UNIVERSITY OF CALIFORNIA,
IRVINE

Efficient Deep Neural Networks on the Edge

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Mohammed Alnemari

Dissertation Committee:
Professor Nader Bagherzadeh, Chair
Professor Jean-Luc Gaudiot
Professor Chen-Yu (Phillip) Sheu

2022

DEDICATION

إِلَى أَبِي الْعَظِيم حَسَّان

إِلَى أُمّي الَحبيْبة لَيْلَة

إِلَى اخواتي وَأَخَصّ مِنْهُم نَبيع اللّه ورشا

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I'd want to express my gratitude to Professor Nader Bagherzadeh, my dissertation advisor, for his insightful comments and suggestions throughout the process. I'm so grateful to work with Professor Nader, who is not only a mentor but also a friend, Which I enjoyed attending his Thanksgiving gatherings. I am grateful to him for providing me with the opportunity to conduct research that is relevant to my interests and for guiding and advising me. Dr. HyunJin Kim was a pleasure to work with him for a year, and I appreciate his input. I'd also like to thank my lab colleagues, for their insightful discussions. I'd also want to express my gratitude to committee members Prof.Jean-Luc Gaudiot and Prof.Chen-Yu (Phillip) Sheu for their insightful comments. I'd like to express my gratitude for my family and friends' encouragement and support. Finally, I'd want to thank the open source community of Python, Pytorch, Keras, Caffe, Tensorly.

العلمُ زينٌ فكن للعلمِ مكتسبا * وكن له طالباً ما عشتَ مقتبسا

اركنْ إليه وثِقْ واغنَ به * وكنْ حليما رزينَ العقلِ مُحْتَرِسا

لا تأثمنَّ فإما كُنْتَ منهمِكا * في العلمِ يوماً وإما كنتَ منغمسا

وكن فتى ماسكا محضَ التقى وَرِعا * للدينِ منغمسا للعلمِ مُفْترِسا

علي بن ابي طالب

VITA

Mohammed Alnemari

EDUCATION

Doctor of Philosophy in Computer Engineering **2022**
University California *Irvine, CA*

Master of Science in Computer Engineering **2017**
University California *Irvine, CA*

Bachelor of Science in Computer Engineering **2011**
Taif University *Taif, Saudi Arabia*

RESEARCH EXPERIENCE

Graduate Research Assistant **2016–2022**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

2011–2012
Tabuk University *Tabuk, Saudi Arabia*

INTERNSHIPS

2016
Summer Intern Research at National Taiwan University *Taipei, Taiwan*

2018
Summer Intern Research at Tokyo Institute of Technology *Tokyo, japan*

2021
Intern R&D at InterDigital *Remote*

REFEREED JOURNAL PUBLICATIONS

Mohammed Alnemari, and Nader Bagherzadeh Ultimate Compression: Joint Method of Quantization and Tensor Decomposition for a Compact Models on the Edge **2022**
IEEE Access

HyunJin Kim, Mohammed Alnemari and Nader Bagherzadeh, A storage-efficient ensemble classification using filter sharing on binarized convolutional neural networks **2022**
PeerJ Computer Science

REFEREED CONFERENCE PUBLICATIONS

Mohammed Alnemari, and Nader Bagherzadeh, "Efficient Deep Neural Networks for Edge Computing **July 2019**
IEEE International Conference on Edge Computing (EDGE)

Mohammed Alnemari, and Nader Bagherzadeh, Toward Accurate Binary Neural Network Using a High Rank Tensor Train Decomposition **April 2022**
IEEE 4th International Conference on Advances in Computer Technology, Information Science and Communications.

Ye Qiao*,Mohammed Alnemari* and Nader Bagherzadeh,A Two-Stage Efficient 3-D CNN for EEG Based Emotion Recognition **Mach 2022**
IEEE International Conference on Industrial Technology

SOFTWARE

Pomegrante AI
Python Framework to Automate Deep learning Research

AWARDS

Second Honor **2011**
Taif University

Full Scholarship **2012**
Tabuk University

Best Paper Award **2019**
IEEE EDGE 2019

ABSTRACT OF THE DISSERTATION

Efficient Deep Neural Networks on the Edge

By

Mohammed Alnemari

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2022

Professor Nader Bagherzadeh, Chair

Deep neural networks have demonstrated outstanding performance in various fields of machine learning, such as computer vision, speech recognition, and natural language processing. In particular, convolutional neural networks (CNNs) perform well in computer vision tasks, such as image recognition, object detection, and image segmentation. The abundance of training data, advanced computation hardware, and use of graphical processing units (GPUs) make the training and deployment of deep CNN models plausible. CNN models usually consist of many layers that contain millions or hundreds of millions of trainable parameters. This large number of parameters necessitates high storage and computation. Deploying these models is challenging for low-energy-constrained devices, such as mobile devices, Internet of Things (IoT) nodes, CPU robotics, and autonomous vehicles. A plethora of software and hardware methods have been introduced over the last five years to compress state-of-the-art deep neural network models for easy deployment at the edge.

This dissertation aims to investigate the use and combination of pruning, quantization, and tensor decomposition methods on state-of-the-art deep neural network models. We compare combined methods with the methods when applied individually in terms of storage and computation cost. Furthermore, we seek to explore and improve the accuracy of these methods using various ensemble techniques and different training routines.

Thus,we propose the FPTT method, which combines a pruning method and a tensor decomposition method. It reduces the number of parameters by 98% for some models and achieves a compression factor of 30.7× for others. Next, we use the ultimate compression method, which combines tensor decomposition with a binary neural network to compress the model. It achieves a compression ratio of 169.1× for some state-of-the-art models. We also present a method for improving the model's inference accuracy, which uses logarithmic representation by averaging multiple quantized inputs using stochastic rounding for the weights. This method achieves the same accuracy as floating-point models while reducing the computation and storage costs. We also improve binary neural network models using ensemble methods and filter sharing to reduce the storage cost of said methods. In addition, we improve binary neural networks by using a fixed rank for tensor train decomposition, which increases the model accuracy by 2%–4%. We employ our methods in two different case studies. In the first case study, we apply the ultimate compression method and achieve a compression of 23× compared with floating-point models. In the second case study, we modify the MobileNet-v2 neural network model for an emotion classification application using EEG signals by binarizing the model and replacing the 2D convolutional layer with a 3D one. We improve the binary model's accuracy using different methods, achieving an accuracy only 2%–5% less than the floating-point model's counterpart while reducing the storage size by 40%.

Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) has grown rapidly, with 41.6 billion devices anticipated to be in use by 2025, generating and consuming 79.4 zettabytes of data [1]. Such a large volume of data requires AI systems for constructing intelligent systems. Two different paradigms exist for constructing such systems. The first paradigm employs cloud computing, where data is offloaded to the cloud for processing and predictions. The second paradigm uses edge computing to perform data processing and prediction near the data sources.

1.1.1 Cloud Computing

In the cloud computing paradigm, data is usually sent to a cloud server for processing and analysis, and subsequent actions are sent to edge devices. Using the cloud computing paradigm consumes a great deal of bandwidth resources, which increases the latency for users. Sending users' data to the cloud also puts them at risk of privacy leakage. Furthermore,

sending data to and processing them in cloud servers requires a significant amount of energy. Moreover, sending data to cloud servers is not scalable. As the number of edge devices increases, the amount of data collected will increase as well. In addition, cloud computing is bounded, making this paradigm unsuitable, especially for AI systems.

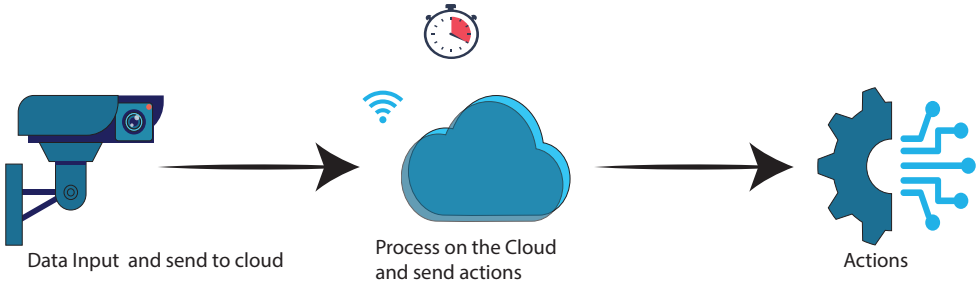


Figure 1.1: Cloud computing data is collected and sent to the cloud, where it is processed and sent to edge devices.

1.1.2 Edge Computing

In the edge computing paradigm, data is collected and analyzed in edge devices or edge servers. Using edge computing reduces the latency because the processing occurs in proximity to the source data. Edge computing saves a significant amount of time as well as data transmission. In addition, processing and analyzing data at the edge protects users' privacy. Furthermore, this paradigm consumes less power at the edge since the devices can offload the computing tasks to edge servers. Moreover, this paradigm is scalable since it is not limited by the resources on the cloud

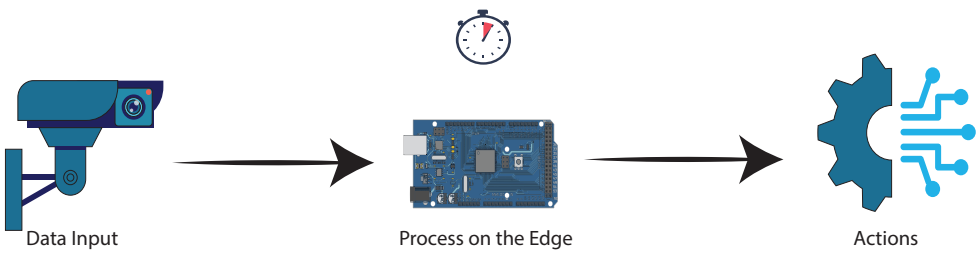


Figure 1.2: Edge computing – Data is collected and processed in edge devices.

1.1.3 Edge AI Applications

Machine learning at the edge benefits autonomous vehicles by allowing more accurate and faster decision making, resulting in the more accurate identification of road traffic elements as well as faster and safer transportation. Machine learning at the edge also benefits smart city applications in a variety of domains, including facial recognition, disaster response, and traffic control. In the agricultural domain, building smart farming that monitors and controls livestock and land resources is one of the many benefits of using machine learning at the edge. Many applications in the Industry 4.0 domain, such as inspection in manufacturing lines, can benefit from machine learning at the edge. Furthermore, using machine learning at the edge for health care has numerous advantages, including the protection of patients' privacy and development of better models for diagnosing various illnesses.

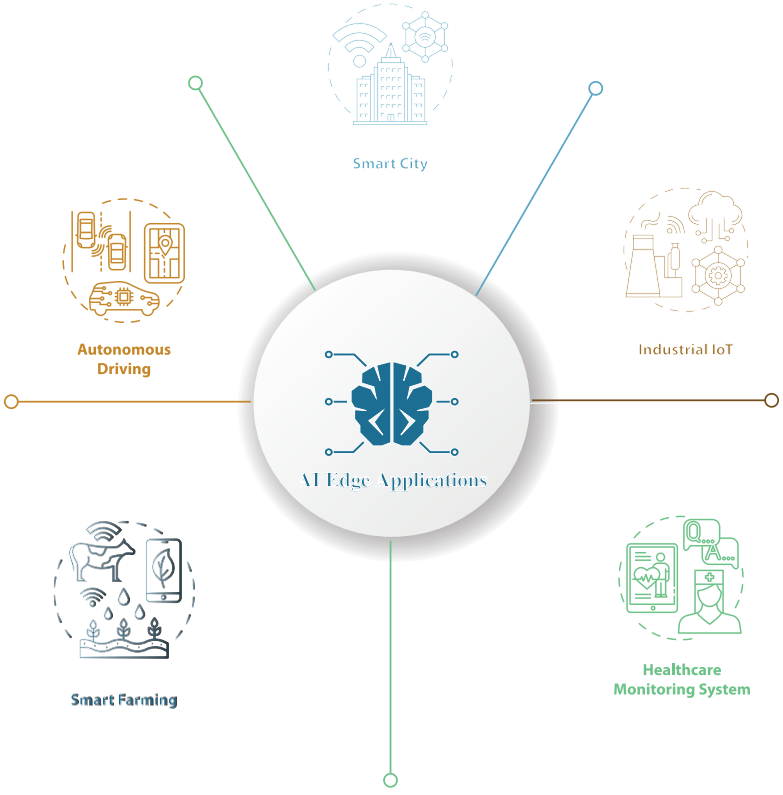


Figure 1.3: Different domains can use machine learning at the edge.

1.2 Machine learning vs. Deep learning

In traditional machine learning, a handcrafted feature extraction step is required, which is typically performed in collaboration with an engineer and a domain expert. Then, as demonstrated in Figure 1.4, these features are fed into a classification algorithm, such as a support vector machine (SVM), decision tree, or random forest, to predict the output.

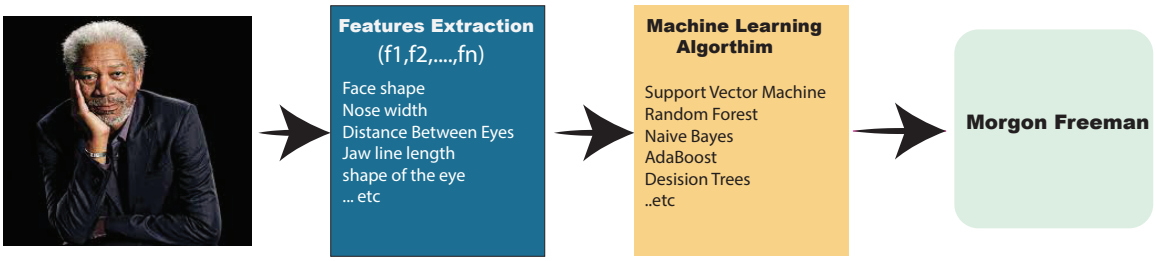


Figure 1.4: Machine learning, where features are extracted from the data before being fed into classification algorithms.

In deep learning, the feature extraction step is not required. The network automatically extracts the features as the data passes through the network layers. The network identifies the pattern within the data to create the features, as illustrated in Figure 1.5.

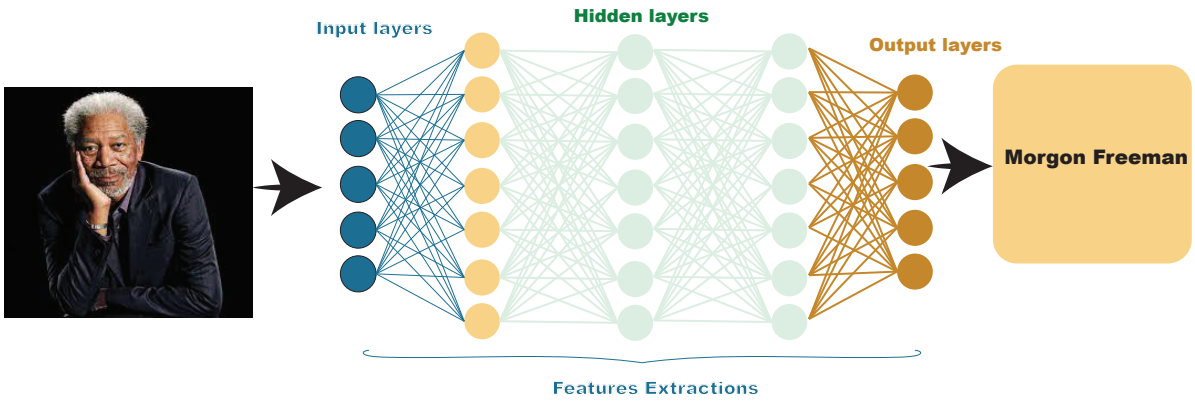


Figure 1.5: Deep learning, where features are extracted as the data pass through the network.

1.3 Deep Learning at the Edge

Because of the storage and computation costs required by these models, deploying deep neural network models at the edge is difficult. For example, a small model called LeNet-5 introduced in 1998 only had 60,000 weights, whereas the AlexNet model introduced in 2012 had 60,000,000 weights. As models grow in size, their accuracy improves, as illustrated in Figure 1.6.

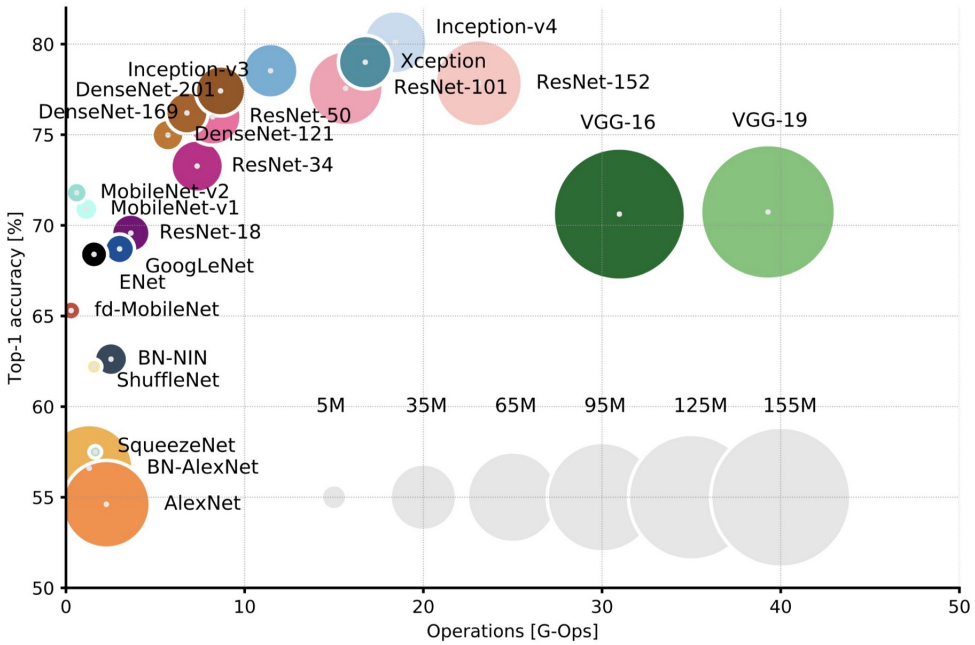


Figure 1.6: Image classification accuracy of different convolutional neural network architectures on the ImageNet dataset for different model sizes. As can be seen, for most architectures the accuracy increases linearly as the number of parameters in the model is increased. Figure reproduced from[10]

To reduce the number of parameters for these models and build more efficient deep neural network models at the edge, three different approaches exist. The algorithmic approach uses algorithms, such as pruning, quantization, tensor decomposition, network distillation, and network architecture search (NAS), to build more efficient models. In the system approach, operations can be optimized to be more efficient. Frameworks such as TensorFlow and Pytorch can be used, for example, to optimize the computational graph and schedule

optimization. Finally, in the hardware approach, hardware can be built either using FPGA or ASIC for specific domains to accelerate deep neural network models on these devices, such as those used on drones and surveillance cameras. Movidius, Nervana, Jeston, and TPU are examples of AI chips.

1.4 Dissertation Contribution

- We introduce the FPTT method, which combines pruning and tensor decomposition using filter pruning on the convolutional layer and tensor train decomposition on the fully connected layer. FPTT reduces the number of parameters, which in turn reduces the number of floating-point operations as well as the storage cost.

- We introduce the ultimate compression method, which is a joint method combining binary neural networks (BNNs) with a tensor decomposition algorithm to build highly efficient models at the edge with low storage and computation costs. This method compresses some models by 169× factors.

- We introduce a method for improving models' inference accuracy, which uses logarithmic representation by averaging multiple quantized inputs with stochastic rounding.

- We improve the inference accuracy of the deep BNN model by employing ensemble methods and sharing filter weights. We also use high-rank tensor train decomposition to improve the inference accuracy of BNN models.

- We apply the ultimate compression method to crowd counting models and compare their performance and storage cost with floating-point counterparts.

- We modify MobileNetV2 to detect emotions from EEG signals at the edge using a 3D convolutional layer instead of a 2D one and binarize the model.

1.5 Dissertation Organization

This dissertation is organized into 11 chapters, the remainder of which are organized as follows. Chapter 2 provides background information on convolutional deep neural networks and differentiates between training and inference. Chapter 3 presents some past studies that are related to our dissertation. Chapter 4 introduces the FPTT method and its performance on state-of-the-art deep neural network models. Chapter 5 introduces the ultimate compression method, which is a joint method combining BNNs and tensor decomposition, and demonstrates their performance with an in-depth study on state-of-the-art deep neural network models. Chapter 6 introduces a trade-off method to improve BNNs using ensemble methods and filter sharing. Chapter 7 introduces a method for improving the model's inference accuracy using logarithmic representation by averaging multiple stochastic rounding of the input parameters. Chapter 8 improves BNNs by modifying the layers of the models using a high rank for tensor train decomposition. Chapter 9 presents the use of the ultimate compression method for crowd counting models. Chapter 10 discusses and modifies MobileNetV2. We build three different versions by replacing the 2D-convolutional layer with a 3D one and binarizing the models for an emotion recognition application using EEG signals. Finally, the conclusions and recommendations for future work are provided in Chapter 11.

Chapter 2

Background

2.1 Convolutional Neural Networks

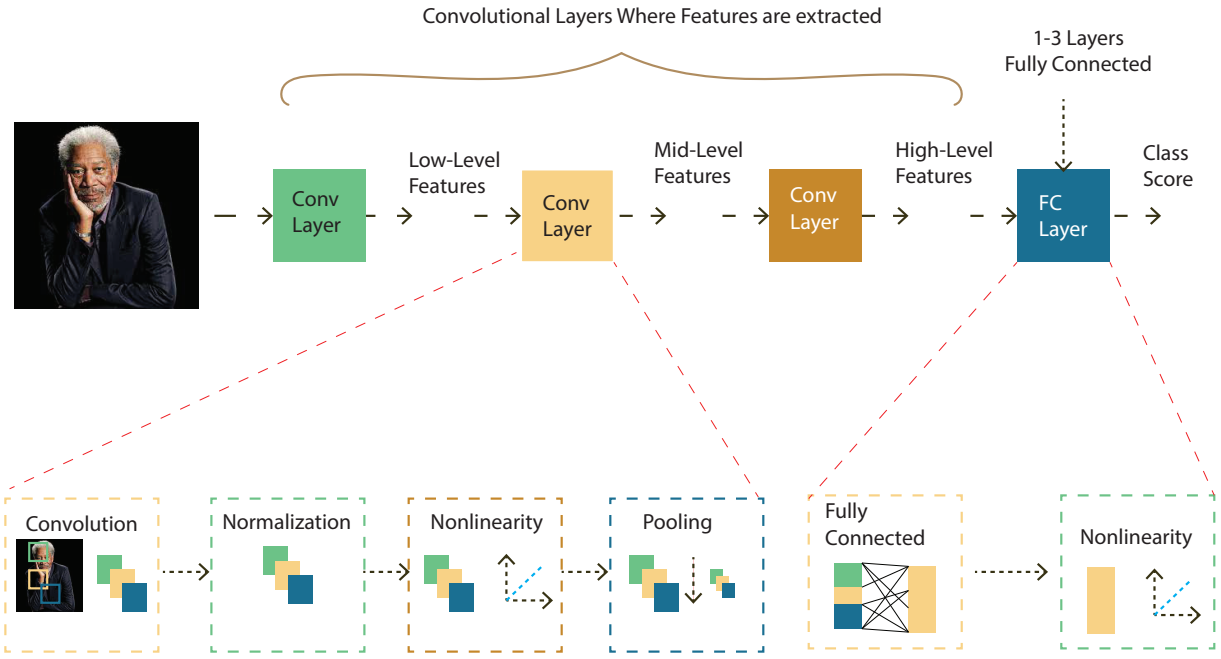


Figure 2.1: A convolutional neural network, in which the convolutional layers extract the features and feed to a fully connected layer for classification.

Convolutional neural networks (CNNs) consist of multiple convolutional layers, where the first layer learns the basic features, such as edges and lines; the middle layer learns more complex features, such as circles and squares; and the following layers extract the most complex features, such as faces and car wheels. Every convolutional block usually contains a convolutional layer, a non-linearity layer, a pooling layer, and finally fully connected layers, as illustrated in Figure 2.1.

Convolution learns the feature representation of the input data. Different filters are used in the convolutional layers to slide on the input and compute the feature maps. Each neuron in the features maps connects to a region of neighboring neurons in the previous layer; this is the neuron's receptive field. The convolutional layer shares the weights among neurons, which reduces the number of model parameters and improves the generalization of the models. There are three different types of convolutional layers, namely 1D convolutional, 2D convolutional, and 3D convolutional layers, as illustrated in Figure 2.2.

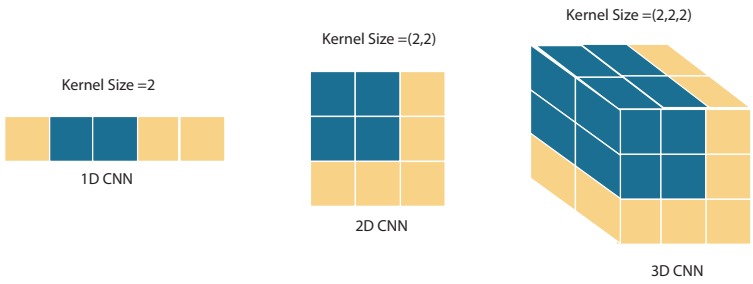


Figure 2.2: The three different types of convolutional layers.

Nonlinearity is a nonlinear activation function that is typically used after convolutional or fully connected layers to introduce nonlinearity into the network. Numerous activation functions exist, such as the sigmoid activation function, which sets all values to have probabilities between 0 and 1, reduces extreme values and zero, and is typically used for two classes. The Tanh activation function, with values ranging from 1 to +1, works best for hidden layers. The ReLU activation function only fires if the input is greater than zero and is the most

commonly used activation for state-of-the-art deep neural network models. Leaky ReLU is similar to the ReLU activation function, but when the input is negative it introduces a small negative slope at approximately 0.01. The aforementioned activation functions are illustrated in Figure 2.3.

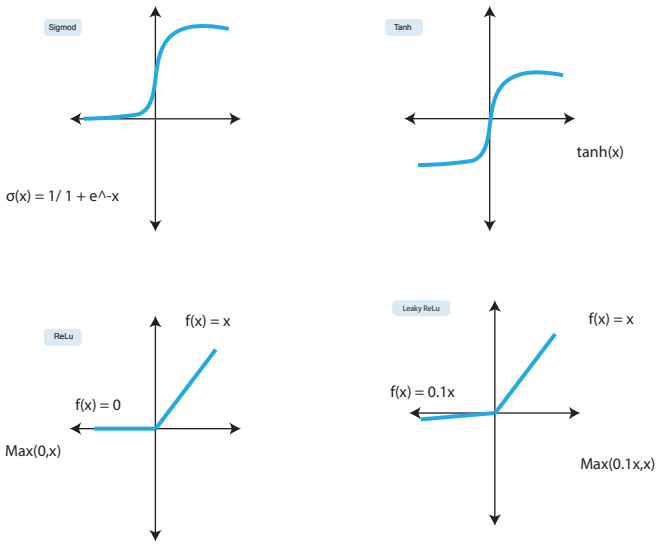


Figure 2.3: Different activation functions: Sigmoid, Tanh, ReLU, and Leaky ReLU.

Pooling is usually used in CNNs to reduce the number of parameters, and it also makes the network robust and invariant to small shifts and distortions [56]. There are two common pooling layers, namely max pooling and average pooling, as illustrated in Figure 2.4.

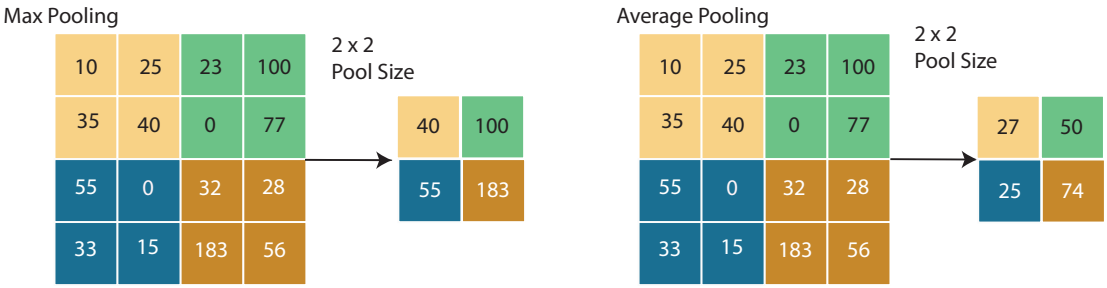


Figure 2.4: Max pooling selected the maximum value, and Average pooling averaged the values

Normalization refers to the distribution of data across the layer being normalized to have a zero mean and unit standard deviation. Doing so improves the accuracy and accelerates the training.

Fully Connected refers to every neuron in the output feature maps connecting to every neuron in the input feature maps. There are usually between one and three fully connected layers, and the features are flattened before being fed into the fully connected layers. The output layer will have the same number as the number of classes.

2.2 Optimization

Optimization in deep learning minimizes the loss function between the real values and the predicted values. This is done using hill-climbing or descent-climbing optimization algorithms, such as gradient descent algorithms and their variants stochastic gradient decent (SGD) and Adam. The step direction (gradient) and step size (learning rate) play significant roles in optimizing the deep neural network models, in which either selecting a large or very small learning rate affects the model accuracy and convergence, as illustrated in Figure 2.5. The weight is updated based on the learning rate and loss function as follows: $\Delta w_i = -\alpha \frac{dL}{dw_i}$

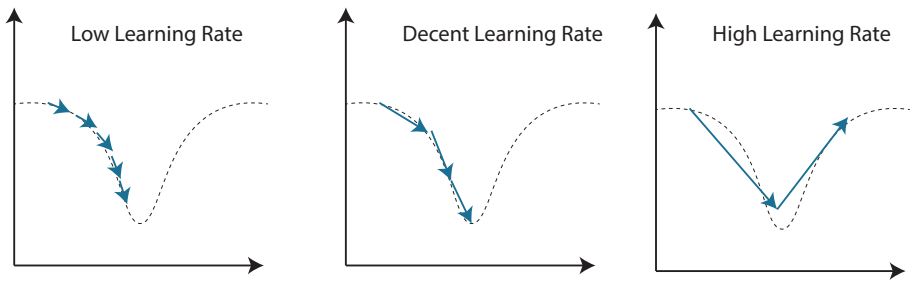


Figure 2.5: Impact of selecting different learning rates on model performance, where selecting a small learning rate makes the model require more time to converge, whereas selecting a high rate causes the model to never reach the local minima.

2.3 Model Training

Training a deep neural network model involves forward propagation, where the data is forwarded to the model and the predicted values are compared with the real values, which is referred to as the loss function $\overline{y} - y$. The parameters are updated using backward propagation. A backward pass in where the derivative of the error is propagated with respect to the weight from the last layer (output layer) to the first input layer. This is done using the chain rule of calculus to compute how the loss is affected by each weight, and the weights are updated as shown in Figure 2.4.

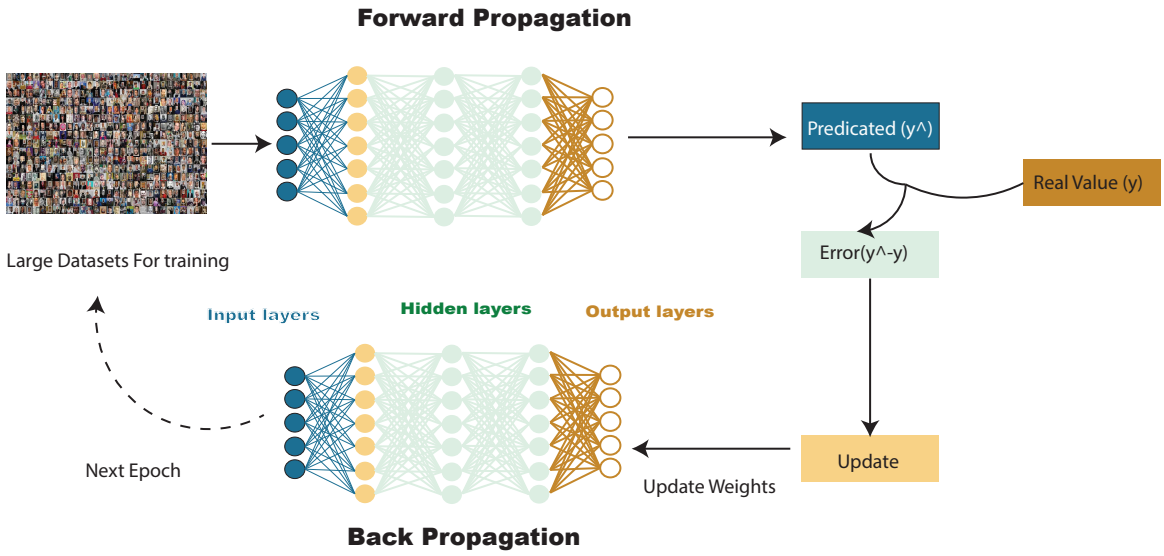


Figure 2.6: Training a neural network, where the data is forwarded to the model and the predicted values are compared with the real values to compute the loss function. Subsequently, the parameters are updated and backpropagated.

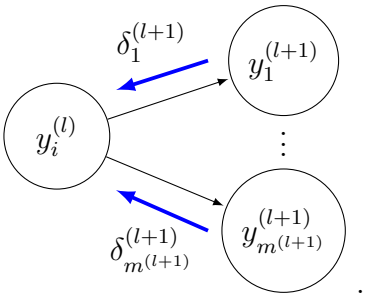


Figure 2.7: Once evaluated for all output units, the errors $\delta_i^{(L+1)}$ can be propagated backwards.

2.4 Model Inference

In the inference stage of the model, only forward propagation is performed, after which the model is trained. A sample of the dataset is forwarded to the neural network model with trained parameters, and then the accuracy of the inference is computed based on the many correct predicted values equal to the real values of the sample of the dataset, as illustrated in Figure 2.8.

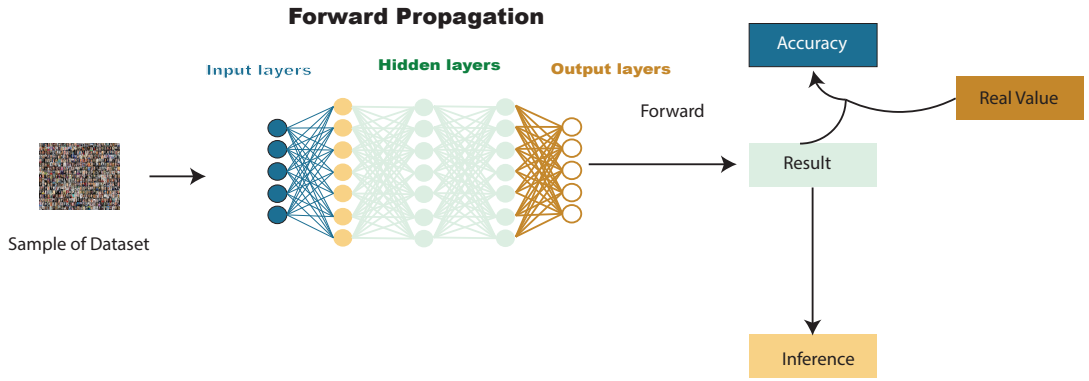
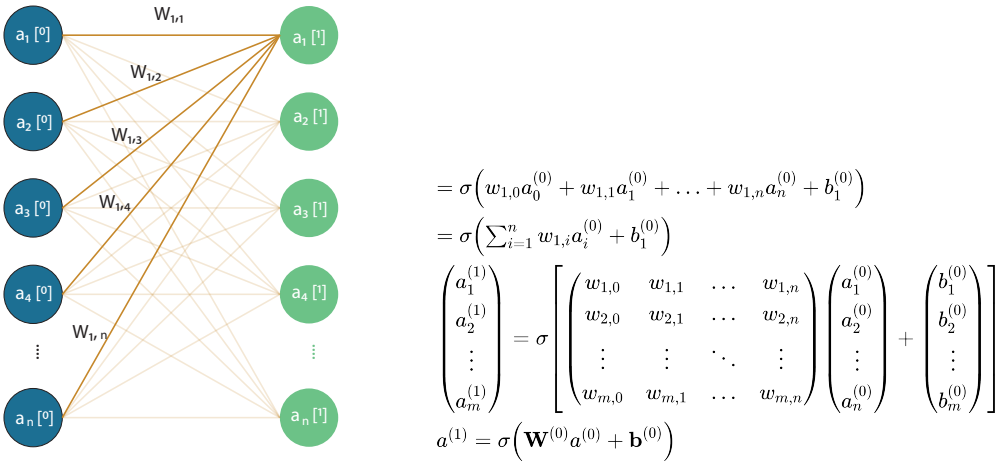


Figure 2.8: Inference in a deep neural network model, in which data is propagated forward and the accuracy of the model is calculated.



$$= \sigma\left(w_{1,0}a_0^{(0)} + w_{1,1}a_1^{(0)} + \ldots + w_{1,n}a_n^{(0)} + b_1^{(0)}\right)$$

$$= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(0)} + b_1^{(0)}\right)$$

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,0} & w_{1,1} & \ldots & w_{1,n} \\ w_{2,0} & w_{2,1} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \ldots & w_{m,n} \end{pmatrix}\begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix}\right]$$

$$a^{(1)} = \sigma\left(\mathbf{W}^{(0)}a^{(0)} + \mathbf{b}^{(0)}\right)$$

(a) (b)

Layers' connection for a convolution: (a) Two fully connected layers showing the inference stage for one neuron; (b) multiply–accumulate operation performed at the inference stage for one neuron.

2.5 Models

2.5.1 LeNet-5

LeNet-5 is a small model that consists of three convolutional layers and two fully connected layers designed for hand written digit recognition. Figure 2.10 illustrates the LeNet-5 architecture.

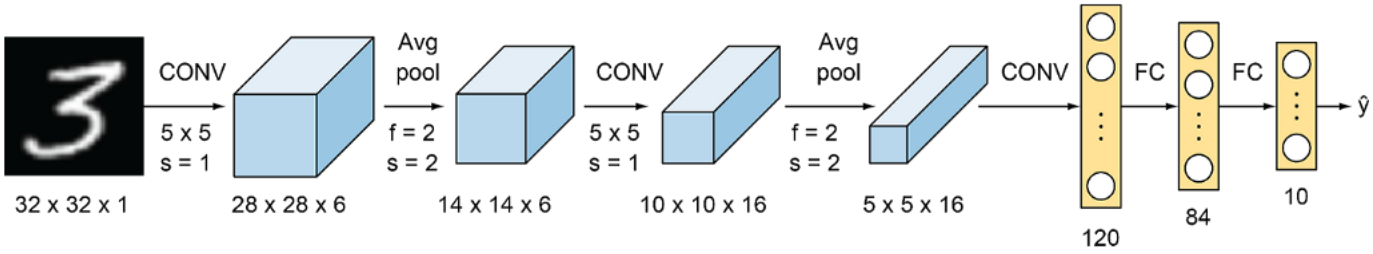


Figure 2.10: Lenet-5 [57] architecture.

2.5.2 Network in Network

The network-in-network (NiN) architecture uses 1×1 convolutions, which provide superior combinational power to the features of the convolutional layers, as depicted in Figure 2.11.

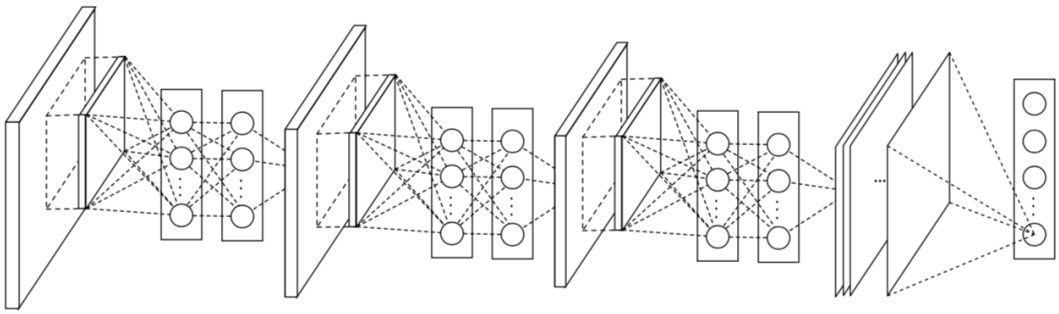


Figure 2.11: Network-in-network [64] architecture.

2.5.3 AlexNet

The AlexNet neural network model is a high-capacity model that consists of five convolutional layers and three fully connected layers, as illustrated in Figure 2.12.

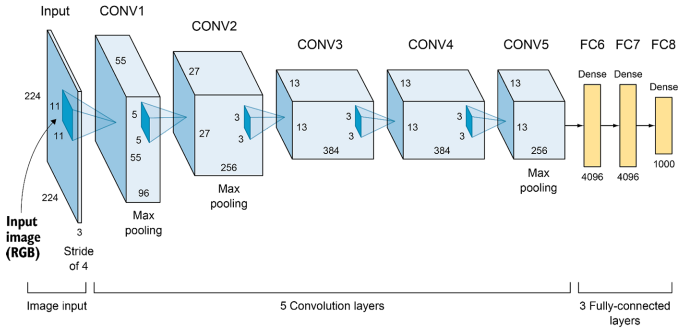


Figure 2.12: AlexNet [54] architecture.

2.5.4 VGG-16

VGG-16 is a high-capacity neural network model that consists of 13 convolutional layers and three fully connected layers, as depicted in Figure 2.13

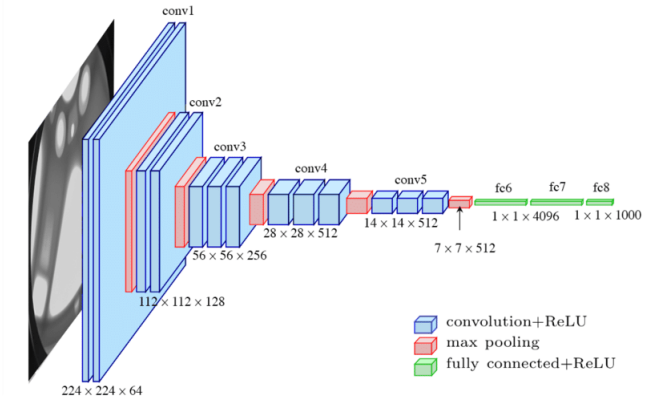


Figure 2.13: VGG-16 [97] architecture.

2.5.5 ResNet

ResNet is a very-high-capacity model that uses a residual block to build very deep models. There are numerous variants of the ResNet architecture, ranging from 18 to 152 layers. Figure 2.14 illustrates the ResNet-18 architecture.

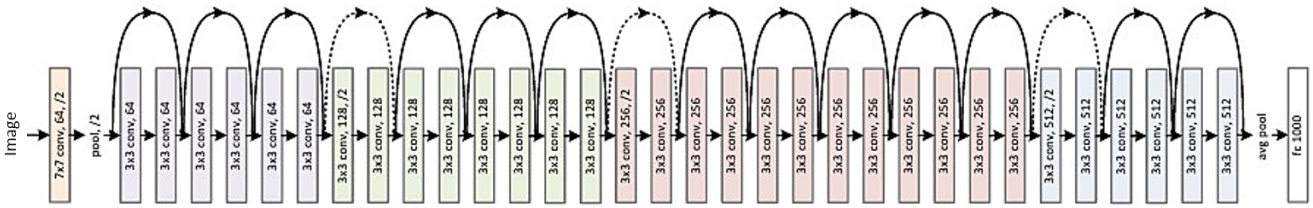


Figure 2.14: ResNet-18 [34] architecture.

2.5.6 GoogLeNet

The GoogLeNet architecture consists of 22 layers (27 layers including pooling layers), and among these layers are nine inception modules. The inception modules consist of four branches with 1×1, 3×3, and 5×5 convolutions as well as downsampling. Figure 2.15 illustrates the GoogLeNet architecture

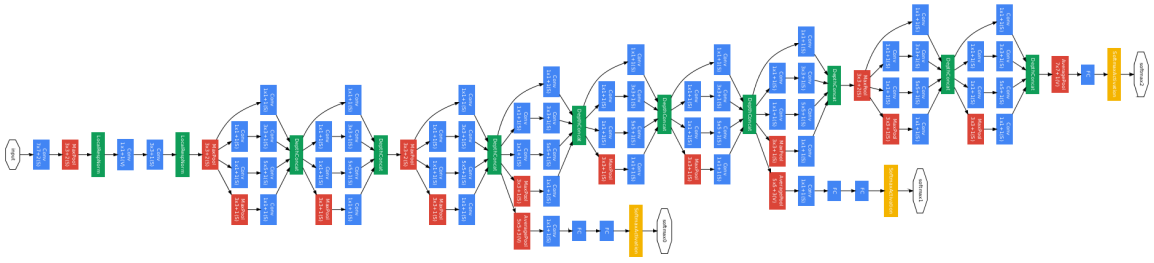


Figure 2.15: GoogLeNet [101] architecture.

2.6 Datasets

2.6.1 MNIST

MNIST is a small handwritten number dataset that consists of 60,000 training images and 10,000 test images of size 28 × 28 pixels and 10 labels ranging from 0 to 9.

Figure 2.16: Sample from the MNSIT dataset [58]

2.6.2 CIFAR-10 and CIFAR-100

CIFAR-10 is a widely used dataset that contains 50,000 RGB images of 32 × 32 pixels for training and 10,000 for testing with 10 different classes. CIFAR-100 differs from CIFAR-10 only in terms of the number of classes, containing 100 different classes instead of 10.

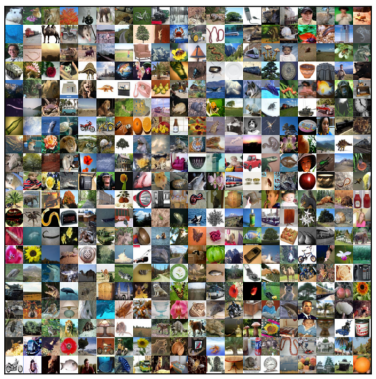


Figure 2.17: Sample from the CIFAR dataset [53]

2.6.3 ImageNet

ImageNet is a widely used dataset with 1.2 million RGB images of size 224 × 224 pixels for training and 50,000 for testing.

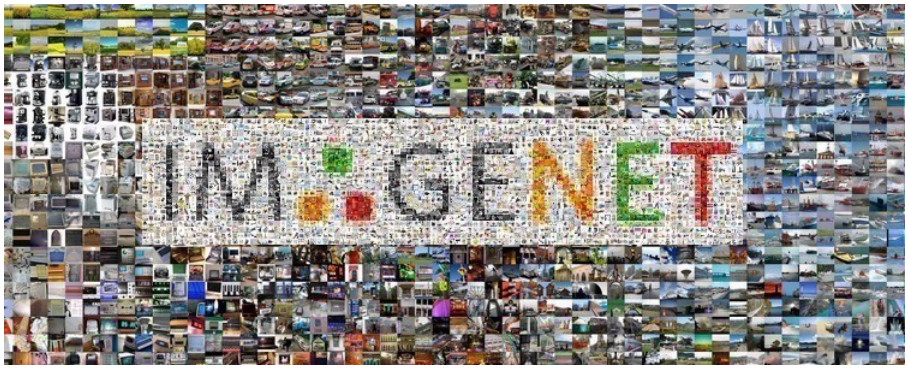


Figure 2.18: Sample from the ImageNet dataset [18]

Chapter 3

Related Works

3.1 Algorithmic Methods

3.1.1 Pruning

Pruning is a well-studied method for reducing the computation and storage costs of deep neural network models. In the early stages of its application to deep neural network models, connections in the models are pruned based on the lowest saliency. The saliency term comes from computing the Hessian matrix or inverse Hessian matrix for every parameter, as described in the Optimal Brain Damage method and Optimal Brain Surgeon method, respectively [59][31]. These methods are used for pruning when the deep neural network model parameters are not a significant burden. However, state-of-the-art deep neural network models, such as AlexNet and VGG-16, have 60 million and 138 million parameters, respectively. Therefore, computing the Hessian matrix or inverse Hessian matrix for every parameter in these models is not plausible. The deep compression method was introduced in 2015, which uses a certain threshold to remove connections below a certain threshold [29]. In

the deep compression method, most of the connections pruned are in fully connected layers, which are responsible for 90% of the total parameters and only 1% of the overall floating-point operations (FLOPs) [62]. Most of the FLOPs for convolutional layers are compressed and accelerated but require Sparse Basic Linear Algebra Subprograms (BLAS) libraries [29] or special hardware to deal with sparse matrices [28]. Researchers have proposed numerous pruning methods for constructing different forms of sparsity that do not require specific hardware, such as vector-level sparsity (1D), kernel-level sparsity (2D), and filter level sparsity (3D). Filter pruning has been proposed in [62] [70] [35], which is a natural structure way of pruning that does not require BLAS or specialized hardware; these methods reduce the FLOPs by more than 30%, 48%, and 52%, respectively.

3.1.2 Quantization

Quantization methods are another option for reducing the storage and computation costs of deep neural network models. Quantization differs from pruning methods by focusing on how many bits can represent the parameters of deep neural network models. Quantization in deep neural networks refers to the conversion of the neural network parameters' values, either the weight, activation, or inputs, from a 32-bit floating point to a lower-precision format. The literature is replete with different quantization methods. Quantization can be applied to the training and inference stages for deep neural network models. In the inference stage, a model with only 8 bits for convolutional layers and 5 bits for fully connected layers can obtain the same accuracy as its floating-point model counterpart [29]. Other methods use an 8-bit integer to train the models and make the inference [107]. Another method is logarithmic representation, which has 3 bits for inference and training with little degradation in terms of accuracy; logarithmic representation also reduces computation, storage, and hardware costs [74]. BNNs are a powerful method for quantizing neural network models and use only 1 bit to represent the deep neural network parameters [17] [16] [65]. These methods can

be applied only in the inference stage of deep neural network models. A recent work [72] demonstrated how to train the model with degradation of only a few percent compared with the floating-point counterpart.

3.1.3 Tensor Decomposition

Tensors are multidimensional arrays or N-way arrays. The array dimensionality specifies the tensor order or the number of tensor modes. Tensor decomposition represents high-order tensor data through multilinear operation over its factors. Tensor decomposition methods have attracted much attention in various fields, such as psychometrics, chemometrics, machine learning, quantum physics, and neuroscience [51][14]. CANDECOMP/PARAFAC (CP) and Tucker decomposition are the most popular and well-known algorithms for decomposing high-order tensors. Both CP and Tucker decomposition are high-order generalizations of principle component analysis (PCA) and singular value decomposition (SVD). CP decomposition is presented in deep learning literature as a tool for compressing the model and reducing the FLOPs required by the convolutional layers and fully connected layers [55]. CP decomposition factorizes a tensor into a sum of the rank-one tensor. Tucker decomposition is also used to compress the model and reduce the required FLOPs required by convolutional layers and fully connected layers [47]. It compresses the data into tensors of small dimensions represented by core tensors, while its factor matrices span the subspace occupied by the fiber of data [66]. CP decomposition produces a compact representation, but finding an optimal solution is challenging. Tucker decomposition is stable and flexible but suffers from the curse of dimensionality, in which the number increases exponentially with the tensor order. Tensor networks are a generalization of tensor decomposition and an excellent tool for large-scale data. They convert high-order tensors into interconnected low-order tensors. Various methods exist for using tensor networks, such as tensor train (TT) [79], hierarchical Tucker (HT) [15], and tensor ring (TR) [115]. TT is the most common algorithm among the

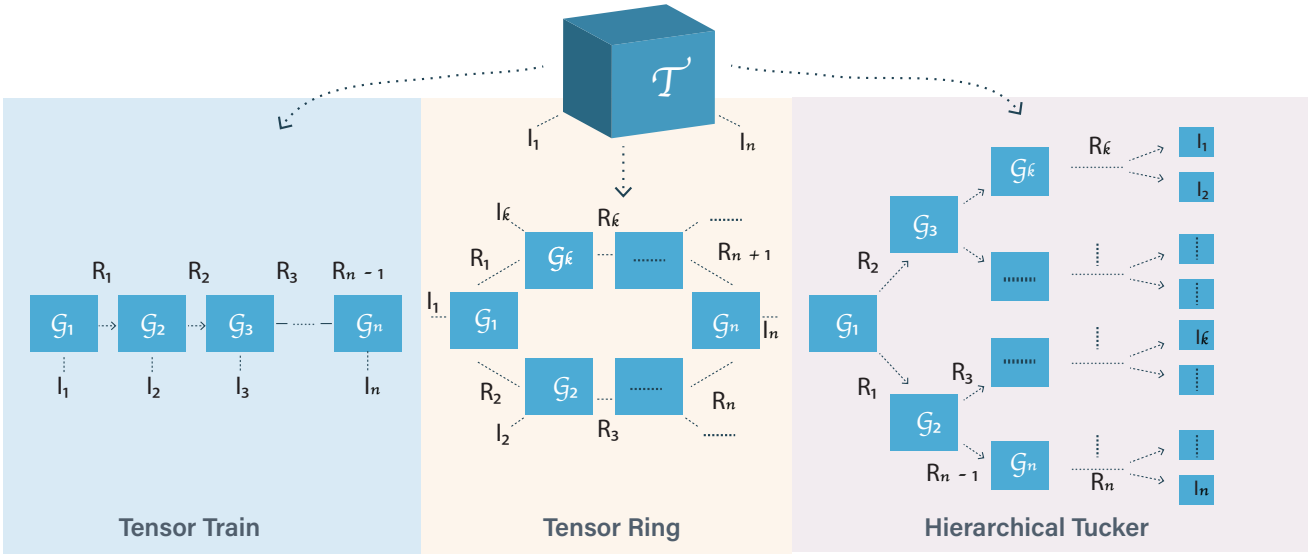


Figure 3.1: Different Tensor Networks Algorithms

tensor network algorithms. TT decomposition provides a better representation of high-order dimensional tensors and does not suffer from the curse of dimensionality. TT decomposition is applied to deep learning models for more efficient models [78][23]. HT is a recursive hierarchical construction of tucker decomposition [15] and accelerates deep neural network models. TR is a generalized form of CP decomposition that uses two-order tensors instead of one-order tensors to multiply the first and last tensor, thus forming a ring structure. TR decomposition was used recently to compress deep neural network models [115]. TT, HT, and TR are illustrated in Figure 3.1.

3.1.4 Network Distillation

Knowledge transfer is one of the methods for compressing deep neural network models. The earliest work on knowledge transfer [9] trained a compressed model to mimic large and complex ensemble models. Knowledge distillation, introduced by Hinton [36], uses the same idea but transfers knowledge from a larger model called the teacher to a smaller model called the student by softening the SoftMax probability distribution. FitNets, on the other hand, not

only uses the SoftMax output probability to guide the student network but also the intermediate representation as a hint for the student network [89]. The FitNets method generates a thin and deeper network that generalizes well [89] and is computationally less intensive than the teacher network. In Born-Again Networks (BANs), the student and teacher are identical in terms of parameterizing, and knowledge transfer occurs from a teacher network to a student network at a similar capacity. However, in BANs, student networks outperform teacher networks in terms of accuracy [20]. Teacher Assistant Knowledge Distillation indicates that the size of the models and the gap between the teacher model's size and the student model's size play a significant role in training improved student models if the gap between the student and teacher model is significant. The student model lowers its performance significantly compared with its teacher based on this gap. Researchers introduced a model or chain of models called teacher assistants to bridge this gap between teacher and student and thus build a better student model [73].

3.1.5 Network Architecture Search

Network architecture search (NAS) is a method for creating neural network architectures by employing a search strategy based on objective evaluation. The objective evaluation can include storage costs, flop counts, and hardware type. NAS can create excellent neural network architectures, but it requires massive computational power. Furthermore, because of the numerous architectures that NAS evaluates, training and evaluating neural network architectures require significant amounts of computational power and time. For example, the Google Brain team's NAS method used 800 GPUs for 28 days, resulting in 22,400 GPU hours, which demonstrates the computation and cost required by NAS [119]. Numerous methods have been introduced to reduce the evaluation procedure, such as low-fidelity performance estimation, weight inheritance, weight sharing, learning-curve extrapolation, network morphism, and single-shot [2].

Chapter 4

Filter Pruning and Tensor Train Decomposition

This chapter presents the FPTT method, which is a two-stage pipeline: filter pruning (FP) on convolutional layers and tensor train (TT) decomposition on fully connected layers. We combined them to reduce the storage and computation requirements of DNN models to be able to deploy them easily on edge devices. In the first stage, we used FP instead of weight pruning to alleviate the need for a BLAS library or specialized hardware. In the second stage, we used TT decomposition, a form of nonrecursive tensor decomposition that does not suffer from the curse of dimensionality, and the parameters were the same as for canonical decomposition (CP). However, it is more stable and based on a low-rank approximation of auxiliary unfolding matrices. Our method is depicted in Figure 4.1.

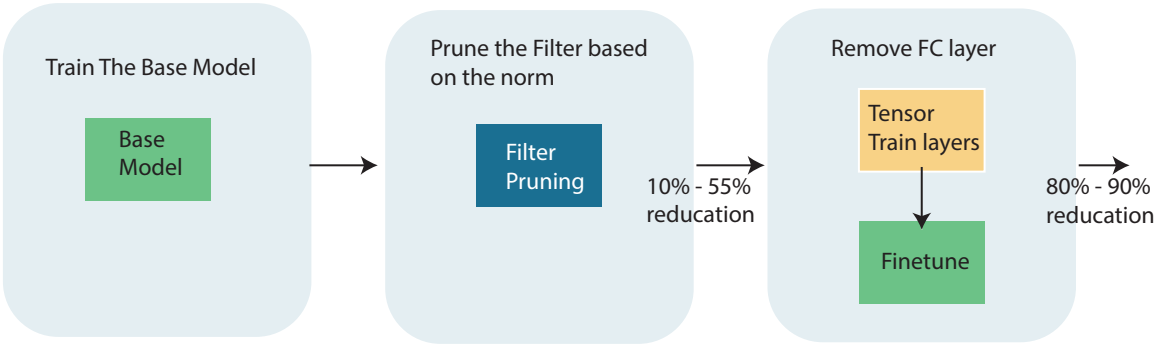


Figure 4.1: The FPTT pipeline: the base model is trained, filter pruning and tensor train decomposition are applied, and finally the model is retrained

4.1 Filter Pruning

Each convolutional layer in deep neural networks transforms an input feature map into an output feature map, which is an input for the next convolutional layer. The filters slide on the input channels, producing feature maps for each filter, as depicted in Figure 4.2. Pruning is applied on the filters and the output feature maps corresponding to the pruned filters are removed. We used the algorithm presented in [62], which uses L1-norm for every filter as pruning criteria. The algorithm is as follows: first, the sum of its absolute kernel weight is calculated, as shown in Equation (4.1); second, the filter is sorted by f_j and the filters with the smallest absolute sum value and their corresponding features map are pruned:

$$f_j = \sum_{n=1}^{n_i} = \sum |K_n| \tag{4.1}$$

We pruned the filters across the network in a greedy fashion, resulting in high accuracy compared with pruning filters layer by layer. Applying FP on the convolutional layers results in effective compression for some models. However, state-of-art models still require high costs in terms of computation and storage to be deployed on edge devices. To compress the model further, we incorporated TT decomposition to compress the fully connected layers of the state-of-art models.

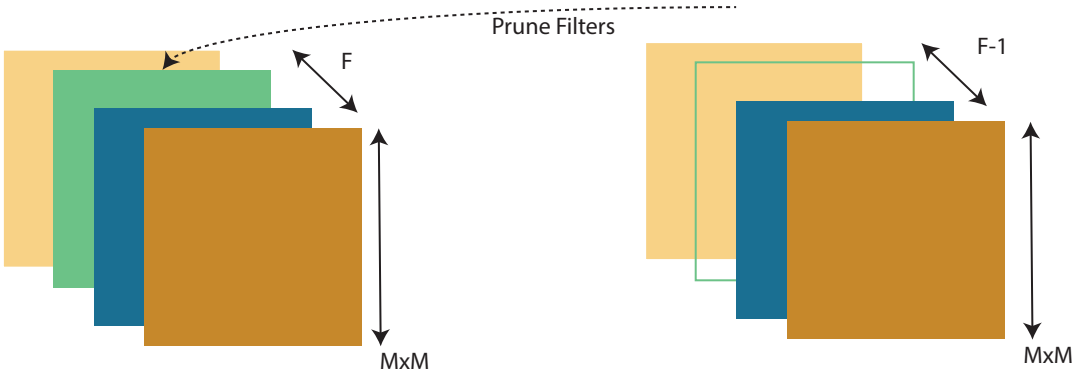


Figure 4.2: Prune filters in which the feature maps that corresponded to the pruned filters are pruned.

4.2 Tensor Train Decomposition

After applying FP to the model, we replaced the fully connected layers with TT layers to compress the models further. TT layers store their weight matrix in the TT format [78]. The TT format, when applied on tensor X, can be represented as in Equation (4.2), such that for each dimension $k = 1, \ldots, d$ and for each value of the k-th th dimension index $i_k = 1, \ldots, n_k$ there is a matrix $G_k[i_k]$ [78].

$$\mathcal{X}(i_1, \ldots, i_d) = G_1[i_1] G_2[i_2] \ldots G_d[i_d] \tag{4.2}$$

All matrices $G_k[i_k]$ related to the same dimension K must be of the same size $r_{k-1} \times r_k$ and $r_0, r_d = 1$ in order to make matrix product Equation (4.2) of size 1×1. The sequence of $\{r_k\}_{k=0}^{d}$ is the rank of the Tensor Train format. The collection of the matrices $(G_k[j_k])_{i_k=1}^{n_k}$ are called cores. Equation (4.2) can be rewritten as shown in Equation (4.3), where using the symbol $G_k[j_k](\alpha_{k-1}, \alpha_k)$ is used as elements of the matrices $G_k[j_k]$.

$$\mathcal{X}(i_1, \ldots, i_d) = \sum G_1[i_1](\alpha_0, \alpha_1) \ldots G_d[i_d](\alpha_{d-1}, \alpha_d) \tag{4.3}$$

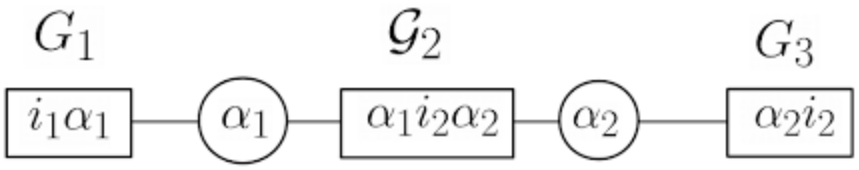


Figure 4.3: Tensor Train Format [79]

To store tensor $\mathcal{X}$ of all its elements requires $\prod_{k=1}^{d} n_k$ numbers compared with TT format which only requires $\sum_{k=1}^{d} n_k r_{k-1} r_k$ to store their elements. This indicates that the TT format is highly efficient in terms of memory. The TT format is represented geographically in Figure 4.3 in where $r_0, r_d = 1$. The ellipses represent indices of the original tensors i_k and indices of some of auxiliary tensors α_k, and circles represent only the indices of the auxiliary tensors α_k representing the link. If the same auxiliary index was present in the two cores, we connected them. To evaluate the entry of tensor, all tensor in ellipses had to be multiplied and we performed summation over all of the auxiliary indices [79].

Fully connected layers can be represented using Equation (4.4) in which $W \in \mathbb{R}^{m \times n}$ is the weight matrix and $b \in \mathbb{R}^m$ is the bias vector.

$$y = W_x + b \tag{4.4}$$

TT layers store their weight matrices in TT format. Using the TT format allows one to use hundreds or even thousands of hidden units with a small number of parameters. The number of these parameters can also be controlled either by using different numbers of units or by changing the TT rank. TT layers can be represented as indicated in Equation (4.5).

$$y\left(i_1, \ldots, i_d\right) = \sum_{j_1, \ldots, j_d} G_1\left[i_1, j_1\right] \ldots G_d\left[i_d\right] X\left(j_1, \ldots, jd\right) + B\left(i_1, \ldots, i_d\right) \tag{4.5}$$

The TT format replaces a tensor $\mathcal{X}$ with its approximation tensor $\mathcal{Y}$ in which $\|\mathcal{X} - \mathcal{Y}\| \leq$ accuracy $\|\mathcal{Y}\|$. We replaced the fully connected layers in the pruned models with TT layers, which reduce the models neurons immensely achieving of more than 90% for some models. We trained DNNs using the SGD algorithm. The models use the backpropagation algorithm to compute the gradient and update the parameters. The backpropagation algorithm computes the gradient of the loss function of the last layer and propagates through layers in reverse order. To use TT layers, we converted the gradient matrix to tensor train format and added them to the estimation of the weight matrices. Doing so when we trained the models would require $\mathcal{O}(MN)$ which is not efficient if we replace the fully connected layer with many tensor train layers. To mitigate this, we use the same approach presented by [78] in which compute the gradient of the loss function directly with respect to the tensor train format of the weights. Doing this requires only $O\left(\mathrm{r}^3 \max\{M, N\}\right)$ which is very efficient if the rank is small. For a further explanation and an in-depth analysis of training TT layers, readers may refer to [78]

4.3 Experiment and Results

In our experiment, we employed three different approaches, as illustrated in Figure 4.4. In the first approach, we trained the base model, pruned the filters, retrained the models, replaced the fully connected layers with TT layers, finetuned the models, and finally computed the models' inference accuracy. In the second approach, we pruned filters from untrained models first and, then trained the models, replaced fully connected layers with TT layers, finetuned the models, and finally computed the models' inference accuracy. In the third approach, we pruned the filters, replaced the fully connected layers with TT layers, trained the models, and then computed the models' inference accuracy.

Our experiments indicated that the third approach worked best with shallow or medium

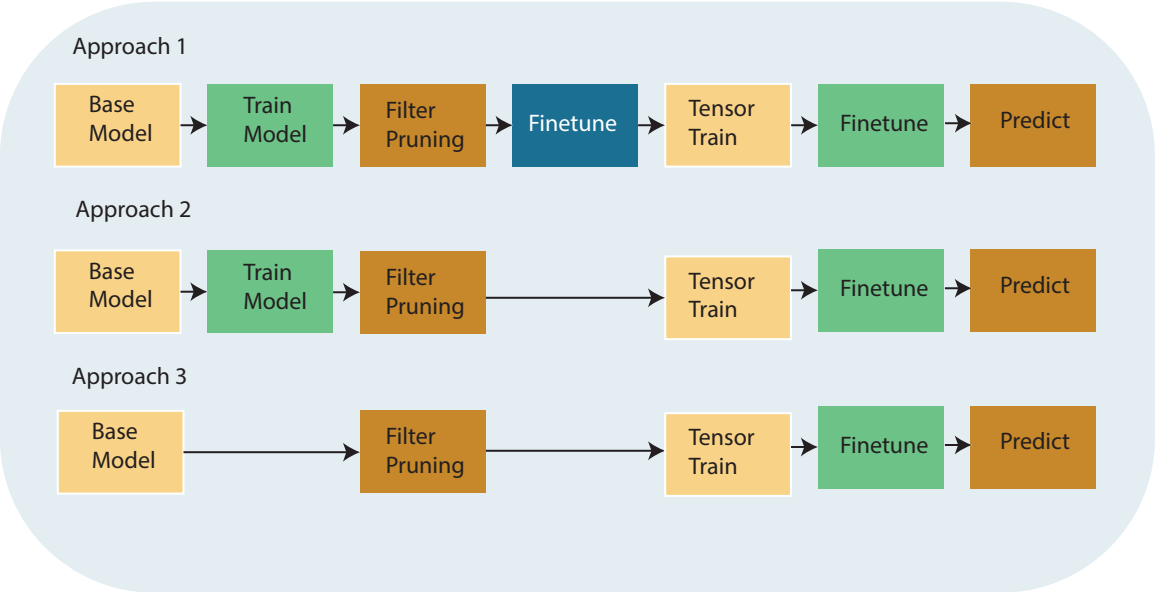


Figure 4.4: Three different approaches applied in the experiment: approach one trained base model, applied filter pruning, retrained the model, applied TT, retrained and finally predicted. Second approach trained base model, applied filter pruning, applied TT, retrain, and finally predicted. Third approach, applied filter pruning on the base model, applied TT, trained, and finally predicted

DNN models such as LeNet-5 and AlexNet, resulting in a small degradation of accuracy. We discovered that the second approach slightly improved model accuracy regardless of the model complexity. When we used the first approach, DNN models such as VGG-16 and VGG-19 performed well in terms of compression ratio and accuracy performance. Our experiment uses three different models, namely VGG-16, AlexNet, and LeNet-5. In our experiment, we used the same metric presented in[62] to compute the number of the neurons that were removed from the model, which is $Reducation = M - C/M$ where M is the parameters number of the original model, and C is the parameters number of the compressed model.

4.3.1 VGG-16

We used the Keras framework in this experiment to build and train the VGG-16 architecture without using dropout or batch normalization layers [97]. We used three different datasets

to train and test our method, namely CIFAR-10, CIFAR-100, and ImageNet. We used horizontal flip, random shift, and random rotation data augmentation for CIFAR-10 and CIFAR-100. We trained VGG-16 for 100 epochs with a batch size of 256 and learning rate of 0.05, and achieved 90.65% accuracy for CIFAR-10 and 62.95% accuracy for CIFAR-100. The number of parameters for the trained VGG-16 was 3.36×10^7 for CIFAR-10 and 3.4×10^7 for CIFAR-100. Next, We applied the second approach of our method, where we pruned the filters and retrained and replace the fully connected layers with TT layers. By doing so for VGG-16 on CIFAR-10, we achieved an accuracy of 90.99% with 5.04×10^6 parameters. Using the metric described in the previous section, we reduced the model by 85% compared with the original model, which is better than [62] and [78], where reductions of 64% and 41%, respectively, were achieved. We also applied the same method using CIFAR-100 and achieved accuracy of 64.07% with 5.43×10^6 parameters. This is reduced the model by 84.03% compared with the original model. For ImagNet dataset we used the pretrained VGG-16 models provided from the Keras framework which had an accuracy of 71.1% and 1.38×10^8 parameters. After applying our method using the second approach, in which we pruned the filter and trained the model for 100 epochs using stochastic gradient decent(SGD) with a learning rate of 0.01 and momentum of 0.9 we achieve accuracy of 73.4% with only 8.71×10^6 parameters meaning that we removed 93.69% parameters of the original model. The Table 4.1 compares between the base models with the FPTT models in terms of accuracy and the number of parameters. Figure 4.5 illustrates the number of parameters when we applied FP, and TT individually and when we applied the FPTT method.

4.3.2 AlexNet

We used the Keras framework in this experiment to build and train the AlexNet architecture without using dropout or batch normalization layers [54]. We used two different datasets to train and test our method, namely CIFAR-10 and CIFAR-100. We used horizontal flip,

Table 4.1: Comparison Between Original VGG-16 and FPTT VGG-16

Network Type	*Accuracy%*	*Parameters*	*Removed*
VGG-16			
CIFAR-10-base	89.9%	3.36×10^7	-
CIFAR-100-base	62.9%	3.4×10^7	
Imagenet-base	71.1%	1.38×10^8	
CIFAR-10-FPTT	90.99%	5.04×10^6	85%
CIFAR-100-FPTT	64.07%	5.43×10^6	84.03%
Imagenet-FPTT	73.4%	8.71×10^6	93.69%

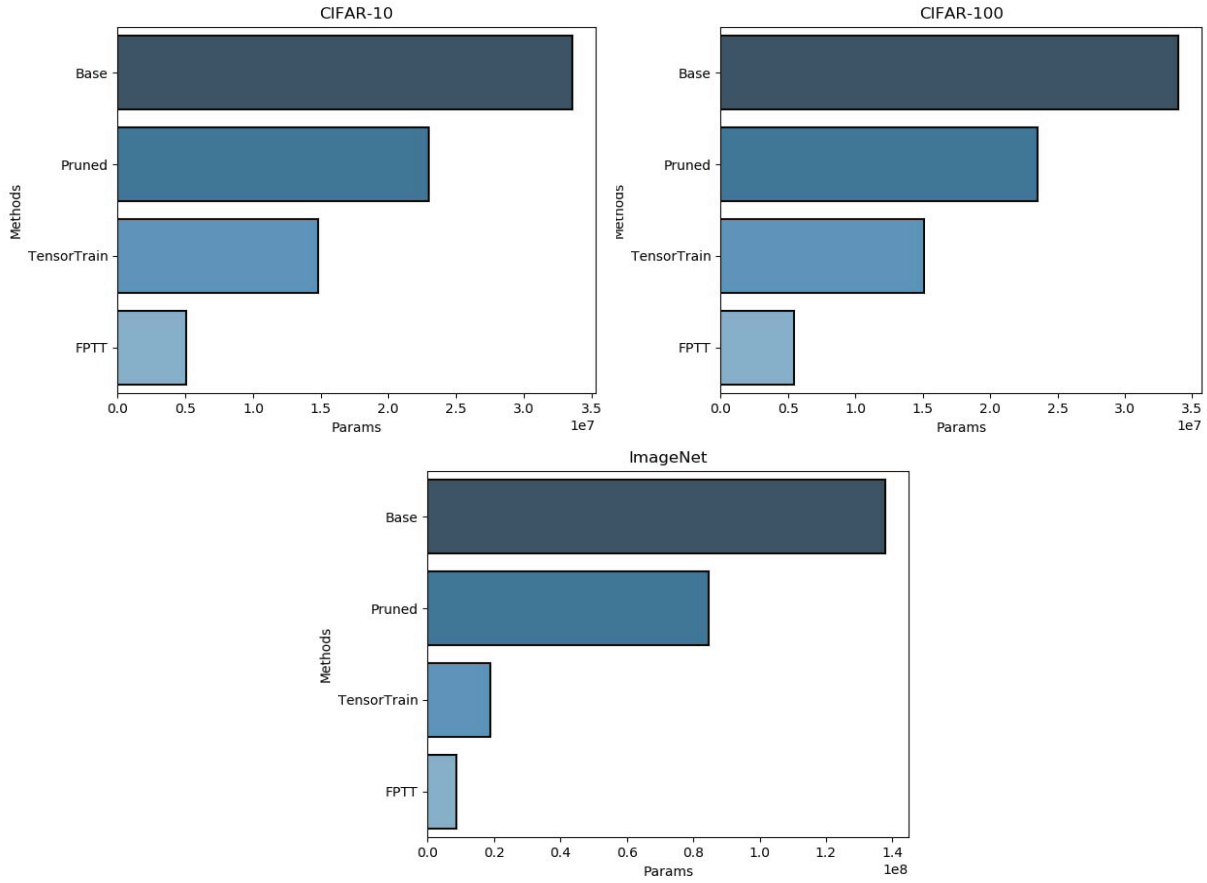


Figure 4.5: VGG-16 model on CIFAR-10,CIFAR-100 and ImageNet showing number of the parameters with base model, filter pruned model, TT decomposed model and FPTT model.

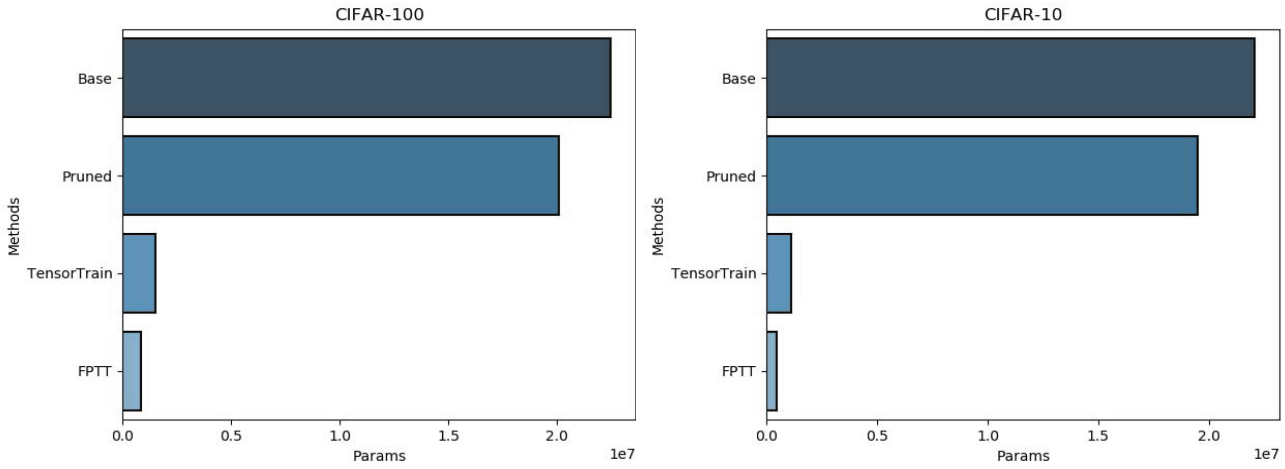


Figure 4.6: AlexNet model on CIFAR-10 and CIFAR-100 showing the number of parameters with the base model, filter pruned model, TT decomposed model and FPTT model

random shift, and random rotation data augmentation for CIFAR-10 and CIFAR-100. We trained AlexNet for 250 epochs with a batch size of 128 and a learning rate of 0.05, and achieved 86.95% accuracy for CIFAR-10 and 56.3% accuracy for CIFAR-100. The number of parameters for the trained AlexNet was 2.21×10^7 for CIFAR-10 and 2.25×10^7 for CIFAR-100. Applied the second and the third approaches of our method on AlexNet, the second approach performed better in terms of accuracy but the third approach was faster in terms of training time. Using the third approach for AlexNet on CIFAR-10 we achieved an accuracy of 85.93% with 7.76×10^5 parameters. Using the metric described in the previous section, we reduce the model by 96.5% compared to the original model. We also apply the same method using CIFAR-100 and we achieve accuracy of 55.56% with 1.1×10^6 parameters. This is reduced the model neurons by 94.5% compared with the original one. Table 4.2 compares between the base models, with the FPTT models in terms of accuracy and number of parameters. Figure 4.6 presents the number of parameters when we applied FP, and TT individually and when we applied the FPTT method.

Table 4.2: Comparison Between Original AlexNet and FPTT AlexNet

Network Type	*Accuracy%*	*Parameters*	*Removed*
AlexNet			
CIFAR-10-base	86.95%	2.21×10^7	-
CIFAR-100-base	56.3%	2.25×10^7	-
CIFAR-10-FPTT	85.93%	7.67×10^5	96.5%
CIFAR-100-FPTT	55.56%	1.1×10^5	94.5%

4.3.3 LeNet-5

We use the Keras framework in this experiment to build and train the LeNet-5 architecture [57]. We used two different datasets to train and test our method, namely CIFAR-10, and MNSIT. Furthermore, use horizontal flip, random shift, and random rotation data augmentation for CIFAR-10, and MNSIT. We trained LeNet-5 for 100 epochs with a batch size of 256 and a learning rate of 0.05, and achieved 75.04% accuracy for CIFAR-10 and 86.39% accuracy for MNSIT. The number of parameters for the trained LeNet-5 was 1.74×10^6 for CIFAR-10 and 1.31×10^6 for MNSIT. When we applied the second and the third approach of our method on LeNet-5, the second approach perform better in terms of accuracy but the third approach was faster in terms of training time. Using the third approach for LeNet-5 on CIFAR-10, we achieved an accuracy of 73.77% with 2.24×10^4 parameters. Using the metric described in the previous section, we reduced the model parameters by 98.71% compared with original model. We used the second approach of our method with MNSIT and achieved an accuracy of 89.99% with 8.86×10^4 parameters. This reduced the model neurons by 93.2% compared with the original model. Table 4.3 compares between the base model, and FPTT models in terms of accuracy and the number of parameters. Figure 4.7 presents the number of the parameters when we applied FP, and TT individually and when we applied the FPTT method.

Table 4.3: Comparison Between Original LeNet-5 and FPTT LeNet-5

Network Type	*Accuracy%*	*Parameters*	*Removed*
	AlexNet		
CIFAR-10-base	75.04%	1.74×10^6	-
Mnsit-base	86.39%	1.31×10^6	-
CIFAR-10-FPTT	73.77%	2.24×10^4	98.71%
Mnsit-FPTT	89.99%	8.86×10^4	93.2%

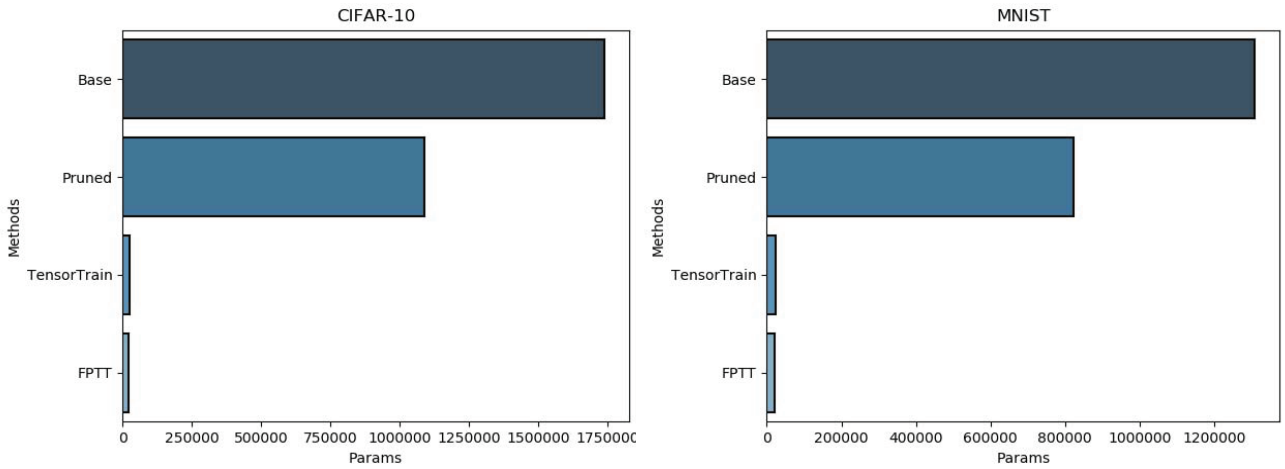


Figure 4.7: LeNet-5 model on CIFAR-10 and MNIST showing number of parameters with base model, filter pruned model, TT decomposed model and FPTT model

4.4 Conclusion

Recent research has indicated that many neural network parameters are redundant, and that state-of-the-art neural network architectures are overparameterized. To address these concerns, we presented the FPTT method, which consists of two stages: the first FP, followed by the replacement of fully connected layers with TT layers. FPTT method outperformed when FP and TT when performed individually. Table 4.4 presents a comparison between the original models and the compressed models using the FPTT method for CIFAR-10.

Table 4.4: Models in Base and FPTT

Network Type	***Original in MB***	***FPTT in MB***	***Compression Factor***
Models			
VGG-16	269.2MB	37.6MB	7.1x
Alexnet	176.8MB	6.2MB	28.5x
LeNet-5	14MB	455.6KB	30.7x

Chapter 5

Ultimate Compression: A Joint Method of Binary Neural Networks and Tensor Decomposition

In this chapter, we apply tensor decomposition methods to floating-point deep neural network models and then binarize the models using a type of BNNs method called XNOR-Net[87]. To summarize, the contributions of this chapter are as follows:

1. We propose an efficient deep neural network model by applying Ultimate Compression method which is a joint method of the tensor decomposition and BNNs.

2. We introduce a rank selection algorithm with which we decompose the models based on the sensitivity of the layer for decomposition.

3. We compare between three different ranks selection algorithm: using Random method, Variational Bayes Matrix Factorization (VBMF), and our method which we select the ranks based on the layers sensitivity.

4. We present our method's results using six different models on four different datasets, namely LeNet-5 on MNIST, Network-in-Network, AlexNet, ResNet-20, and ResNet-32 on CIFAR-10, ResNet-20 and ResNet-32 on CIFAR-100, and finally Alexnet and Resnet- 18 on ImageNet.

5. We present a discussion and analysis on how to improve the accuracy of the decomposed binary models using different optimizers, different activation functions, and different methods for training the models.

6. We demonstrate that the decomposed binary models yield a deeper model, which takes more time to converge. Furthermore, by applying orthogonal initialization, the model converges faster.

5.1 Tensor Decomposition

We used three different tensor decomposition methods: CP, Tucker, and TT. The tensor decomposition methods were used on both convolutional and fully connected layers. We applied the three different methods on a 3D tensor, as illustrated in Figure 5.1.

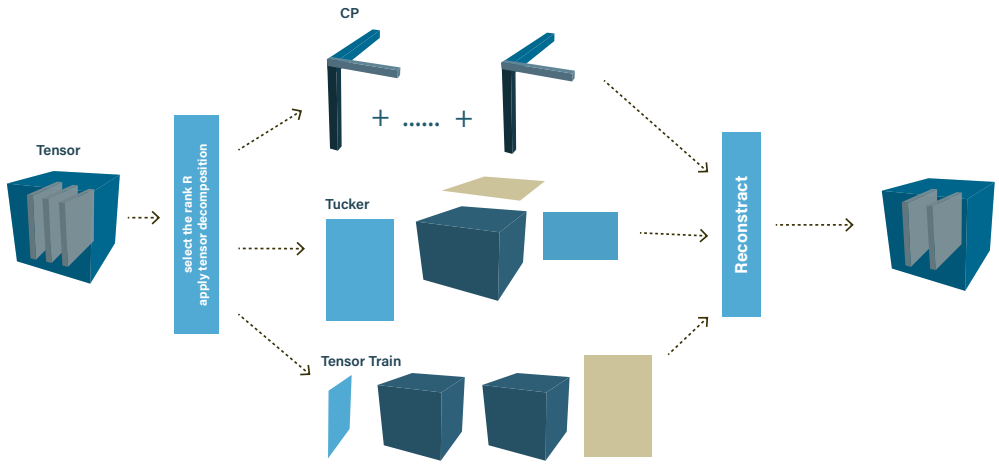


Figure 5.1: Three different methods of tensor decomposition on a three-order tensor

5.1.1 CP Decomposition

CP factorizes a tensor into a linear combination of the rank of one tensor[51]. Figure 5.1 depicts a three-order tensor. The formal definition of CP decomposition for a Nth tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ decomposes in the outer product matrices and R is the rank $\boldsymbol{U}_1, \boldsymbol{U}_2, \ldots \boldsymbol{U}_d$ as follows :

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{u_1}_r \circ \mathbf{u_2}_r \circ \ldots \mathbf{u_d}_r \tag{5.1}$$

The factor matrices are a combination of the vectors from the rank one components such as $\mathbf{u} = \begin{bmatrix} \mathbf{u_1}_1 & \mathbf{u_1}_2 & \cdots & \mathbf{u_1}_R \end{bmatrix}$ and likewise for $\mathbf{U_2}$ and $\ldots \mathbf{U_d}$. The column $\mathbf{U_1}$, $\mathbf{U_2}$, and $\ldots \mathbf{U_d}$ is very often normalized to the unit length with weights absorbed into a vector $\boldsymbol{\lambda} \in \mathbb{R}^R$, as follows:

$$\mathcal{X} = \sum_{r=1}^{R} \lambda_r \mathbf{u_1}_r \circ \mathbf{u_2}_r \circ \ldots \mathbf{u_d}_r \tag{5.2}$$

For a given tensor, several algorithms exist for computing CP decomposition. In this study, we employed an alternative least square (ALS) algorithm. The core idea of this algorithm is to optimize each factor matrix individually, keeping all tensor factor matrices fixed except the one that is optimized, and then repeating this task for each matrix until the stopping criterion is satisfied [51].

We used CP tensor decomposition on the convolutional and fully connected layers. The rank of the tensor was required in order to apply the decomposition. Finding the tensor's rank is an NP-hard problem. To approximate the tensor rank, numerous algorithms and methods exist. In this study, we implemented and investigated three different approaches to select the ranks. First, we used a random rank for all of the layers, which is a random number based on the size of the tensor. In the second approach, we chose the rank based on the layer's sensitivity to decomposition. In

Algorithm 1 ALS for CP decomposition[14]

Input: Data tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ and rank R.
Output: Factor matrices

$$\boldsymbol{U}_1 \in \mathbb{R}^{n_1 \times R}, \boldsymbol{U}_2 \in \mathbb{R}^{n_2 \times R}, \ldots \boldsymbol{U}_d \in \mathbb{R}^{n_d \times R}$$

procedure ALS-CP($\mathcal{X}$,R)
    Initialize $\boldsymbol{U}_1, \boldsymbol{U}_2, \ldots \boldsymbol{U}_d$
    while Not having converged or arrived at a satisfied criterion **do**
        $\boldsymbol{U}_1 \longleftarrow \mathbf{X}_{(1)}(\mathbf{U}_d \odot \mathbf{U}_{d-1} \odot \cdots \mathbf{U}_2)$
        $\left(\mathbf{U}_d^{\mathrm{T}}\mathbf{U}_d \circledast \mathbf{U}_{d-1}^{\mathrm{T}}\mathbf{U}_{d-1} \cdots \circledast \mathbf{U}_2^{\mathrm{T}}\mathbf{U}_2\right)$
        Normalize the column vector U_1 to unit length.
        $\boldsymbol{U}_2 \longleftarrow \mathbf{X}_{(2)}(\mathbf{U}_d \odot \mathbf{U}_{d-1} \odot \cdots \mathbf{U}_1)$
        $\left(\mathbf{U}_d^{\mathrm{T}}\mathbf{U}_d \circledast \mathbf{U}_{d-1}^{\mathrm{T}}\mathbf{U}_{d-1} \cdots \circledast \mathbf{U}_1^{\mathrm{T}}\mathbf{U}_1\right)$
        Normalize the column vector U_2 to unit length.
        $\vdots$

        $\boldsymbol{U}_d \longleftarrow \mathbf{X}_{(d)}(\mathbf{U}_{d_1} \odot \mathbf{U}_{d-2} \odot \cdots \mathbf{U}_1)$
        $\left(\mathbf{U}_{d-1}^{\mathrm{T}}\mathbf{U}_{d-1} \circledast \mathbf{U}_{d-2}^{\mathrm{T}}\mathbf{U}_{d-2} \cdots \circledast \mathbf{U}_1^{\mathrm{T}}\mathbf{U}_1\right)$
        Normalize the column vector U_d to unit length.
        Store the norms in vector λ
    end while
    return $\boldsymbol{U}_1, \boldsymbol{U}_2, \ldots \boldsymbol{U}_d$ and λ
end procedure

the third approach, we used VBMF to determine the rank [77]. These methods are explained in more detail in the next section.

5.1.2 Tucker Decomposition

Tucker tensors are composed of core tensors multiplied by each matrix along the mode[51]. Tucker decomposition for Nth tensors decomposes to the outer product matrices, where r is the rank, and g is the core tensor as follows:

$$\mathcal{X} = \sum_{r_1=1}^{R_!} \sum_{_2=1}^{R_2} \cdots \sum_{r_n=d}^{R_d} g_{r_1 r_2 \cdots r_d} \mathbf{u_1}_{r_1} \circ \mathbf{u_2}_{r_2} \circ \ldots \mathbf{u_d}_{r_d} \tag{5.3}$$

The factor matrices $\mathbf{U_1}$,$\mathbf{U_2}$, and $\ldots \mathbf{U_d}$ can be considered as Principle Component Analysis (PCA) for every mode. The core tensor $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \cdots \times R_d}$ indicates the different interactions between the different components [51].

Several algorithms exist for determining the Tucker decomposition for a given tensor, including high-order SVD (HOSVD) and high-order orthogonal iteration (HOOI). HOSVD can be considered the basic definition of PCA, in which the component that best captures the variations in mode n is found. In this study, we used HOOI, an ALS algorithm that uses the HOSVD outcome as the factor matrix initialization [14].

Tucker decomposition is used on convolutional and fully connected layers. Finding the best Tucker approximation is also an NP-hard problem. We used the same approaches that we used for CP decomposition to select the Tucker rank.

5.1.3 Tensor Train Decomposition

TT decomposition decomposes a tensor of order n into a chain of product tensors of order-two or order-three tensors. TT decomposition is a type of non-recursive tensor decomposition that, unlike Tucker decomposition, does not suffer from the curse of

Algorithm 2 HOOI for Tucker Decomposition[14]

Input: Data tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$

R Ranks in each mode.

Output: Core Tensors $\mathcal{G}$, Factor matrices

$$\boldsymbol{U}_1 \in \mathbb{R}^{n_1 \times R}, \boldsymbol{U}_2 \in \mathbb{R}^{n_2 \times R}, \ldots \boldsymbol{U}_d \in \mathbb{R}^{n_d \times R}$$

1: **procedure** HOOI-TUCKER($\mathcal{X}$,R)
2: Initialize $\boldsymbol{U}_1, \boldsymbol{U}_2, \ldots \boldsymbol{U}_d$ Using HOSVD
3: **while** Criteria Not Satisfied **do**
4: **for** n **do**=1, ...,N do
5: $\boldsymbol{Y}_n \longleftarrow \mathcal{X} \times \mathbf{U_1}^T \times \cdots\cdots\cdots \times \mathbf{U_{n-1}}^T \times \mathbf{U_{n+1}}^T \cdots \times \mathbf{U_N}^T$
$\mathbf{U}_n \longleftarrow \mathbf{R}_n$ Leading Singular Vector of $\mathbf{Y}_n$
6: **end for**
7: **end while**
8: **return** $\mathcal{G}$,$\boldsymbol{U}_1, \boldsymbol{U}_2, \ldots \boldsymbol{U}_d$
9: **end procedure**

dimensionality [79]. The formal definition of the Nth order tensor decomposes to second- or third-order tensors, where r is the rank, as follows:

$$\mathcal{X}(u_1, ..., u_d) = G_1[u_1]G_2[u_2]...G_d[u_d]. \tag{5.4}$$

Where G_d is a core tensor and can be of order two or three. All tensors $G_d[u_d]$ related to the same dimension d must be of the same size $r_{d-1} \times r_d$ and $r_0, r_d = 1$. The chain of $\{r_d\}_{d=0}^{d}$ is the rank of the TT format.

TT-SVD, TT-ALS, TT rounding, and other algorithms are used to compute TT decomposition. We used recursive TT-SVD on the tensors in this study, and the algorithm for decomposing the convolutional layers was based on [23]. Finding the approximate TT decomposition for a given tensor is an NP-hard problem. In this study, we used a method similar to the previous decomposition to find the rank, but with a greater emphasis on the layer sensitivity approach due to the promising results, as demonstrated in the following sections. On both convolutional and fully connected layers, we used TT decomposition.

Algorithm 3 SVD for Tensor Train Decomposition[79]

Input: Tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, accuracy ϵ

Output: Cores tensors $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_d$ of TT approximation $\mathcal{X}$ with TT ranks $r_0 = r_d = 1$

procedure SVD-TT($\mathcal{X}$,R)
    Initialize
    Temporary Tensor $\mathcal{T} = \mathcal{X}, r_0 = 1$
    for k=1 to d-1 **do**
        Compute truncated SVD: $\mathcal{T} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$
        $r_k \coloneqq rank(\mathcal{T})$
        $G_k \coloneqq reshape(\mathbf{U}, [r_{k-1}, n_k, r_k])$
        $\mathcal{T} \coloneqq \Sigma\mathbf{V}^T$
    end for
    $\mathcal{G}_d \coloneqq \mathcal{T}$
    return $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_d$
end procedure

5.1.4 Layer Sensitivity and Rank

We used a heuristic method based on layer sensitivity. We tested the layers' sensitivity using six different ranks: 1, 5, 20, 40, 60, and 80. The CP, Tucker, and TT tensor decomposition methods were used. Although no significant differences exist in accuracy between Tucker and TT decomposition, TT decomposition has a higher compression ratio than Tucker, as indicated in Table 5.1.

Table 5.1: ResNet-20 Architectures on CIFAR-10.

Network Type	*Top1%*	*Params Size MB*	*Compression*
ResNet-20			
FP-Model	92.60%	1.1	-
CP-Model	77.60%	0.368	2.9x
Tucker-Model	91.180%	1	1.1x
TT-Model	91.330%	0.947	1.16x

We applied TT decomposition on the layers. Figure 5.2 presents the sensitivity of AlexNet model layers after decomposition with the TT method on convolutional and fully connected layers before and after finetuning the model. Figures 5.2.a and 5.2.b depict the model before finetuning, revealing that the layers became more robust the

deeper we went in the model. Using a small rank such as 5 or 20 for the deeper layers yielded the same results as for the undecomposed layers. Table 5.2 presents the study and compression of each layer when we decomposed with different ranks. Figures 5.2.c and 5.2.d depict the accuracy of the layers after we finetuned the models for between 20 and 25 epochs. Table 5.2 also studies the layers' accuracy and compression ratio after finetuning. From Table 5.2 and Figures 5.2.c and 5.2.d, we can see that selecting a rank between 40 and 80 would yield the same accuracy or a small degradation in accuracy compared with undecomposed models.

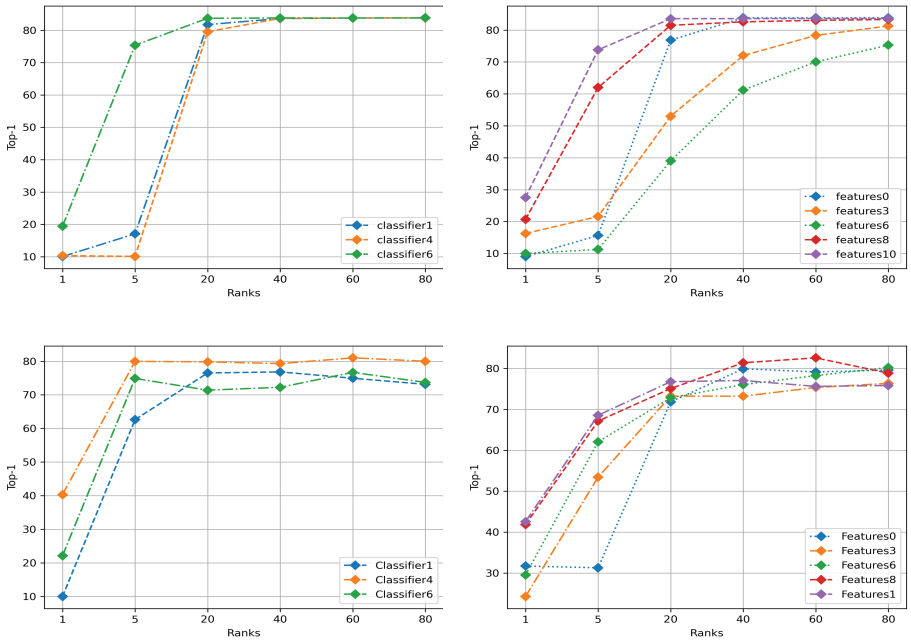


Figure 5.2: Layer sensitivity of the AlexNet model after applying tensor decomposition with different ranks: (a) Sensitivity of the classifier layers that are fully connected layers before finetuning; (b) sensitivity of the feature extraction layers that are convolutional layers before finetuning; (c) sensitivity of the classifier layers that are fully connected layers after finetuning; (d) sensitivity of the feature extraction layers that are convolutional layers after finetuning.

In addition, we investigated the layer sensitivity of the ResNet-20 model. ResNet-20 has 19 convolutional layers and one fully connected layer. Figure 5.3 presents four graphs that depict the layer sensitivity across Resnet-20's three basic blocks. From Figure 5.3 and Tables 5.4, 5.5, and 5.6, we inferred that with only a rank of 20 for

Table 5.2: Layer Sensitivity for AlexNet Model Feature Layers Before and After Applying Tensor Train Decomposition

Rank	*Top1% before*	*Top1% After*	*Compression*
Features-0			
FP-Model	83.80%	-	-
1	9.03%	31.71%	23.6x
5	15.63%	31.29%	3.84x
20	76.830%	71.81%	0.905x
40	83.809%	79.90%	0.67x
60	83.809%	79.13%	0.67x
80	83.809%	79.59%	0.67x
Features-3			
FP-Model	83.809%	-	-
1	16.25%	24.29%	422.1x
5	21.590%	53.41%	77.3x
20	53.0%	73.23%	14.7x
40	72.009%	73.30%	5.57x
60	78.299%	75.34%	2.99x
80	81.26%	76.34%	2.04x
Features-6			
FP-Model	83.809%	-	-
1	9.98	29.51	1140.1x
5	11.23%	62.06%	218.9x
20	39.01%	72.83%	47.6x
40	61.18%	76.049%	20.32x
60	70.02%	78.23%	11.8x
80	75.26%	80.22%	7.85x
Features-8			
FP-Model	83.809%	-	-
1	20.69%	41.84%	1369.5x
5	61.97%	67.129%	264.1x
20	81.479%	75.080%	58.20x
40	82.559%	81.37%	25.13x
60	83.0400%	82.57%	14.7x
80	83.299%	83.12%	9.87x

Table 5.3: Layer Sensitivity for AlexNet Model Features-10 and Classifiers Layers Before and After Applying Tensor Train Decomposition

Features-10			
FP-Model	83.809%	-	-
1	27.53%	42.59%	1138.65x
5	73.779%	68.54%	217.64x
20	83.5599%	76.75%	46.6x
40	83.5699%	77.059%	19.60x
60	83.619%	75.610%	11.27x
80	83.619%	75.83%	7.43x
Classifier-1			
FP-Model	83.809%	-	-
1	10.03%	10.0%	21845.33x
5	17.079%	62.63%	1456.35x
20	81.73%	76.51%	104.025x
40	83.629%	76.83%	28.54x
60	83.82%	78.94%	10.11x
80	83.619%	79.89%	7.43x
Classifier-4			
FP-Model	83.809%	-	-
1	10.329%	40.34%	65536x
5	10.090%	79.95%	4369.06x
20	79.52%	79.81%	312.07x
40	83.669%	79.29%	79.921x
60	83.790%	81.009%	35.81x
80	83.799%	82.02%	25.28x
Classifier-6			
FP-Model	83.809%	-	-
1	19.38%	22.12%	303.407x
5	75.36%	74.89%	20.74x
20	83.709%	71.40%	2.88x
40	83.80%	72.25%	1.449x
60	83.809%	76.68%	0.96x
80	83.809%	78.72%	0.906x

the first basic block, we could achieve the same performance or a small degradation based on the layers. However, the second basic block required a rank between 60 and 80 to achieve the same accuracy as undecomposed layers. For the third basic block, we achieved the same accuracy with a rank of only 60, as indicated in Figure 5.3 and Table 5.6. The model became more robust as we went deeper into the model, where using a small rank would not yield a large degradation, as was the case when we decomposed the layers in the second basic block.

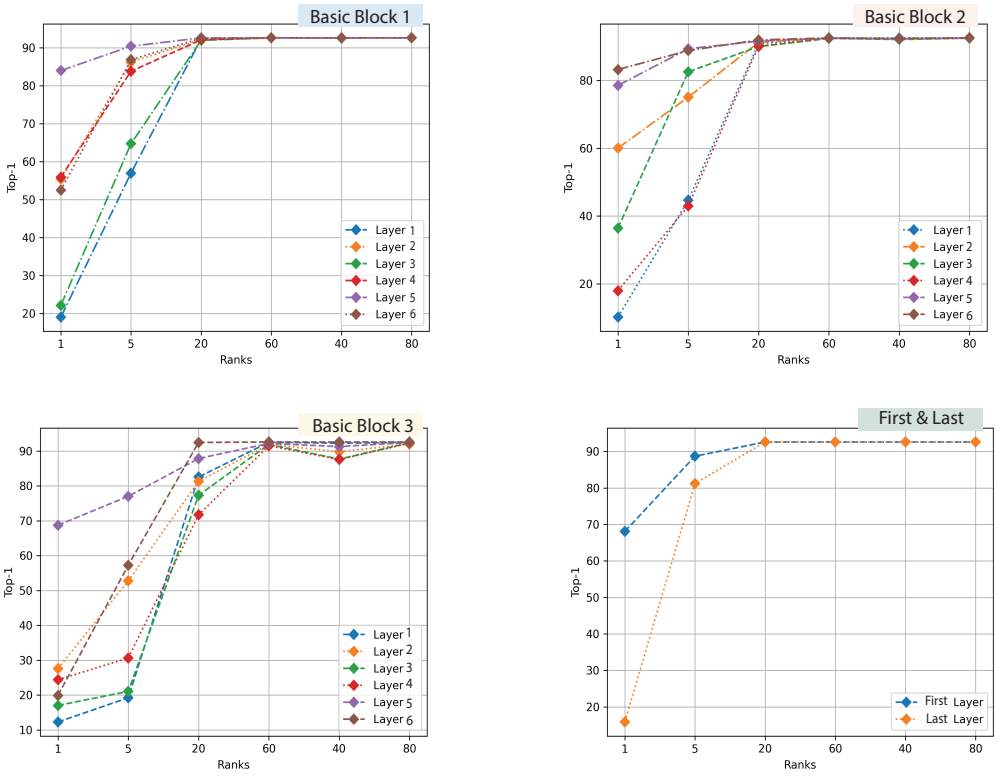


Figure 5.3: Layer sensitivity of the ResNet-20 model after applying tensor decomposition with different ranks: (a) Sensitivity of the first basic block before finetuning; (b) sensitivity of the second basic block before finetuning; (c) sensitivity of the third basic block before finetuning; (d) sensitivity of the first basic block after finetuning.

Rank	*Top1% Bef*	*Top1% Aft*
Basic Block 1		
FP-Model	92.6%	-
1	19.03%	91.26%
5	56.93%	91.75%
1	55.39%	91.59%
5	86.20%	91.84%
1	22.1%	91.80%
5	64.70%	91.89%
1	55.90%	91.36%
5	83.790%	91.34%
1	83.970%	91.75%
5	90.400%	91.79%
1	52.45%	91.72%
5	86.82%	91.89%

Table 5.4: Layer Sensitivity for the ResNet-20 Model's First Basic Block Before and After Applying Tensor Train Decomposition

Rank	*Top1% Bef*	*Top1% Aft*
Basic Block 2		
FP-Model	92.6%	-
1	10.159%	90.91%
5	44.71%	91.54%
1	60.06%	91.36%
5	75.09%	91.86%
1	36.41%	91.36%
5	82.58%	91.72%
1	17.97%	91.15%
5	42.98%	91.01%
1	78.55%	91.32%
5	89.38%	91.47%
1	83.25%	91.58%
5	88.83%	91.72%

Table 5.5: Layer Sensitivity for the ResNet-20 Model's Third Basic Block Before and After Applying Tensor Train Decomposition

5.2 Binary Neural Networks

BinaryConnect was one of the first BNNs quantization methods [16]. BinaryConnect limits the weight of the neural network to +1 or 1, replacing the multiply–accumulation operation with simple additions or subtractions. The weight binarization for the inference stage is presented in Equation (5.5), which is referred to as deterministic binarization. Real values are quantized during forward propagation using the equations in deterministic binarization (6). However, the error cannot propagate during backpropagation because the gradient is zero almost everywhere. To mitigate this problem, a straight-through estimator (STE) is used, which is a heuristic method for estimating the gradient of the stochastic neuron, as presented in Equation (5.6), where (x) is the value before binarization [7].

Table 5.6: Layer Sensitivity for ResNet-20 Model Basic Before and After Applying Tensor Train Decomposition

Rank	*Top1% Before*	*Top1% After*
Basic Block 3		
FP-Model	92.6%	-
1	12.28%	89.05%
5	19.25%	90.47%
20	82.55%	92.11%
40	92.25%	91.94%
60	92.54%	91.97%
1	27.61%	90.03%
5	52.77%	90.31%
20	81.31%	91.199%
40	89.80%	91.45%
60	91.889%	91.86%
1	17.04%	89.0%
5	21.06%	90.61%
20	77.30%	91.62%
40	87.680%	91.89%
60	91.91%	91.95%
1	24.40%	88.47%
5	30.63%	89.09%
20	71.709%	90.72%
40	87.56%	91.68%
60	91.53%	91.97%
1	68.73%	90.29%
5	77.02%	90.47%
20	87.86%	91.43%
40	91.27%	91.52%
60	92.189%	92.03%
1	19.90%	89.790%
5	57.25%	91.40%
20	92.47%	92.22%
40	92.589%	92.12%
60	92.6%	92.22%

BinaryConnect only binarizes weights, whereas XNOR-net, which was used in the present study, binarizes both the weight and the input of the convolutional layers [87].

$$w_b = \begin{cases} +1 & if\ w \geq 0, \\ -1 & else. \end{cases} \tag{5.5}$$

$$STE(x) = \begin{cases} 0 & if\ x < -1, \\ 1 & if\ -1 \geq x \leq 1, \\ 0 & if\ x > 1. \end{cases} \tag{5.6}$$

The weight values in XNOR-net are approximated using binary filters, as demonstrated below; by treating quantization as an optimization problem, as in the equation, a better scale factor can be selected.

$$I * W \approx (I \oplus \beta)\alpha \tag{5.7}$$

$$J(\beta, \alpha) = \|W - \alpha\beta\|^2 \tag{5.8}$$

Here,W denotes real value filters, B denotes binary filters, and $alpha$ denotes a positive scaling factor. The binary weight filter is the sign of the weight values after solving this optimization problem, and the scaling factor is the average of the absolute weight values.

$$\beta^* = sign(W),\ \alpha^* = \frac{1}{n} \|W\|_{l\,1} \tag{5.9}$$

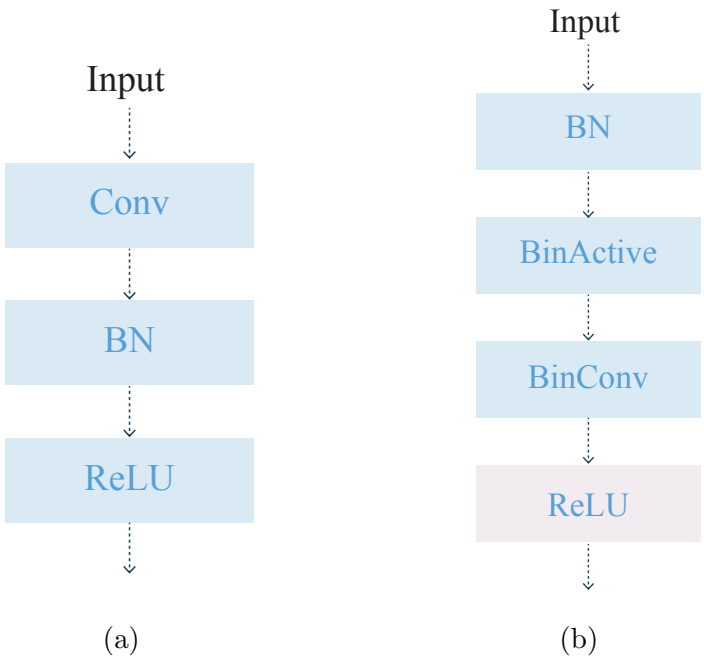


Figure 5.4: Layers' connection for a convolution: (a) for conventional fp32; (b) for BNN

A block of XNOR-net differs from a block of a CNN, as depicted in Figure 5.4.

The batch normalization (BN) [40] layer is placed before the binary activation layer. The BN is formulated as follows:

$$\hat{x}_i \rightarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}}, \tag{5.10}$$

where μ_β and σ are the mini-batch mean and variance for a channel. Iterative mini-batches from the previous layer's outputs are used in the training of μ_β and σ. A convolution output x_i means each element in a channel. Term ϵ prevents the division by zero. After the normalization, the BN layer scales and shifts the normalized feature $\hat{x}_i$ into x_i in a channel, which can be equated as:

$$x_i \rightarrow \lambda \hat{x}_i + \beta, \tag{5.11}$$

where the affine parameters λ and β are learnable during the CNN training.

The BN layer can change the range of the convolution output distribution, and the adjusted convolution output is used as the input to the binary activation layer. Figure 5.4 illustrates the layer connection for a convolutional layer, where *Conv* and *BinConv* denote the conventional fp32-based convolutional and binarized convolutional layers, respectively. Morever, the term *BN* and *BinActive* mean the batch normalization and binary activation layers, respectively. In a conventional fp32-based CNN of Figure 5.4 (a), the convolutional layer outputs go towards the next BN layer, and the BN layer outputs go towards the activation layer such as *ReLU*. Figure 5.4 (b) depicts the layer connection of a binarized convolution in [87]. The BN layer is located before the BinActive layer to adjust features with its learnable parameters. The features binarized from the BinActive layer go towards the BinConv layer. It is noted that the output format of the BinConv layer becomes fp32 after its scaling.

5.2.1 Tenosrized quantized models

We selected the rank in order to decompose the models using three different methods, the first of which was VBMF. To apply VBMF algorithm tensors, the tensor must be 2D. Thus, we unfolded the convolutional layer based on modes 0 and 1 and then applied VBMF on the unfolded tensor; the rank was the first dimension of the diagonal matrix computed by the VBMF algorithm [77][11]. The second method was a heuristic method based on the sensitivity of the layers to decomposition. We selected six different fixed ranks and tested the models layer by layer as explained in the previous section.

For the third method, we used a random method. We used a random number to decompose the model layer by layer. The random number was selected based on the layer dimensions in which we selected a low and a high range for each layer. The low range in the convolutional layer was the size of the kernel. Low range for the fully connected layers was the low number of the matrix shape. For the high range

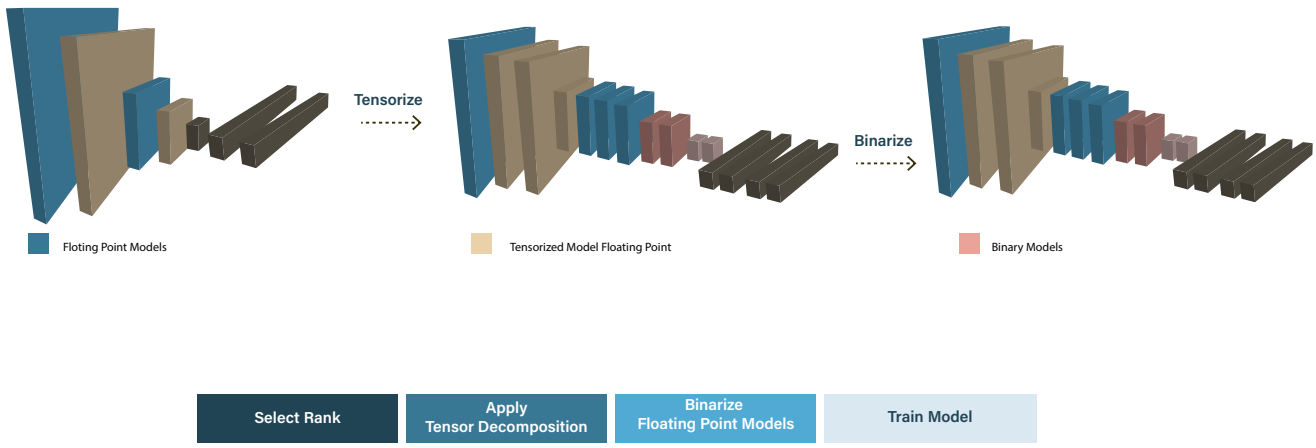
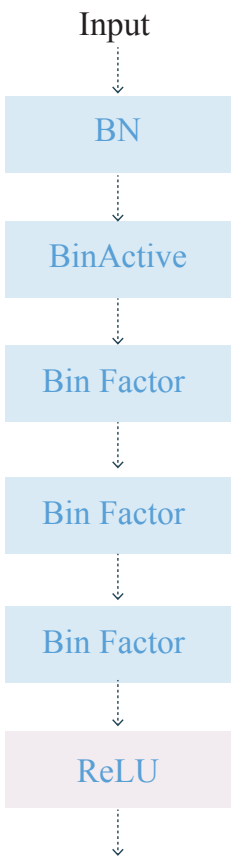


Figure 5.5: Our propsed method in which We select the rank to decompose the models, based on the layer senstivty and after binarize the model using xnor-net method and finally train the the model

for the convolutional layers, we unfolded the tensor to two dimensions for the layers and selected the high number of the unfolded tensor. For the high range for the fully connected layers, we used the same method; that is, the large dimension of the matrix was used for decomposition.

After applying our tensor decomposition method, we applied the XNOR-Net method to binarize the decomposed model, where we binarized the decomposed layers using the method explained in the previous section. A block of decomposed XNOR-Net differs from an XNOR-Net block and a CNN block, as illustrated in Figure 5.6.

In a CNN, the convolutional operation maps the input tensor X of size H x W x S to the output tensor y of size S x W' X H' using a tensor kernel of size $D \times D \times S \times T$ in which T, and S are the output and input, respectively, and D is the spatial dimension.



(a)

Figure 5.6: Layer's connection for a convolution: (a) For Tensorized BNN

$$\mathcal{Y}_{h',w',t} = \sum_{i=1}^{D}\sum_{j=1}^{D}\sum_{s=1}^{S}\mathcal{K}_{i,j,s,t}\mathcal{X}_{h_i,w_j,s} \tag{5.12}$$

We applied CP decomposition with rank R as shown in the following equation. Spatial dimensions usually small are not decomposed like filter of size 1.

$$\mathcal{K}_{t,s,j,i} = \sum_{r=1}^{R}\mathbf{U}^{(1)}_{r,s}\mathcal{U}^{(2)}_{r,j,i}\mathbf{U}^{(3)}_{t,r} \tag{5.13}$$

$\mathbf{U}^{(1)}_{r,s}\mathbf{U}^{(2)}_{r,j,i}\mathbf{U}^{(3)}_{t,r}$, are of size $R \times S$, $R \times D \times D$, and $T \times R$ respectively.

We applied CP decomposition from input tensor X to output tensor y, which is presented in the following equation after substituting Equation 5.13 into Equation 5.12 [55].

$$\mathcal{Y}_{t,w',h'} = \sum_{r=1}^{R} \mathbf{U}_{t,r}^{(3)} \left(\sum_{j=1}^{D} \sum_{i=1}^{D} \mathcal{U}_{r,j,i}^{(2)} \left(\sum_{s=1}^{S} \mathbf{U}_{r,s}^{(1)} \mathcal{X}_{s,w_j,h_i} \right) \right) \tag{5.14}$$

We applied Tucker decomposition with rank R as shown in the following equation:

$$\mathcal{K}_{i,j,s,t} = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} \mathcal{C}'_{r_1,r_2,r_3,r_4} U_{i,r_1}^{(1)} U_{j,r_2}^{(2)} U_{s,r_3}^{(3)} U_{t,r_4}^{(4)} \tag{5.15}$$

Where $\mathcal{C}'$ is the core tensor of size f $R_1 \times R_2 \times R_3 \times R_4$ and $U_{i,r_1}^{(1)} U_{j,r_2}^{(2)} U_{s,r_3}^{(3)} U_{t,r_4}^{(4)}$ are of a factor of size $D \times R_1$, $D \times R_2$, $S \times R_3$ and $T \times R_4$ respectively [47]. $U_{i,r_1}^{(1)} U_{j,r_2}^{(2)}$ could be ignored when we applied tucker decomposition because it referred to the spatial information which mostly small. We applied Tucker decomposition from input tensor $\mathcal{X}$ to output tensor Y, which is presented in the following equation after substituting Equation 5.15 in Equation 5.12 [47].

$$\mathcal{Y}_{t,w',h'} = \sum_{r=1}^{R_4} \mathbf{U}_{t,r}^{(4)} \left(\sum_{j=1}^{D} \sum_{i=1}^{D} \sum_{r_3=1}^{R_3} \mathcal{C}'_{r_1,r_2,r_3,r_4} \left(\sum_{s=1}^{S} \mathbf{U}_{r,s}^{(3)} \mathcal{X}_{s,w_j,h_i} \right) \right) \tag{5.16}$$

We applied TT decomposition with rank R. The convolutional layers were formulated through matrix-by-matrix multiplication in which the four-way tensor reshaped into a matrix K of size $D^2 S \times T$, and we applied TT format in which G is TT-cores, as discussed in [23]. We obtained the following decomposition of the convolutional kernel:

$$\mathcal{K}_{t,s,j,i} = \boldsymbol{G}_0[i,j] \boldsymbol{G}_1 [t_1, s_1] \dots \boldsymbol{G}_d [t_d, s_d] . \tag{5.17}$$

We used the same substitution as in the previous methods to map tensor X to a tensor

Y by convolving X with kernel K as follows:

$$\mathcal{Y}_{t,w',h'} = \sum_{j=1}^{D} \sum_{i=1}^{D} \sum_{s_1,\ldots,s_d} \mathcal{X}_{s,w_j,h_i} \boldsymbol{G}_0[i,j] \boldsymbol{G}_1[t_1, s_1] \ldots \boldsymbol{G}_d[t_d, s_d]. \tag{5.18}$$

5.3 Experiment Results

To test our method, we used four different datasets, namely MNIST, CIFAR-10, CIFAR-100, and ImageNet. To train the LeNet-5 model on MNIST, we used random horizontal flip data augmentation, transformed the images to tensors and normalizing them.

In terms of data augmentation for CIFAR-10, we used a random crop of 32 with padding of 4, and we also used random horizontal flip. We transformed the images into tensors and normalized them using standard PyTorch parameters for mean and standard deviation. To test our method, we used four different models: NiN, AlexNet, ResNet-20, and ResNet-32.

For CIFAR-100, we used the same data augmentation as for CIFAR-10. To test our method, we used two different architectures, namely Resnet-20 and Resnet-32.

For ImageNet, we used the same data augmentation as for the previous datasets. We used two different architectures to test our method, namely AlexNet and ResNet-18.

To train the models for MNSIT, CIFAR-10, and CIFAR-100, we used one Nvidia GeForce GTX 1080 Ti GPU. For the ImageNet dataset, we used four Nvidia Tesla V100 GPUs. We created and trained the model with the PyTorch library [81] and then used the Tensorly library [52] to apply tensor decomposition algorithms on the models.

Next, we modified the AlexNet model by adding two layers of 0.5 dropout after the first fully connected layer and before the last fully connected layer, as well as by using

the ReLU activation function and batch normalization, which are not implemented in the vanilla version of the AlexNet model [54]. For the CIFAR-10, AlexNet, and NiN floating-point models, we used a batch size of 32 to train the models for 320 epochs. We used the Adam optimizer with an initial learning rate of 3e-4 and a weight decay of 1e-4. Furthermore, we used the ReduceLROnPlateau scheduler with a patience of 10 to reduce the learning rate by a factor of 0.001.

Then, we decomposed the AlexNet and NiN floating-point models using the layer sensitivity method described in the previous section. We then finetuned the models for between 25 and 50 epochs using AdamW as the optimizer with the same hyperparameters used to train the previous models.

To binarize the AlexNet and NiN models, we modified them using the XNOR-Net method explained in the previous section, and we use 0.5 and 0.2 for dropout layers, respectively, instead of 0.5 for the floating-point models. We trained the model for 500 epochs using Adam with a learning rate of 1e-4 and weight decay of 1e-5. Then, we used the ReduceLROnPlateau scheduler with a patience of 50 to reduce the learning rate by a factor of 0.005.

To binarize the decomposed model, after applying the sensitivity method to the models. The models had a small number of parameters and decomposed layers, and they degraded by 1%–2% compared with nondecomposed floating-point models, as indicated in Tables 5.7, 5.8, 5.9, and 5.10. We then binarized the decomposed layers using the XNOR-Net method, training the models for 320 epochs with an Adam optimizer of 0.01 and weight decay of 1e-5. Next, we used the ReduceLROnPlateau scheduler with a patience of 50 to reduce the learning rate by a factor of 0.005. After applying the XNOR-Net method on the decomposed model, the accuracy degraded by 1%–2% compared with the nondecomposed binary models. For ResNet-20 and ResNet-32 on CIFAR-10 and CIFAR-100, we trained the floating-point models using Adam with a learning rate of 1e-4 and weight decay of 1e-7 for 320 epochs. We also use the ReduceL-

RonPlateau scheduler with a patience of 50 to reduce the learning rate by a factor of 0.005. After decomposing the model using layer sensitivity, as presented in Table 2, we finetuned the models for 25–50 epochs with AdamW with the same hyperparameters as the floating-point models.

To binarize the decomposed model, after applying the sensitivity method on models without decomposing and binarizing the first and last layers. We trained the models for 320 epochs with Adam with a learning rate of 0.01 and weight decay of 1e-5. We used the ReduceLRonPlateau scheduler with a patience of 50 to reduce the learning rate by a factor of 0.005. For more tuning, we trained the model for 100 epochs with AdamW with a learning rate of 3e-4 and weight decay of 1e-5. The accuracy of the models degraded by 1%–2% compared with nondecomposed binary models but a greater compression ratio. For AlexNet and ResNet-18 for ImageNet. For the floating-point models, we used the pre- trained models from the PyTorch Torchvision library [81]. We decomposed the model based on the layer sensitivity and finetuned it using Adam with a learning rate of 1e-4 and weight decay of 1e-7 for 20–25 epochs. We binarized the model except for the first and last layers and trained it from scratch for 70 epochs using Adam with a learning rate of 0.01 and weight decay of 1e-4; moreover, we used the ReduceLRonPlateau scheduler with a patience of 10 to reduce the learning rate by a factor of 0.005.

For binary decomposed models, we neither binarized nor decomposed the residual of the first and last layers. We trained the model for 70 epochs with Adam with a learning rate of 3e-4 and weight decay of 1e-5. We also used ReduceLRonPlateau scheduler with a patience of 10 to reduce the learning rate by a factor of 0.005. Subsequently, we finetuned the model further using AdamW with a learning rate of 1e-5 and weight decay of 1e-7 for 10–15 epochs.

Finally, for all of the decomposed binary models, we used norm gradient clipping, which prevented the models from gradient exploding and accelerated the training.

Table 5.7: LeNet-5 Architectures on MNSIT

Network Type	*Top1%*	*Params Size MB*	*Compression*
LeNet-5			
FP-Model	99.06%	0.244	-
FP-Tensorized	98.75%	0.116	2.1x
BNN-Model	99.02%	0.11	10.8X
Ours	98.73%	0.067	17.9X

Table 5.8: Comparison of Different Architectures on CIFAR-10

Network Type	*Top1%*	*Params Size MB*	*Compression*
Network In Network			
FP-Model	87.72%	3.7	-
FP-Tensorized	86.640%	1.3	2.642x
BNN-Model	83.35%	0.299	12.37x
Ours	82.45%	0.113	32.74x
AlexNet			
FP-Model	87.240%	91.7	-
FP-Tensorized	86.68%	9.8	9.35x
BNN-Model	81.79%	2.85	32.2x
Ours	80.91%	0.542	169.1x
Resnet-20			
FP-Model	92.60%	1.1	–
FP-Tensorized	90.98%	0.62	1.7x
BNN-Model	81.87%	0.047	23.40x
Ours	80.92%	.0342	32.16x
Resnet-32			
FP-Model	93.53%	1.9	-
FP-Tensorized	91.56%	1.1	1.72x
BNN-Model	83.53%	0.071	26.760x
Ours	81.05%	0.054	35.18x

Table 5.9: Comparison of Different Architectures on CIFAR-100

Network Type	*Top1%*	*Params Size MB*	*Compression*
Resnet-20			
FP-Model	68.730%	1.2	-
FP-Tensorized	65.89%	1	1.2x
BNN-Model	50.17%	0.069	17.4X
Ours	48.66%	0.040	30.0X
Resnet-32			
FP-Model	70.12%	2	-
FP-Tensorized	68.54%	1.3	1.5X
BNN-Model	51.2%	0.093	21.5x
Ours	48.01%	0.076	26.3X

Table 5.10: Comparison of Different Architectures on ImageNet

Network Type	*Top1% / Top5%*	*Params Size MB*	*Compression*
AlexNet			
FP-Model	56.66%/79.09%	244	–
FP-Tensorized	54.24%/76.83%	28.32	8.61x
BNN-Model	46.69%/70.21%	22.83	10.68x
Ours	44.25%/69.78%	15.9	15.3x
Resnet-18			
FP-Model	69.75%/89.08%	46.8	-
FP-Tensorized	66.31%/86.21%	5.84	8.01x
BNN-Model	52.16%/72.24%	4.01	11.67x
Ours	50.06%/70.14%	2.64	19.02x

5.4 Discussion and Analysis Studies

In this section, we discuss in detail how we improved the accuracy of the model while keeping the number of parameters the same. We used two different models, namely AlexNet and Resnet-20 on CIFAR-10, which can be generalized for other decomposed binary models. We discuss how to improve the accuracy of these models by studying the impact of three different methods: initialization, activation functions, and finally rank selection algorithms. Model initialization plays a significant role in improving model accuracy and accelerating training and model convergence. We studied different initialization algorithms by applying them to BNN models. First, we studied and applied Xavier initialization on the BNN models. Xavier initialization is applied to make the weights such that the variance of the activations is the same across every layer. This helps to avoid the vanishing gradient problem [24]. We also studied Kaiming initialization, which is also used to avoid the vanishing gradient problem and is mostly used with the ReLU activation function [33].

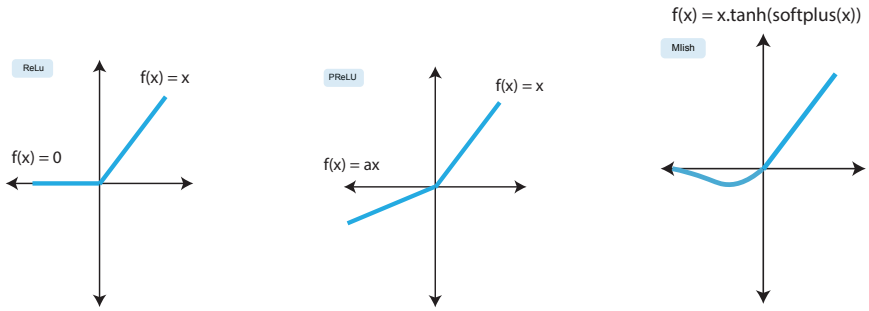
The depth of neural network models is crucial for model convergence. BNNs, introduced in the previous section, use batch normalization before the binary layer and an activation function after the binary layer, as depicted in Figure 5.4.b. The decomposition of binary layers, as illustrated in Figure 5.5, results in more deep neural network models. Training the decomposed BNN models that we introduced is a challenging task because the models are so deep that it takes longer for them to converge, and furthermore, they suffer from the vanishing gradient problem if not designed properly. The weights for Xavier and Kaiming initialization are drawn from iid Gaussian distributions. When the model's weights are drawn from iid Gaussian distributions, the error is stretched and skewed as the signal propagates back through the network.

To avoid this issue, rigorous studies by researchers have demonstrated that the input–output Jacobian is empirically linked to better convergence, and the stronger

condition where all singular values of the Jacobian concentrate near one is known as dynamical isometry [110] [111] [94] [98]. Dynamical isometry can be achieved through orthogonal weight initialization and has been empirically demonstrated to have excellent performance in terms of accuracy as well as to accelerate model convergence. Using orthogonal weight initialization improved our model accuracy, as Tables 5.11 and 5.11 present, for both the AlexNet and ResNet-20 models.

To improve our models even further, we investigated various activation functions and their impact on model accuracy. We employed the ReLU activation function, as presented in Equation 5.16 and Figure 5.7.a [3]. The problem with ReLU activation functions is that they produce dead neurons with weights less than x, thus preventing the model from properly fitting the data. To address this, a parametric ReLU (PReLU) with a different activation function introduced a learnable parameter that updated during training, as presented in Equation 5.17 and Figure 5.7.b [4]. We also used a Mish activation function, which is a nonmonotonic function that behaves like ReLU and PReLU in the positive region and is nonmonotonic in the negative region. The Mish activation function has positive derivatives at some points and negative derivatives at others, thus increasing the model expressivity, as demonstrated in Equation 5.18 and Figure 5.7.c. Furthermore, the fact that Mish is self-regularized helps in the optimization of deep models [69]. When these three different activation functions are applied to BNN models, the PReLU activation function with orthogonal initialization yielded the best results for AlexNet and ResNet-20. We used three different methods to select the rank for decomposing the models: VBMF, sensitivity, and random, as explained in the previous section. Using the sensitivity method, we lost accuracy of between 1%–2% compared with the binary counterpart and compressed the model by 169× and 32. × for AlexNet and ResNet-20, respectively, compared with the floating-point models.

We use three different methods to select the rank to decompose the models: VBMF, sensitivity, and random, as explained in previous section. Using the sensitivity method,



(a) (b) (c)

Figure 5.7: Layer's connection for a convolution: (a) ReLU activation function. (b) PeRLU activation function. (c) Mish activation function.

we lose in accuracy between 1%˜2% compared to its binary counterpart and compressed the model by 169x and 32.16x for Alexnet and ResNet-20, respectively, compared to the floating-point models.

$$\text{ReLU(x)} = \begin{cases} x & if\ x > 0, \\ 0 & if\ x \leq 0, \end{cases} \tag{5.19}$$

$$\text{PReLU(x)} = \begin{cases} x & if\ x_i > 0, \\ a_i x_i & if\ x_i \leq 0, \end{cases} \tag{5.20}$$

As shown in Equation (5.21), the Mish activation function apply Tanh on Softplus activation function.

$$\text{Mish(x)} = x \cdot \tanh(\text{softplus}(x)) \tag{5.21}$$

In where $tanh(x)$ is :

$$\text{Tanh(x)} = \left(e^{x} - e^{-x}\right) / \left(e^{x} + e^{x}\right) \tag{5.22}$$

and $sofplus(x)$ is :

$$\text{Softplus(x)} = \ln\left(1 + e^{x}\right) \tag{5.23}$$

Table 5.11: AlexNet Architecture Results on CIFAR-10

Method	*Top1%*	*Params Size MB*	*Compression*
Initialization			
Xaviar	81.79%	2.85	-
Kaiming	81.70%	-	-
Orthongal	82.01%	-	-
Activation Function			
ReLU	81.79%	2.85	-
PReLU	84.01%	-	-
Mish	82.68%	-	-
Rank			
VBMF	73.12%	0.162	566x
Senstivity	81.05%	0.542	169.1x
Random	75.98%	0.89	103x

Table 5.12: ResNet-20 Architectures Results on CIFAR-10

Method	*Top1%*	*Params Size MB*	*Compression*
Initialization			
Xaviar	81.87%	0.047	-
Kaiming	81.24%	-	-
Orthongal	82.55%	-	-
Activation Function			
ReLU	80.62%	0.047	-
PReLu	83.93%	-	-
Mish	82.30%	-	-
Rank			
VBMF	75.09%	0.015	70.96x
Senstivity	81.89%	0.034	32.16x
Random	78.17%	0.040	26.89x

Table 5.13: Comparison of Different Architectures on CIFAR-10

Network Type	*Top1%*	*Params Size MB*
ResNet-20		
Resnet20-FP[34]	92.60%	1.1
Mobile-net[37]	90.18%	12.4
Mobile-netv2[93]	91.29%	9.0
Efficient-net[102]	91.330%	11.4
ResNet20-Xnor[87]	83.93%	0.047
Ours	81.89%	0.034

Chapter 6

A Storage-Efficient Ensemble Classification Using Filter Sharing on Binarized Convolutional Neural Networks

This chapter proposes a storage-efficient ensemble classification for overcoming the low inference accuracy of BNNs. When external power is sufficient in a dynamic power system, classification results can be enhanced by aggregating the outputs of multiple BNN classifiers. However, memory requirements for storing multiple classifiers are a significant burden for devices at the edge. The proposed scheme shares filters from a trained CNN model to reduce storage requirements in the binarized CNNs instead of adopting fully independent estimators. While several filters are shared, the proposed method only trains unfrozen learnable parameters in the retraining step. We compared and analyzed the performances of the proposed ensemble-based systems depending on ensemble types and BNN structures on CIFAR datasets. Our experiments concluded that the proposed method involving filter sharing can be scalable with the number of

classifiers and effective at enhancing classification accuracy. With a binarized ResNet-20 model on the CIFAR-100 dataset, the proposed scheme achieved 56.74% final Top-1 accuracy with 10 BNN classifiers, which enhanced the performance by 7.58% compared with using a single BNN model.

6.1 Introduction

An ensemble-based system can improve the performance of CNNs by averaging the classification results from different models [30]. Each model acts as a single base classifier, and the combined prediction of multiple base classifiers is provided from the ensemble-based system with CNNs. In the same manner, ensemble BNNs can obtain better classification results by using multiple models s [104, 118], which increases the regularization of target solutions and enhances the inference accuracy. The ensemble in [104] stored weights of base classifiers derived from a BNN model by applying stochastic rounding to each real-valued weight multiple times. The authors in [118] demonstrated the trade-offs on the number of classifiers with BNNs. The methods in [104, 118] increased the inference accuracy using multiple base classifiers, so that memory requirements were proportional to the number of weight files in the base classifiers. They would not be suitable for an embedded system with limited storage resources.

Our study focused on a method for overcoming the storage cost limitation of BNN ensembles. By sharing the filters in convolutional layers, we were able to reduce the storage costs required by BNNs ensemble models. When the BNN base model classifiers share the filter weights from the pretrained BNN convolutional layers, the batch normalization layer weights and fully connected layer weights for the ensemble-based system are the only layers that should retrain. We summarize our contributions as follows:

- We proposed an ensemble method based on shared filter weights that reduces the amount of storage required for ensemble-based systems.
- In the proposed ensemble system, we improved the model's classification accuracy by introducing scalability with the number of base classifiers.
- We used three different method to binarize the models, namely XNOR-Net [87], Bi- Rea-Net [69], and ReActNet [68].
- We introduced shared filters weights of the base classifier with four different configurations for weight sharing. This enhances the inference accuracy of binary models without affecting the storage cost.
- We used various ensemble methods, and we describe and compare the details of each method in our evaluations.
- We used these ensemble methods with sharing filter weights to build models that outperformed the base models of ResNet-20 and ReAcNet-10 by 7.58% and 3%, respectively, with CIFAR-100 while retaining the same storage as the base models.
- We found that different ensemble methods had different effects on different models. We found that the fusion method worked the best with ResNet-20, while the begging method led to performance enhancement when used with ResNet-18.

6.1.1 Ensemble Learning

Ensemble methods, also known as ensemble learning, are a powerful tool for improving deep neural network model performance and model generalization. An ensemble-based system is defined as the implementation of ensemble learning. An ensemble-based system combines multiple machine learning algorithms or models to outperform those that use a single algorithm or model. An ensemble-based system comprises multiple base estimators that are combined to form a strong estimator [30]. Ensemble methods include fusion, voting, begging, and gradient boosting, among many others. Finding

the best model in the search space that yields the fewest errors is difficult in statistical learning, which is the foundation of machine learning and deep learning. It is difficult because the datasets are always smaller than the search space. Researchers discovered a way to mitigate this by using various ensemble methods such as voting, which reduces the risk of selecting a bad model, bagging, and boosting with different starts, which result in improved approximation and fusion, thus expanding the model's function space [21]. Furthermore, using the ensemble methods to ensemble the same model with a different activation function or initialization method can increase diversity while decreasing correlation [71].

6.1.2 Ensemble Methods

Fusion and voting are fundamental ensemble methods used to construct ensemble-based systems. In the fusion-based ensemble, the averaged prediction from base estimators is used to calculate the training loss. When M base estimators $e^1, e^2, ..., e^m, ..., e^M$ are used,the output can be $\mathbf{o_i} = \frac{1}{M}\sum_{m=1}^{M} \mathbf{o_i}^m$ for a given sample $\mathbf{x_i}$. In the fusion, when y_i is the target output for $\mathbf{x_i}$, its loss function is $L(\mathbf{o_i}, y_i)$. On data batch B, its training loss can be $\frac{1}{B}\sum_{i=1}^{B} L(\mathbf{o_i}, y_i)$ in which training loss is used to update the parameters of all base estimators. When using a voting-based ensemble, each base estimator is created independently. With M base estimators, each base estimator e^m can be trained independently of the other base estimators. For the voting-based ensemble, the class is selected based on the majority of the hard votes from the base estimators in the inference.

For example, let us assume that there are two classes (dog, cat) in a dataset and three base estimators $(e^1, e^2, e^3$ in a voting-based ensemble. When estimators e^1 and e^2 classify a sample into *dog*, class *dog* is voted on the classification. Even if estimator e^3 votes for *cat*, class *dog* is selected by a majority vote. On the other hand, soft vot-

ing sums the prediction probabilities from estimators and then averages the summed values. The class with a high probability is selected in soft voting. For example, let us assume that a base estimator e^m outputs its predicting probabilities of classifying a sample for a set of class dog, cat, which is formulated as $P(e^m) = P_{dog}, P_{cat}$. When two base estimators have $P(e^1) = 0.7, 0.3$ and $P(e^2) = 0.2, 0.8$, their averaged probabilities can be $P_{avg} = \frac{0.7+0.2}{2}, \frac{0.3+0.8}{2}$. Using soft voting, the sample is classified into the class of cat. In the bagging-based ensemble [8], the subsampling with replacement produces multiple datasets to train each base estimator. In [108], different data batches for each base estimator were sampled with replacement and used to train base estimators independently. The boosting-based ensemble trains base estimators sequentially, where a base estimator is trained considering the errors from the previously trained base estimator. The snapshot ensemble [38], unlike other ensemble-based systems, has only one model and collects model parameters at each minima during training. As a result of the obtained different parameters obtained for the single model, multiple base estimators are obtained.

Notably, base estimators can also function as base classifiers for image classification. When base classifiers are used, their averaging of base classifiers is typically performed on the predicted probabilities of target classes through a SoftMax function as follows:

$$\sigma(\mathbf{z}^j)_i) = \frac{e^{z_i^j}}{\sum_{k=1}^{K} e^{z_k^j}}, \tag{6.1}$$

where term K is the number of the classes. Term z_i^j is an element of the input vector $\mathbf{z}$ on the j-th base classifier, so $\mathbf{z}^j = (z_i^j, ..., z_K^j) \in \mathbb{R}^K$ in which predictions from the base classifiers are averaged in the inference.

6.1.3 Binary Neural Network

In this chpater, we use three different versions of BNNs namely Xnor-Net which we explained in details in the previous chapter, and we refer to it in the chapter as the BNNs model. The second method we use is Bi-Real Net which significantly enhance Xnor-Net with adding a negligible computation cost. The thrid method is the ReActNet which improves Bi-Real Net method further.

Bi-Real Net

Bi-Real Net was specifically applied on ResNet architectures, where Bi-Real Net connected the activation function that after 1 bit convolution or batch normalization to the activation of consecutive block through an identity shortcut. This significantly improved the model's accuracy while incurring no additional computational costs.

To train the Bi-real net, obtain a close approximation for the non-differentiable sign function with respect to activation. In addition, a magnitude-aware gradient with respect to the weight was used to update the weight parameters, and finally, a clip function was used to reather that ReLU activation function for better initialization [69].

ReActNet

ReactNet is a method for further improving binary neural networks by modifying and binarizing the model and bypassing all intermediate convolutional layers, including the downsampling layer. In addition, with depth analysis for activation distribution variations, generalize the Sign and PReLU functions to RSign and RPReLU, which enable explicit learning of distribution reshape and shift at near-zero extra cost. In addition, distributional loss is used in this method to enable the binary network to learn the same distribution as the floating point counterpart. [68].

6.1.4 Ensemble BNNs

Ensemble BNNs combine multiple base classifiers to create a single strong classifier that outperforms the base model in terms of prediction accuracy. Several ensemble-based systems have used BNNs. The ensemble-based system in [104] applied stochastic rounding to a real-valued weight to obtain its binary weight. The stochastic rounding of a weight w can be performed in Eq. (6.5) as follows:

$$\mathrm{sr}(w) = \begin{cases} \lfloor w \rfloor & \text{with probability } 1 - (w - \lfloor w \rfloor) \\ \lfloor w \rfloor + 1 & \text{with probability } w - \lfloor w \rfloor \end{cases} \tag{6.2}$$

This system performs a kind of soft voting with base estimators that contain different binary weight files from one high-precision neural network. Each inference evaluation with a binary weight file could be considered a base classifier. The ensemble-based system averages prediction probabilities to enhance the classification accuracy. Although this ensemble-based system lowers the classification variance of the aggregated classifiers [104], its target system should store multiple weight files. The binary ensemble neural network (BENN) in [118] used bagging and boosting ensemble methods to obtain multiple models to be aggregated. These models with ensemble methods improve the inference accuracy, but multiple weights should be stored as well. This is a significant burden when deploying ensemble BNNs at the edge.

Various libraries assist in building ensemble methods with state-of-the-art deep neural network models. In this chapter, we use the *Ensemble-Pytorch* library [108], which is an open source library that supports different types of ensemble methods.

6.2 Proposed Method

6.2.1 Motivations

Using BNN models reduces both storage and computational costs. Using binary weights makes the BNN models use binary bitwise operations instead of FLOPs. Therefore, BNN models are feasible options for a low-resource constrained power-hungry embedded system. The problem with BNNs is that inference accuracy is degraded immensely compared with floating-point models. Furthermore, the power source status of a power-hungry embedded system typically varies. When the embedded system is connected to an external power source, sufficient power is available, resulting in high inference accuracy even when the system consumes much power. An embedded system with energy harvesting equipment, on the other hand, can use it as an additional power source. The model for this type of embedded system does not need to be run in low power mode. However, this is not the case for the vast majority of embedded systems. As a result, depending on the power source status, a trade-off occurs between inference accuracy and the amount of computation that can be adjusted. We note that an ensemble-based system based on BNNs can provide this trade-off. Multiple BNN models are aggregated to produce better classification results in the ensemble-based system using BNNs, with each trained model acting as a base classifier. Whereas only one BNN model can be used in low power mode, multiple BNN models can be aggregated when a sufficient power source is available. Using multiple BNN models, the model parameters are stored to train the base classifiers. This ensemble-based system, however, is not applicable if storage resources are limited.

Given the primary benefits of BNNs, the ensemble-based system's increasing storage requirements may limit its applications. To address this limitation, we introduce a method for sharing the weight parameters between BNN models, where we choose which parameters to share.

In CNNs, the convolutional layers consist of filters in which their weights are learned during the training stage. When sliding each filter on an image, features maps will be constructed that contain certain features for that image. These features can vary from abstract features, such as lines and edges, to more complex features, such as faces and cars. Using multiple BNN models for ensembling, the filters in convolutional layers can be shared among these models, which reduces the storage cost required by these models.

6.2.2 Proposed Ensemble-Based System Using BNNs

In our method, we build multiple base classifiers from a given pretrained model. Figure 8.1 illustrates our method for building two weak base classifiers A and B with identical architectures using the pretrained model. When filters are shared between models, the parameters of BinConv layers in the same position are identical. Between the base classifiers, the parameters of the BinConv layers in the same position can be identical to the shared filter weights. The shared filter weights are based on the filter weights of the pretrained model. In our method, we initialize the base classifiers with the same filter weights of BinConv layers across all base classifiers. During the retraining process, shared filter weights are *frozen* in all base classifiers. As a result, the filter weights of the BinConv layers in each base classifier retain their initial values and do not change during retraining.

We define the real-valued activation and weight filter as $\mathbf{X}$ and $\mathbf{W}$, respectively. In convolutional layers of BNNs, the dot product is approximated between $\mathbf{X}, \mathbf{W} \in \mathbf{R}^n$ such that $\mathbf{X}^\intercal \mathbf{W} \approx \beta \mathbf{H}^\intercal \alpha \mathbf{B}$, where $\mathbf{H}, \mathbf{B} \in \{+1, -1\}^n$ and $\beta, \alpha \in \mathbf{R}^+$. In other words, $\mathbf{H}$ and $\mathbf{B}$ denote the binary activation and filter, respectively. Terms β and α are scaling factors for weights and activations, respectively. The dot product can be optimized as

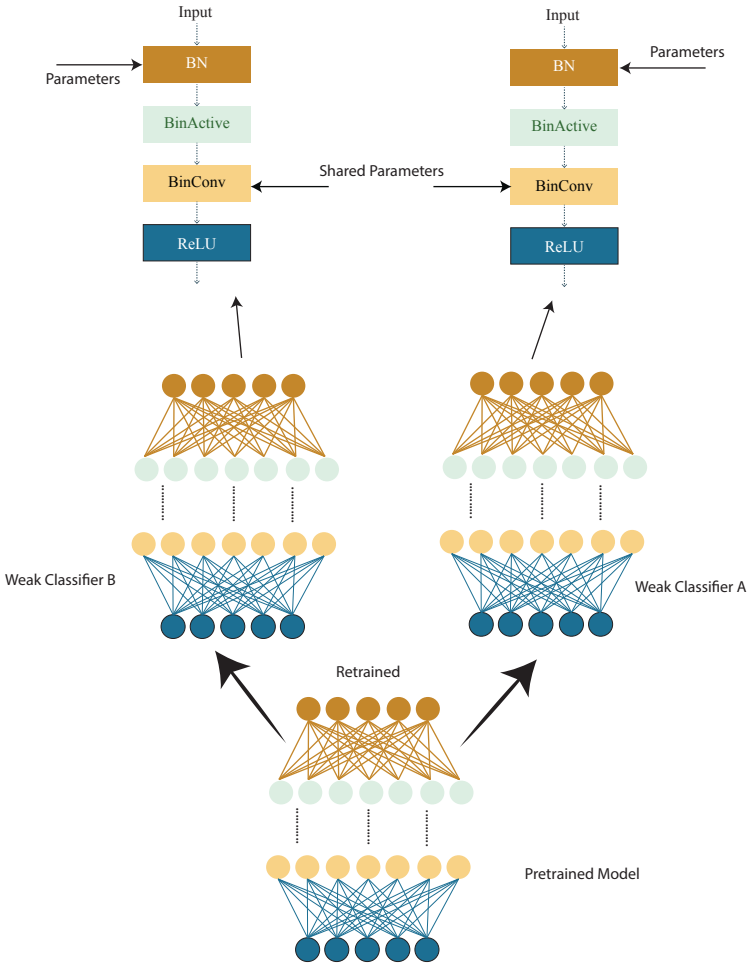


Figure 6.1: Conceptual figure of proposed ensemble-based system using BNNs.

follows:

$$\alpha^*, \mathbf{B}^*, \beta^*, \mathbf{H}* = \alpha, \mathbf{B}, \beta, \mathbf{H} \| \mathbf{X} \odot \mathbf{W} - \beta\alpha\mathbf{H} \odot \mathbf{B} \|. \tag{6.3}$$

We define the binary weight filter $\mathbf{B}^m$ from the m-th base classifier, where $m \in i | i \leq$. When the filter weights are reused, $\mathbf{B}^m = \mathbf{B}$. Term $\mathbf{H}^m$ denotes the binary activation from m-th base classifier.

On the other hand, the parameters of batch normalization layers in the same position

are different in the base classifiers. Formally, $x_i^m \rightarrow \lambda^m \hat{x}_i^m + \beta^m$, where $x_i^m \in \mathbf{X}^m$. Parameters λ^m and β^m in Eq. (5.10) are learnable in the retraining process. The scaling and shifting with parameters λ and β for each channel can adjust the normalized features to optimize the ensemble-based system. When activation value x_i becomes close to zero, the quantization error is maximized, which could produce large bias and variance in a classifier. When a base classifier produces a small quantization error with x_i^m, it is assured that the maximized quantization error with a single x_i can be mitigated. Morever, the dot products for each base classifier retain their optimization, as presented in Eq. (6.3). Morever, each last fully connected layer could have different learnable weights, adjusting the accumulation of the final features.

Algorithm 4 Training of the ensemble-based system using BNNs.

procedure TRAINING(pretrained model $BNN_{pretrained}$, training dataset $dataset$, number of base classifiers n, number of training steps T)
    $BNN_{base}(n) \Leftarrow$ Initialize($BNN_{pretrained}$, n)
    for $BinConv \in BNN_{base}(n)$ **do**
        Freeze($BinConv$)
    end for
    for $k \Leftarrow 1$ to T **do**
        $BNN_{base}(n) \Leftarrow$ Train($BNN_{base}(n)$)
    end for
    $weights \Leftarrow$ GetWeights($BNN_{base}(n)$)
    $weights \Leftarrow$ RemoveOverlap($weights$)
    return weights
end procedure

Algorithm 4 formally describes the aforementioned retraining process mentioned above. A pretrained model $BNNs_{pretrained}$ is used to initialize multiple n base classifiers $BNN_{base}(n)$. A function Freeze($BinConv$) prevents updating the weights of $BinConv$ layers in all base classifiers. During training T steps, the ensemble-based system with n base classifiers is trained. A function GetWeights($BNN_{base}(n)$) produces the weights of the retrained ensemble. A function RemoveOverlap($weights$) removes the overlapped weights of $BinConv$ layers between base classifiers. Finally, the trained $weights$ are returned.

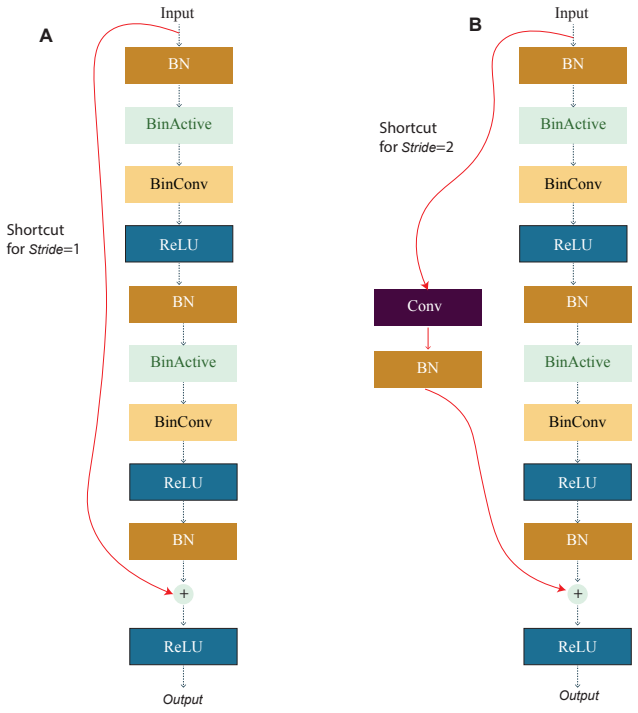


Figure 6.2: Basic blocks of binarized ResNet [34]: (a) $stride = 1$ (b) $stride = 2$.

For improved classification results, we can choose which filters are shared or not. Figure 6.2 illustrates basic blocks of the binarized ResNet [34], in which the basic blocks are stacked, thus maintaining a pyramid structure. The non-zero stride can reduce the resolution of output features by their height and width. When the number of channels is doubled in the ResNet, $stride = 2$ is used in the convolution.

In Figure 6.2 (a), a basic block contains the shortcut the sums the input features to the output of the last batch normalization layer. The heights and widths of input and output features are the same, respectively. The basic block of Figure 6.2 (a) contains two BinConv layers. In this case, it is possible that only one of them can share its filter weights in an ensemble-based system. In Figure 6.2 (b), when $stride = 2$, the height and width of the output features are half of those of the input features. An exact 1×1 Conv layer can be used in the shortcut when shrinking the feature dimension.

6.3 Hardware Analysis

6.3.1 Storage Resource Requirements

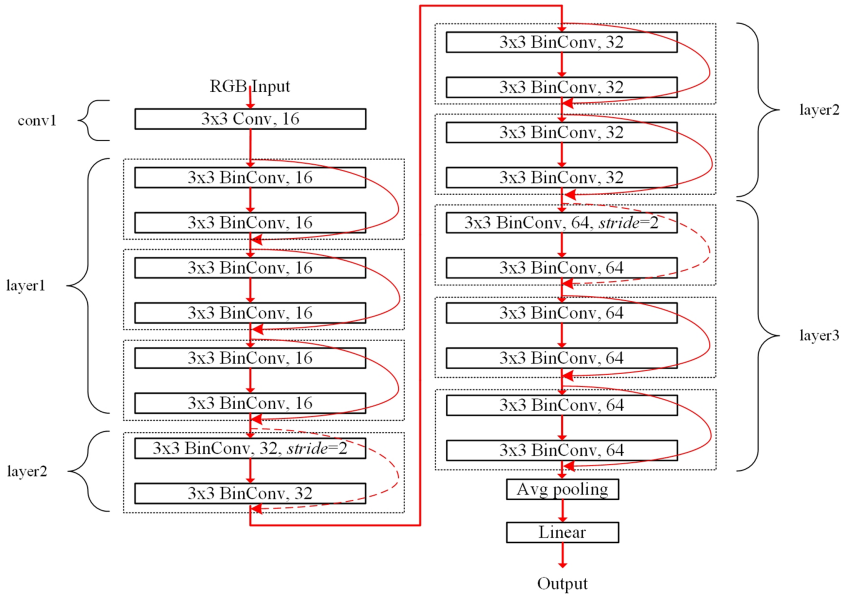


Figure 6.3: Binarized ResNet-20 structure for CIFAR dataset.

The filter sharing in our ensemble-based systems reduces the storage resource requirements. For example, the binarized ResNet-20 structure for the CIFAR dataset is presented in Figure 6.3. The *conv*1 and fully-connected linear layers are 32-bits uses floating-points operations [87]. The layer1, layer2, and layer3 blocks contain six 3×3 BinConv layers, respectively, where a basic block of the dotted box contains two BinConv layers. Each basic block contains the shortcut, which is indicated by the rounded red arrows. The dotted rounded red arrows indicate exact 1×1 convolutional layers used as a shortcut for shrinking the feature dimension with $stride = 2$. Finally, the average pooling layer (denoted as *Avg pooling*) averages the final convolutional outputs. The linear layer has full connections to all averaged outputs to produce the final classification result.

Table 6.1 lists the output size, layer description, and storage requirements. The weight

Table 6.1: Details of a Binarized ResNet-20 Model and Storage Resource Requirements on CIFAR-10

Block Name	*Output Size*[a]	*Layer Description*[b]	*(bits)*[c]
conv1	$16 \times 32 \times 32$	3×3, 3, $stride = 1$	$13,824 \times 32$
layer1	$16 \times 32 \times 32$	binarized 3×3, 16, $stride = 1$	$13,824 \times 1$
layer2	$32 \times 16 \times 16$	binarized 3×3, 16, $stride = 2$	$67,072 \times 1$
layer3	$64 \times 8 \times 8$	binarized 3×3, 32, $stride = 2$	$268,288 \times 1$
average pooling	$1 \times 1 \times 64$	8×8 average pooling	-
linear	10	1×1, 64, no stride	$20,480 \times 32$

[a] When the number of output channels and the width and height of output features are denoted as c_{out}, w_{out}, and h_{out}, the output size is calculated as $c_{out} \times w_{out} \times h_{out}$.
[b] Weight filter size $w \times h$, the number of input channels c_{in}, stride in the first convolutional layer of the basic block. When $stride = 2$, $c_{out} = c_{in} \times 2$.
[c] Memory requirements for storing weights.

size of each binarized convolutional layer can be calculated as $c_{in} \times w \times h \times c_{out}$ bits. On the other hand, the weight size of the first fp32 convolutional layer (conv1) is calculated as $c_{in} \times w \times h \times c_{out} \times 32$ bits. In the linear layer, c_{out} can be the same as the number of classes. The storage requirements for the linear layer increase with the number of classes. For example, the storage requirements of the linear layer for the CIFAR-100 dataset can be 204,800 bits, providing 100 image classes.

6.3.2 Computational Resource and Power Consumption

An ensemble-based system using these base classifiers can increase computations proportional to the number of base classifiers in the inference stage. Although filter weights are shared, each base classifier follows the same binarized CNN structure, performing the same number of multiply-accumulate operations when using fusion, voting, bagging, and boosting schemes. For example, in [90, 46], the binarized ResNet-18 for the CIFAR-10 dataset required 58.6×10^7 floating-point operations (FLOPs). When n base classifiers perform $n \times 58.6 \times 10^7$ FLOPs. Power consumption is e also proportional to the number of base classifiers. For example, in [27], the estimated power consumption

Table 6.2: Details of binarized ResNet-18 Model and Storage Resource Requirements on CIFAR-10

Block Name	*Output Size*[a]	*Layer Description*[a]	*(bits)*
conv1	$64 \times 32 \times 32$	3×3, 3, *stride* $= 1$	$(5.53E + 4) \times 32$
layer1	$64 \times 32 \times 32$	binarized 3×3, 64, *stride* $= 1$	$(1.47E + 5) \times 1$
layer2	$128 \times 16 \times 16$	binarized 3×3, 64, *stride* $= 2$	$(7.78E + 5) \times 1$
layer3	$256 \times 8 \times 8$	binarized 3×3, 128, *stride* $= 2$	$(3.11E + 6) \times 1$
layer4	$512 \times 4 \times 4$	binarized 3×3, 256, *stride* $= 2$	$(1.25E + 7) \times 1$
average pooling	$1 \times 1 \times 64$	8×8 average pooling	-
linear	10	1×1, 512, no stride	$(1.64E + 5) \times 32$

[a] Each layer block contains two basic blocks.

of the XNOR-Net model for the ImageNet dataset [18] was 1.92 mJ. In this case, if n base classifiers are adopted, the power consumption can be $n \times 1.92$ mJ.

6.4 Experimental Results and Analysis

6.4.1 Binarized ResNets on CIFAR Datasets

We trained the binarized ResNet-20 and ResNet-18 models on CIFAR datasets to evaluate our method. BNN architectures with a residual block with the same number of layers as plain BNN architectures performed better in terms of accuracy and model convergence time. Therefore, we used binarized ResNet models. In Figure 6.3 and Table 6.1, the binarized ResNet-20 model is a simple lightweight BNN model. The binarized ResNet-18 model, which was designed to classify ImageNet datasets, requires substantial amounts of computation and storage resources. Table 6.2 contains information about the binarized ResNet-18 model and the storage resource requirements. The storage requirements of the binarized ResNet-18 are approximately 47 times greater than those of the binarized ResNet-20. Furthermore, the binarized ResNet-18 model requires 13.5 times the computational resources.

6.4.2 Ensembles with Binarized ResNet

To evaluate the ensemble-based system, we trained the binarized ResNet-20 and ResNet-18 models on the CIFAR datasets [53]. Specifically, we evaluated the performance and scalability of the ensemble-based system using the CIFAR-10 and CIFAR-100 datasets.

To obtained the binarized ResNet-20 with initial weights, that denoted as $BNN_{pretrained}$ in Algorithm 4, we modified the model for CIFAR-100 by inserting dropout layers with a dropout rate of 0.5 [100]. We also replaced the ReLU activation function layers with the PReLU [33], which performs better, particularly for BNNs [25, 82, 72]. We applied data augmentation to the input images, where a 32×32 input image was randomly cropped from a 40×40 padded image and randomly flipped in the horizontal direction. We trained the model for 400 epochs with a batch size of 256 using Adam optimizer [48] with momentum of 0.9. We used a learning rate of 0.0001 that changed based on the polynomial policy in which learning rate lr decreased by $base_{lr} \times (1 - \frac{iteration}{epochs})$. We achieved a top-1 accuracy of 49.16% while the floating-point model had an accuracy of 64.24%. Then, using the Ensemble PyTorch library [108] described in the previous section, we applied several ensemble methods. The pretrained initial weights were used to initialize all filter weights of the binarized convolutional layers in each base weak classifier, which were then frozen during the retraining process. The filter weights were shared among classifiers regardless of the number of weak classifiers n. The learnable weights of the batch normalization layers and fully connected layers, on the other hand, were unfrozen and updated during the retraining process. We retrained the ensemble for 200 epochs with a batch size of 256 using the Adam optimizer with a momentum of 0.9. For ensemble models, we employed a learning rate of 0.01. Following retraining, both the boosting and snapshot ensemble methods degraded the accuracy of the binarized base model. The accuracy of the boosting method with $n = 2$ was only 35.68%, which was lower than the accuracy of the binarized base model, which

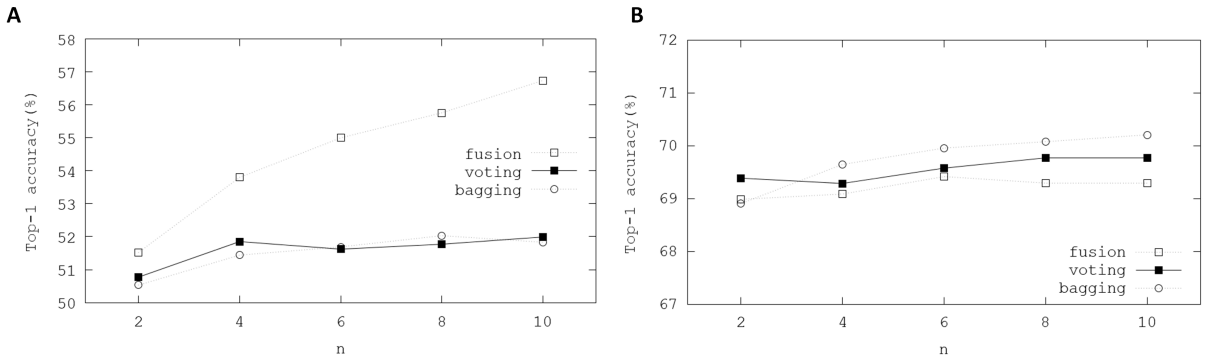


Figure 6.4: Top-1 inference accuracies of ensemble schemes using binarized ResNet models on CIFAR-100 dataset: (a) binarized ResNet-20 (b) binarized ResNet-18.

was 49.16%. As a result, we only used fusion, voting, and bagging for the experiment because these methods improve inference accuracy when $n = 2$.

In the fusion ensemble method, we found that improved accuracy was proportional to the number of ensemble n. By contrast, for voting and bagging ensemble methods, the accuracy was not improved with $n \geq 4$. We achieve 56.74% final Top-1 accuracy for the fusion method when $n = 10$, which was 7% higher than the binarized base model. Despite the fact that the filter weights were shared among the weak classifiers, the ensemble methods yielded higher accuracy, as indicated in 6.4.a.

We used the same model modification and training procedure that we used for ResNet-20 to obtain binarized ResNet-18 initial weights on CIFAR-100. We achieved a top-1 accuracy of s 68.59% for the binarized model, while the floating-point model had a top-1 accuracy of 75.61%.To apply the ensemble method, we shared the filters weights among the weak classifiers, which were frozen during the retraining process. The learnable weights of the batch normalization layers and fully connected layers, on the other hand, were unfrozen and updated during the retraining process, similar to binarized Resnet-20. Using two Nvidia Tesla V100 graphic processing units (GPUs), we retrained the model for 200 epochs with a batch size of 256. We also used the Adam optimizer with a learning rate of 0.01 and momentum of 0.9. In the fusion method, when $n \geq 8$, We use a batch size of 128 instead of 256 to avoid the issue that

GPU memory requirements exceeded the maximum resources. Figure 6.4.b presents the top-1 accuracy with different ensemble methods.

When we compared the ensemble binarized ResNet-20 with the ensemble binarized ResNet-18, we found that ResNet-18 performed better regardless of the ensemble method. While the fusion method achieved the highest top-1 accuracy for ResNet-20, the begging method with $n = 10$ achieved a top-1 accuracy of 70.21. Furthermore, we noticed that when we use ensemble methods in ResNet-18, the accuracy enhancement was lower than in ResNet-20, which was due to the initial weight having higher accuracy. Figure 6.4 indicates that ResNet-20 had a greater accuracy enhancement than ResNet-18.

We only use ReNet-20 for CIFAR-10. We modified the models in the same manner as for CIFAR-100, inserting dropout layers with a dropout rate of 0.5 and replacing the ReLU activation function with the PReLU activation function. We used the same data augmentation by cropping a 32×32 input image from a 40×40 padded image and using random horizontal flip. We trained the model for 400 epochs with a batch size of 256, using the Adam optimizer with a learning rate of 0.001 and momentum of 0.9. The learning rate changed based on the polynomial policy described previously. We achieved a top-1 accuracy of 84.06% for the base binarized ResNet-20. Using the ensemble method and the same retraining procedure as described previously, we achieved a top-1 accuracy of 85.30% for the fusion method when n = 2, and 86.99% when n = 10. We were able to conclude that ResNet-20 on CIFAR-10 exhibited a marginal improvement in accuracy when compared with CIFAR-100 because the base model has a high accuracy. Figure 6.5 presents the inference accuracies when we applied different ensemble methods.

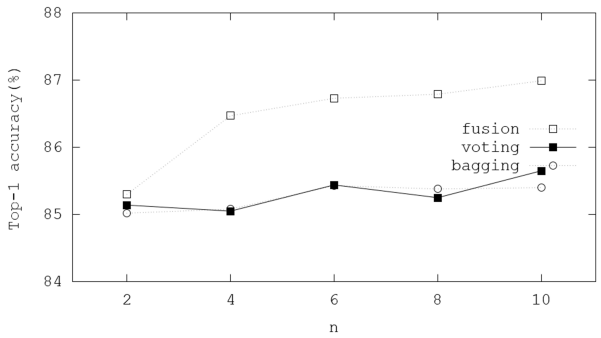


Figure 6.5: Top-1 inference accuracies of ensemble schemes using binarized ResNet-20 models on CIFAR-10 dataset.

6.4.3 Comparison of Different Configurations of Weight Sharing

This section introduces and compares various weight-sharing configurations, each of which improves accuracy in a different way. The configuration methods are described as follows:

- *No shared*: n base classifiers do not share weights in any layers.
- *Fusion*: The base classifiers share the weights of all binary convolutional layers and do not share the weights of batch normalization layers or the linear layers.
- *All Frozen*: The base classifiers share the weights of all convolutional and linear layers, and do not share the weights of batch normalization layers.
- C_1: The base classifiers share the weights of all convolutional and linear layers, except for the first convolutional layer.
- C_2: The base classifiers shared the weights of all convolutional layers, except for 1×1 convolutional layers used as shortcuts and linear layers.
- C_3: The base classifiers shared the weights of all convolutional layers, except for both the last convolutional and linear layers.

When using these weight-sharing configurations, a trade-off occurs between inference

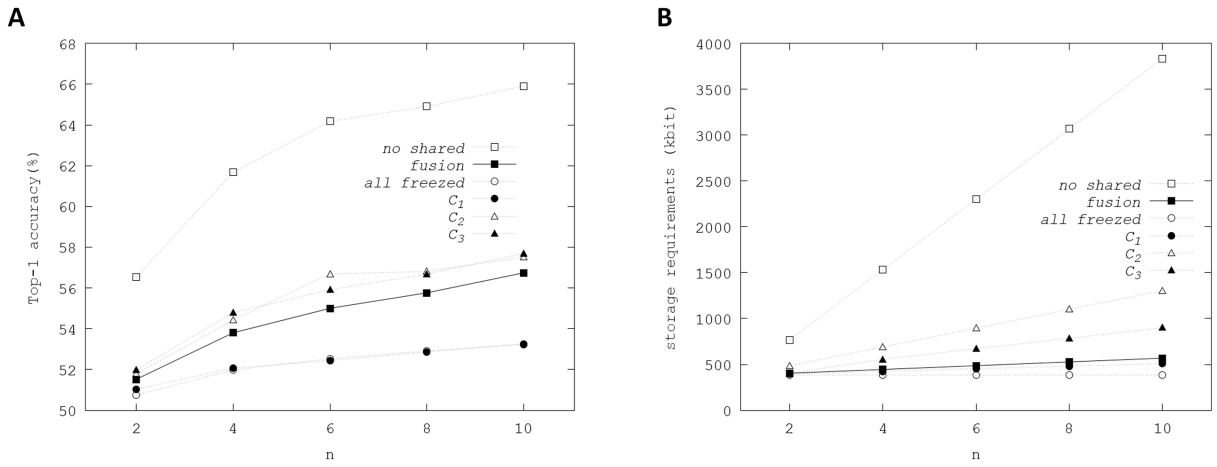


Figure 6.6: Top-1 inference accuracies and storage requirements of different configurations of ensembles using binarized ResNet-20 on CIFAR-100 dataset: (a) Top-1 inference accuracy (b) storage resource requirements.

accuracy and storage requirement resources, as illustrated in Figure 6.6, which is implemented using the fusion ensemble method.

Using the *no shared* configuration with $n = 10$, the model achieved a top-1 accuracy of 65.9%.However, this configuration requires a large amount of storage resources. In *all frozen* configuration with $n = 10$, when the weights of all convolutional and linear layers were shared, the model achieved top-1 accuracy of 53.23%, greater than the base classifier by 4%, and the improved accuracy was proportional to n. This configuration requires to retrain the weights of the batch normalization layers, but the storage resource requirements for this configuration are negligible. The *C1* configuration exhibited the same top-1 inference accuracy as the *all frozen* configuration when $n \geq 2$, but the storage requirement resource slightly increased, which implied that the first layer does not play a significant role in the model inference accuracy. Both the *C1* and *C2* configurations enhanced the top-1 inference accuracy compared to *fusion* configuration; however, as n increased, the storage requirement resources for both of these configurations increased as well. Figure 6.6 illustrates the effects of the weight-sharing configurationa that we introduced on the top-1 inference accuracy along with

the storage resources requirements.

6.4.4 Ensembles with Bi-Real-Net and ReActNet on CIFAR Datasets

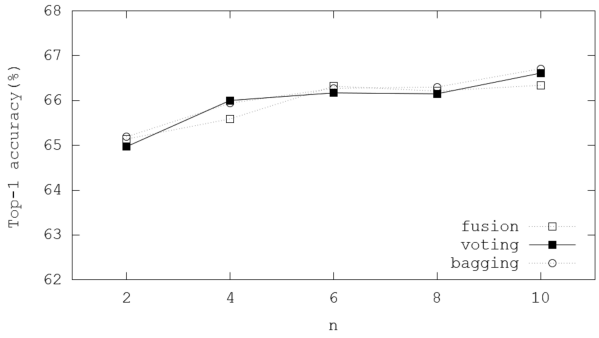


Figure 6.7: Top-1 inference accuracies of ensemble schemes using Bi-Real-Net-18 on CIFAR-100 dataset.

We employed two different BNN methods for further analysis of our ensemble-based system, namely Bi-Real-Net [69], and ReActNet-10 [67]. We binarized ResNet-18 using the Bi-Real-Net method with training on CIFAR-100. All of the layers binarized except the first convolution layer and linear classifier layer, which retained their 32 floating-point representation. Bi-Real-Net uses 32 floating-points of a 1×1 convolutional layer per four binarized convolutional layers as a downsampling. We replaced the ReLU activation function with the PReLU activation function without adding dropout layers to the model. We trained the model for 200 epochs with a batch size of 256, using the Adam optimizer with a learning rate of 0.001, and momentum of 0.9 with a polynomial scheduler to decay the learning weight as described previously. We achieved a top-1 accuracy of 63.97% for the Bi-real-Net base model. Then, we employed three different ensemble methods, namely fusion, soft voting, and bagging. We used the fusion configuration for weight sharing, which froze all of the initial weights except those of the batch normalization and linear layers weights. As a result, these weights were updated during the retraining process. We then retrained the ensemble models

with 200 epochs, using the Adam optimizer with a learning rate of 0.001 and momentum of 0.09, and a polynomial scheduler to decay the learning weight. We used a batch size of 128 instead of 256 when using the fusion method with $n \geq 8$ because experienced the same GPU memory problem explained earlier.

The voting and bagging methods performed better than the fusion method. The top-1 inference accuracy improved by between $2.74\% \sim 1.17\%$. The bagging scheme achieved a top-1 inference accuracy of 66.71% with n = 10. As n increased, the improvement in accuracy became marginal, as demonstrated in Figure 6.7.

Next, we used ReAcNet-10 to evaluate our method with CIFAR-100, which is similar to ResNet-10 in term of layer architecture. ReacNet-10 differs in that it has eight binarized convolutional layers, each of which has a shortcut and is connected with Rsign and RPReLU functions [67]. We used downsampling with a 1×1 binarized convolutional layer for every two binarized convolutional layers.

We trained the model with 400 epochs with a batch size of 256. We used the Adam optimizer with a learning rate of 0.0005 and momentum of 0.9 with a polynomial scheduler to decay the learning weight. The ReActNet-10 model achieved a top-1 inference accuracy of 66.69% on the CIFAR-100 dataset.

Furthermore, we employed three different ensemble configurations, namely fusion, soft voting, and bagging, to evaluate our method. We used the fusion configuration for weight sharing, which froze all of the initial weights except for those of the batch normalization and linear layers. As a result, these weights were updated during the retraining process. Moreover, during the retraining process, the parameters that control the thresholds of RSign layers were also updated. Furthermore, PReELU activation was not frozen, and the parameters updated during the retraining process to control the slope of the negative parts as well as to distribute moving values. We did not share the weights of the downsampling layers among the weak base classifiers.

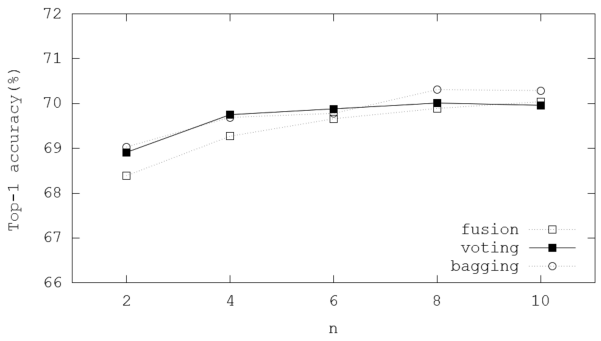


Figure 6.8: Top-1 inference accuracies of ensemble schemes using ReActNet-10 on CIFAR-100 dataset.

Subsequently, we retrained the ensemble models with 200 epochs and a batch size of 256. We used the Adam optimizer with a learning rate of 0.001 and momentum of 0.09 with a polynomial scheduler to decay the learning weight. The begging method outperformed the voting and fusion methods as the number of n increased, as illustrated in Figure 6.8. The bagging method achieved a top-1 inference accuracy of 70.29% with n = 10, which was an enhancement of 3.6% compared with the base model.

Figure 6.8 illustrates the top-1 inference accuracy for the fusion,voting and bagging ensemble methods using ReActNet-10 on the CIFAR-100 dataset. The ensemble methods led to enhancements of the top-1 inference accuracy. In the evaluations, the final top-1 inference accuracies were enhanced by 3.6%1.7%.The evaluation results allowed us to conclude that as the number of n increase the top-1 inference accuracy increased, the top-1 inference accuracy improved until reaching a point where only a marginal improvement occurred, as illustrated in Figure 6.8. However, with this even $n = 2$ exhibited better performance than the base model with negligible costs in terms of hardware.

6.5 Conclusion

We proposed an ensemble-based system that shares the filter weight of convolutional layers. This system reduces storage resource requirements and has a scalable capability. We used different BNN methods to binarize our models, which we ensembled using fusion, voting, and bagging. We evaluated these models and demonstrated the trade-off between storage requirement resources and top-1 inference accuracy for these models. We found fusion method with C2 configuration yield the best result in term of accuracy improvement and storage cost. Finally, we found that across all of the experiments, as the number of n increased, the top-1 accuracy improved until reaching a point where the improvement became marginal.

Chapter 7

A Scalable CNN-Based Inference System Using Multiple Logarithmic Stochastic Rounding

This chapter presents a method for an efficient CNN at the edge using stochastic rounding of logarithmic quantization. To improve the inference accuracy, we used multiple samples that had been quantized using stochastic rounding. The use of logarithmic quantization replace the inefficient multiply–accumulate operations with the more efficient multi-shifting and weight-summation operations. When we used two to eight samples, we achieved an accuracy close to that of floating-point models.

7.1 Introduction

Several models have demonstrated exceptional performance in terms of inference accuracy. These models, however, require a million or even a billion FLOPs, such as that in [54, 97]. Several studies have used quantization to reduce this complexity, and one of the most competitive types in terms of inference accuracy and hardware complexity

is logarithmic representation [74]. When using logarithmic representation, shifting operations replace inefficient multiplication operations; however, deterministic rounding introduces an error that accumulates and magnifies, making it an unacceptable option. Deterministic rounding produces tiny weight changes that may or may not result in a weight update. As a result, using the appropriate rounding method in quantization results in higher inference accuracy.

Using stochastic rounding instead of deterministic rounding benefits the models by allowing neural networks to escape, resulting in a higher inference accuracy. After applying the quantization, stochastic rounding uses a probabilistic approach to approximate the dropped information[96]. However, when compared with deterministic rounding, stochastic rounding introduces additional errors. These errors can be reduced by performing stochastic rounding multiple times and averaging the results.

This chapter describes a method for using logarithmic representation with stochastic rounding multiple times and averaging the results. Doing so eliminated the need for the multiplications required by CNNs, making the model more efficient and easier to deploy at the edge. The model's inference accuracy was improved by performing stochastic rounding multiple times and averaging the results. The experimental results revealed that the inference accuracy was proportional to the number of samples for each input. Using between two and eight samples yielded the same results as the floating-point model.

7.2 Background

Let us assume that fractional number A is represented by $A = (-1)^s \cdot (1 + f_A) \cdot 2^k, k \in \mathbb{Z}$ and $0 \leq f_A < 1$ where s means the sign bit of A. The rounding method is applied to term $1 + f_A$ in the logarithmic number representation, so that A is approximated as

follows:

$$A \approx (-1)^s \cdot \left(2^{k+\log_2 \text{ round } (1+f_A)}\right) \tag{7.1}$$

Rounding methods are divided into two types, namely deterministic and stochastic. Deterministic methods such as rounding up, rounding down, and rounding to the nearest, always produce the same deterministic value for an input value. For example, when $A \approx 2^{-2+\log_2 2} = 2^{-1}$ at any time, it is denoted that A is approximated into A_{approx} .The rounding to the nearest method produces a better result to compensate for lost information after quantization because it minimizes rounding errors as follows: $|A - A_{\text{approx}}|$.On the other hand, in the stochastic method, the logarithmic stochastic rounding converts A to $(-1)^s \cdot 2^{k_A}$ or $(-1)^s \cdot 2^{k_A+1}$ with the probability depending on f_A, so that A is approximated using the logarithmic stochastic rounding as follows:

$$A \approx \begin{cases} (-1)^s \cdot 2^{k_A}, & \text{with probability } 1 - f_A \\ (-1)^s \cdot 2^{k_A+1}, & \text{with probability } f_A \end{cases} \tag{7.2}$$

For example, let us assume that $A = -0.0111_2$ where $k_A = 2$ and $f_A = 0.11_2$. The stochastically rounded value of A is approximated as follows:

$$A \approx \begin{cases} -2^{-2}, & \text{with probability } 1 - 0.11_2 = 0.25_{10} \\ -2^{-1}, & \text{with probability } 0.11_2 = 0.75_{10}. \end{cases} \tag{7.3}$$

7.3 Proposed Design

Quantization refers to the process of converting values from a floating-point fp32 format to a lower precision format by removing several bits. We usually use a rounding method to compensate for the information lost due to the removal of these bits. In

the literature, rounding to the nearest integer is the most commonly used method for deep neural network models. However, while rounding to the nearest reduces the absolute error between the real and quantized values, it also introduces an error that accumulates and magnifies during the multiply–accumulate operations required by the CNN's inference stage. On the other hand, stochastic rounding does not minimize the error of the quantized values. Still, with multiple stochastically rounded values, the values achieve comparable values to the floating-point counterpart. Stochastic rounding entails the following trade-off: precise values require more rounded and averaged values, whereas faster computation times require fewer rounded values. As a result, our method is dynamic, with the selection based on the resources of the edge devices. Accumulating and averaging operations can add a significant amount of hardware complexity, which is unacceptable for edge devices. As a result, we implemented stochastic rounding using a linear feedback shift register (LFSR) hardware block. This hardware block generates pseudo-random numbers compared with each real-valued number to produce stochastically rounded values.

On the inference stage of CNNs models, we can use quantization methods on input, neuron, and activation weights. Several studies have investigated the effects of logarithmic quantization on neuron weights, inputs, and activation weights[60, 74, 61]. Because the number of activations is greater than the number of reused weights during convolution operations, quantizing activation weights is more robust than quantizing neuron weights[74]. In pretrained state-of-the-art models, the weight range of the input is larger than neuron scaled weights $\in$ [1,1][43]. As a result, the errors when applying quantization to input weights are small, and during the inference stage, neuron weights are not as critical. In this study, we used logarithmic quantization with stochastic rounding on the weights of the inputs based on this observation.

7.3.1 Proposed Logarithmic Stochastic Rounding

In our design, we use logarithmic quantization on the inputs, which means that the weight of the inputs is uses logarithmic arithmetic rather than floating-point arithmetic. The logarithmic representation for an input x_i in the range of $2^{\kappa x_i} \leq |x_i| < 2^{\kappa x_i+1}$ for some integer k_{x_i} is $(-1)^s \cdot 2^{k_{x_i}}$ or $(-1)^s \cdot 2^{k_{x_i}+1}$. Using the logarithmic representation, we replace the the multiplication of neuron weights w_j and neuron inputs x_i with shift operation(denoted as $\ll$) as follows:

$$w_j \cdot x_i = \begin{cases} (-1)^s \cdot w_j \ll k_{x_i}, & x_i \approx (-1)^s \cdot 2^{k_{x_i}} \\ (-1)^s \cdot w_j \ll (k_{x_i}+1), & x_i \approx (-1)^s \cdot 2^{k_{x_i}+1} \end{cases} \tag{7.4}$$

Logarithmic quantization with stochastic rounding converts x_i to $(-1)^s \cdot 2^{k_{x_i}}$ or $(-1)^s \cdot 2^{k_{x_i}+1}$ For implementing (7.1), x_i is approximated using the logarithmic stochastic rounding based on (7.4) as follows:

$$x_i \approx \begin{cases} (-1)^S \cdot 2^{k_{x_i}}, & \text{with probability } 1 - f_{x_i}^p \\ (-1)^S \cdot 2^{k_{x_i}+1}, & \text{with probability } f_{x_i}^p \end{cases} \tag{7.5}$$

In (7.5), we use the truncated fraction $f_{x_i}^p$ to decide for the rounded number. When using the logarithmic representation, the IEEE single-precision floating-point format requires 23 bits to store the fraction of each value, which is not efficient. We mitigated this in our design by instead of storing all fractional bits;that is, only p high-order bits in the fraction of activation output are stored, which is denoted as $f_{x_i}^p$. This stored information after truncation is referred to as the probability value.

For instance, if x_i =0.011011_2, $K_{xi} = 2$ and $f_{xi} = 0.1011_2$, where $f_{x_i}^2$= 0.10_2 is the probability value to decide for the number using the stochastic rounding.

7.3.2 Proposed Design

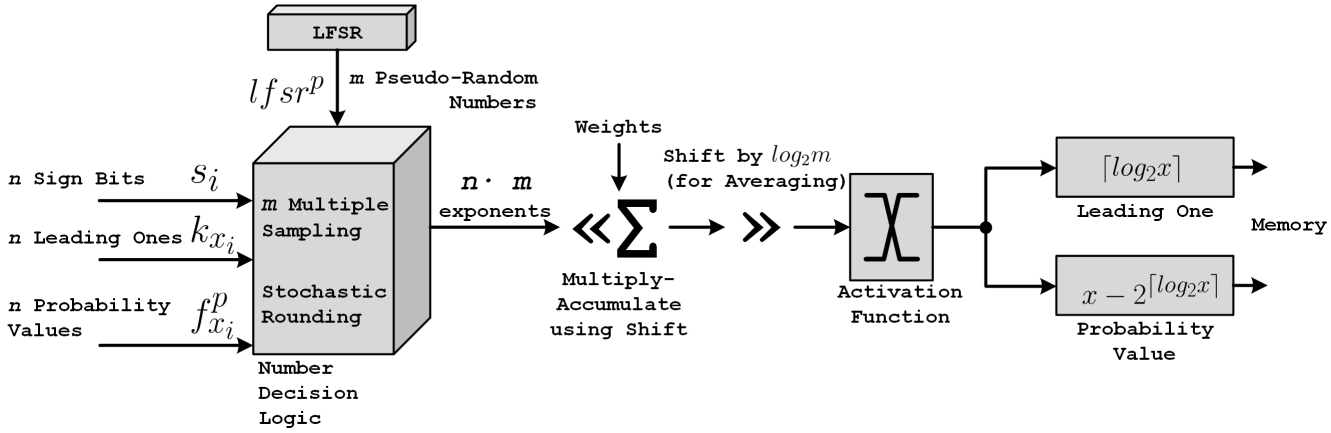


Figure 7.1: The proposed design for CNNs using logarithmic representation with stochastic rounding.

In our design we use multiple sampling based on the logarithmic representation with stochastic rounding, as illustrated in Figure 7.1. We obtain n leading ones $k_{xi}, 0i < k$ and probability values $f^p_{x_i}, 0i < k$ from memory. The stochastic rounding makes the decision of the rounded number $2^k_{x_i}$ or $2^{k+1}_{x_i}$ in number decision logic (NDL) block based on the obtained information from memory. We use LFSR blocks to generates pseudo-random numbers in order to decide for the stochastic rounding number. We also share output of LFSR block when multiple NDL blocks are performed in parallel to reduce the hardware complexity. .In our design, we compare the generated low-order p bits of the LFSR $lfsr^p$ with the probability value $f^p_{x_i}$ in a comparator. To decide for the tie-breaking of $lfsr^p = f^p$ randomly, the $(p+1)$-th bit we use LFSR, which is denoted as tiebit. The number decision of x is formulated as follows:

$$x_i \approx \begin{cases} (-1)^s \cdot 2^k_{x_i}, & \text{if } lfsr^p > f^p \\ (-1)^s \cdot 2^k_{x_i}, & \text{if } lfsr^p = f^p \text{ and tiebit } = 1 \\ (-1)^s \cdot 2^{k+1}_{x_i}, & \text{if } lfsr^p = f^p \text{ and tiebit } = 0 \\ (-1)^s \cdot 2^{k+1}_{x_i}, & \text{if } lfsr^p < f^p. \end{cases} \tag{7.6}$$

Then, weights w_j are converted into the fixed-point values for the multiplication with the output of the NDL block. We replace the multiplication operation of CNNs by shifting the weights to the left by k or $k+1$. The NDL block generates m multiple samples using stochastic rounding, and the number decision in NDL block is repeated m times. Using the shift operation for multiply-accumulate using m multiple samples, we then average the accumulated values. To average the values, we restrict our design for m to be a power of 2 (e.g., 2, 4, 8, 16, and so on), to approximate the division using a shift operation by $\log_2 m$, instead of a divider which would add more complexity to hardware;this would be unacceptable for edge devices. Formally, for n-element weight vector $\mathbf{w} \in R^n$ and n-element input vector x, the multiply accumulate operation that yield $\mathbf{y} = \mathbf{w}^T \cdot \mathbf{x}$ as presents in Figure 7.1. Using our design, the multiply-accumulate operation is defined as follows:

$$\mathbf{w}^T \cdot \mathbf{x} = \sum_{i=1}^{n} \sum^{m} (-1)^{s_i} \cdot w_i \ll \left(k + \log_2 \left(\text{round}\left(1 + f_{x_i}^{p}\right)\right)\right) + b_i \tag{7.7}$$

Then, as depicted in Figure 7.1, the averaged values pass the activation function. The values of the activation function both the leading one position K_{y_i} value, and the probability value $f_{x_i}^{p}$ are temporarily stored in the memory for the next layer.

When using logarithmic quantization with stochastic rounding and m=1, our design reduces the hardware complexity, but the accuracy is degraded significantly. Therefore, our design with using $m \geq 2$, increase the hardware complexity but we avoid that by sharing LFSR block among multiple NDL blocks. Furthermore, using shifting for the averaging instead of using a normal divider adds a negligible complexity to the hardware. Therefore, our design is optimized to have a trade-off between the inference accuracy and both hardware complexity and power consumption. In our design, the number of m is proportional to the inference accuracy; that is, as we increase m, the inference accuracy increases, which makes this design an appropriate option especially

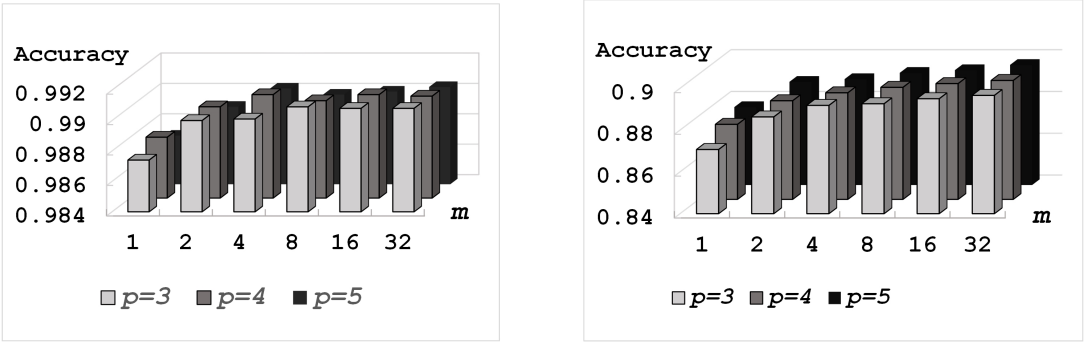
for devices at the edge.

7.4 Experimental Results and Analysis

We programmed and designed the NDL block and the shifter as combinational blocks in the Verilog hardware description language (HDL), and then evaluated these blocks in terms of area and power. We generated the random number with the help of an external pseudo-random sequence generator. To synthesize the codes, we used Synopsys Design Compiler and a 28/32nm standard cell library from Synopsys[3], with a target frequency of 250MHz in Ultra mode. In our design, the area and the power for the Radix-4 Booth multiplier are $1,682um^2$, $93.0uW$ respectively which are 83.4% and 92.8% reduced compared with the floating-points representation.

We designed the deep neural network models using the Caffe framework, and we modified the matrix multiplication module in the Caffe to implement our design for stochastic rounding. We then studied the effect of samples m on the inference accuracy for LeNet-5 with MNIST, NIN with CIFAR-10, AlexNet, and GoogLeNet with ImagNet.

We investigated the effects of the number of samples m and the size of the probability value p on the top-1 inference accuracy for the LeNet-5 and NiN models, as illustrated in Figure 7.2. As the figure indicates, the inference accuracy of the LeNet-5 model did not reflect the true effects of m and p on the inference accuracy because of the simplicity of the model. However, with NiN models (Figure 7.2.b), the top-1 inference accuracy was proportional to the size of m, and probability values p. The top-1 inference accuracy improved as the number of samples increased and the precision of the probability increased. We found that $p = 4$ was sufficient for the precision of our probability values, which we use for all of the models in our experiments. Furthermore, we found that as m increased, the top-1 inference accuracy increased until it reached a size of m, at which point the accuracy improvement became negligible.



(a) (b)

Figure 7.2: Layer's connection for a convolution: (a) LeNet-5 (b) NIN.

The top-1 inference accuracy improved as the number of samples increased and the precision of the probability increased. We discovered that $p = 4$ is sufficient for the precision of our probability values, which we use for all of the models in our experiment. Also, we found that as m increased the accuracy increased as well until reach to a size of m where the accuracy improvement is negligible. For example, when m became 2 instead of 1, the model increased by 1% to 2%, but when m increased from 16 to 32, the improvement was less than 0.2 %. As a result, we restricted our experiments to be $2sim8$ samples which led to a significant improvement.

We applied our method with p=4 and investigated how to quantize only the input (denoted as inlog), quantize only the weight (denoted as wlog), and quantize both the input and weights (denoted as inwlog) to affect the inference accuracy. For the inputs, we also use floating-point (denoted as float) and rounding to the nearest integer (denoted as rd), as presented in Figures 7.3 and 7.4. Table 7.1 indicates that the top-1 inference accuracy increased as m increased, with m=4 achieving the same accuracy as the floating-point counterpart.

Figure 7.3 depicts how the inference improved as the number of m increased as well as compares the different methods, demonstrating that using inlog quantization yielded the same accuracy as floating with only m = 4

Table 7.2 demonstrates that the top-1 inference accuracy increased as m increased,

Table 7.1: Comparison of Different Methods and Different on MNSIT

Method	*m*	*Top-1%*
LeNet-5		
Float	-	99.06
rd	-	98.95
lsr	1	98.80
lsr	2	99.00
lsr	4	99.08
lsr	8	99.04
lsr	16	99.08
lsr	32	99.07

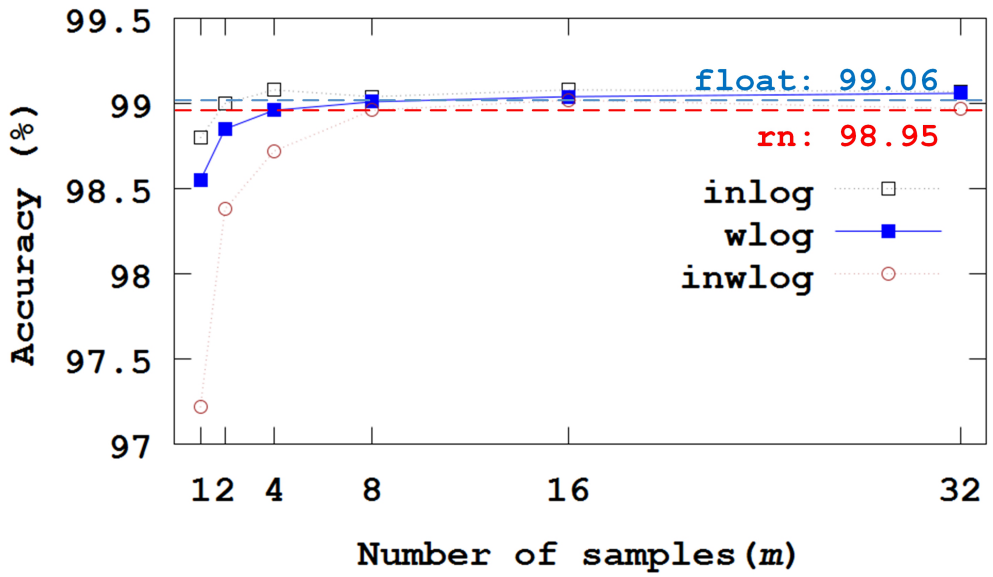


Figure 7.3: Inference accuracy using logarithmic quantization for LeNet-5 with MNSIT

with m=8 also achieving the same accuracy as the floating-point counterpart.

Figure 7.4 depicts how the inference improved as the number of m increased. It also compares the different methods, demonstrating that using inlog quantization yielded the same accuracy as floating with only m = 8.

Table 7.2: Comparison of Different Methods and Different on CIFAR-10.

Method	*m*	*Top-1%*
NIN		
Float	-	89.57
rd	-	88.03
lsr	1	87.59
lsr	2	88.72
lsr	4	89.10
lsr	8	89.36
lsr	16	89.54
lsr	32	89.69

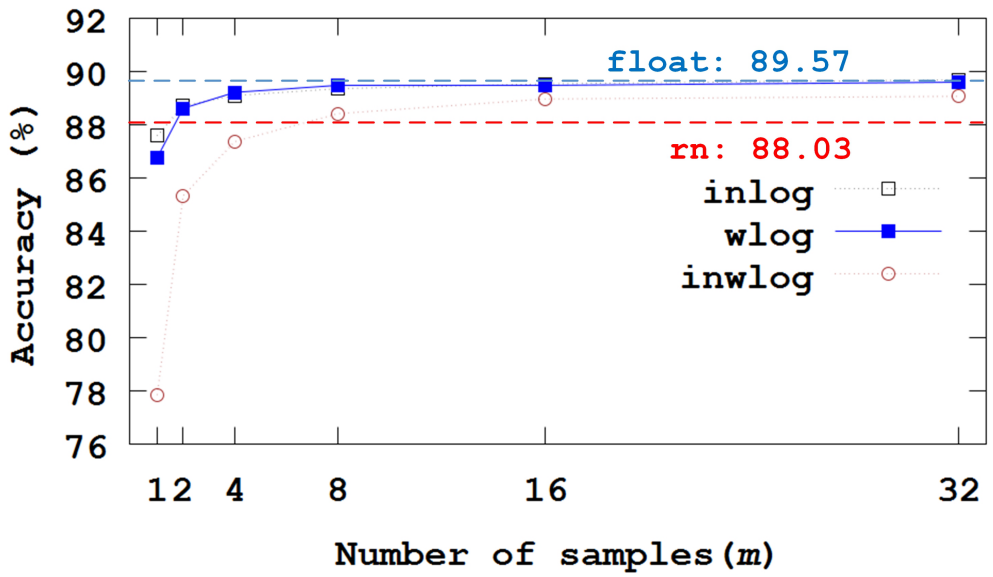


Figure 7.4: Inference accuracy using logarithmic quantization for NiN with CIFAR-10

For AlexNet and GoogLeNet from Table 7.3, we achieved the same floating-point accuracy when m = 8.

For the GoogLeNet model, rounding to the nearest greatly reduced the inference accuracy, which was only 49.19%, due to the fact that GoogLeNet comprises 22 layers and as errors pass they become magnified. As a result, Figure 7.5 compares only the stochastic rounding method with different m.

Table 7.3: Comparison of Different Architectures on ImageNet

Method	*M*	*Top1%*	*Top5%*
AlexNet			
Float	-	56.82%	79.95%
rn	-	47.07%	71.42%
lsr	1	45.61%	69.93%
lsr	2	51.57%	75.69%
lsr	4	54.55%	78.04%
lsr	8	55.72%	79.18%
lsr	16	56.29%	79.53%
lsr	32	56.48%	79.72%
GoogleNet			
Float	-	-%	89.1%
rn	-	-%	49.19%
lsr	1	-	71.81%
lsr	2	-	83.82%
lsr	4	-	87.4%
lsr	8	-	89.14%
lsr	16	-	89.21%
lsr	32	-	89.21%

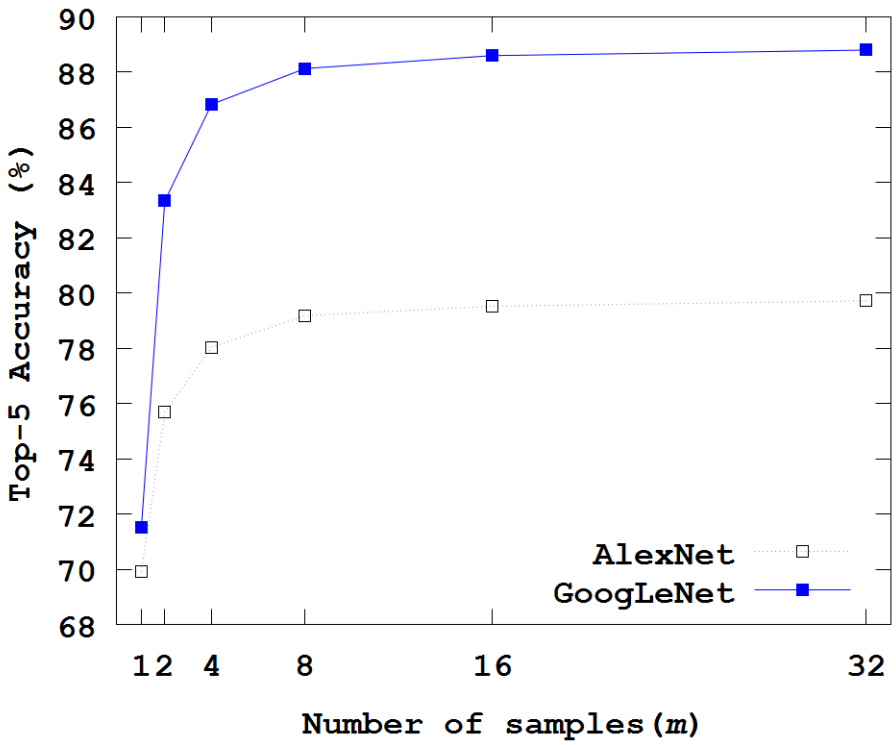


Figure 7.5: Inference accuracy using logarithmic quantization for both AlexNet and GoogLeNet with ImageNet.

7.5 Conclusion

In this chapter, we have presented a method for improving logarithmic quantization by sampling multi-samples with stochastic rounding. We reduced the hardware complexity and power consumption of our method by employing two distinct logic blocks, namely NDL and LSFR. Furthermore, our method is a trade-off method, and as the number of m increased, so did the model's inference accuracy. According to our experiments, two to eight samples are sufficient for achieving the same accuracy as floating-point models. Furthermore, we discovered that when the model was deep and had many layers, the deterministic rounding method significantly reduced the inference accuracy because errors increased and magnified when passing through the layers. Stochastic rounding, on the other hand, was not affected by this problem and is a viable option when the model has many layers.

Chapter 8

High Rank Tensor Train For Binary Neural Network

This chapter proposes a method for improving BNN accuracy. BNNs use binary weights and activation, allowing them to be deployed on low-power devices. In terms of computation and storage costs, BNNs offer numerous advantages. However, the accuracy of the models suffers a significant loss in performance. In this chapter, we improve the accuracy of BNN models based on latent variables obtained through high-rank TT decomposition. Through this work, we make the following contributions: (1) We present a method that improves the accuracy of state-of-the-art BNN models by 2%–4% while only increasing a small number of model parameters; and (2) we apply our method on two different datasets, namely CIFAR-10 and CIFAR-100, using Resnet-20.

8.1 Tensor Train Ranks

In general, finding a tensor rank is an NP-hard problem. The goal of using tensor decomposition for neural network models is to compress them and reduce the number of parameters. This is accomplished by finding the tensors' lower rank approximation.

Numerous approximation algorithms exist for finding low-rank tensors. Algorithms such as VBMF [77] [11] as well as reinforcement learning can be used to select the rank [13].

In this study, we selected three different fixed ranks and investigated their effects on binary deep neural network models. To add more parameters to the decomposed tensors, we chose a fixed large number for the rank. A large number of model parameters can help the model to generalize better, resulting in an increase in model accuracy of 2% to 3%.

8.2 Xnor-Net

XNOR-net is type of BNN where the weights and input of the convolutional layers are 1-bit. The weight values in XNOR-Net are approximated using binary filters, as presented below; by treating quantization as an optimization problem, as in the equation, a better scale factor can be selected:

$$I * W \approx (I \oplus \beta)\alpha \tag{8.1}$$

$$J(\beta, \alpha) = \|W - \alpha\beta\|^2 \tag{8.2}$$

Here, W denotes real value filters, B denotes binary filters, and alpha denotes a positive scaling factor. The binary weight filter is the sign of the weight values after solving this optimization problem, and the scaling factor is the average of the absolute weight

values.

$$\beta^* = sign(W),\ \alpha^* = \frac{1}{n} \|W\|_{l\,1} \tag{8.3}$$

This work is conducted using XNOR-net methods, which has the advantage of saving up to 32x memory and increasing CPU speed by 58x. Furthermore, the computational heavy matrix multiplication operations are transformed into bitwise XNOR operations and bit-count operations [87]..

Real values are quantized during forward propagation using the equations in deterministic binarization (5). However, the error cannot propagate during the back-prorogation because the gradient is zero almost everywhere. To mitigate this, the STE is used, which is a heuristic method for estimating the gradient of neuron, as shown in equation (8.5), where (x) is the value before binarization[7].

$$w_b = \begin{cases} +1 & if\ w \geq 0, \\ -1 & else. \end{cases} \tag{8.4}$$

$$STE(x) = \begin{cases} 0 & if\ x < -1, \\ 1 & if\ -1 \geq x \leq 1, \\ 0 & if\ x > 1. \end{cases} \tag{8.5}$$

8.3 Our Proposed Method

Typically, deep neural network models for image classification have the following five stages: the input layer; the second-stage layer, which extracts low information such as edges; the third-stage layers, which extract intermediate information such as patterns;

the fourth-stage layers, which extract high information such as objects; and finally, the features flatten for a classifier layer.

In CNN, the convolutional operation maps the input tensor $\mathcal{X}$ of size H x W x S to the output tensor † of size S x W' X H' using a tensor kernel of size $D \times D \times S \times T$ in which T, and S are the output and the input respectively and D is the spatial dimension.

$$\mathcal{Y}_{h',w',t} = \sum_{i=1}^{D} \sum_{j=1}^{D} \sum_{s=1}^{S} \mathcal{K}_{i,j,s,t} \mathcal{X}_{h_i,w_j,s} \tag{8.6}$$

Applying TT decomposition with rank R, the convolutional layers are formulated as matrix-by-matrix multiplication in which the four-way tensor reshapes into a matrix K of size $D^2S \times T$;furthermore, TT-Format is applied in which $\boldsymbol{G}$ is TT-cores as discussed in [23]. We would obtain the following decomposition of the convolutional kernel.

$$\mathcal{Y}_{t,w',h'} = \sum_{j=1}^{D} \sum_{i=1}^{D} \sum_{s_1,\ldots,s_d} \mathcal{X}_{s,w_j,h_i} \boldsymbol{G}_0[i,j] \boldsymbol{G}_1 [t_1, s_1] \ldots \boldsymbol{G}_d [t_d, s_d] . \tag{8.7}$$

Models that are over-parameterized generalize better. Typically, as the number of parameters in the models increases, so does the model's accuracy, as evidenced by numerous empirical studies[54][34][97]. These empirical results hold true for various types of deep neural network models, including plain neural network models such as AlexNet and VGGNet, as well as networks that use residual blocks such as ResNet and its variants. Most of of these parameters are not used for classification but rather for the optimization algorithm to converge for a better local minima[83].

Based on these empirical findings, we increased the number of parameters in the model by selecting a high rank, and then decomposed the convolutional layers using the TT algorithm, resulting in a chain of convolutional layers whose sum of parameters was greater than the original convolutional layer. Subsequently, we applied the XNOR-Net

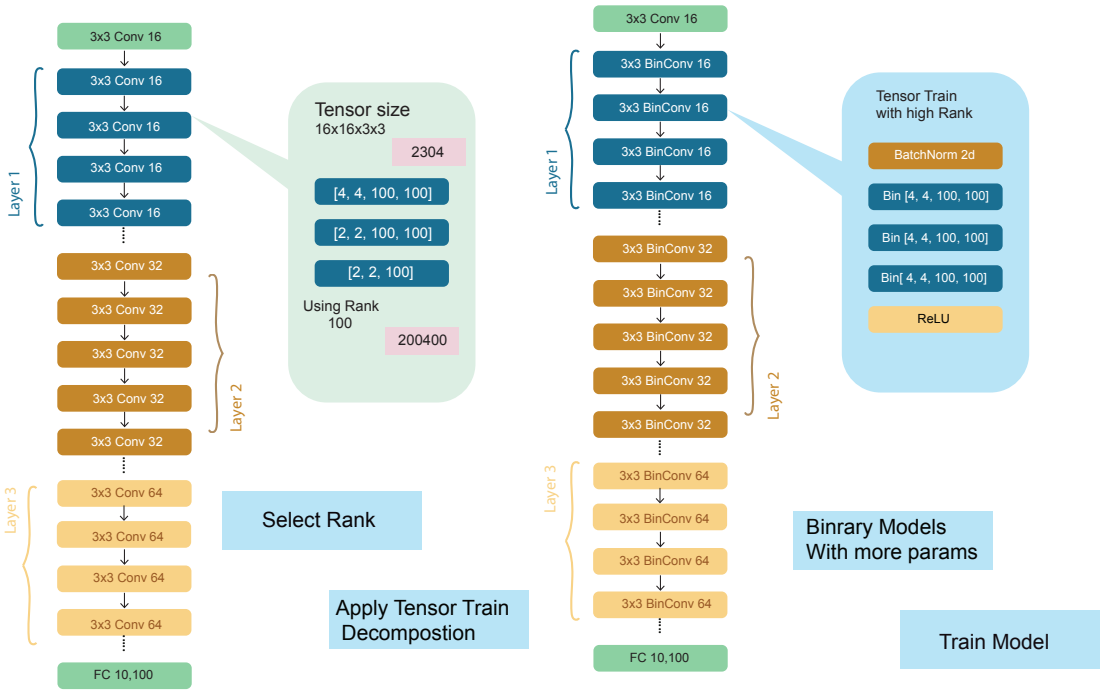


Figure 8.1: The proposed method, which includes decomposing the models with high rank tensor train decomposition, binarizing the decomposed layers, and finally training the models.

method to the decomposed layers before training the models. Figure 8.1 depicts a layer with 16 input channels and 16 output channels with a filter size of 3×3. This layer had a total of 2304 parameters. The number of parameters increased by $86.9\times$ when we used our method, allowing the model to generalize better and improving its accuracy.

Our method is similar to others that improve BNNs by adding more parameters, such as EBNN [118] and Bi-Real-Net [69]. However, we use a more efficient approach for adding binary parameters because it retains the advantage of using the XNOR-Net method, which saves memory by $32\times$ and CPU computation by $58\times$, which EBNN and Bi-Real-Net do not.

8.4 Results and Experiments

Table 8.1: Comparison of Floating and Binary Models with High Rank on CIFAR-10

Network Type	*Rank*	*Params Size MB*	*Top 1%*
Floating-poing			
Base	-	1.1	92.36%
TT-Model-1	50	4.9	91%
TT-Model-2	100	19.1	92.1700%
TT-Model-3	200	76.0	92.360%
XNOR-Net			
Base	-	0.047	79.133%
XNOR-Net-TT-1	50	0.209	80.73%
XNOR-Net-TT-2	100	0.820	82.12%
XNOR-Net-TT-3	200	2.8	84.259%

To build, train, and test deep neural network models for our experiment, we used the PyTorch framework. To simulate a BNN, we used the detach function from the PyTorch library. Using the detach function implies that the graph of the model will only have floating-point parameters during backpropagation but will retain binary parameters during forward propagation. We used the TedNet library to apply decomposition, which is an open source library designed to decompose neural network layers [80].

CIFAR-10 and CIFAR-100, both of which are widely used datasets with 50,000 RGB images of 32×32 pixels for training and 10,000 for testing, with 10 classes for CIFAR-10 and 100 classes for CIFAR-100. In terms of data augmentation, we used a random crop of 32 with padding of 4 as well as random horizontal flip. We transformed the images into tensors and normalized using PyTorch parameters for mean and standard deviation.

ResNet-20 is a ResNet variant that consists of 19 convolutional layers and one fully connected layer with residual connections between the layers. We used ResNet-20 on both datasets for the floating-point and binary models, as presented in Table 1.

ResNet-20 is a ResNet variant that consists of 19 convolutional layers and one fully connected layer with residual connections between the layers. ResNet-20 was used on both datasets for the floating-point and binary models, as shown in Table 1.

Table 8.2: Comparison of Floating and Binary Models with High Rank on CIFAR-100

Network Type	*Rank*	*Params Size MB*	*Top 1%*
Floating-poing			
Base	-	1.2	68.730%
TT-Model-1	50	4.9	67.170%
TT-Model-2	100	19.1	68.14%
TT-Model-3	200	76.0	68.73%
XNOR-Net			
Base	-	0.069	50.17%
XNOR-Net-TT-1	50	0.280	51.37%
XNOR-Net-TT-2	100	1.0	52.54%
XNOR-Net-TT-3	200	3.6	55.35%

We applied TT decomposition on all of the layers of ResNet-20 except for the first layers. We used ranks of 50, 100, and 200 for all of the layers, as Table 1 indicates. Then, we used the XNOR-Net method as explained in the previous sections to binarize the layers of the model, except for the first and last layers.

Next, we trained the models on a single GTX 1080 TI GPU for 120 epochs with a batch size of 32 using the Adam optimizer with a learning rate of 0.01 and weight decay of 1e-7; then, we used the ReduceLRonPlateau scheduler with a patience of 25 to reduce the learning rate by a factor of 0.005. In addition, we used norm grad clip to prevent the model from exploding gradients and to accelerate model training

8.5 Discussion and Conclusions

In this chapter, we have presented a method for improving a BNN model by adding more parameters using a high-rank TT decomposition algorithm. We used three dif-

Table 8.3: Comparison of Different Architectures on CIFAR-10

Network Type	*Top1%*	*Params Size MB*
	ResNet-20	
Resnet20-FP[34]	92.60%	1.1
Mobile-net[37]	90.18%	12.4
Mobile-netv2[93]	91.29%	9.0
Efficient-net[102]	91.330%	11.4
ResNet20-Xnor[87]	79.13%	0.047
XNOR-Net-TT-1	80.89%	0.209
XNOR-Net-TT-2	82.12%	0.80
XNOR-Net-TT-3	84.529%	2.8

ferent ranks of 50, 100, and 200 to decompose all of the layers except the first, and then binarized all of the layers of the models except the first and last. We improved the model's accuracy by 2%–4% while retaining the benefits of using the XNOR-Net method, which saved memory storage by 32× and computational costs by 58×.

Chapter 9

Crowd Counting Application

9.1 Background

One of the most critical applications in smart cities is crowd counting. Building a comprehensive computational model capable of analyzing and monitoring high-density crowds is a difficult task. Crowd analysis and flow monitoring are critical in high-risk environments, such as stadiums, spiritual gatherings, and music concerts, for avoiding crushing and blockage. Furthermore, analyzing and understanding crowd density and movement enables the development of better security services as well as better logistics and infrastructure for monitoring and easing crowd flow. Crowd counting is a difficult task, with difficulties arising from a variety of sources, including background noise, the nonuniform distribution of people, blurred images, and distorted and affected images [95]. To address these problems, the deep learning literature has introduced numerous deep neural network models and datasets of various types. Crowd images are frequently derived from the video feed of a surveillance camera, and the majority of analyses are performed in the cloud rather than on the surveillance camera itself. We used crowd counting as a case study for our Ultimate Compression method introduced in Chapter 5. We specifically used two different models, namely MCNN and CSRNet, with the

following four datasets: ShanghaiTech B, UCF_CC_50, WorldEXPO'10, and UCF-QNRF datasets[63]. We compared the Ultimate Compression method with floating-point with MCNN and CSRNet in terms of mean absolute error (MAE), root mean square error (RMSE), and storage costs.

9.2 Models and Datasets

9.2.1 Models

MCNN

MCNN consists of three columns of CNNs in which their filters have different sizes. Except for the sizes and numbers of filters, the columns have the same network structure (conv–pooling–Activation–conv–pooling); and the pooling is a max-pooling of size two; and the activation function is ReLU. The input to the MCNN model is an image, and the output is a density map whose integral provides the crowd counting. Differently sized filters result in different receptive fields of the same images, making the learned features by each column more robust to variations in people and head size due to the perspective effect across different images. To map the feature maps to the density map, MCNN replaces the fully connected layer with a convolutional layer of size 1x1[114].

CSRNet

Compared with MCNN, CSRNet significantly reduces the training time while producing high-quality density maps. To support input images with variable resolutions, CSRNet employs convolutional layers with a filter of size (3x3) for all layers as the backbone. The front-end of CSRNet is comprises of the first 10 layers of VGG-16. The back-end, uses dilated convolution layers to enlarge receptive fields and extract deeper

features without sacrificing resolution. CSRNet does not use the pooling layer, but it does use the ReLU activation function[63].

9.2.2 Dataset

ShanghaiTech B

The ShanghaiTech Part B dataset contains 716 images with sparse crowd scenes taken from Shanghai streets, examples of which are provided in Figure 9.1.

Figure 9.1: Samples from the ShanghaiTech Part B dataset [114].

UCF_CC_50

The UCF_CC_50 dataset consists of 50 images with varying perspectives and resolutions[39]. The number of annotated people per image ranges from 94 to 4543, with an average of 1280.

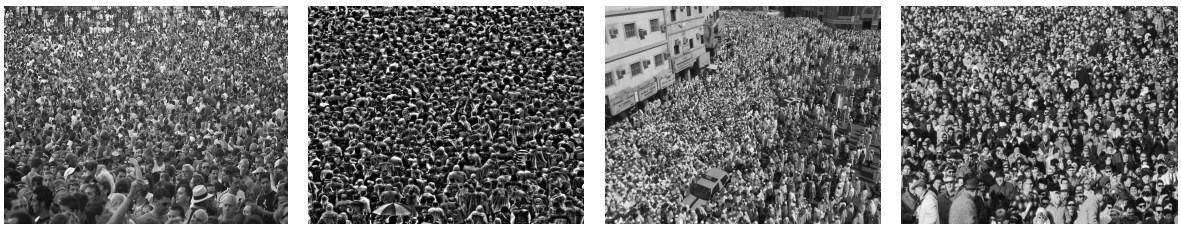


Figure 9.2: Samples from UCF_CC_50 datasets [39].

UCF-QNRF

Compared with other datasets, the UCF-QNRF dataset contains 1535 images of the highest resolution. Furthermore, it contains a broader range of scenes with the most varied viewpoints, densities, and lighting variations. The maximum number of annotated people per image is 12,865, with an average of 815.

[39]

Figure 9.3: Samples from the datasets[39]

WorldEXPO'10

The WorldExpo'10 dataset contains 3980 annotated frames from 1132 video sequences recorded by 108 different surveillance cameras. This dataset is split into two parts: a training set (3380 frames) and a testing set (600 frames) from five different scenes.

Figure 9.4: Samples from the WorldEXPO'10 dataset[113].

9.3 Experiment and Results

To train the models, we used the Euclidean distance as the loss function, measuring the difference between the estimated density map and the ground truth as follows:

$$L(\Theta) = \frac{1}{2N} \sum_{i=1}^{N} \|F\left(X_i; \Theta\right) - F_i\|_2^2 \tag{9.1}$$

where is N is the number of training images; and Θ is the learnable parameter in the models; X_i is the input images; and F_i is the ground truth density map of image X_i; $F\left(X_i; \Theta\right)$ is the estimated density map generated by the models; and L is the loss function between the estimated density map and the ground truth density map[114].

We applied our Ultimate Compression method using these both of these models on the four different datasets which are the most common in the literature for crowd counting applications. We used the same data augmentation and training routines that presented in[22] to train the floating-point models. Then, we applied Ultimate Compression by applying binarization on the decomposed layers and training the models using the same routine that we used for the floating-point models. We used different metrics to test the model performance: MAE, RMSE, and storage cost, as presented in Table 9.1.

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |\hat{y}_i - y_i| \tag{9.2}$$

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{\left(\hat{y}_i - y_i\right)^2}{n}} \tag{9.3}$$

	UCF-QNRF		WE		ShanghaiTech B		UCF-CC-50		Floating	
	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE	Params	Compress.
MCNN	365.2	577.2	18.8	0.0	40.3	60.0	566.4	715.3	545KB	–
CSRNet	111.4	199.4	14.3	0.0	9.8	14.6	155.2	254.4	65MB	–
									Ous	
	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE	Params	Compress.
MCNN	416	621.5	18.6	0.0	50.4	70.2	377.6	509.1	152KB	3.58x
CSRNet	121.2	198.3	15.6	0.0	16.25	25.03	233.54	266.17	2.74MB	23x

Table 9.1: Comparison Between the Floating-Point Models of MCNN and CSRNet and the Decomposed Binary Models

We used the original image sizes in MCNN because resizing the images introduces additional distortion in the density map, which is difficult to estimate. Then, we applied our Ultimate Compression method on the second and third layers only in each column, and the first, third, and conv 1×1 layer is floating-point, as illustrated in Figure 9.5.

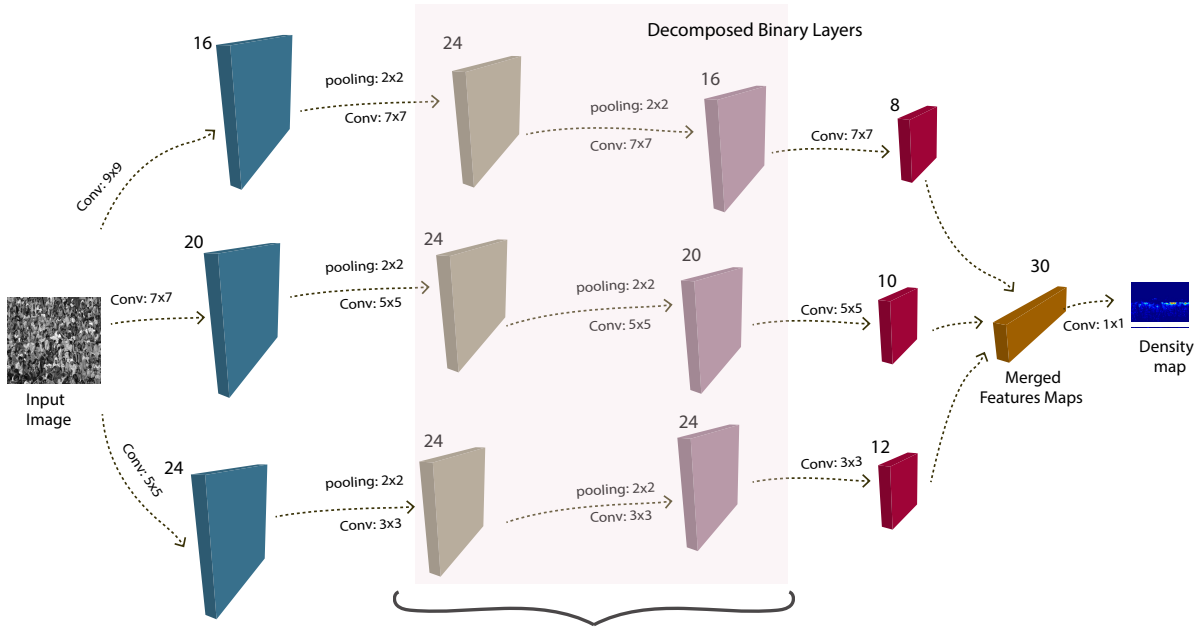


Figure 9.5: MCNN model: we first binarized the second and third layers of each column, and then decomposed them using tensor train decomposition.

For CSRNet, we used Ultimate Compression for both back-end layers, which use 10 layers from VGG-16, and front-end layers, which consist of six layers with a dilation of 2. Here, we applied it to all layers except the first and last layers in the back-end

and front-end, as depicted in Figure 9.6. Thus, we compressed the model by 23×.

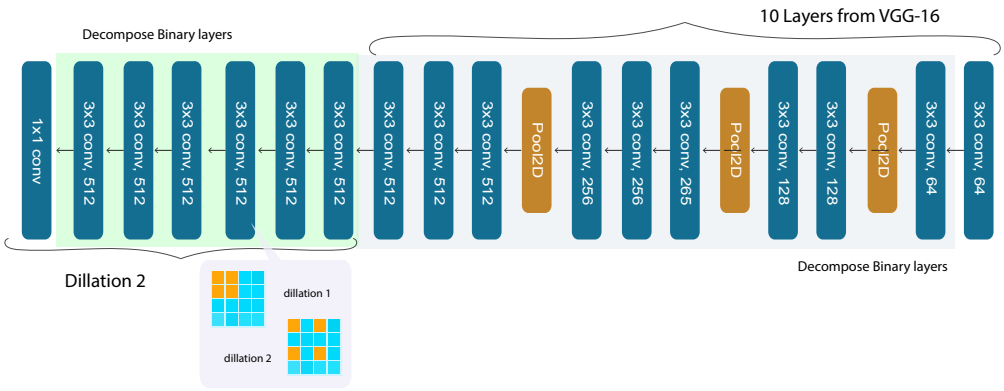


Figure 9.6: CSRNet: we binarized and decomposed the 10 layer of the VGG-16 back-end and five layers in the front-end with a dilation of 2.

9.4 Conclusion

In this chapter, we have presented how we studied and applied our Ultimate Compression method on a crowd counting application using MCNN and CRNet. The MCNN model captured different receptive fields using multiple column convolutional layers with different kernels and fused them together to generate a density map. The CSRNet model used CNNs as the front-end and dilated CNNs for the back-end. The dilated CNNs retained spatial information. We compressed the model by 3.5× and 23× for MCNN and CRNet, respectively.

Chapter 10

A Two-Stage Efficient 3D CNN Framework for EEG-Based Emotion Recognition

This chapter proposes a novel two-stage framework for emotion recognition using EEG data that outperforms state-of-the-art models while keeping the model size small and computationally efficient. The framework consists of two stages; the first stage involves constructing efficient models named EEGNet, which is inspired by the state-of-the-art efficient architecture and employs inverted-residual blocks that contain depth-wise separable convolutional layers. The EEGNet models on both valence and arousal labels achieve average classification accuracies of 90%, 96.6%, and 99.5% with only 6.4k, 14k, and 25k parameters, respectively. In terms of accuracy and storage cost, these models outperform the previous state-of-the-art results by up to 9%. In the second stage, we binarize these models to further compress them and deploy them easily on edge devices. BNNs typically degrade model accuracy. We improve the EEGNet binarized models by introducing three novel methods, achieving a 20% improvement over the baseline binary models. The proposed binarized EEGNet models achieve accuracies

of 81%, 95%, and 99% with storage costs of 0.11 Mbits, 0.28 Mbits, and 0.46 Mbits, respectively. These models will help to deploy a precise human emotion recognition system in the edge environment.

10.1 Introduction

Deep neural networks have achieved incredible results in computer vision, speech recognition, and natural language processing [53]. Deep neural network models have also performed exceptionally well in the fields of brain–computer interaction (BCI) and human–computer interaction (HCI). Electroencephalography (EEG) signals are used in a variety of BCI applications, such as control prosthetics, neurofeedback, and emotion recognition [5]. In real-time, a BCI system records EEG signals in a noninvasive manner and produces a message or computational command from the recorded signals. A BCI system comprises three different components: sensors (mostly electrodes mounted on the scalp to record EEG signals); translation and communication (mostly translating EEG signals into commands or computational language); and real-time actions (actions based on EEG signals). CNNs have produced promising results in computer vision applications, such as image recognition, object detection, and semantic segmentation. Various types of CNNs exist, but in terms of dimensionality, 1D CNNs are the most commonly used for time-series applications such as human activity identification [99] and physiological signals, as demonstrated in [49].Two-dimensional CNNs are the most commonly used for image data and computer vision applications, such as image classification and segmentation. Three-dimensional CNNs, which are mostly useful for volumetric data, have been adopted successfully in video analysis and object recognition tasks [42, 45]. Furthermore, 3D CNNs take 3D inputs and apply 3D filters to them. The filters move along three axes to form 3D shape outputs. Long sequence data such as video, audio, electrocardiogram (ECG), and EEG signals can benefit from

the extra convolutional dimension due to the spatiotemporal correlations between data segments in the long sequence.

Emotions play a critical role in human reasoning and are linked to rational decision making, perception, human interaction, and even human intelligence itself [93]. Various methods exist for modeling human emotions, and one of the most effective approaches is to use multiple dimensions or scales for emotion categorization. In such a model, emotions are defined by two major perception dimensions: valence and arousal. Arousal ranges from low to high, whereas valence ranges from positive to negative. For example, fear has a negative valence and high arousal, whereas excitement has a positive valence and high arousal.

EEG signals are brain waves that measure eclectic field behaviors from the human scalp. They can be naturally applied to human emotion recognition due to their reflection of human response and linkage to the cortical activities [91]. Several studies have investigated EEG-based emotion recognition by extracting EEG features using deep neural network algorithms. These methods have demonstrated high recognition accuracies compared with classical machine learning models but still require feature extraction and selection prior to the classifier [117] [6]. Using CNNs for emotion recognition is not novel. In [85] and [75], a 2D CNN was used for the emotion recognition of power spectrum density (PSD) features from original EEG signals as input to the neural network and demonstrated good results. However, CNNs typically require a large amount of computational resources and the computing power spectrum density is not efficient. As a result, deploying such models on low-resource-constrained devices is not possible. In this chapter, we propose a novel two-stage 3D CNN framework for emotion recognition without special signal transformations on EEG data. Our method significantly outperformed the state-of-the-art results while keeping the model size small and computationally efficient, thus allowing model deployment in edge environments where devices have limited resources.

10.2 Related Works

A variety of approaches, including traditional machine learning and deep learning algorithms, have been introduced to identify and classify emotions. Traditional machine learning algorithms require feature extraction and selection before the classifier is applied. One study used the support vector machine (SVM) on the publicly available dataset DEAP to identify valence and arousal perceptions using feature vectors based on statistical measurements of the frequency bands in the EEG signal, and achieved 67% by using all features [92]. A deep belief network with glia chain (DBN-GC) was also adopted in this area. The authors extracted the intermediate representation of raw EEG signals from each domain separately, and then used the glia chain to mine correlation information. Lastly, they fused all information together using the restricted Boltzmann machine (RBM) to implement emotion recognition, achieving 75.92% and 76.83% for arousal and valence, respectively [12]. On the other hand, the deep learning method proposed by [26] with a 2D CNN achieved comparable performance to the SVM. However, there are two significant drawbacks to applying 2D CNNs to raw EEG signals, namely covariance shift and the unreliability of the emotional ground truth. Covariance shift refers to the difference in statistical distribution between training and testing data, which is severe in EEG signals due to the non-stationary nature of the signal [44]. Usually, raw EEG signals are segmented into several input sequences to augment the data. Emotion EEG trials should correspond to their ground truths, which are self-reported. The difference between the average of the segmented signals and ground truths causes the unreliability of each epoch, which influences the model training [26]. To address these two problems, the 3D CNN structure has been introduced because of its ability to simultaneously extract spatial and temporal features. Salama et al. [91] proposed a 3D CNN strategy for EEG-based emotion recognition with a data augmentation method by adding normalized random Gaussian noise and achieved

better results compared with the previous methods. Yang et al. [112] developed a different multi-column CNN structure whose prediction is produced by a weighted sum of the decisions from all individual recognizing modules, and they obtained approximately 90% on both valence and arousal labels. Zhao et al. [116] constructed another 3D CNN model with reshaped channel matrices that achieved the state-of-the-art results of 96.43% and 96.61% for valence and arousal, respectively. However, the large parameter counts of the redundant model prevent its usage in practice.

10.3 EEG Signals

EEG signals can be classified into different ranges based on their frequency, namely delta, theta, alpha, beta, or gamma waves.

Delta waves are the slowest and primarily exist between 1 Hz and 4 Hz. They are often known as deep sleep waves because they are more common while the human body is in deep meditation or under relaxing conditions and deep sleep. During this cycle of waves, the body is recovering and regenerating from the previous day's events [86].

Theta waves are mostly generated around the 4–7 Hz range. These waves are correlated with both light meditation and sleep. When the brain generates an increasing amount of theta waves, it is said to be in "dream mode." Humans undergo rapid eye movement (REM) sleep in this condition. According to [86], frontal theta waves are associated with information retrieval, learning, and memory retention.

Alpha waves exist primarily in the range of 8–12 Hz and are also known as deep calming waves. They reflect the brain's resting state and are prevalent during periods of daydreaming or meditation. Alpha waves have effects on imagination, visualization, understanding, memory, and focus. According to [86], alpha waves are linked to reflecting sensory, motor, and memory functions.

Beta waves are most common at frequencies ranging from 12 Hz to 25 Hz. These waves are linked to a person's awareness and alertness. When we are wide awake or aware, engaging in some kind of mental task, such as problem-solving or decision-making, these waves are more prevalent [86].

Gamma waves are the fastest and are, like beta waves, most common while a person is alert and awake. Cognition, knowledge retrieval, attention span, and memory are all correlated with these waves. Gamma waves are thought to represent a person's "higher virtue," such as altruism, compassion, and spiritual emergence [86].

10.3.1 Emotions Detection

Emotions vary from one person to another. Several methods have been presented in the literature for modeling human emotions. One model depicts simple emotions such as happiness and sadness [106], while the other depicts fear, anger, depression, and satisfaction [41]. Another modeling method employs several measurements or scales to categorize human emotions [84]. The most prevalent human emotion model uses two key dimensions to model the emotions, namely valence and arousal. Valence varies from positive to negative, while arousal ranges from low to high. Fear, for example, has negative valence and high arousal, while excitement has positive valence and high arousal [88].

In the frequency domain, power features are often used in researches. The alpha band power spectral density (PSD) of EEG correlates with valence [103]. Furthermore, when the delta and theta bands of the power spectral density (PSD) of EEG are extracted from three central channels, they contain information that is associated with valence and arousal [19]. We focused on the arousal and valence scale in this study.

10.4 Proposed Method

10.4.1 Efficient 3-D CNN Models with Inverted Residual Block

The 3D CNN expands on the traditional 2D CNN by adding an additional dimension. 3D CNN is written as follows:

$$y^l_{i,j,k} = \sum_{a=0}^{m}\sum_{b=0}^{n}\sum_{c=0}^{p} \omega_{a,b,c} * x^{l-1}_{i-a,j-b,k-c} \tag{10.1}$$

where x^{l-1} is the output from the previous layer after activation is applied, ω is the 3D convolutional kernel with size $m*n*p$, and y^l is the convolution output. To the best of our knowledge, only a few studies have used 3D CNNs specifically for EEG signals. To test the benefits of using 3D CNNs, we modified the most widely used architectures in the literature, namely ResNet-18 [34] and MobileNetV2 [93], by replacing 2-D CNNs with 3-D CNNs. The use of these models on our EEG dataset resulted in excellent accuracy at the expense of very high storage and computational costs, as demonstrated in Section 10.5.4 . However, deploying these models on edge BCI devices is not a viable option.

Model	*Total Param.*	*Width Factor*	t	*Output Neurons*
EEGNet V1	6.4K	0.4	2	320
EEGNet V2	14.6K	0.5	3	640
EEGNet V3	24.8K	0.8	4	640

Table 10.1: Proposed Models with Their Corresponding Parameter Setting

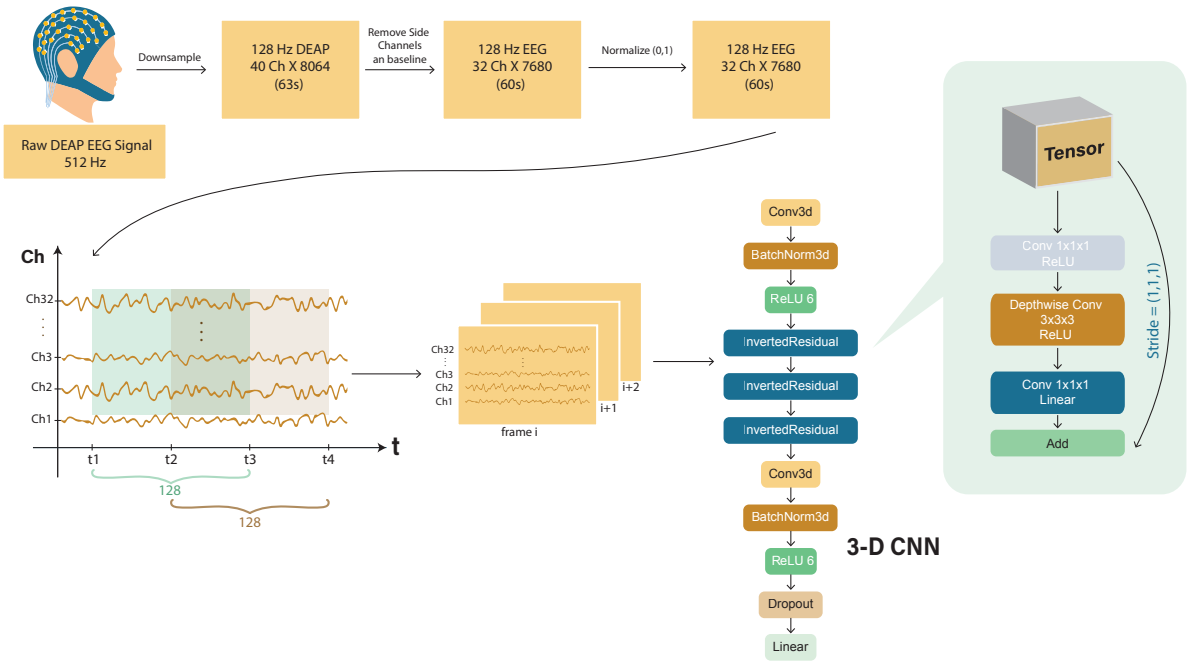


Figure 10.1: The data process steps and proposed EEGNet architecture

We created three extremely efficient network models named EEGNet V1, V2, and V3, in which we modified the block structure introduced by [93] by replacing the inverted residual blocks and depthwise separable convolutions to 3D operations instead of 2D, as illustrated in Figure. 10.1.

Depthwise separable convolution decoupled the standard convolution into a 3×3 depthwise convolution and a 1×1 pointwise convolution, which reduced the learning parameters and computational costs of the networks [37]. In a standard residual block, inputs are followed by multiple bottleneck layers, which are then followed by expansions. As demonstrated with 3D inverted residual blocks in Fig. 10.1, we reversed the bottleneck and expansion layers, and then applied shortcut connections directly between the two bottlenecks when both the input and output tensors of the block had the same shape. The blocks also had their 3D batch normalization layer and ReLU activation layer inside. Our models consisted of three inverted residual blocks and then were followed by

another point-wise convolution layer to expand the feature maps for classification. The last two layers were a dropout layer, which was included for regularization to prevent an overfitting and a fully connected linear layer for generating classification logits, which then went through a SoftMax function to produce final classification probabilities with two classes (positive and negative).

As shown in Table 10.1, we tuned three hyperparameters to create three versions of EEGNet models: the expansion factor t, the width multiplier factor (WF), and the number of output neurons of the last convolution layer. The expansion factor t determines the number of times to expand the channels of the input tensor to the hidden expansion layers of the second and third inverted residual blocks. WF determines the overall model channel width. The number of output neurons controls the number of output neurons of the last point-wise convolution layer. V1, V2, and V3 are in ascending order of model complexity, but also increasing test accuracy as well. They are targeting edge platforms with different resource constraints and latency requirements.

10.4.2 Model Binarization

To further compress the models, we binarized the weights and activation within the inverted residual blocks as the second stage of our framework. Binarization is an extreme case of quantization in which only one bit is used to represent the number in the weights and activation, thereby greatly reducing the computation and memory footprint. The XNOR-Net is a common binarization method that achieves up to 58x of the inference speed and 32x memory saves [87]. The issue with BNNs is that model accuracy often degrades significantly. Hence, we introduced three techniques to improve the accuracy of the binarized models with little to no additional costs in terms of storage and computational resources.

Adding real-value residual connections to each block

Residual connections provide the possibility of constructing very deep models without performance degradation. Normally, feature maps produced by convolution operations are directly calculated, but [34] suggested that the convolution layers have difficulty learning the identity mapping, assuming the solution is already optimal from the previous layer. With residual connection between layers, the feature map $H(x)$ is now constructed by two parts: $H(x) = x + F(x)$, where x is identity mapping and $F(x)$ is the residual that is learned by the current layer; thus, the performance of a deep model will at the worst be equal to its shallow counterpart. Our baseline models applied residual connection only when the given input channel was equal to output channel and the stride was (1,1,1) as illustrated in Fig. 10.1, (i.e. only the first block).

The previous residual setting was sufficient for real-value baseline models since our network architecture was not very deep. However, the binarized models would suffer significantly due to information loss when applying nonlinear activation even with a shallow network structure. Hence, for the binarized network, we preserved the real-value feature map from the previous layer and added it to the output activation. Then, we applied the real-value residual connections to all three blocks in the proposed models. If the convolution stride was not (1,1,1) or the channels of input/output were different, we first downsampled the real-value feature map to its desired shape and then added it as illustrated in Fig. 10.2. Those dense real-value connections increased the representation capability significantly due to the limited knowledge that the binarized activation maps contained. The only additional cost of adding more real-value shortcuts is a small amount of element-wise addition operations with no extra memory cost because the addition operations are computed on the fly during the inference stage.

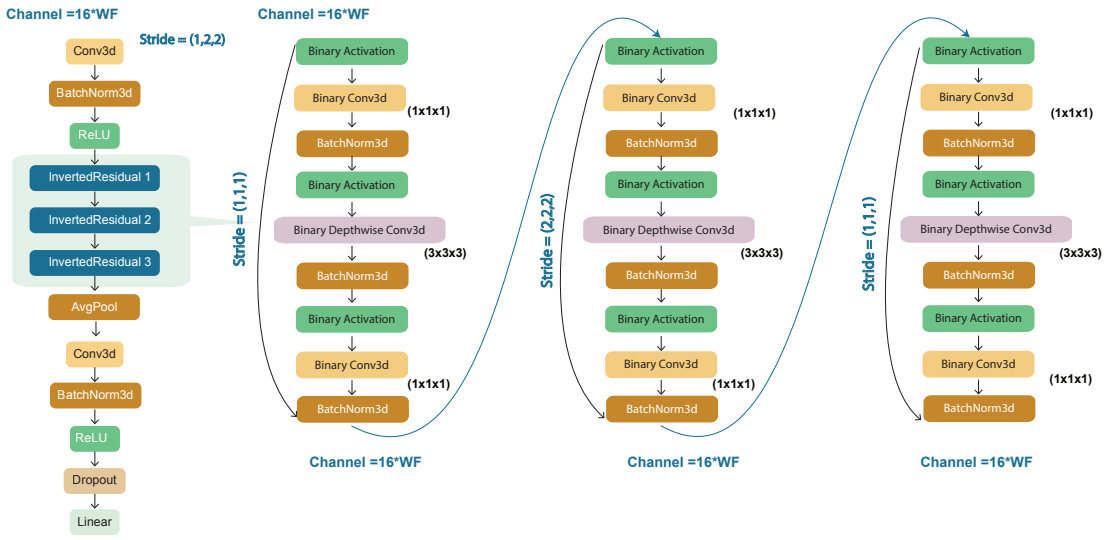


Figure 10.2: Proposed Binary EEGNet Architecture

Applying channel-wise scaling factor

Binarizing a neural network by applying the sign function $X_b = sign(X_r)$ to both the weights and the activation would result in crucial information loss. To address this problem, Rastegari et al. [87] suggested the scaling factors α and K to approximate floating-point weights and activation after binarization, as follows:

$$A * W \approx (sign(A) \odot sign(W))\alpha K \tag{10.2}$$

Where A and W are weights and activation, $*$ is the real value convolution operator, $\odot$ represent binary convolution with XNOR and bits shift operations, α is a weight scaling factor such that $W_r \approx \alpha W_b$, and K is the scalar factor matrix of the corresponding activation. We removed K due to its high computational cost and negligible impact on performance, as suggested by [87]. However, we found that analytically calculated $\alpha = \frac{||W||}{n}, n = c \times w \times h$ was not optimal. Specifically, it averaged out all of the weighting channels but ignored their significance and overall magnitude levels. Therefor, we proposed a channel-wise scaling factor, and added the third dimension to share model

representation ability with $\alpha_i = \frac{||W_i||}{n}, n = k \times w \times h, \alpha \in \mathbb{R}^c$, where c is weight channels and $k \times w \times h$ is data dimensions. The use of the channel-wise scaling factor improved the performance of the binarized models significantly, as the experiment and results section will be demonstrated.

Architecture modification for additional compression

In the proposed models, we binarized the layers inside the inverted residual blocks for better performance. The first and last full-precision convolution layers, which require a high computation and storage costs, became the most expensive parts of the model during training and inference. Therefore, to further compress the binarized models, we reduced the number of filters in the first channel from 32×WF to 16×WF. This avoided the need to perform millions of multiply–accumulate operations with negligible performance loss due to the redundant nature of CNNs. Furthermore, we modified the final full-precision convolution layer, which usually requires a high computation and storage costs due to its property of generating a large number of feature maps for the final prediction, has been modified by adding the average pooling layer ahead. This resulted in the computations being conducted at a $1 \times 1 \times 1$ spatial-temporal resolution in the final layer rather than $3 \times 8 \times 32$. Additionally, the last stage of tuning not only reduced the computation but also increased the accuracy as well because the early average pooling helped to avoid location sensitivity of the input features in the activation.

Combining these three methods vastly improved the performance of the binarized models.

Method	*Accuracy*	*Improve Δ*
Plain BNN	75.14%	-
1- Connection Real Values	80.3%	+5.16
2- Channel-Wise	91.93%	+16.79
3 - Tuning Last Stage	77.28%	+2.14
1 & 2	92.97%	+17.83
1 & 3	78.89	+3.75
2 & 3	95.33	+20.19
1 & 2 & 3	94.96%	+19.82
Full Percision	96.37%	-

Table 10.2: Methods with Binary Neural Networks Using EEGNet V2

10.5 Experiment and Results Analysis

10.5.1 DEAP Dataset

DEAP[50] is a well-known public EEG dataset for the analysis of human affective states. It consists of 32 participants with recordings of each watching 40 1-minute-long excerpts of music videos. Then, the participants rated each video for levels of valence, arousal, liking, and dominance [50]. The result was 63 seconds with 8064 sample points for each channel of every 1-minute trial. The first 3 seconds of the trial was the baseline prior to the actual experiment and was removed in our data process step. There were 40 channels that had been recorded with 32 EEG channels and eight side channels. The EEG signals were recorded using the standard international 10-20 system with 32 active AgCl electrodes [50]. The rated score of each video clip ranged from 1 to 9 per label.

	4	***6***	***8***	***10***
Arousal	64.2%	99.5%	98.5%	97.1%

Table 10.3: Arousal Test Accuracy vs. Number of Frames per Chunk with MobileNetV2-3D on the DEAP Dataset

10.5.2 Data Preprocessing and 3D Representation

We first adopted a preprocessing method that is widely used in the literature [32], in which we downsampled the data from 512 Hz to 128 Hz and removed electrooculography (EOG) artifacts. Then, to eliminate unwanted frequency components, we introduced a bandpass filter, which only preserved the 4-45Hz frequency range covering theta, alpha, and beta waves. The 3-seconds pre-trial baseline and the eight side channels were then removed. Subsequently, we then normalized the data for each channel of each trial to be between 0 and 1. Lastly, we divided each trial into 32 1-second data frames with a window size of 128 points and an overlapping ratio of 50%, as illustrated in Fig. 10.1. In a short period of time, this method preserves the temporal stationarity of EEG signals.

To prepare the datasets for 3D CNNs, we stacked up multiple 32-channels by 1-second-long consecutive data frames to form 3D data chunks [91]. After experimenting with various frame sizes, we selected six frames to continue as they yielded the best accuracy, as presented in Table 10.3. Then, we assigned each chunk the labels that were the same as the ground-truth labels of its corresponding trial. In total, we constructed 25,600 chunks of data with a size of 6*32*128 as the method presented. We experimented with valence and arousal perception in this study with their corresponding labels. A threshold value of 5 was applied to assign positive and negative labels from the provided 1-9 scores, as this is common in the area of research literature. Finally, our proposed method is presente in Figure. 10.1.

10.5.3 Training Setting

All models were trained on an NVIDIA Tesla V100 GPU and implemented with the PyTorch framework [81]. We split the training and validation datasets with 80% and 20% respectively. The dropout layer was set to a 0.2 dropout rate. We trained the

models for 100 epochs with a batch size of 256 and adopted the Adam optimizer [48] . The initial learning rate was set to 0.001 and the multi-step learning rate scheduler was applied with a milestone set to 75, and gamma equal to 0.5. We used the cross entropy loss function across all models and applied additional label smoothing [76] with $\epsilon = 0.1$ specifically to the binarized models. Label smoothing provides additional regularization by constructing soft labels as

$$label_{soft} = (1 - \epsilon) * label - \frac{\epsilon}{K} \tag{10.3}$$

where K is the number of label classes, which we set to 2.

10.5.4 Results of Baseline Models

Our proposed method significantly outperformed previous studies without special feature extraction or signal transformation while still maintaining a compact size. As Table 10.4 demonstrates, our EEGNet V1 achieved better results than [112] and [91] but with over 10x fewer learning parameters, which is only 6.4K. Our proposed EEGNet V2 achieved performance comparable to with the state-of-the-art 3D CNN approach [116] and 3D ResNet18 in the experiment but with only 14.6K learning parameters, which is less than 1% of the parameter counts compared to with [116]. Ultimately, our largest V3 variant achieved an average classification accuracy of 99.5% with only 24.8K parameters and significantly outperformed the previous studies. To the best of our knowledge, these three variants of our proposed method outperformed the state-of-the-art models in terms of both accuracy and model size. To elaborate the performance analysis in more detail, we have included validation precision, recall, and F1 score of each model to show its effectiveness and robustness in Table 10.4.

The 3-D CNNs with batch normalization, dense prediction, inverted residual blocks, and depthwise separable convolution made our models efficient and yielded outstanding

Table 10.4: The Precision, Recall, and F1-Score of the Proposed EEGNet and Binary EEGNet Models (DEAP)

Models	*Precision (%)*		*Recall (%)*		*F1 Score (%)*	
	Valence	*Arousal*	*Valence*	*Arousal*	*Valence*	*Arousal*
EEGNet V1	90.00	90.22	89.37	92.89	89.70	91.65
EEGNet V2	96.73	96.63	96.80	97.54	96.77	97.10
EEGNet V3	99.72	99.59	99.30	99.56	99.50	99.58
Bi-EEGNet V1	80.20	81.55	84.96	86.80	82.63	84.25
Bi-EEGNet V2	95.02	96.90	96.46	94.98	95.75	95.88
Bi-EEGNet V3	99.23	99.20	99.54	99.83	99.40	99.53

results in terms of accuracy and model size. Batch normalization helped to solve the covariance shift problem [44], and dense prediction solved the unreliability issue [26]. Inverted residual blocks maintained the manifolds of interest in neural networks and reduced the information loss when performing non-linear transformation [93]. Finally, applying depthwise separable convolution reduced the parameters and the computation complexity of the models. These advantages make efficient model deployment become possible for low power and resource-constrained devices at the edge.

10.5.5 Result of Binarized Models

Our model binarization algorithm adopted the piece-wise polynomial function proposed by [69] to estimate the derivative of the sign function to successfully propagate the binary models. Table 10.2 presents the effectiveness of our three proposed techniques, which resulted in model performance improvements of 5%, 17%, and 2%, respectively, on the arousal label with EEGNet V2 setting. The valence label exhibited a similar performance gain in our experiments. By combining these, we achieved an improve-

Model	*Valence*	*Arousal*	*Param.*	*Mem. Usage*
Samara et al. [92]	66.9%	66.69%	-	-
Chao et al. [12]	76.83%	75.92%	-	-
Wang et al. [105]	72.1%	73.3%	-	-
Yanagimoto et al. [109]	81.16%	-	-	-
Salama et al. [91]	87.44	88.49	2.5M	75.13 Mbits
Yang et al. [112]	90.01	90.65	314K	9.58 Mbits
Zhao et al. [116]	96.43	96.61	170M	5435 Mbits
Resnet18-3D	92.77	97.53	33.2M	1012 Mbits
MobileNetV2-3D	99.68	99.70	2.4M	71.87 Mbits
EEGNet V1	**88.44%**	**90.04%**	**6.4K**	**0.20 Mbit**
EEGNet V2	**96.37%**	**96.60%**	**14.6K**	**0.45 Mbit**
EEGNet V3	**99.45%**	**99.51%**	**24.8K**	**0.76 Mbit**
Bi-EEGNet V1	**80.00%**	**80.95%**	**3.5K+9.1K***	**0.11 Mbit**
Bi-EEGNet V2	**94.42%**	**95.05%**	**8.1K+33K***	**0.28 Mbit**
Bi-EEGNet V3	**99.32%**	**99.43%**	**11K+130K***	**0.46 Mbit**

Note: * denote 1-bit binary parameters.

Table 10.5: Performance Comparison with Previous Studies

ment of over 20% over the plain binary model while still maintaining similar storage and computational costs. As indicated in Table 10.5, the binarized models with V2 and V3 settings achieved a performance comparable to their full-precision counterparts while further compressing the size by over 40%. The memory usage (i.e, model size) was calculated by summing up the 32-bit multiples the number of real-valued parameters and 1-bit multiples the number of binary parameters in each model. Hence, our binarized models can take advantage of the proposed method and speed up the inference significantly when deploy on edge systems while still maintaining state-of-the-art performance.

10.6 Conclusion

In this chapter, we have proposed a two-stage 3D CNN framework that specializes in emotion recognition tasks using time-domain EEG signals. It extracts spatiotemporal feature representations automatically. The public DEAP dataset was used to conduct experiments with data processing techniques to form the 3D inputs. Our models demonstrated superior performance

in classifying both valence and arousal perceptions, which can easily be processed into actual human emotions afterwards. The introduced baseline EEGNet V1, V2, and V3 in the first stage achieved average classification accuracies of 90%, 96.6%, and 99.5%, respectively, with a small number of learning parameters and a compact model size. In the second stage, we binarized these models with three novel techniques to perform further compression and help to take advantage of efficient bitwise operation. Model binarization saved storage costs and computational resources by over 40% compared with the baseline EEGNet while still maintaining comparable performance to their full-precision counterparts. Finally, the efficient models that we presented helped to make the real-time deployment of a precise human emotion recognition system in a resource-constrained environment a viable option

Chapter 11

Conclusion and Discussion

This dissertation has presented two different methods for compressing deep neural networks. Our FPTT method worked best for neural network models with convolutional and fully connected layers, in which we used the FP method on the convolution layers and TT decomposition on the fully connected layers; then, we manually modified the filters until we achieved optimal compression while keeping the accuracy intact or the degradation to less than 2%. Our second method is called Ultimate Compression, which is agnostic of the deep neural network architecture, in which we employed three different tensor decomposition methods, namely Tucker, CP, and TT, and compared them in terms of accuracy and compression ratio. We applied them on all convolutional and fully connected layers, except for the first and last layers. We found that the TT method outperformed the other methods in terms of compression and inference accuracy. Furthermore, we introduced a method that selects the rank for decomposition based on the layer sensitivity and compared it with other methods. We concluded that the sensitivity method outperforms the other rank selection algorithms in terms of compression and accuracy. We also investigated how the initialization and activation functions affected the convergence and accuracy of the models, and concluded that using orthogonal initialization with the PReLU activation function and sensitivity as the

selection rank algorithm for the decomposed binarized models outperformed the other methods. We believe that orthogonal initialization outperforms other initialization methods due to the dynamic isometric condition, which is defined as the equilibration of singular values of the input–output Jacobian matrix.

In addition, this dissertation has investigated and presented various methods for improving quantized deep neural network models without increasing hardware complexity. We presented a method for improving BNNs by ensembling them and sharing the weight filters among the models. We compared three methods for ensembling, namely fusion, voting, and bagging, with six different weight filter configurations. A trade-off exists to our method in that using nonshared configurations yields better accuracy but increases hardware complexity, whereas using an all-frozen configuration yields the best performance in terms of hardware complexity but greatly degrades the accuracy. We also introduced various configurations that demonstrated improved accuracy while maintaining hardware complexity to a minimum. Our method is dynamic, based on the power resources available to edge devices. The model can switch between different weight filter sharing configurations. In the second method, we improved the models quantized using logarithmic representation by stochastic rounding on input weights multiple times and then averaging them. To avoid the hardware complexity required for accumulation and division of the stochastic rounding of input weights, we modified the hardware by using the efficient logic blocks NDL and LSFR. Thus, we were able to build our method while maintaining the model complexity low. In addition, we improved the performance of the BNN by decomposing the models using TT decomposition with a large rank, which increased the number of parameters. As a result, the model's accuracy increased by 4%–5%. A large rank in this method increased the size of the models to almost the same as the floating-point method; however, these models have the advantage of using binary arithmetic instead of floating-point arithmetic, which greatly reduces hardware and computation costs, especially for devices at the

edge.

In this dissertation, we have implemented two different applications at the edge. The first application is crowd counting, for which we used the Ultimate Compression method to create efficient models that are deployed on surveillance cameras rather than sending the camera feed to the cloud. The second application is emotion detection using EEG, in which we modified and built efficient models by replacing 2D convolution layers in state-of-the-art mobilnetv2 with 3D convolution layers. We then binarized the model and introduced various methods aimed at different inference and storage costs.

There are numerous future works that can be built using the methods presented in this dissertation. Using network distillation to create a chain of quantized or decomposed models of varying ranks. In addition, our FPTT method can be combined with an 8bit quantization to further compress the models. We can also use these methods, such as pruning, quantization, and tensor decomposition, to build different models for specific domains using multi-objective baysaian optimization, where the objective function is the model's accuracy and storage cost. Federated learning is a machine learning technique that trains an algorithm across multiple decentralized edge devices or servers holding local data samples without exchanging them, and uses compression methods to accelerate model training and inference.

Bibliography

[1] The growth in connected iot devices is expected to generate 79.4zb of data in 2025, according to a new idc forecast — business wire. https://www.businesswire.com/news/home/20190618005012/en/The-Growth-in-Connected-IoT-Devices-is-Expected-to-Generate-79.4ZB-of-Data-in-2025-According-to-a-New-IDC-Forecast. (Accessed on 12/23/2021).

[2] How nas was improved from many days to hours in search time. https://peltarion.com/blog/data-science/nas-search. (Accessed on 12/31/2021).

[3] Synopsys 28/32nm standard cell libraries — synopsys. https://www.synopsys.com/dw/ipdir.php?ds=dwc_standard_cell. (Accessed on 18/06/2019).

[4] A. F. Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[5] R. Alkawadri. Brain–computer interface (bci) applications in mapping of epileptic brain networks based on intracranial-eeg: An update. *Frontiers in neuroscience*, 13:191, 2019.

[6] M. Alnemari. *Integration of a Low Cost EEG Headset with The Internet of Thing Framework*. PhD thesis, UC Irvine, 2017.

[7] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[8] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[9] C. Bucil, R. Caruana, and A. Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining-KDD*, volume 6, page 535.

[10] A. Canziani, A. Paszke, and E. Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[11] B. Casvanden and C. Bogaard. VBMF, 11 2017.

[12] H. Chao, H. Zhi, L. Dong, and Y. Liu. Recognition of emotions using multichannel eeg data and dbn-gc-based ensemble deep learning framework. *Computational intelligence and neuroscience*, 2018, 2018.

[13] Z. Cheng, B. Li, Y. Fan, and Y. Bao. A novel rank selection scheme in tensor ring decomposition based on reinforcement learning for deep neural networks. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3292–3296. IEEE, 2020.

[14] A. Cichocki, N. Lee, I. Oseledets, A.-H. Phan, Q. Zhao, and D. P. Mandic. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. *Foundations and Trends® in Machine Learning*, 9(4-5):249–429, 2016.

[15] N. Cohen, O. Sharir, and A. Shashua. On the expressive power of deep learning: A tensor analysis. In *Conference on learning theory*, pages 698–728. PMLR, 2016.

[16] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. *arXiv preprint arXiv:1511.00363*, 2015.

[17] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[19] C. A. Frantzidis, C. Bratsas, C. L. Papadelis, E. Konstantinidis, C. Pappas, and P. D. Bamidis. Toward emotion aware computing: an integrated approach using multichannel neurophysiological recordings and affective visual stimuli. *IEEE Transactions on Information Technology in Biomedicine*, 14(3):589–597, 2010.

[20] T. Furlanello, Z. Lipton, M. Tschannen, L. Itti, and A. Anandkumar. Born again neural networks. In *International Conference on Machine Learning*, pages 1607–1616. PMLR, 2018.

[21] M. Ganaie, M. Hu, et al. Ensemble deep learning: A review. *arXiv preprint arXiv:2104.02395*, 2021.

[22] J. Gao, W. Lin, B. Zhao, D. Wang, C. Gao, and J. Wen. C^ 3 framework: An open-source pytorch code for crowd counting. *arXiv preprint arXiv:1907.02724*, 2019.

[23] T. Garipov, D. Podoprikhin, A. Novikov, and D. Vetrov. Ultimate tensorization: compressing convolutional and fc layers alike. *arXiv preprint arXiv:1611.03214*, 2016.

[24] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[25] J. Gu, C. Li, B. Zhang, J. Han, X. Cao, J. Liu, and D. Doermann. Projection convolutional neural networks for 1-bit cnns via discrete back propagation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 8344–8351, 2019.

[26] H. Gunes and B. Schuller. Categorical and dimensional affect analysis in continuous input: Current trends and future directions. *Image and Vision Computing*, 31(2):120–136, 2013.

[27] N. Guo, J. Bethge, H. Yang, K. Zhong, X. Ning, C. Meinel, and Y. Wang. Boolnet: Minimizing the energy consumption of binary neural networks. *arXiv preprint arXiv:2106.06991*, 2021.

[28] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[29] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[30] L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (10):993–1001, 1990.

[31] B. Hassibi and D. G. Stork. *Second order derivatives for network pruning: Optimal brain surgeon*. Morgan Kaufmann, 1993.

[32] N. Hazarika, J. Z. Chen, A. C. Tsoi, and A. Sergejew. Classification of eeg signals using the wavelet transform. *Signal processing*, 59(1):61–72, 1997.

[33] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[34] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[35] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, 2019.

[36] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[37] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[38] G. Huang, Y. Li, G. Pleiss, Z. Liu, J. E. Hopcroft, and K. Q. Weinberger. Snapshot ensembles: Train 1, get m for free. *arXiv preprint arXiv:1704.00109*, 2017.

[39] H. Idrees, I. Saleemi, C. Seibert, and M. Shah. Multi-source multi-scale counting in extremely dense crowd images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2547–2554, 2013.

[40] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[41] C. E. Izard, J. Kagan, and R. B. Zajonc. *Emotions, cognition, and behavior*. CUP Archive, 1984.

[42] S. Ji, W. Xu, M. Yang, and K. Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2012.

[43] Y. Jia and E. Shelhamer. Caffe — model zoo. `https://caffe.berkeleyvision.org/model_zoo.html`. (Accessed on 01/25/2019).

[44] S. Jirayucharoensak, S. Pan-Ngum, and P. Israsena. Eeg-based emotion recognition using deep learning network with principal component based covariate shift adaptation. *The Scientific World Journal*, 2014, 2014.

[45] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[46] H. Kim. Aresb-net: accurate residual binarized neural networks using shortcut concatenation and shuffled grouped convolution. *PeerJ Computer Science*, 7:e454, 2021.

[47] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[48] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[49] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, and D. J. Inman. 1d convolutional neural networks and applications: A survey. *Mechanical systems and signal processing*, 151:107398, 2021.

[50] S. Koelstra, C. Muhl, M. Soleymani, J.-S. Lee, A. Yazdani, T. Ebrahimi, T. Pun, A. Nijholt, and I. Patras. Deap: A database for emotion analysis; using physiological signals. *IEEE transactions on affective computing*, 3(1):18–31, 2011.

[51] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[52] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic. Tensorly: Tensor learning in python. *arXiv preprint arXiv:1610.09555*, 2016.

[53] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[54] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[55] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[56] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[57] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[58] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[59] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[60] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong. Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904. IEEE, 2017.

[61] S. Lee, H. Sim, J. Choi, and J. Lee. Successive log quantization for cost-efficient neural networks using stochastic computing. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[62] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[63] Y. Li, X. Zhang, and D. Chen. Csrnet: Dilated convolutional neural networks for understanding the highly congested scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1091–1100, 2018.

[64] M. Lin, Q. Chen, and S. Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[65] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*, 2015.

[66] Y. Liu, J. Pan, and M. Ng. Tucker decomposition network: Expressive power and comparison.

[67] Z. Liu, W. Luo, B. Wu, X. Yang, W. Liu, and K.-T. Cheng. Bi-real net: Binarizing deep network towards real-network performance. *International Journal of Computer Vision*, 128(1):202–219, 2020.

[68] Z. Liu, Z. Shen, M. Savvides, and K.-T. Cheng. Reactnet: Towards precise binary neural network with generalized activation functions. *arXiv preprint arXiv:2003.03488*, 2020.

[69] Z. Liu, B. Wu, W. Luo, X. Yang, W. Liu, and K.-T. Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European conference on computer vision (ECCV)*, pages 722–737, 2018.

[70] J.-H. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.

[71] G. Maguolo, L. Nanni, and S. Ghidoni. Ensemble of convolutional neural networks trained with different activation functions. *Expert Systems with Applications*, 166:114048, 2021.

[72] B. Martinez, J. Yang, A. Bulat, and G. Tzimiropoulos. Training binary neural networks with real-to-binary convolutions. *arXiv preprint arXiv:2003.11535*, 2020.

[73] S. I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh. Improved knowledge distillation via teacher assistant. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5191–5198, 2020.

[74] D. Miyashita, E. H. Lee, and B. Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[75] S.-E. Moon, S. Jang, and J.-S. Lee. Convolutional neural network approach for eeg-based emotion recognition using brain connectivity and its spatial information. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2556–2560. IEEE, 2018.

[76] R. Müller, S. Kornblith, and G. Hinton. When does label smoothing help? *arXiv preprint arXiv:1906.02629*, 2019.

[77] S. Nakajima, M. Sugiyama, and R. Tomioka. Global analytic solution for variational bayesian matrix factorization. *Advances in Neural Information Processing Systems*, 23:1768–1776, 2010.

[78] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov. Tensorizing neural networks. *arXiv preprint arXiv:1509.06569*, 2015.

[79] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[80] Y. Pan, M. Wang, and Z. Xu. Tednet: A pytorch toolkit for tensor decomposition networks. *arXiv preprint arXiv:2104.05018*, 2021.

[81] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[82] H. Phan, Y. He, M. Savvides, Z. Shen, et al. Mobinet: A mobile binary network for image classification. In *The IEEE Winter Conference on Applications of Computer Vision*, pages 3453–3462, 2020.

[83] B. Poole, S. Lahiri, M. Raghu, J. Sohl-Dickstein, and S. Ganguli. Exponential expressivity in deep neural networks through transient chaos. *Advances in neural information processing systems*, 29:3360–3368, 2016.

[84] J. Posner, J. A. Russell, and B. S. Peterson. The circumplex model of affect: An integrative approach to affective neuroscience, cognitive development, and psychopathology. *Development and psychopathology*, 17(3):715, 2005.

[85] R. Qiao, C. Qing, T. Zhang, X. Xing, and X. Xu. A novel deep-learning based framework for multi-subject emotion recognition. In *2017 4th International Conference on Information, Cybernetics and Computational Social Systems (ICCSS)*, pages 181–185. IEEE, 2017.

[86] R. A. Ramadan, S. Refat, M. A. Elshahed, and R. A. Ali. Basics of brain computer interface. In *Brain-Computer Interfaces*, pages 31–50. Springer, 2015.

[87] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[88] B. Reuderink, C. Mühl, and M. Poel. Valence, arousal and dominance in the eeg during game play. *International journal of autonomous and adaptive communications systems*, 6(1):45–62, 2013.

[89] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[90] Sagartesla. flops-cnn. `https://github.com/sagartesla/flops-cnn`, 2020. Accessed: 2020-06-03.

[91] E. S. Salama, R. A. El-Khoribi, M. E. Shoman, and M. A. W. Shalaby. Eeg-based emotion recognition using 3d convolutional neural networks. *Int. J. Adv. Comput. Sci. Appl*, 9(8):329–337, 2018.

[92] A. Samara, M. L. R. Menezes, and L. Galway. Feature extraction for emotion recognition and modelling using neurophysiological data. In *2016 15th international conference on ubiquitous computing and communications and 2016 international symposium on cyberspace and security (IUCC-CSS)*, pages 138–144. IEEE, 2016.

[93] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[94] S. S. Schoenholz, J. Gilmer, S. Ganguli, and J. Sohl-Dickstein. Deep information propagation. *arXiv preprint arXiv:1611.01232*, 2016.

[95] U. Shabbir, J. Sang, M. S. Alam, J. Tan, and X. Xia. Comparative study on crowd counting with deep learning. In *Pattern Recognition and Tracking XXXI*, volume 11400, page 114000X. International Society for Optics and Photonics, 2020.

[96] H. Sim and J. Lee. Log-quantized stochastic computing for memory and computation efficient dnns. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 280–285, 2019.

[97] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[98] J. Sirignano and K. Spiliopoulos. Mean field analysis of deep neural networks. *Mathematics of Operations Research*, 2021.

[99] R. S. Srinivasamurthy. Understanding 1d convolutional neural networks using multiclass time-varying signals. 2018.

[100] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[101] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[102] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[103] G. K. Verma and U. S. Tiwary. Multimodal fusion framework: A multiresolution approach for emotion classification and recognition from physiological signals. *NeuroImage*, 102:162–172, 2014.

[104] S. Vogel, C. Schorn, A. Guntoro, and G. Ascheid. Efficient stochastic inference of bitwise deep neural networks. *arXiv preprint arXiv:1611.06539*, 2016.

[105] Y. Wang, Z. Huang, B. McCane, and P. Neo. Emotionet: A 3-d convolutional neural network for eeg-based emotion recognition. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2018.

[106] B. Weiner. Attribution, emotion, and action. 1986.

[107] S. Wu, G. Li, F. Chen, and L. Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018.

[108] Y.-X. Xu. Ensemble pytorch. `http:https://github.com/xuyxu/Ensemble-Pytorch`, 2020. Accessed: 2021-05-03.

[109] M. Yanagimoto and C. Sugimoto. Recognition of persisting emotional valence from eeg using convolutional neural networks. In *2016 IEEE 9th International Workshop on Computational Intelligence and Applications (IWCIA)*, pages 27–32. IEEE, 2016.

[110] G. Yang, J. Pennington, V. Rao, J. Sohl-Dickstein, and S. S. Schoenholz. A mean field theory of batch normalization. *arXiv preprint arXiv:1902.08129*, 2019.

[111] G. Yang and S. S. Schoenholz. Mean field residual networks: On the edge of chaos. *arXiv preprint arXiv:1712.08969*, 2017.

[112] H. Yang, J. Han, and K. Min. A multi-column cnn model for emotion recognition from eeg signals. *Sensors*, 19(21):4736, 2019.

[113] C. Zhang, H. Li, X. Wang, and X. Yang. Cross-scene crowd counting via deep convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 833–841, 2015.

[114] Y. Zhang, D. Zhou, S. Chen, S. Gao, and Y. Ma. Single-image crowd counting via multi-column convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 589–597, 2016.

[115] Q. Zhao, G. Zhou, S. Xie, L. Zhang, and A. Cichocki. Tensor ring decomposition. *arXiv preprint arXiv:1606.05535*, 2016.

[116] Y. Zhao, J. Yang, J. Lin, D. Yu, and X. Cao. A 3d convolutional neural network for emotion recognition based on eeg signals. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6. IEEE, 2020.

[117] W.-L. Zheng, H.-T. Guo, and B.-L. Lu. Revealing critical channels and frequency bands for emotion recognition from eeg with deep belief network. In *2015 7th International IEEE/EMBS Conference on Neural Engineering (NER)*, pages 154–157. IEEE, 2015.

[118] S. Zhu, X. Dong, and H. Su. Binary ensemble neural network: More bits per network or more networks per bit? In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4923–4932, 2019.

[119] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.