and R. Camposano, "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 301–318, 1990.

90. Gokhale, M., J. Stone, J. Arnold and M. Kalinowski, "Stream-Oriented FPGA Computing in the Streams-C High Level Language", *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 49–56, IEEE, 2000.

91. ARM, "AMBA AXI-Stream Protocol Specification", 2010, `https://developer.arm.com/documentation/ihi0051/latest/`, accessed on January 10, 2023.

92. "Vitis High-Level Synthesis User Guide (UG1399)", `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction`, accessed on January 10, 2023.

93. Csiszar, I., "A Geometric Interpretation of Darroch and Ratcliff's Generalized Iterative Scaling", *The Annals of Statistics*, Vol. 17, No. 3, pp. 1409–1413, 1989.

94. LeCun, Y., L. Bottou, Y. Bengio and P. Haffner, "Gradient-Based Learning Applied to Document Recognition", *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2278–2324, 1998.

95. Xiao, H., K. Rasul and R. Vollgraf, "Fashion-Mnist: A Novel Image Dataset for Benchmarking Machine Learning Algorithms", ArXiv:1708.07747, 2017.

96. Harrison Jr, D. and D. L. Rubinfeld, "Hedonic Housing Prices and the Demand for Clean Air", *Journal of Environmental Economics and Management*, Vol. 5, No. 1, pp. 81–102, 1978.

97. Pace, R. K. and R. Barry, "Sparse Spatial Autoregressions", *Statistics & Probability Letters*, Vol. 33, No. 3, pp. 291–297, 1997.

98. Dataport, "Pecan Street", 2017, `https://www.pecanstreet.org/`, accessed on August 14, 2022.

99. Kingma, D. P. and J. Ba, "Adam: A Method for Stochastic Optimization", ArXiv:1412.6980, 2014.

100. Robbins, H. and S. Monro, "A Stochastic Approximation Method", *The Annals of Mathematical Statistics*, Vol. 22, No. 3, pp. 400–407, 1951.

101. Van der Maaten, L. and G. Hinton, "Visualizing Data Using t-SNE", *Journal of Machine Learning Research*, Vol. 9, No. 11, pp. 2579–2605, 2008.

102. Wu, W., L. He, W. Lin, R. Mao, C. Maple and S. Jarvis, "SAFA: A Semi-Asynchronous Protocol for Fast Federated Learning With Low Overhead", *IEEE Transactions on Computers*, Vol. 70, No. 5, pp. 655–668, 2020.

103. Thorgeirsson, A. T. and F. Gauterin, "Probabilistic Predictions With Federated Learning", *Entropy*, Vol. 23, No. 1, p. 41, 2020.

104. Wang, F., H. Zhu, R. Lu, Y. Zheng and H. Li, "A Privacy-Preserving and Non-Interactive Federated Learning Scheme for Regression Training With Gradient Descent", *Information Sciences*, Vol. 552, pp. 183–200, 2021.

105. Zhang, K., *Tiresias: A Peer-to-Peer Platform for Privacy Preserving Machine Learning*, Ph.D. Thesis, Massachusetts Institute of Technology, 2020.

106. Rodríguez-Barroso, N., G. Stipcich, D. Jiménez-López, J. A. Ruiz-Millán, E. Martínez-Cámara, G. González-Seco, M. V. Luzón, M. A. Veganzones and F. Herrera, "Federated Learning and Differential Privacy: Software Tools Analysis, the Sherpa. AI FL Framework and Methodological Guidelines for Preserving Data Privacy", *Information Fusion*, Vol. 64, pp. 270–292, 2020.

107. Venkataramanan, V., S. Kaza and A. M. Annaswamy, "Der Forecast Using Privacy Preserving Federated Learning", ArXiv:2107.03248, 2021.