Aalto University
School of Science
Master's Programme in Security and Cloud Computing

Arnab Rahman Chowdhury

Distributed deep learning inference in fog networks

Master's Thesis
Espoo, July 30, 2020

Supervisors: Professor Mario Di Francesco, Aalto University
Professor Frank Alexander Kraemer, Norwegian University of Science and Technology
Advisor: Thaha Mohammed M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Security and Cloud Computing

ABSTRACT OF
MASTER'S THESIS

Author: Arnab Rahman Chowdhury

Title:
Distributed deep learning inference in fog networks

Date:	July 30, 2020	**Pages:**	82
Major:	Security and Cloud Computing	**Code:**	SCI3084
Supervisors:	Professor Mario Di Francesco Professor Frank Alexander Kraemer		
Advisor:	Thaha Mohammed M.Sc. (Tech.)		

Today's smart devices are equipped with powerful integrated chips and built-in heterogeneous sensors that can leverage their potential to execute heavy computation and produce a large amount of sensor data. For instance, modern smart cameras integrate artificial intelligence to capture images that detect any objects in the scene and change parameters, such as contrast and color based on environmental conditions. The accuracy of the object recognition and classification achieved by intelligent applications has improved due to recent advancements in artificial intelligence (AI) and machine learning (ML), particularly, deep neural networks (DNNs).

Despite the capability to carry out some AI/ML computation, smart devices have limited battery power and computing resources. Therefore, DNN computation is generally offloaded to powerful computing nodes such as cloud servers. However, it is challenging to satisfy latency, reliability, and bandwidth constraints in cloud-based AI. Thus, in recent years, AI services and tasks have been pushed closer to the end-users by taking advantage of the fog computing paradigm to meet these requirements. Generally, the trained DNN models are offloaded to the fog devices for DNN inference. This is accomplished by partitioning the DNN and distributing the computation in fog networks.

This thesis addresses offloading DNN inference by dividing and distributing a pre-trained network onto heterogeneous embedded devices. Specifically, it implements the adaptive partitioning and offloading algorithm based on matching theory proposed in an article by Mohammed et al. [25]. The implementation was evaluated in a fog testbed, including Nvidia Jetson nano devices. The obtained results show that the adaptive solution outperforms other schemes (Random and Greedy) with respect to computation time and communication latency.

Keywords: DNN inference, task partitioning, task offloading, distributed algorithm, DNN frameworks and architectures

Language: English

Acknowledgements

I would like to thank my thesis supervisors, Mario Di Francesco, Frank Alexander Kraemer, and my advisor Thaha Mohammed for helping and guiding me throughout the thesis process.

I am grateful to my friends for giving me moral support to complete the thesis on time.

Finally, I must express my gratitude to my parents for encouraging me to stay strong during the pandemic situation as they reside 4000 kilometers away from me. This accomplishment would not have been possible without them.

Thank you.

Espoo, July 30, 2020

Arnab Rahman Chowdhury

Abbreviations and Acronyms

DNN	Deep Neural Network
AI	Artificial Intelligence
ML	Machine Learning
RBM	Restricted Boltzmann Machine
DBN	Deep Belief Network
GAN	Generative Adversarial Network
RNN	Recurrent/Recursive Neural Network
LSTM	Long Short Term Memory
GRU	Gated Recurrent Unit
CNN	Convolutional Neural Network
NiN	Network in Network
ReLU	Rectified Linear Unit
DDNN	Distributed Deep Neural Network
DINA	Distributed INference Acceleration
DINA-P	Distributed INference Acceleration-Partitioning
DINA-O	Distributed INference Acceleration-Offloading
gRPC	gRPC Remote Procedure Call
MQTT	Message Queuing Telemetry Transport
BDD100k	Berkeley Deep Drive dataset

Contents

Chapter 1

Introduction

In recent years, the development of high-performance computing is steadily increasing. In particular, the computing capability of modern computers and embedded devices has increased due to the improvement of central processing unit and graphics processing unit architectures. As a result, these devices are capable of computing complex functions of artificial intelligence (AI) and machine learning (ML) for different applications, such as image and natural language processing, speech recognition, augmented reality, and cognitive assistance. Moreover, embedded devices equipped with sensors for data collection are mostly targeted for applications leveraging machine learning. According to Cisco, the number of smart devices connected via the internet will grow more than 12 billion by 2022 [5]. Therefore, the scope of data generation will increase for machine learning-based applications. In addition, the availability and the maintenance cost of cloud infrastructures have decreased in recent times, inspiring AI/ML-based applications to execute heavy computation in the cloud.

Generally, most of the machine learning applications are computed at smart devices, and sometimes the heavy computations are offloaded to the cloud. Smart devices are unable to execute heavy computation of AI/ML-based applications due to a lack of computing power and limited resources. Instead, they offload the computation to the cloud or employ a simple machine learning model, such as Support Vector Machines. However, offloading computation to the cloud adds extra costs due to communication: a high response time and a low throughput; moreover, the simpler machine learning models provide results with reduced accuracy. Fog computing has emerged to address these problems. In fact, fog computing provides decentralized computing infrastructure that brings the benefits and computing power of the cloud closer to the source of data [1].

Deep Neural Networks (DNNs) have recently been used in a variety of

intelligent applications. DNNs require heavy computing power and resources, significantly increase the computation burden at the end devices. Moreover, DNNs are progressing towards deeper structures to provide more system accuracy and precision [14, 32, 36]. Researches in deep neural networks have proposed different techniques that involve partitioning and offloading the DNN structure (model) into a distributed computing hierarchy [16, 18, 23, 38]. A distributed computing hierarchy consists of the local network, the fog, and the cloud. This process distributes the computation costs and parallelizes the computation to adopt scalability and efficiency.

1.1 Problem statement

Despite their growth, AI/ML-based applications encounter important key challenges. Smart (embedded) devices have lower energy and computing resources, whereas intelligent applications (e.g., image recognition, natural language processing, and speech detection) require high processing power. Moreover, some of these applications are latency-sensitive and safety-critical, such as augmented reality and cooperative autonomous driving. Dynamic network conditions also affect collaborative partitioning and distribution of computation [16, 18]. Considering the challenges, this thesis aims to answer the following questions:

- Which approach is appropriate to address the challenges imposed by AI/ML-based applications?
- Are solutions for addressing the challenges proposed in the literature practical for use at resource-constrained embedded devices?

This thesis addresses offloading DNN inference by dividing and distributing a pre-trained network onto heterogeneous embedded devices. In particular, it implements an adaptive partitioning and a distributed offloading algorithms based on matching theory introduced in the work by Mohammed et al. [25]. DNN inference refers to the technique of predicting the results of a DNN process while encountering new data. The implementation was evaluated in a testbed consists of heterogeneous fog devices (Nvidia Jetson Nano devices).

The main contributions of this thesis are as follows. First, it implements an adaptive partitioning scheme based on the computing capability and the utilities of the fog nodes. Utility is used to express association preference among the end devices and the fog nodes. The source DNN is divided into sublayers by the adaptive partitioning algorithm in order to offload to fog

networks. Second, this thesis implements a distributed algorithm based on swap matching for offloading the sublayers of DNN from end devices to the fog networks. Finally, experiments are carried out by considering three different schemes: Adaptive, Random, and Greedy. The obtained results based on a large dataset and four state-of-the-art DNN architectures demonstrate that the adaptive solution outperforms other schemes (Random and Greedy) in terms of computation time and communication latency.

1.2 Structure of the thesis

The rest of the thesis is organized as follows. Chapter 2 overviews artificial intelligence and machine learning, then presents different DNN architectures in some detail. Chapter 3 discusses the most important research targeting DNN inference acceleration. Chapter 4 details the implementation of the offloading framework based on matching theory proposed by Mohammed et al. [25]. Chapter 5 evaluates the related adaptive partitioning approach against the other two strategies, namely, random partitioning and greedy offloading. Finally, Chapter 6 concludes the thesis and describes possible future research.

Chapter 2

Deep neural networks (DNNs)

A deep neural network (DNN) is a member of a broad family in artificial intelligence; especially, it is considered a sub-field of machine learning. It is a computational model inspired by the biological structure of the brain cells, which consists of interconnected nodes (neurons) that can learn from experience by modifying their connections and parameters. DNN is massively used in different application areas, such as speech recognition, natural language processing, computer vision, image, and video surveillance. With the help of DNN, AI can be trained by using supervised and unsupervised learning data and predict results.

The rest of this chapter describes the overview, the DNN architectural components, and the categories of DNN architectures in general.

2.1 Overview

Since an early age, computer scientists have been deriving their thoughts to design and build such a machine that would act identical to humans. Alan Turing, the father of theoretical computer science, developed a Turing test, which could evaluate the intelligent behavior of a machine in comparison to humans. In 1956, for the first time, the term Artificial Intelligence (AI) was introduced by John McCarthy.

Artificial intelligence refers to such a computer system that can perceive and learn knowledge, make reasoning and decisions based on the acquired knowledge, and act accordingly in an uncertain and complicated environment. There are two categories of AI in the modern era: the Narrow AI/Weak AI and the General AI/Strong AI. The narrow AI has the ability to handle only one sophisticated task. In contrast, several narrow AI can be combined to achieve a wide range of goals in the general AI. For example, in a self-

driving car, a trained object recognition system can only classify classes of objects based on the observation which refers to narrow AI approach, in contrast, if the system can detect an object with the best accuracy, process voice commands and adopt in any complicated situations at the same time indicates as general AI.

The first intelligent machine-building approach is known as symbolic AI. This approach works manually based on human-readable symbols and a set of rules for manipulating the symbols. A symbol may represent in a character string to describe the task in words. For example, a symbolic representation of a "Wooden chair" is given in Figure 2.1, inspired by [24].

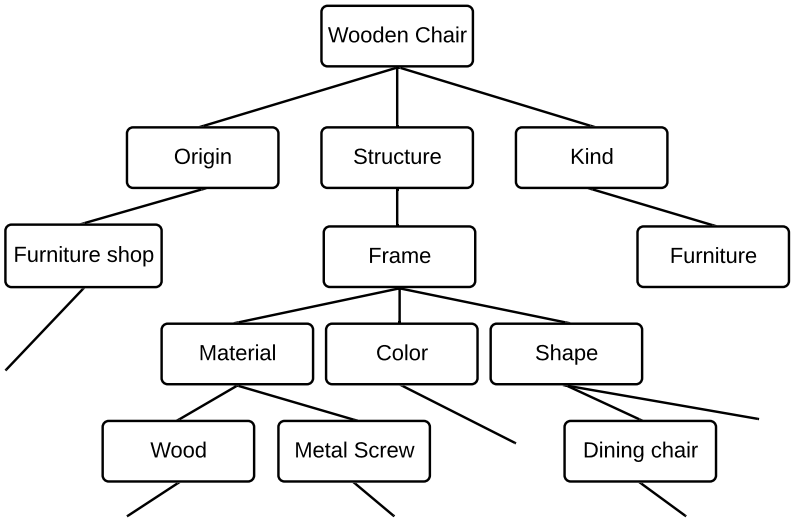

Figure 2.1: The symbolic representation of a wooden chair.

The manual input process of symbolic AI system is inappropriate for handling real-life problems. It also imposes a significant challenge to declare all the symbols and the rules to fit with real-world applications. To alleviate the problems, researches apply machine learning approach. Machine Learning (ML) is a statistical method that provides the system ability to learn directly from the data and improve from experience without human intervention. As the example mentioned earlier in the symbolic approach, if different categories of chair images are fed into the machine learning algorithm, the algorithm will be able to detect the chair and its category by learning features, such as edges, patterns, and textures.

The domain of machine learning can be divided into two major areas.

Supervised learning: In supervised learning, the training input data and the target output are known. The supervised learning algorithm learns from the input data by comparing the generated prediction result with the target output. It is an iterative learning process and achieves an acceptable level of performance after a certain iterations. Linear regression, Random forest, and Support vector are some popular supervised learning algorithms.

Unsupervised learning: In unsupervised learning, only the training data is available. The unsupervised learning algorithm learns by discovering the features of the input data and accumulates the learning into its knowledge domain. This method generates new input data based on the observation of the same set of data. K-means and Apriori are some practiced unsupervised learning algorithms.

With the advancement of computing capabilities and big data technologies, machine learning introduces a system architecture known as Artificial Neural Network (ANN). ANN was introduced in the late '40s, requiring a high computational power and access to a large dataset. In recent times, the powerful CPU and GPU, and the cloud infrastructures revive the ANN technology to take care of the complex AI tasks proficiently. A deep neural network is one of the kinds of artificial neural networks.

2.2 DNN architectures

The architectural components of a neural network are layers, parameters, hyperparameters, activation functions, loss functions, and optimization methods. The layers are constructed by neurons, which are known as units [29]. Units are a symbolic representation of biological neurons. Figure 2.2 illustrates a basic structure of a neural network.

Layers: A layer represents a collection of units that operates together at a specific depth of the neural network. There are two types of layers in a neural network, such as the input layer and the hidden layers. The output layer is categorized under the hidden layers. The input layer represents the input features that go through the units of hidden layers to generate prediction in the output layer. The hidden layers learn different aspects of the input data by breaking it down into different levels of abstraction. To understand the hidden layer's concept, let us consider an image recognition example of a four-wheel vehicle. In abstract terms, the first hidden layer is responsible for identifying pixels of light and dark, and the second hidden layer detects edges and simple shapes. Primarily, the hidden layers at the beginning of the DNN identify simple structural components, but the deeper hidden layers detect more complex structure of the image.

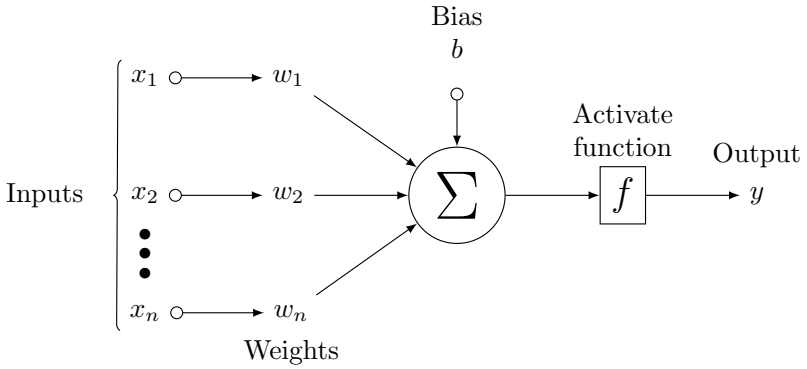

Figure 2.2: A basic structure of a neural network.

Parameters: Parameters in neural networks are configuration variables. They are subjective to the model and may vary based on different neural network architecture. The weights and the biases are such parameters which are used to form the network connection.

Hyperparameters: Hyperparameters are used to tune the neural network and configure settings to achieve effective performance. The size of the layer, learning rate, dropout, mini-batch size, number of epochs, and normalization of input data are included in the hyperparameters.

- **Layer size**: The layer size defines the number of units in a layer.
- **Learning rate**: The learning rate governs how much to adjust in the model based on the estimated error calculated during model training. A high learning rate may cause the convergence to sub-optimal solutions, whereas a low learning rate takes a more prolonged period for the training process to complete.
- **Dropout**: The dropout mechanism discards units from the hidden layers to improve the neural network training and to overcome the over-fitting problem. The over-fitting problem occurs when the trained model works efficiently with the training data (seen data) but works poorly with the testing data (unseen data).
- **Mini-batch size**: The mini-batch size refers to how many samples will propagate through the network at a time. The network trains faster with mini-batches as the model parameters are updated after each propagation. For example, let us assume the total sample size is

1280, and the mini-batch size is 128. The model parameters will be updated 10× for each propagation.

- **Number of epochs**: The number of epochs defines the number of times the complete data set propagates through the learning algorithm.
- **Normalization**: In vector representation of the input data, the data are scaled to a specified range, such as [0,1], [-1,1], which affects the activation in the neural network.

Activation functions: An activation function is used to restrict the output of a layer to a specific limit and add non-linearity in the neural network. The activation function helps the network to learn the complex pattern in the data set. Some commonly used activation functions are `sigmoid`, `tanh` and `Rectified Linear Unit (ReLU)`.

Loss functions: Loss functions assess the compliance between the predicted output and the supervised learning output. The loss functions determine the misclassification cost. Based on the calculated loss, model parameters are updated accordingly through the backward propagation method. Some commonly used loss functions are `Mean Squared Error` (Regression problem) and `Cross-Entropy Loss` (binary and multi-class classification problem).

Optimization methods: Optimization methods help to minimize loss functions by updating the parameter values through optimization algorithms. These methods aim to find the best set of parameter values, in other words, a trained model that gives minimal loss. The optimization algorithms are categorized into two types, such as first-order optimization algorithms (e.g., `Gradient Descent`, `Stochastic Gradient Descent`) and second-order optimization algorithms (e.g., `Newton method`, `Conjugate Gradient`). The most widely used first-order optimization algorithm is `Adam`, which computes the adaptive learning rate for each parameter [19].

A historical evolution of DNN architectures is shown in Figure 2.3. According to Patterson et al. [29], there are four major DNN architectures in practice.

1. Unsupervised Pretrained Networks (UPNs)
2. Convolutional Neural Networks (CNNs)
3. Recurrent Neural Networks (RNNs)
4. Recursive Neural Networks (RNNs)

Figure 2.3: The Evolution of DNN architectures.

2.3 Unsupervised Pretrained Networks (UPNs)

Unsupervised pretraining helps to extract features from an unlabeled data set. There are many architectures used for unsupervised pretraining, and the followings are the most popular neural networks.

1. Restricted Boltzmann Machines (RBMs)
2. Autoencoders
3. Deep Belief Networks (DBNs)
4. Generative Adversarial Networks (GANs)

2.3.1 Restricted Boltzmann Machines (RBMs)

Restricted Boltzmann Machines (RBMs) are two-layered neural networks widely used for dimensionality reduction (feature selection and extraction), classification, regression, collaborative filtering, and topic modeling. They can learn a probability distribution over its input dataset. A fully bipartite graph connects both layers (visible and hidden layer) of RBMs. Units of the same layer are not connected, that makes them restricted and different from Boltzmann Machine.

In RBMs, there are two additional bias layers (visible bias and hidden bias), which makes it different than Autoencoders. The hidden and the visible bias layers are used in the forward pass and reconstruction phase, respectively. Figure 2.4 illustrates the forward pass and the reconstruction phase of RBMs. The RBMs construct error by subtracting the inputs from the reconstructed inputs and tune the model parameters (e.g., weights) accordingly. Due to the behavior of the RBMs, they are known as generative models as they generate new samples of data from the same distribution.

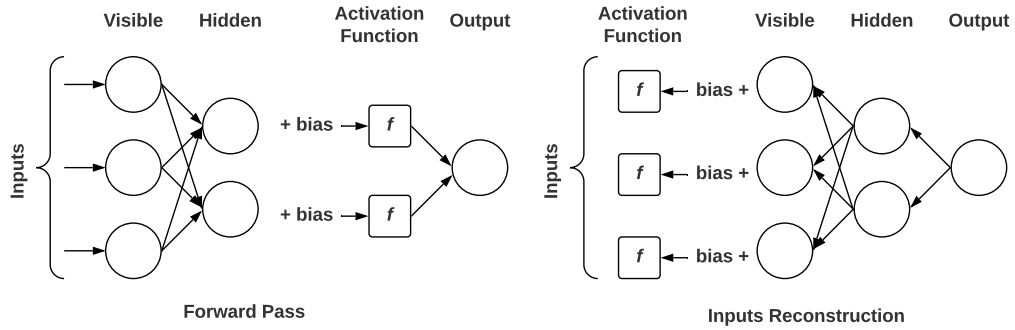

Figure 2.4: RBMs forward pass and reconstruction phase.

2.3.2 Autoencoders

Autoencoders are used to learn an efficient way of compressing and encoding data to construct new data, closer to the original input. They reduce data dimensions by ignoring noise in the data. Autoencoders are constructed by four significant parts: encoder, bottleneck (hidden layers), decoder, and reconstruction loss. Figure 2.5 shows a basic structure of autoencoders. The bottlenecks are the autoencoders' key feature, which constrains a compressed knowledge representation of the input.

Autoencoders are almost similar to multi-layer perceptron with some differences. One of the differences is that both the input and the output layers of autoencoders have the same number of units. Autoencoders use unlabeled data in unsupervised learning and build a compressed representation of inputs with bottlenecks. Variational autoencoders, sparse autoencoders, denoising autoencoders, and contractive autoencoders are some autoencoder variants widely used in practice. The application areas of autoencoders are anomaly detection and image denoising.

2.3.3 Deep Belief Networks (DBNs)

Deep Belief Networks (DBNs) are probabilistic unsupervised learning algorithms. They are constructed by the stack of Restricted Boltzmann Machines (RBMs), consisting of multiple hidden and visible layers of stochastic and latent variables. Unlike other models, each layer of DBNs knows the entire input data. The process through which DBNs learn is a greedy approach. In a greedy learning algorithm, the optimal choice is made at each step of the process, which leads to the global optimum. Figure 2.6 shows a basic architecture of DBNs.

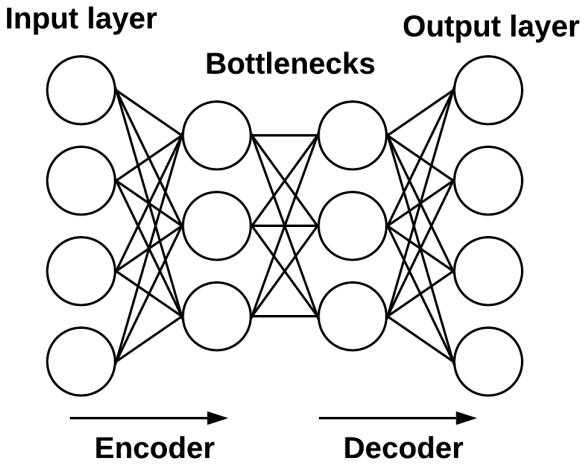

Figure 2.5: A basic structure of autoencoders.

The DBN's overall process can be categorized into two phases, such as the pre-train phase and the fine-tuning phase. Learning features in an unsupervised fashion is considered as the pre-train phase. The RBMs learn higher-level features from the distribution of data through each layer progressively, which eventually use the learned features as input to the higher level. When the learned features are used as initial model parameters in a feed-forward network; this phase is called the fine-tuning phase. In the feed-forward networks, backpropagation is the key for tuning the model parameters. Normal backpropagation is used with a low learning rate to find the best value for model parameters.

2.3.4 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) [9] are a combination of generative and discriminative learning algorithms. The generative algorithm tries to generate sample inputs for the discriminative algorithm to classify whether the data is populated from the generator or the training data set. GANs are used to build efficient classifiers by generating sample images and videos. Moreover, they are also used to create fake media content, such as Deepfakes. Figure 2.7 illustrates a visual overview of the GANs architecture.

The GANs are formed by two neural networks, such as a generator and a discriminator, and simultaneously train two models. The generator produces synthetic sample data from a random noise and trains the model to provide more realistic sample data, which are fed through the discrimina-

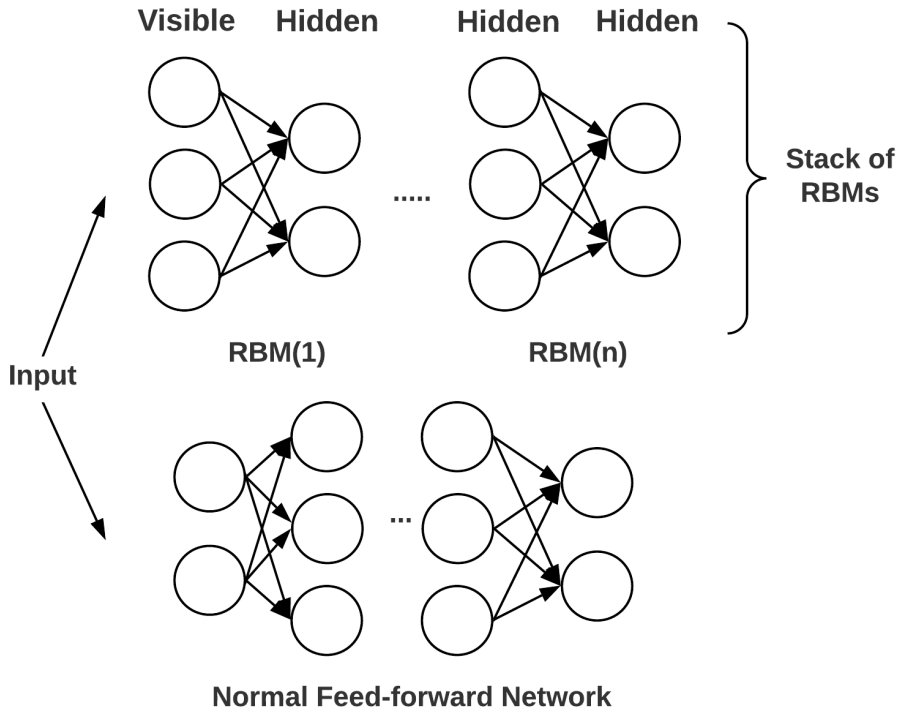

Figure 2.6: A basic architecture of DBNs.

tor to determine whether the data is real or synthetic. The discriminators are usually standard Convolutional Neural Networks (CNNs). The generators use deconvolutional networks to generate synthetic sample images. The gradient of the output of discriminators helps the generator to produce more realistic data by making small changes. In both networks, backpropagation is used to train the model efficiently. GAN is considered a recursive process of generating more realistic data and a useful classifier that can detect the difference between synthetic and training data.

2.4 Convolutional Neural Networks (CNNs)

Convolutional Neural Network (CNN), also known as ConvNet, is a class of neural networks where a linear convolution operates instead of matrix multiplication in network layers. This architecture has taken inspiration from biological processes, especially the connectivity pattern of the visual cortex neurons. Some regions of visual cortex cells trigger to a specific pattern of

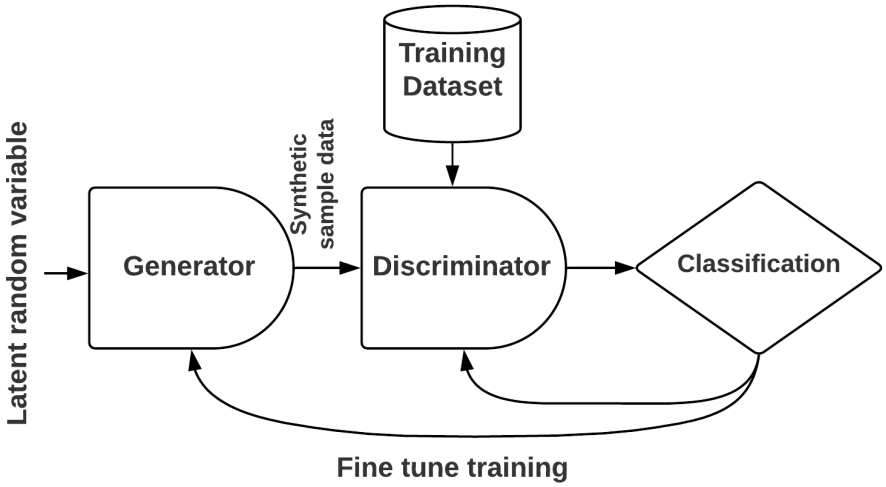

Figure 2.7: A visual overview of GANs architecture.

visual fields such as edges, color, and curves. All the sensitive neurons are organized in a columnar fashion to produce visual perception and try to locate specific characteristics of the observed object [17]. The basic progression of the CNN is similar to the visual cortex working process. The CNN learns higher-order features of data through convolution operation. First, it starts by classifying the lower-level features, and more sophisticated features are analyzed as the network progresses. Such architecture is useful for object recognition in images and videos, translating natural language and sentiments.

The goal of CNN is to transform input data starting from the input layer through intermediate connected layers into a set of classes. The output layer shows the difference of the input images. The high-level overview of the CNN architecture is as follows.

- Input layer
- Learning layer (feature extraction)
- Classification layers (fully-connected)

The learning layer or the feature extraction layer executes a repetitive pattern of sequence:

- Convolution layer (linear operation)
- Activation function (e.g. `ReLU`, `sigmoid`, `tanh`)
- Pooling layer

The first layer in the CNN is always a convolutional layer. The input is a tensor of shape [number of images/batch] × [width of image] × [height of image] × [depth of image (channels)]. A feature identifier, known as a filter/a kernel, convolves over the receptive field from left to right through the input images to extract lower-level features and generate an activation map or a feature map. The receptive field refers to a region in the input data that stimulates when the filter/the kernel is applied to extract features, and the feature map represents the result of the extraction. The depth of the kernel must be kept the same as the depth of input. A visual illustration is given in Figure 2.8. In the series of the feature-extraction layers, the output of one convolutional layer interprets as the input to the next convolutional layer. As it progresses, more complex features are extracted. Zeiler et al. [40] represents a visualization of intermediate features and the classifier's operation in a CNN. The following formula provides the output size of a given convolutional layer. Here, W = input volume size, K = kernel size, P = padding and S = stride.

$$output = \frac{W - K - 2P}{S} + 1$$

During the design consideration of CNN, some of the hyperparameters need to be considered, such as the filter/kernel size, the stride, and the padding. The stride represents the number of shifts while the kernel is sliding through the input images. It controls the sliding movement of the kernel over input volume. The spatial dimension decreases as the convolution operation progresses. Zero padding is added to the image's border to preserve more information of the original input volume to detect lower-level features. The choice of an appropriate hyperparameter largely depends on the type of training dataset. Figure 2.9 visualizes the stride and the padding of the convolution layer.

After each convolution layer, the activation function is applied to introduce non-linearity. In recent years, the `Rectified Linear Unit (ReLU)` becomes very popular due to its faster training efficiency and capability to alleviate the vanishing gradient decent problem. The vanishing gradient problem occurs during back-propagation phase of network training. In the back-propagation algorithm, the gradient is calculated based on the loss function

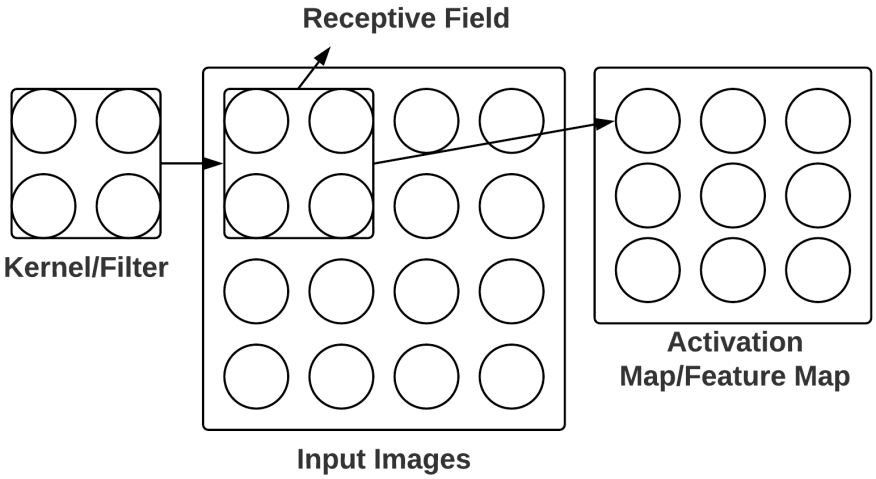

Figure 2.8: A visual illustration of CNN convolution operation.

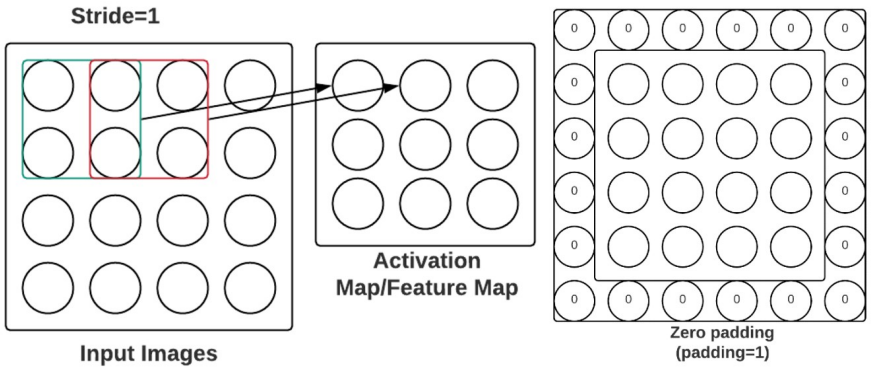

Figure 2.9: A visual illustration of stride and padding in CNN convolution operation.

and it becomes infinitely smaller in a deep network. Thus, the performance gets saturated and degrades rapidly. The `ReLU` also improves the performance of neural network [26]. It applies the function $f(x) = \max(0, x)$ to all the values of the convolution layer output. Some other non-linear activation functions, such as `sigmoid` ($\sigma(x) = 1/(1 - e^{-x})$), `tanh` ($\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$) are popular as well in neural network.

Depending on the structure and the type of dataset, after the activation layer, the pooling layer is applied. It is also known as non-linear down-sampling. The purpose of pooling layer is two-fold. First, reducing the compu-

tation cost by compressing the number of model parameters, such as weights and second, controlling overfitting problem. In the overfitting problem, the model is trained in such a way that it learns the details and the noise of the training data, which impacts negatively on the performance of the model on new data. There are different options for pooling to choose, such as max pooling, average pooling, L2-norm pooling. Figure 2.10 shows how max-pooling works in convolution neural network. Dropout [35] is another way to reduce overfitting. Units of CNN layers deactivate randomly in dropout phase. Dropout is applied mostly in fully connected layers of CNNs.

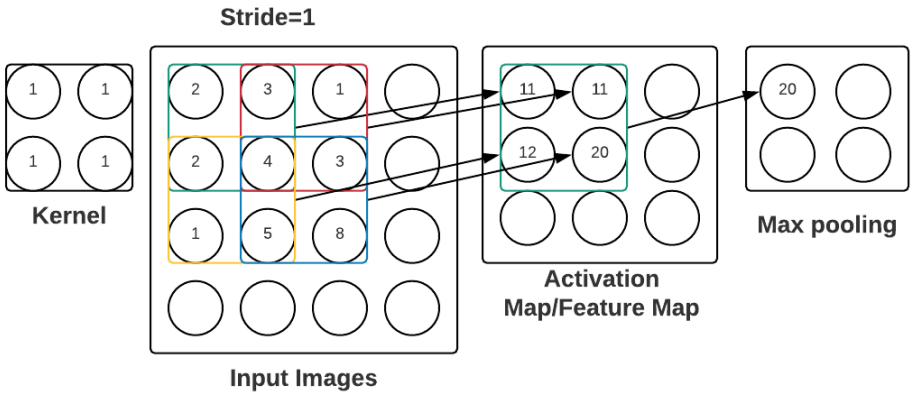

Figure 2.10: Example of max-pooling in CNN.

After several convolutional and pooling layers, the network introduces fully-connected layers to co-relate the higher-level features to a particular class, which provides the outcome of the network. Fully-connected layers take the output of the last convolutional or the pooling layer as input and output N-dimensional vectors. N-dimensional vector represents the number of classes/categories that can be identified from the data. Figure 2.11 illustrates a complete overview of CNN.

Many state-of-the-art convolutional neural networks are introduced by AI researchers, which can classify and predict objects from a set of images and video streaming. For DNN inference, data are trained by different image classification architectures, such as AlexNet [20], ResNet [14], VGG [32], and NiN [22] in this thesis. These architectures are also used for predictions and classifications. This section will briefly explain the benchmark CNNs for object recognition and image classification tasks.

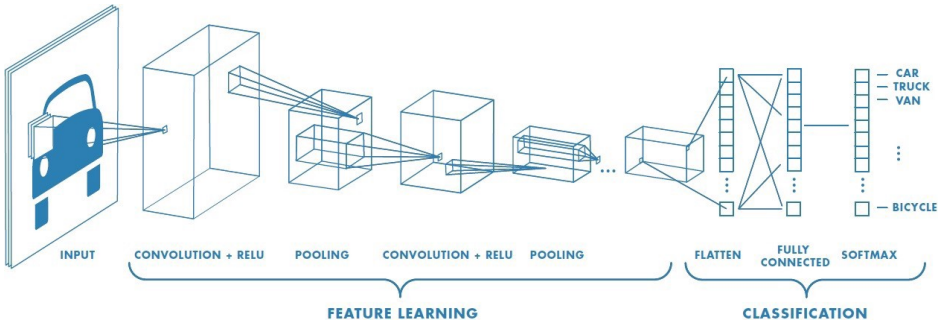

Figure 2.11: A complete overview of convolutional neural network.

2.4.1 AlexNet

One of the most notable publications in the field of computer vision is "ImageNet Classification with Deep Convolutional Networks" [20]. Authors have proposed a deep convolutional neural network that can classify images with the top-5 test error rate of 15.3%. AlexNet can classify images in 1,000 classes.

AlexNet has eight learned layers composed of five convolutional layers and three fully-connected layers.

- The first convolutional layer takes $224 \times 224 \times 3$ as input where 224 represents the dimension of the image, and three represents the channel (e.g., RGB). The input is filtered with 96 kernels of size $11 \times 11 \times 3$ with a stride of 4 pixels. The following convolutional layer filters the output of the former convolutional layer. The second convolutional layer filters with 256 kernels of size $5 \times 5 \times 48$.

- The last three convolutional layers do not have any pooling or normalization layers in between and are connected.

- The third, fourth, and the last convolutional layers filters with 384 kernels of size $3 \times 3 \times 256$, 384 kernels of size $3 \times 3 \times 192$ and 256 kernels of size $3 \times 3 \times 192$, respectively.

- Each fully-connected layer has 4096 neurons.

Instead of `sigmoid` and `tanh` activation functions, AlexNet uses the `ReLU`. The `ReLU` provides several times faster training time compare to `sigmoid` and `tanh` activation function [26]. Krizhevsky et al. [20] have used two Nvidia GTX 580 GPUs to train the model. One of the advantages of modern GPU is cross-parallelism. GPU can read and write each other's memory directly without going through the host memory. With the help of cross-parallelism,

the authors have put half of the kernels to each GPU, and it has been programmed in such a way that the layer n will take input from the layer $n-1$, which resides in the same GPU. AlexNet has also introduced the local response normalization technique, which has reduced top-1 and top-5 error rates by 1.4% and 1.2%, respectively. Dropout helps to avoid overfitting while training. Figure 2.12 illustrates the AlexNet architecture.

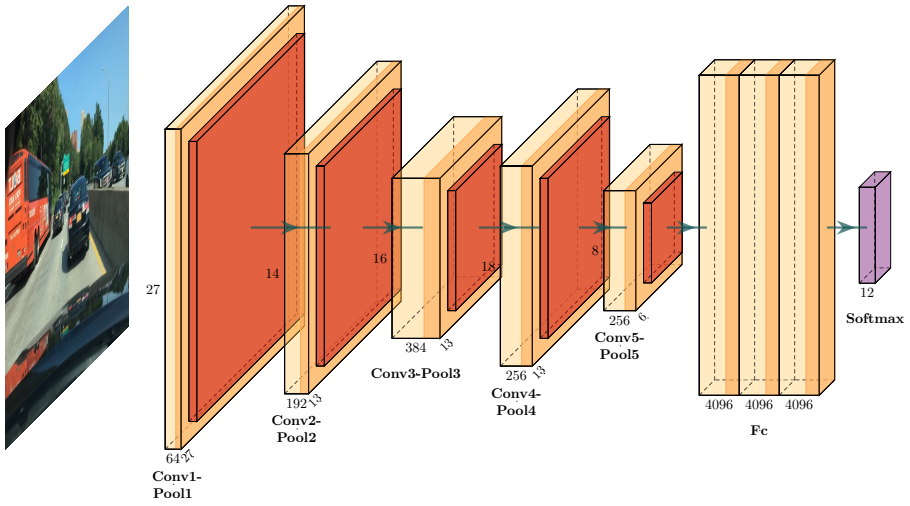

Figure 2.12: The AlexNet architecture.

2.4.2 VGG

Imagenet Large Scale Visual Recognition Challenge (ILSVRC) [31] introduces many state-of-the-art deep neural network architectures that have specific characteristics of their own to improve recognition task and result more accurately than the past achievements. After the immense success of AlexNet, many enthusiastic people have attempted to enhance the original architecture. VGG [32] is one of the improved architectures. The work has concentrated more on the depth of CNN architecture and increased the depth by adding more convolutional layers. A small convolution filter of 3×3 is used in all layers.

VGG has five CNN configurations that follow the generic design concept, only differs in-depth (number of layers). Table 2.1 illustrates different configuration strategies of VGG.

- A fixed-size 224×224 RGB image is fed through the VGG network.

CNN Configurations				
A	B	C	D	E
11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
Input (224 × 224 RGB image)				
conv3-64	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool				
conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool				
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool				
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool				
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool				
FC-4096				
FC-4096				
FC-1000				
soft-max				

Table 2.1: The VGG CNN configurations. Source: [32]

- A filter size of 3 × 3 is used throughout the network except for one configuration where 1 × 1 filter size is used for linear transformation of the input channel. The padding and stride are considered as 1 pixel for all the convolutional layers.
- Five max-pooling layers are applied after the convolutional layers. Each max-pooling is performed over a filter of 2 × 2 and a stride of 2.

- After the pile of convolutional layers, three fully-connected layers are applied. The first two layers have 4096 channels, and the last layer has 1000 channels, representing 1000 classes for recognition.
- After each convolution layer, non-linearity (e.g., `ReLU`) is applied.

It is worth noting that a stack of two and three convolution layers produce an effective receptive field of 5×5 and 7×7, respectively. This design choice has some benefits. First, the VGG approach has made the decision function more discriminative by applying more non-linearity. For instance, the first convolutional layer has a filter-size of 7×7, followed by a non-linearity in AlexNet. In contrast, a stack of three convolutional layers followed by non-linearity after each layer is applied in VGG to achieve the same receptive field. Second, the number of model parameters is decreased by 44%. The convention of using a small-size filter to do classification tasks is used previously, but due to less depth, the result is not efficient for large scale image datasets. Goodfellow et al. [10] have applied a deep CNNs to recognize street numbers and showed that depth has a real effect on efficiency and accuracy, and increased depth led to significant performance. Figure 2.13 illustrates the VGG16 architecture.

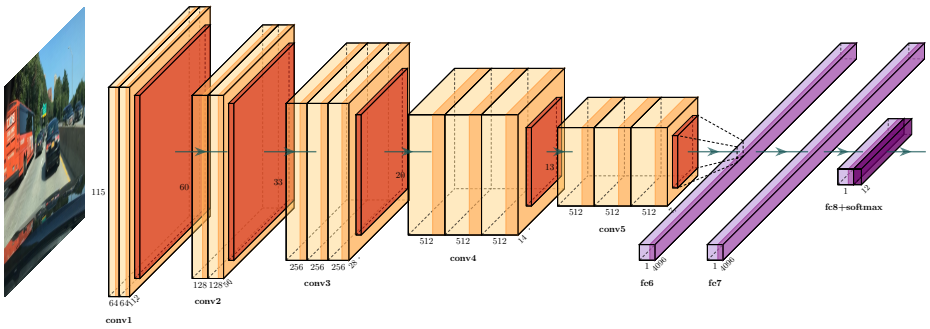

Figure 2.13: The VGG16 architecture.

2.4.3 ResNet

As the neural network goes in-depth, the network training becomes problematic. This is due to the vanishing gradient problem. Moreover, the model parameters increase in deep networks. Tuning a higher number of model parameters may forge the network to train with a higher error rate. Microsoft

Residual Network (ResNet) [14] has addressed these problems and presents a residual learning framework. It is a deep neural network consisting of 152 layers, which is eight times larger than VGG [32]. It has won the first prize in ILSVRC with a 3.57% error rate on the imagenet data set.

The authors have introduced a residual mapping technique to overcome the vanishing gradient problem. Figure 2.14 illustrates residual learning building blocks.

- In traditional convention, input x goes through a stack of convolutional layers and output a function $H(x)$. In ResNet, the original input x is added with the output function's result denoted as $F(x)$ instead of the straight transformation. The way the original input is carried to the output is called an identity shortcut connection. Identity shortcut connection does not affect on the network by adding extra model parameters and computational complexity.

$$H(x) = F(x) + x$$

- The identity shortcut connection has been used in two different ways. If the input and the output are of the same dimension, the input is added directly with the output. When the dimension increases, authors have considered two options, i.e., identity mapping with extra zero entries padded for increasing dimensions and a 1×1 convolutional layer. In both cases, the stride of 2 is used due to the dissimilarity of dimensions.

Residual learning eases the training process during the backward pass of backpropagation, which helps the gradient flow easily through the graphs by distributing it. According to the authors, optimizing the residual mapping is more relaxed than the unreferenced mapping [14].

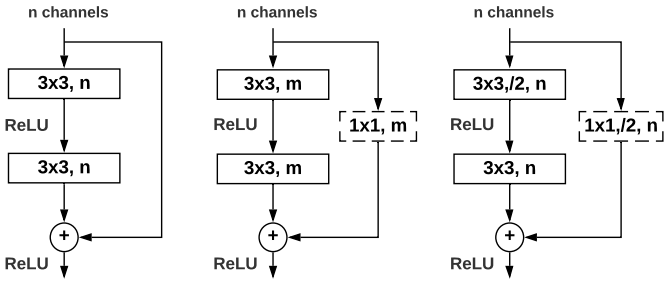

Figure 2.14: ResNet residual learning building block.

2.4.4 Network-in-Network

Network-in-Network (NiN) [22] is a novel deep network structure that replaces the conventional generalized linear model (GLM) by a universal function approximator to enhance the model segregation for local patches within the receptive field. In the standard convolutional network, the feature map is produced by the execution of linear layers followed by a non-linear activation function. The convolution filter used in CNN is a generalized linear model (GLM) with a lower abstraction level. Replacing GLM by non-linear function approximator can enhance the level of abstraction.

In NiN, a multilayer perceptron (MLP) is used as a non-linear function approximator. The decision is made based on two reasons. First, the MLP is compatible with a convolutional neural network which is trained by back-propagation techniques. Second, the MLP itself is considered as a deep model.

NiN is organized by several stacks of MLP convolutional layers followed by a global average pooling layers. It does not have any fully-connected layer at the end of the convolution layers. Instead, the spatial average of the feature map results in the confidence of categories . The global average pooling does the process. A softmax function is applied to the resulting vector after the pooling. Figure 2.15 represents a micro-network (MLP) based NiN structure, which has three MLP convolutional layers and one global average pooling layer.

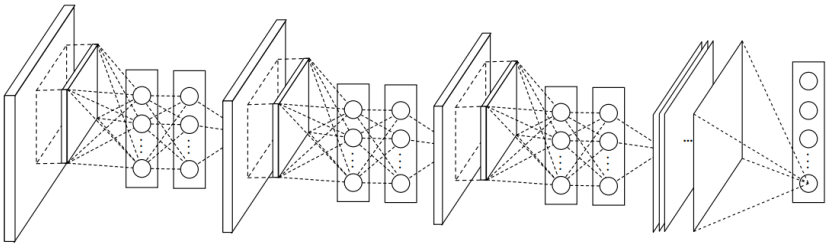

Figure 2.15: The overall structure of Network-in-Network. Source: [22]

2.5 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks are considered a class of neural networks which process data in a sequential order. The word recurrent means the same process is executed for each element of a sequence, and the output depends on the previous computation. Some researchers interpret the dependency process of RNNs as memory where information from the previous calculation

is captured for further processing. A general architecture of the recurrent neural network is given in Figure 2.16.

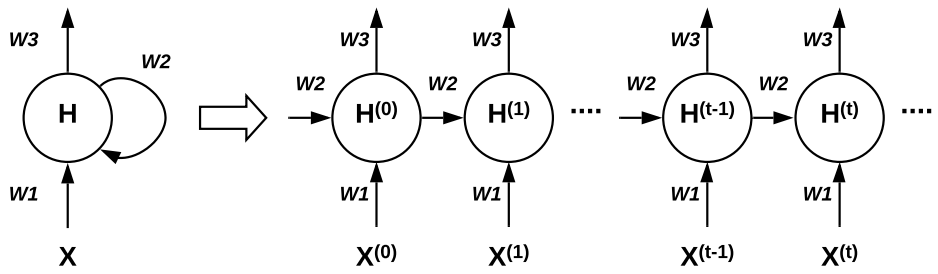

Figure 2.16: A general architecture of recurrent neural network. Source: [21]

Figure 2.16 represents an unrolled/unfolded recurrent neural network. For example, if the data represents a sentence and the sentence has five words, then the network will be unrolled into 5-layers, one layer for each word. The internal computation of the recurrent neural network works as follows:

- $W1$, $W2$, and $W3$ represent the model parameters or weights of the network. An interesting factor to notice is that all the layers share the same parameters across all steps that reduce the total number of model parameters to be learned.

- $X^{(t)}$ is the input at time step t. $H^{(t)}$ is the hidden state at time step t. $H^{(t)}$ is calculated based the following formula $H^{(t)} = f(X^{(t)} * W1 + H^{(t-1)} * W2)$. Here, function represents the activation function (i.e., `sigmoid`, `tanh` and `ReLU`). The hidden state interprets as the memory of the network. The first hidden state $H^{(-1)}$ is set to all zeros.

- $Y^{(t)}$ is the output of corresponding input $X^{(t)}$ at time step t. The output at each time step depends on the application domain. It is not mandatory to calculate output at each time step. For instance, predicting sentiments of a sentence.

The training process of recurrent neural networks is almost similar to the neural network with a slight difference. The backpropagation is called Backpropagation Through Time (BPTT) in RNNs as the gradient at each output depends not only on the calculations of t time step but also on the $t-1$ time step.

The general recurrent neural networks, which are also known as vanilla recurrent neural networks, face the vanishing gradient problem. The gradient is used to adjust the weights of the network. The adjustment depends on how the value of the gradient evolves. If the amount of the gradient is large, the impact of the weight adjustment is also significant, and if the value of the gradient decreases, it affects the weight adjustment. As the BPTT progresses, the gradient drastically shrinks, and the layers at the beginning of the network fail to learn as the weights are slightly adjusted due to a small gradient. This problem is called the vanishing gradient problem, and the network suffers from short-term memory.

There are two particular kinds of recurrent neural networks in practice, such as Long Short Term Memory networks [15] and Gated Recurrent Units [4] to mitigate the short-term memory of recurrent neural networks.

2.5.1 Long Short Term Memory (LSTM)

Long Short Term Memory (LSTM) is a special kind of recurrent neural network capable of learning long term dependencies. It has control flow like vanilla recurrent neural network with some additional functionalities, which helps to keep and remove information from the sequence data. LSTM also forms the chain of repeating modules similar to the vanilla recurrent neural networks. The illustration of LSTM building module is shown in Figure 2.17

- The horizontal line at the top of the figure is known as the cell state, which is the core component for ensuring long-term memory. LSTM can add new information and remove unnecessary old information in the cell state through the regulation of some gates.

- There are three gates responsible for protecting and controlling the cell state. The `sigmoid` function governs all the gates. The `sigmoid` function provides output between 0 and 1. If the output of the `sigmoid` function is close to zero, the information can be forgotten or removed. If the output is nearly 1, the information should be kept.

- The first gate is a forget gate. This gate decides what information should be kept or removed. The red color circle indicates the forget gate. The input of the current state X_t at time step t, and information from the previous hidden state H_{t-1} are merged and fed through the forget gate. The output of the forget gate multiplies with the cell state to preserve valid information.

- The second gate is an input gate, which stores new information in the cell state. The input gate decides which value to update. The green color circle represents the input gate. $H_{t-1} + X_t$ are passed through a `tanh` function, which squishes the value between -1 to 1 to regulate the network. The input gate decides which information should be kept from the `tanh` function output and added to the cell state. Thus, the new information is added in the cell state.

- The third and the last gate is output gate, which generates the next hidden state information. The blue color circle denotes the output gate. Similar to the input gate, $H_{t-1} + X_t$ are passed through an output gate, and the newly modified cell state is passed through a `tanh` function. Both outputs are multiplied to decide which information should be carried through the hidden state.

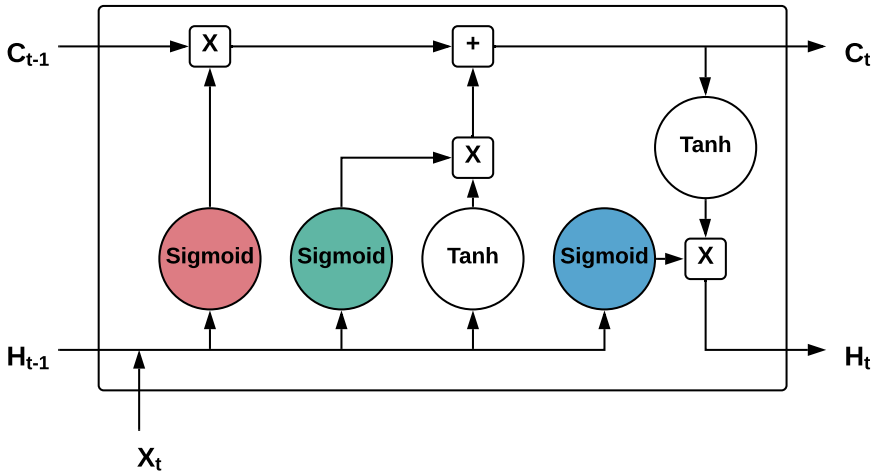

Figure 2.17: Building module of LSTM.

2.5.2 Gated Recurrent Units (GRUs)

Gated Recurrent Unit (GRU) aims to solve the vanishing gradient problem and work almost similar to the LSTM. GRU also has two gates that help coordinating the relevant information to pass through the chain of the GRU modules. The illustration of the GRU building module is shown in Figure 2.18.

- GRU has two gates, such as an update gate and a reset gate. The update gate helps to keep relevant information to pass through the GRU module, and the reset gate forgets irrelevant information to improve the prediction result. In Figure 2.18, green and red color circles indicate the update and the reset gate, respectively.

- The update gate uses the `sigmoid` function, which produces an output result between 0 and 1. Close to 0 means forget the information, and close to 1 indicates to keep the information. The output of the update gate is denoted as u_t. Here, W_{1u} and W_{2u} represents the weights of the input x_t and the weights of the previous module's information h_{t-1}, respectively. The equation of the update gate is given below.

$$u_t = \sigma(W_{1u}.x_t + W_{2u}.h_{t-1})$$

- The reset gate works similar to the update gate as it also has the same activation function. The output of the reset gate is denoted as r_t. Only the difference between the gates is corresponding weights and usages. The equation of the reset gate is given below.

$$r_t = \sigma(W_{1r}.x_t + W_{2r}.h_{t-1})$$

- The output of the reset gate introduces a candidate content that stores as relevant information from the previous hidden unit. The output of the reset (r_t) and the product of the previous hidden unit (h_{t-1}) and its weights (W_2) are element-wise multiplied to determine what to remove from the previous time step. The `tanh` function is used to generate the output of current memory content. For instance, consider a smartphone review sentiment analysis problem. Some reviewer wrote, "The X phone has all the common features. ... But the camera does not perform well, which I need most." Here, the most relevant part of the review is in the last sentence, which means the previous information are not relevant to determine the reviewer's satisfactory level. The general formula is given below.

$$h_t^{'} = \tanh\left(W_1.x_t + r_t * W_2.h_{t-1}\right)$$

- In the final step, the result of the update gate is used to determine what information should be kept from the candidate contents and the previous hidden unit. Let us consider the previous example. The last sentence holds the relevant information. At this time step, the most

relevant information is positioned at the beginning of the statement. So, the majority portions of the previous information should be kept and the candidate content should be ignored. The update gate helps to determine this by generating results close to 0 and 1. The formula for the hidden unit at time step t is given below.

$$h_t = u_t * h_{t-1} + (1 - u_t) * h_t^{'}$$

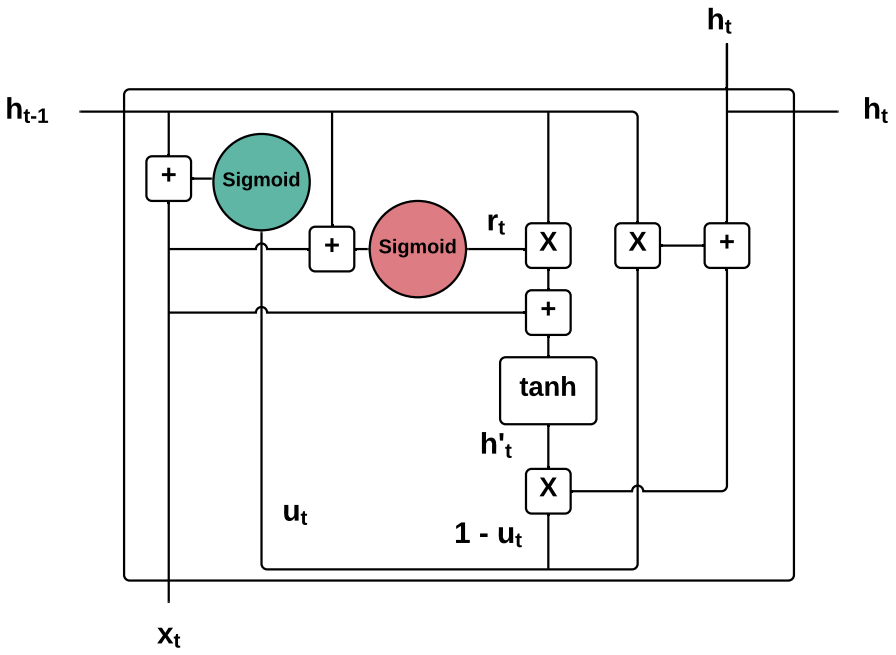

Figure 2.18: Building module of GRUs.

2.6 Recursive Neural Networks (RNNs)

Recursive neural networks are non-linear adaptive models, which learn and predict based on the deep structured data. In the early '90s, the neural networks were successful while processing fixed-length data and variable-length sequences. Still, they cannot apply efficiently on structured data, such as the logical terms, trees, and graphs. Christoph Goller et al. [8] first presented a novel approach that was capable of solving inductive inference problems on complex symbolic structures of variable size. Since then, the researchers

are working to represent and classify structured data with the help of neural networks. Sperduti et al. [34] have proposed an approach of generalized recursive neurons, where all the supervised sequence classification networks can be generalized into structures. The authors have presented a framework to solve the problem of processing structured information [6].

The principle motivation behind the recursive neural networks is to work with the information of different sizes and topologies. In contrast, the feature-based approach works with a fixed-size data. A backpropagation technique is used, known as Backpropagation Through Structure (BPTS) to train a recursive neural network. Recursive structures are found in natural scenes and natural language sentences [33]. The authors have proposed a structure predicting algorithm based on a recursive neural network that parses natural scenes, natural language sentences and predicts with appropriate class labels. Figure 2.19 illustrates the parsing process of natural scene and sentence in RNNs.

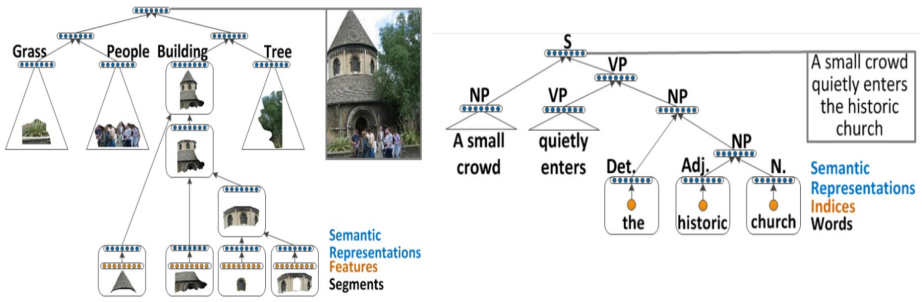

Figure 2.19: Parsing natural scene and sentence in RNNs. Source: [33]

Advantages of recursive neural networks

- Robust against the vanishing gradient problem.
- Efficiently learn hierarchical, structured and variable length data.

Disadvantages of recursive neural networks

- The hierarchical structure of every input data must be known while training.
- Difficult to work in mini-batches as the hierarchical structure changes for every training data.

Chapter 3

Distributed DNN inference

Intelligent applications enable by advanced artificial intelligence solutions have been deployed in places ranging from homes to far away satellites. In fact, machine learning techniques allow to obtain high accuracy and reliability in several tasks, especially deep neural networks, natural language processing, image and voice recognition, and computer vision. Self-driving vehicles, smart assistant applications are some of the contributions of advanced machine learning technologies. Deep neural networks (DNNs) require high computational resources to train and predict models. Today's cloud technologies provide extensive resources for running DNN tasks. Heavy computation of intelligent applications are taking advantage of this cloud facilities. The main disadvantage of the cloud-only approach lies in transmitting a significant amount of data through the wireless communication channel with high latency. With the advancement of the powerful System on Chip, the end devices are becoming resourceful and capable of executing a DNN task. Moreover, researchers are also focusing on offloading the DNN tasks or partitioning DNNs to the neighboring edge/fog devices, which are relatively more potent than the end devices for accelerating DNN computation by alleviating device-level computing cost and memory usage. Generally, DNN inference is offloaded to the edge. A trained network (model) is used to predict/infer the test samples to predict the output in DNN inference. The rest of this chapter reviews research works in DNN inference acceleration that are more relevant.

3.1 Distributed mobile computing system

Most DNN applications are performed on client-server architecture due to adequate computing capability required for executing DNN tasks. Mao et

al. [23] have proposed a local distributed mobile computing system for DNN (MoDNN). This first work has considered heterogeneous mobile devices in a local cluster communicating over Wi-Fi.

The article has described the methodologies as follows. First, the authors have built a heterogeneous mobile computing cluster, connecting through Wi-Fi direct, a standard for peer-to-peer wireless communication, which has a high transfer rate ($\sim$ 250 Mbps) compared to a cellular network. Second, two layer-aware partitioning schemes have been proposed. Most state-of-the-art DNN architectures are designed by the convolutional layers, followed by the fully-connected layers. The convolutional layers are computing-intensive, whereas the fully-connected layers consume excessive memory. Based on experiments, authors have observed that the convolutional layer's computing cost depends on the size of the input, and the number of weights affects the memory usages in case of the fully connected layers. They have proposed a Biased One-Dimensional Partition (BODP) based on computing capabilities of mobile devices for partitioning the convolutional layers. A weight partitioning algorithm is introduced to partition the fully-connected layers inspired by the spectral clustering techniques which cluster the network using the eigenvalues of the similarity matrix of the data. Finally, the authors have implemented a scheduler for each mobile device as a middleware to plan the overall execution process.

The implementation is accomplished on the MXNet deep learning framework. The authors have used the VGG16 pre-trained DNN model, which is a very popular image recognition algorithm. The evaluation report of MoDNN suggests that the execution time decreases significantly based on the increasing number of worker nodes (mobile devices). The execution time improves by 2.17–4.28 times when the number of worker nodes increases from two to four. Figure 3.1 overviews the distributed mobile computing system.

3.2 Neurosurgeon

Kang et al. [18] have investigated both cloud-only execution and computation partitioning methodology between the end devices and the cloud infrastructure. They have experimented with eight intelligent applications in the domain of computer vision and natural language processing. They have also designed a scheduler that automatically partitions the DNN computation to find the best computation time and communication latency.

In the cloud-only approach, the authors have offloaded DNN computation to the cloud infrastructure through a wireless communication medium. The Jetson TK1 mobile platform is used as an end device, and a powerful

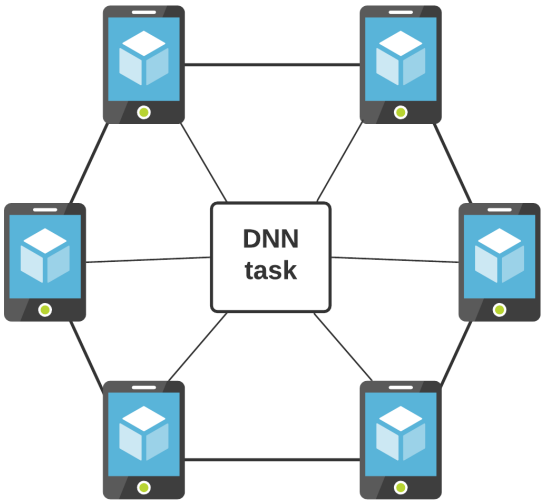

Figure 3.1: An overview of distributed mobile computing system.

computer equipped with a high performing GPU is used as a server. They have also experimented with DNN task execution in the mobile/end devices. The cloud processing has a higher computation impact than the mobile/end devices, but the communication latency is significantly high. Communication latency varies based on transmitted data size. As the end devices are getting smarter and resourceful every day, DNN task execution achieves significant improvement, but often leads to extreme energy consumption. The cloud-only processing provides significant computation efficiency over mobile processing and achieves better performance with a fast communication medium.

Fine-grained computation partitioning is another approach proposed by the authors. The DNN computation is partitioned between the end/mobile device and the cloud. They have analyzed the computation behavior of popular DNN architectures at the layer granularity. Most state-of-the-art DNN architectures are formed based on conventional layers, such as fully connected, convolutional and pooling. Some additional computations are performed to provide more accuracy, such as normalization, softmax, argmax, and dropout. Authors have partitioned the DNN architectures to execute some of the layers on the end/mobile platform and the rest on the cloud infrastructure. Based on the observations, the fully-connected layers are most costly with respect to computation time than other layers. It is also observed that the best partition point depends on the DNN architectures and application areas. For instance, a computer vision application provides the best

efficiency if the partitioning point is in the middle of the DNN, but in the case of speech recognition and natural language processing, partitioning at the beginning or the end is more suitable.

The best efficient point of partition depends mostly on the topology of the architecture. However, some other factors also affect the partitioning point, such as the state of the wireless medium and the cloud infrastructure load factor. The wireless medium has a variance, which affects the transmission latency. Therefore, the query service time gets increased due to the load pattern of the data center. Considering the above aspect, authors have proposed an intelligent DNN partitioning engine called Neurosurgeon, which selects the best partitioning point to optimize computation, communication latency, and energy consumption. Neurosurgeon consists of two phases, such as deployment phase and runtime phase. The deployment phase generates a prediction model by profiling the end/mobile devices and the server. The information is stored in the mobile devices for future prediction of latency and energy cost of each layer. In the runtime phase, four steps are performed. First, analyzing and extracting the DNN layer configurations. Second, predicting layer performance from the layer prediction model. Third, evaluating the partition point considering the wireless bandwidth and the server load, and finally, executing the DNN partition between the end/mobile platform and the server/cloud. Figure 3.2 illustrates the overview of the Neurosurgeon.

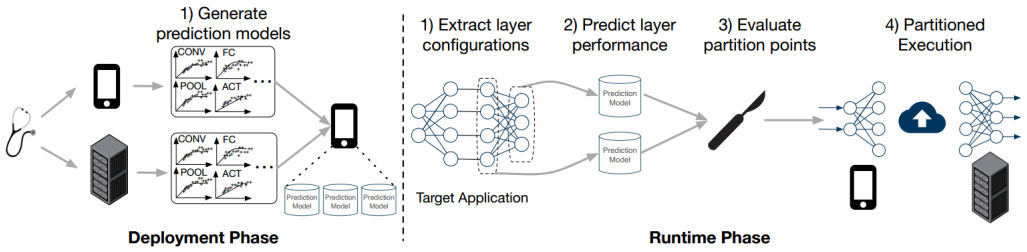

Figure 3.2: An overview of the Neurosurgeon. Source: [18]

3.3 DNN Surgery

Hu et al. [16] have proposed a framework, Dynamic Adaptive DNN Surgery (DADS), that supports partitioning the DNN architectures between the edge and the cloud.

While designing the DNN Surgery, authors have given importance to two aspects. First, the dynamic behavior of communication networks, and sec-

ond, the structural behavior of the DNN architecture. The partition decision depends on the network conditions. During peak hours, the throughput decreases due to the massive amount of traffic in the LTE (4G) network. On the contrary, the computation latency increases under high throughput conditions. The best possible cut decision needs to be ensured considering both the scenarios. In the case of peak hours, the layer with a smaller data size than the input is considered as partition point. The DNN computation is partitioned at the input layer during the higher network capacity period. Another concerning issue is that recent DNN architectures are based on directed acyclic graphs (DAGs) topology instead of chain topology. Partitioning the DAG topology needs more graph-theoretic analysis than the chain topology leading to the NP-hard problem. To mitigate both the network and the topology problem, authors have proposed two adaptive dynamic schemes, such as DNN Surgery Light (DSL) and DNN Surgery Heavy (DSH). The proposed DADS continuously monitors the network condition and decides which scheme is appropriate for the current state. If the network is in a lightly loaded condition, the DSL is applied; otherwise, the DHL is suitable for heavily loaded conditions. The DSL's primary goal is to minimize the overall delay to process one frame, and the DHL aims to maximize the throughput. To reduce the overall delay, the problem is converted into a min-cut problem to find the globally optimal solution. Figure 3.3 shows the conversion of the min-cut problem. An approximation approach is applied to solve the NP-hard problem, which is not solvable in polynomial computational complexity.

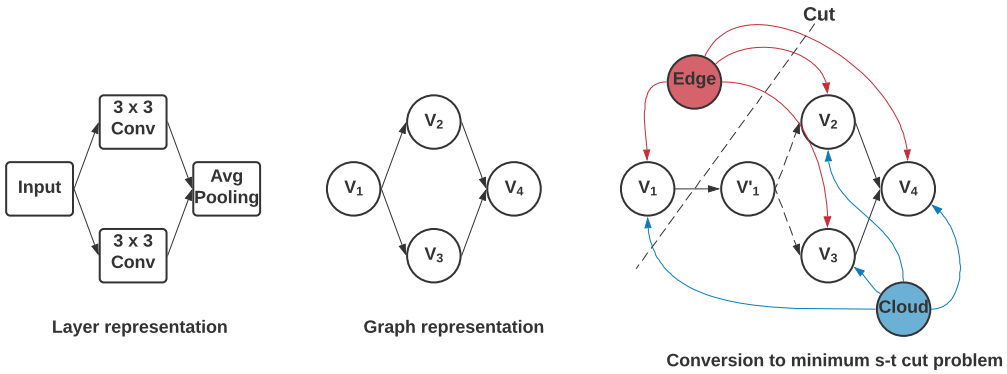

Figure 3.3: The conversion to the min-cut problem. Source: [16]

Authors use Raspberry pi 3 model B as the edge device and Cloud Ali as the cloud. The client-server interface is implemented using the gRPC protocol. The edge device extracts the sample frame from the self-driving car

video data set. The edge also makes the partition decision based on the network condition and accordingly processes the allocated layers. The partition decision and the intermediate results are transmitted to the cloud after the execution phase at the edge. The cloud receives the partition decision and executes the rest of the computation. The DNN surgery is compared with the edge-only and the cloud-only approaches. This technique has a computation latency speedup of 6 and 8 times (approx.) compared with the edge-only and the cloud-only approaches, respectively. In terms of throughput gain, it has a speedup of 8 and 11 times (approx.) compared with the other two approaches. The work also compares with Neurosurgeon [18]. The DNN surgery and the Neurosurgeon perform equally in the chain topology, but the DNN surgery outperforms the Neurosurgeon in heavy workload condition. The overall argument of this research suggests that if the network provides a high bandwidth and a high data rate, it is advisable to offload the large portion of the DNN computation to the cloud for better performance. The edge-only approach is feasible only in low data rate conditions.

3.4 Distributed deep neural network

Distributed Deep Neural Network (DDNN) [38] is a framework over distributed computing hierarchy, which consists of the distributed end devices (local network), the edge, and the cloud. A single end-to-end DNN model is jointly trained by the framework over three-tier (i.e., a local network, the edge, and the cloud) for fast and efficient localized inference with better prediction confidence.

In conventional distributed DNN computation, a small neural network (NN) model is executed in the end devices for initial feature extraction, and a large NN model is executed in the cloud for more robust feature extraction and classification. These approaches need to encounter some challenges, such as limited computation capability of resource-constrained devices due to insufficient memory and low battery power, aggregation of sensor data from multiple end devices for a single DNN task execution, and joint training of the models on the end devices, the edge, and the cloud. Teerapittayanon et al. [38] have proposed a framework that is capable of mapping sections of a DNN over distributed computing hierarchy, training a DNN model in a distributed manner, and allowing automatic sensor fusion through aggregation schemes. The DDNN framework has exit points defining the sample classification at earlier points in a NN. In the distributed environment, the DDNN framework has three exit points, e.g., local exit, edge exit, and cloud exit. When a classification task achieves a certain level of prediction confidence on

the end devices, the process exits from the local exit point without proceeding to the edge or the cloud (see Figure 3.4). This application also works with multiple geographically distributed end devices, and edge networks, which are aggregated together for performing classification tasks. Authors have presented three approaches for aggregating the output of the end devices or the edge networks: Max pooling (MP), Average pooling (AP), and Concatenation (CC). The joint training follows the GoogleNet [37] training process. During the back-propagation phase, the calculated loss from each exit point is accumulated together to train the entire network jointly.

The DDNN reduces communication cost 20× compared to offloading raw sensor data from the end devices to the cloud.

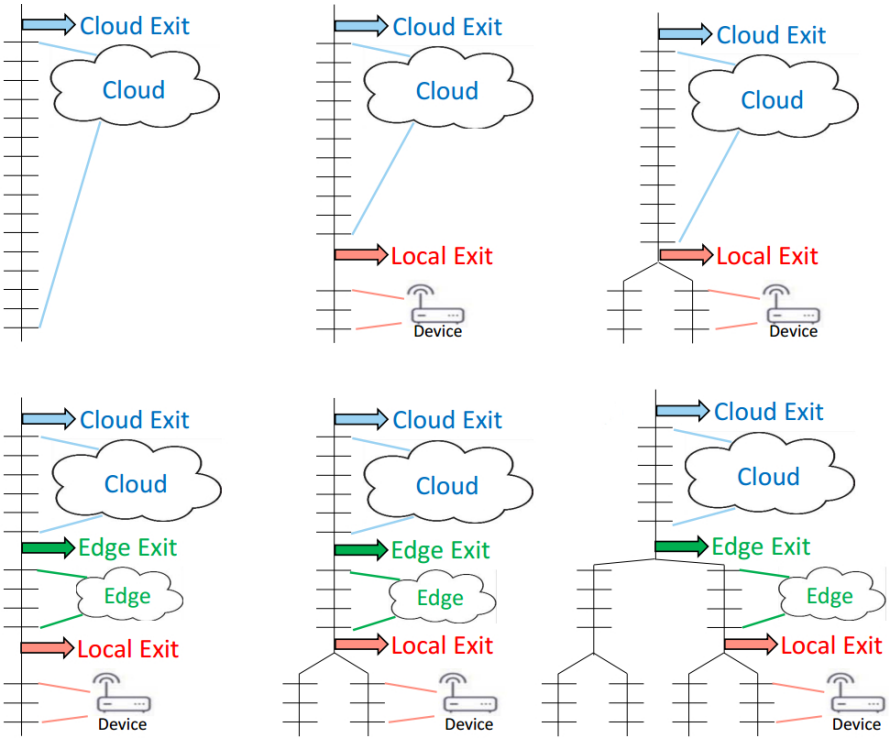

Figure 3.4: An overview of distributed deep neural network framework. Source: [38]

3.5 Distributed inference acceleration (DINA)

This thesis has implemented a DNN adaptive task partitioning and offloading approach based on the research paper by Mohammed et al. [25], which

is inspired by the recent contributions in the DNN inference acceleration mentioned above.

Most of recent approaches split the DNN into two parts: one part executes on the end or edge device, and the other part on the cloud. These approaches have some trade-off between computation time and transmission time. Mohammed et al. [25] have proposed a technique to split the DNN computation into multiple partitions that can be processed locally on end devices or distributed the partitions across one or multiple fog nodes in a network. Authors have introduced two schemes: an adaptive DNN partition and a distributed algorithm based on matching theory to offload the DNN. Figure 3.5 overviews the underlying model. In Figure 3.5 (a), the DNN inference task d_i divides into sub-tasks (i.e., $v_0, v_1, \ldots, v_n$). The sub-tasks are the DNN layers in the context of the DNN inference. Moreover, the sub-task can be partitioned into more smaller tasks (i.e., $v_i = v_{i0}, v_{i1}, \ldots, v_{in}$). These sub-divided tasks are then offloaded to the neighboring fog nodes selected based on the utilities shown in Figure 3.5 (b).

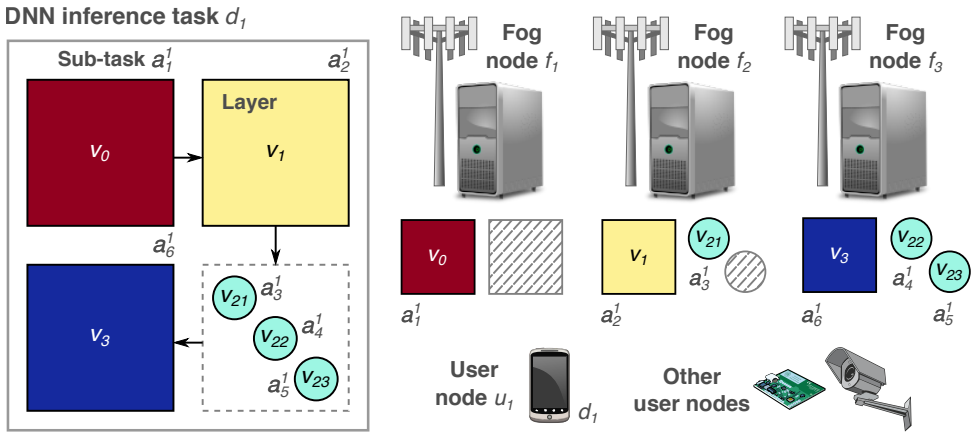

Figure 3.5: An overview of the DINA. Source: [25]

Mohammed et al. [25] have presented Distributed INference Acceleration (DINA), which is based on a matching game approach. The matching game theory, also known as search and matching theory, is an arithmetical framework introduced in economics. This theory describes the relational behavior between two sets of entities with preferences on each other [30]. It has been applied in wireless networking for resource allocation [13]. The significant contribution of this work is given below. All the notations are summarized in Table 3.1.

Symbol	Description
$\mathcal{U}$	Set of user nodes
$\mathcal{F}$	Set of fog nodes
$\mathcal{A}$	Set of DNN layers/partitions
φ_{fa}	Service utility of fog node f
φ_{uf}	Service utility of user node u
$\succ_u$	Preference relation of user node u
$\succ_f$	Preference relation of fog node f
μ_u	Preference list of user node u
r_{uf}	Data rate of user node u towards fog node f
x_{fa}	Binary variable denoting task a assigned to fog node f
τ_a	Delay threshold for task a
δ	Edge delay
θ	Maximum fog nodes for offloading
Γ	Target transmission rate threshold
T_{fa}	Total execution time for task a at fog node f
T_{fa}^{tr}	Transmission time for task a to fog node f
T_{fa}^{exe}	Execution time for task a at fog node f
T_{fa}^{que}	Queuing time for task a at fog node f
$\overline{T}_{fa}^{que}$	Average queuing time for task a at fog node f

Table 3.1: Summary of notations.

3.5.1 Fine-grained adaptive partitioning (DINA-P)

The DNN task is divided into multiple fragments that are smaller than a single layer. The partitioning scheme considers the specific characteristics of layer types in different DNN architectures and represents the partition into a matrix to reduce communication overhead.

The partitioning algorithm is called DINA-P and relies on utility functions for both user nodes and fog nodes. The utility of a fog node denoted as f for executing a task denoted as a of a user node denoted as u is:

$$\varphi_{fa} = x_{fa}(t)(\tau_a - T_{fa}^{tr} - T_{fa}^{exe} - \overline{T}_{fa}^{que}) \tag{3.1}$$

where τ_a represents the delay threshold for task a. T_{fa}^{tr} is the transmission time for task a to fog node f and T_{fa}^{exe} is the execution time for task a at fog node f. $\overline{T}_{fa}^{que}$ denotes the average queuing latency. $x_{fa} \in \{0, 1\}$ is a binary variable defines the offloading problem.

The utility of the user node u for a matching fog node f is defined as:

$$\varphi_{uf} = \frac{1}{T_{fa}} \tag{3.2}$$

$$T_{fa} = T_{fa}^{tr} + T_{fa}^{exe} + T_{fa}^{que} + \delta \tag{3.3}$$

where T_{fa} denotes total execution time for task a at fog node f and δ is the edge delay. Table 3.1 summarizes the notations.

Algorithm 1: Adaptive DNN partitioning (DINA-P)

Input : p: convolution kernel size; φ_{fa}: utility of f; $|f_n|$: number of fog neighbors f of $u \in \mathcal{U}$; $\mathcal{G}$: DAG for DNN inference task d; ${R^{m \times n}}_a$: matrix associated with subtask a, $\forall a \in \mathcal{A}$; c: compute power of f.

Output : DNN partitions $P = \{P_1, P_2, P_3, \ldots\}$

Init : $P \leftarrow \emptyset$, $c_0 = 0$, $w \leftarrow \max(m, n)$

1 **forall** a belonging to parallel paths in $\mathcal{G}$

2 **for** $i \in [1, |f_n| + 1]$

3 $\rho_i \leftarrow \dfrac{\sum_{j=0}^{i-1} (c_j/\varphi_{fa})}{\sum_{j=0}^{|f_n|} (c_j/\varphi_{fa})}$

4 **if** convolutional layer

5 $\omega_i \leftarrow \lfloor \rho_i \cdot (w - (p-1)) \rfloor$

6 $P_i \leftarrow P_i \cup R_a[\omega_{i-1}][\omega_i + (p-1)]$

7 **else** `// fully-connected layer`

8 $\omega_i \leftarrow \lfloor \rho_i \cdot w \rfloor$

9 $\hat{P} = R_a[\omega_{i-1}][\omega_i]$

10 $P_i \leftarrow P_i \cup \hat{P}$

11 $P \leftarrow P \cup P_i$

Algorithm 1 describes the adaptive partitioning algorithm. Input, output, and init sections initialize all the variables. All the parallel paths in the DNN inference task are considered (line 1). For each neighboring fog f, the user node calculates the partition ratio ρ_i based on the computing capabilities and the utilities of the fog nodes (line 2). This partitioning ratio is used to divide the input task into the sub-task depending on the type of layers, such as the convolution layers (lines 4-6) and the fully-connected layers (lines 7-10). The convolution layers need redundant information (i.e., pixels) for computing convolution operation that makes the partitioning ratio different from the fully-connected layers. The adaptive partition depends on the network condition to provide maximum utilities for both the user nodes and the fog nodes.

3.5.2 Distributed DNN offloading (DINA-O)

This section describes a distributed algorithm for offloading the DNN task to preferable fog nodes based on computation time, communication latency, and queuing time.

After the partition, the sub-tasks are offloaded to the fog nodes. A single DNN task d can be offloaded to multiple fog nodes; moreover, a single fog node can be associated with multiple use nodes, which leads to a many-to-many matching. The matching is obtained based on preference relations. The association between a user and a fog node depends on the association of other users to the same fog node. This kind of dependency is known as externality in matching theory. Due to dynamic preference, stability is not guaranteed with standard matching algorithms (e.g., Gale-Shapley [7]) under externality [2]. Two-side exchange stability [3] addresses the problem where no user remains unmatched. The offloading algorithm (DINA-O) adopts the concept of two-sided exchange stability and addresses the optimization problem with the help of swap matching. Algorithm 2 describes the offloading algorithm.

The distributed swap matching algorithm has two phases: initialization and swap matching. Lines 1-6 describe the algorithm's initialization, starting with discovering the neighboring fog nodes. User nodes then calculate the data rate toward the fog nodes. On this occasion, both the fog nodes and the user nodes calculate their utilities based on the Eq (3.1) and the Eq (3.2), and create a preference list. Before executing Algorithm 1 to partition the DNN tasks, an initial random assignment between the user nodes and the fog nodes is performed by satisfying the following *constraints*: a DNN task should offload to at most θ fog nodes at a given time, the transmission rate should not be lower than a threshold Γ and the total time to execute the task should not be more than the time τ_a for executing the same task locally.

In the swap matching phase (lines 7-27), the randomly matched pair executes the swap matching if there is a swap-blocking pair. The swap blocking pair satisfies the following conditions: the utilities should not decrease for both the user nodes and the fog nodes after the swap, and the utilities of at least one node should improve following the swap. Each user node sends a request to its first preference fog node if the fog node is not associated with it already. At this point, the fog node f calculates the utilities based on the DNN sub-task a from the node u. If the utilities improve after the swap, the fog node accepts the request. If the request is instead rejected, the user node considers the second fog node from the preference list for swap. The iteration comes to an end if all the user nodes attach with their preferred fog nodes, and no more swaps are possible; thus, stable matching is obtained, and sub-tasks are offloaded to the fog nodes.

Algorithm 2: DNN Inference Offloading (DINA-O)

Input : A set of FNs $f, f' \in \mathcal{F}$ and UNs $u, u' \in \mathcal{U}$
Output : A two-sided exchange-stable matching μ
`// Phase 1: Initialization`
Each $u \in \mathcal{U}$ discover its neighboring fog nodes $f \in \mathcal{F}$
All nodes $\forall u \in \mathcal{U}, \forall f \in \mathcal{F}$ calculate r_{uf}
$\forall f \in \mathcal{F}$ creates a preference list with Eq. (3.1)
$\forall u \in \mathcal{U}$ creates a preference list with Eq. (3.2)
Randomly match $(u, f), \forall f \in \mathcal{F}, u \in \mathcal{U}$ such that the *constraints* are satisfied
Execute Algorithm 1 to partition DNN tasks $d \in \mathcal{D}$
`// Phase 2: Swap matching`
while $\exists \mu_{u,f}^{u',f'} : (f', \mu_{u,f}^{u',f'}) \succ_u (f, \mu), (u', \mu_{u,f}^{u',f'}) \succ_f (u, \mu), (f, \mu_{u,f}^{u',f'}) \succ_{u'} (f', \mu), (u, \mu_{u,f}^{u',f'}) \succ_{f'} (u', \mu)$
  Update φ_{fu} and φ_{fu} based on μ
  Sort fog nodes $f \in \mathcal{F}$ based on preference $\succ_u$
  Sort user nodes $u \in \mathcal{U}$ based on preference $\succ_f$
  if $\mu_{u,f} = \varnothing$ `// There is an unmatched item` o
    u sends proposal to most preferred f
    f computes $\varphi_{fu}\left(\mu_{u,o}^{o,f}\right)$
    if $(u, \mu_{u,o}^{o,f}) \succ_f (u, \mu)$ and *constraints* hold
      Accept proposal, $\mu \leftarrow \mu_{u,f}^{u',f'}$
      $\Lambda_f \leftarrow \Lambda_f \cup \{u\}$, $\Lambda_u \leftarrow \Lambda_u \cup \{f\}$
    else Reject proposal and keep matching μ
  if $(f', \mu_{u,f}^{u',f'}) \succ_u (f, \mu)$ and $(f, \mu_{u,f}^{u',f'}) \succ_{u'} (f', \mu)$
    u sends proposal to f' and u' to f
    f, f' compute $\varphi_{f'u}(\mu_{u,f}^{u',f'}), \varphi_{fu'}(\mu_{u,f}^{u',f'})$
    if $(u', \mu_{u,f}^{u'f'}) \succ_{f'} (u, \mu)$ and *constraints* hold
      Accept proposal, $\mu \leftarrow \mu_{u,f}^{u',f'}$
      $\Lambda_{f'}, \Lambda_u \leftarrow \Lambda_{f'} \setminus \{u'\} \cup \{u\}, \Lambda_u \setminus \{f\} \cup \{f'\}$
    else if $(u, \mu_{u,f}^{u'f'}) \succ_f (u', \mu)$ and *constraints* hold
      Accept proposal, $\mu \leftarrow \mu_{u,f}^{u',f'}$
      $\Lambda_f, \Lambda_{u'} \leftarrow \Lambda_f \setminus \{u\} \cup \{u'\}$, $\Lambda_{u'} \setminus \{f'\} \cup \{f\}$
    else Reject proposal and keep matching μ

The notation $\mu_{u,f}^{u',f'}$ indicates the swap matching.

$$\mu_{u,f}^{u',f'} = \{\mu \backslash \{(u, \mu(u)), (u', \mu(u'))\}\} \cup \{(u, \{\{\mu(u) \backslash f\} \cup f'\}), (u', \{\{\mu(u') \backslash f'\} \cup f\})\}$$

The notation $(f', \mu_{u,f}^{u',f'}) \succ_u (f, \mu)$ indicates that the user node u prefers the fog node f' and the swap matching $\mu_{u,f}^{u',f'}$ over the fog node f and μ, respectively. An example with random utilities of the user nodes and the fog nodes explains the swap matching more clearly as follows. The letters and the numerical values represent the user nodes and the fog nodes, respectively. Table 3.2 and Table 3.3 depicts the random utilities of the user nodes and the fog nodes. The preference list of the user nodes and the fog nodes based on the utilities are illustrated in Table 3.4 and Table 3.5.

Node A	Node B	Node C
1 : 93 2 : 97 3 : 84 4 : 23	1 : 2 2 : 15 3 : 97 4 : 94	1 : 49 2 : 7 3 : 71 4 : 72

Table 3.2: Utilities of fog nodes calculated by user nodes.

Node 1	Node 2	Node 3	Node 4
A : 70 B : 21 C : 45	A : 91 B : 23 C : 20	A : 50 B : 92 C : 59	A : 22 B : 71 C : 82

Table 3.3: Utilities of user nodes calculated by fog nodes.

Node A	Node B	Node C
2 > 1 > 3 > 4	3 > 4 > 1 > 2	4 > 3 > 1 > 2

Table 3.4: User node preference list based on the fog node's utilities.

Node 1	Node 2	Node 3	Node 4
A > C > B	A > B > C	B > C > A	C > B > A

Table 3.5: Fog node preference list based on the user node's utilities.

Figure 3.6 illustrates simple network considered in this example. The circles and the triangle represent the user node and the fog node, respectively. The edge indicates the strength of the communication link. For example, the user node A prefers to offload the DNN sub-tasks to the fog nodes 1, 2, 3 instead of 4 considering the utilities.

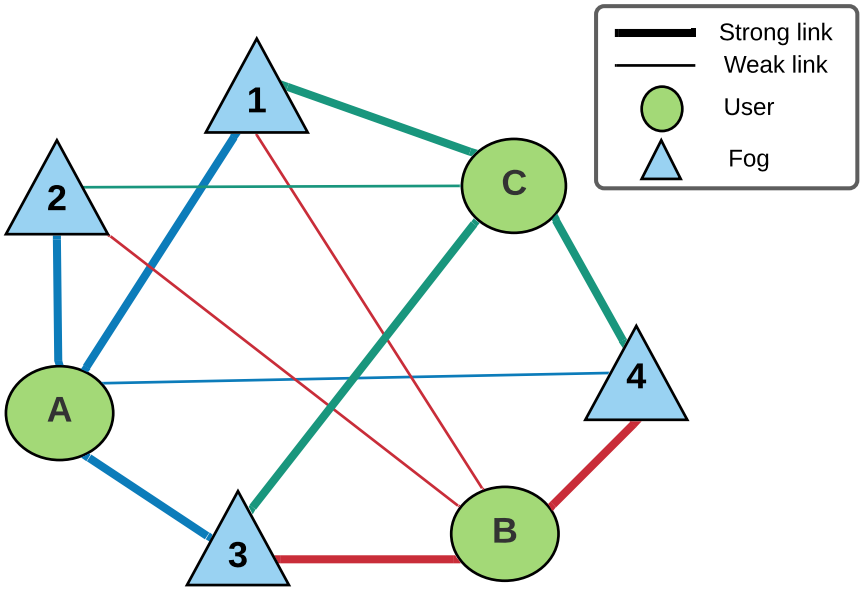

Figure 3.6: Topological view of user and fog nodes in the considered example.

Let us assume that, after applying the random match (line 5 of Algorithm 2), the pairs are (A, 3), (B, 1), (C, 2). The swap matching applies if the pairs are swap-blocking pairs and the conditions for swap-blocking pairs satisfy. Table 3.6 shows a scenario when the swap executes. (A, 3) and (B, 1) are a swap-blocking pair and satisfies the conditions. Thus, a swap occurs between the pairs resulting in new pairs (A, 1) and (B, 3). The new pair (A, 1) and random match pair (C, 2) are not a swap-blocking pair. They do not perform a swap. After the swap matching, the stable pairs are (A, 1), (B, 3), (C, 2).

(A, 3) and (B, 1)		(A, 1) and (C, 2)	
$1 \succ_A 3$	✓	$2 \succ_A 1$	✓
$3 \succ_B 1$	✓	$1 \succ_C 2$	✓
$A \succ_1 B$	✓	$C \succ_1 A$	×
$B \succ_3 A$	✓	$A \succ_2 C$	✓
State: Swap		**State**: No swap	

Table 3.6: Swap matching.

Performance Evaluation: A custom python network simulator has been employed to evaluate the performance of the research work by simulation. The authors use the Berkeley Deep Drive [39] dataset and experiment on four state-of-the-art DNN architectures, i.e., AlexNet, ResNet32, VGG16, and NiN. DINA is compared with three schemes: a random offloading with DINA partitioning (RANDP), a random offloading without partitioning (RAND), and a greedy offloading without partitioning (GANC) and showed an outstanding performance in terms of total execution time compared to the other three approaches. It indicates approximately 1.7-5.2 times better performance. In the case of transmission time, DINA performs around 1.7-2.9 times better than the RAND and the RANDP. DINA is also compared with the DNN Surgery [16] and outperforms by 2.6-4.2 times approximately in total execution time.

Chapter 4

Implementation

This chapter explains the methodologies used to implement DINA [25] as a software framework for heterogeneous embedded devices. Two schemes of DINA: a fine-grained adaptive partitioning and a distributed swap-matching algorithm-based matching theory are implemented and evaluated in this thesis. This chapter focuses on the framework, libraries, tools used, and the implementation approach of both the schemes.

4.1 Frameworks, libraries and tools

PyTorch [28] is used as the development framework in this thesis. PyTorch is a free and open-source library for machine learning research based on the Torch library developed by the Facebook AI research lab. It operates in two interfaces, such as Python and C++. Python interface is more stable and used than C++. The reasons to choose PyTorch as a development framework are listed below.

- **Pythonic**: Python is a widely-used programming language in the machine learning research area. Python integration is easier in PyTorch, and NumPy, one of the most popular Python libraries, is also integrated with PyTorch. Python makes PyTorch framework easy to learn, code, and debug.
- **Data parallelism**: PyTorch can distribute computation among multiple CPUs or GPUs.
- **Dynamic computational graph**: PyTorch supports dynamic computation, where every level of computation can be accessed and changed programmatically at run-time based on the network's behavior. Whereas

other machine learning frameworks with static graph support work like a black box.

- **Cloud support**: PyTorch supports all well-known cloud platforms for large-scale datasets and deep learning models.
- **Community support**: PyTorch has small but very efficient community support, which is monitored by some incredible, talented, and focused contributors and developers.

Section 2.4 presented four state-of-the-art CNN architectures: AlexNet, VGG16, ResNet34, and NiN, which were used to train models for DNN inference. These four architectures are the most popular and widely used for image recognition in different applications. Moreover, two of them, namely, AlexNet and ResNet34, follow the Directed Acyclic Graphs (DAGs) topology, and the other two architectures: VGG16 and NiN, follow the chain topology. The PyTorch incorporates a subpackage called `torchvision.models` that contains the definition of the model architectures and pre-trained models on ImageNet, such as AlexNet, VGG16, ResNet34 for image classification application. This thesis has used the `torchvision.models` to train the models on BDD100k dataset [39] except for the Network-In-Network (NiN). The definition of NiN is written separately and imported for training the model for DNN inference. The trained models are copied to heterogeneous fog nodes with the help of gRPC remote procedure call (gRPC) protocol. Figure 4.1 visualizes the topologies of the benchmark DNN architectures.

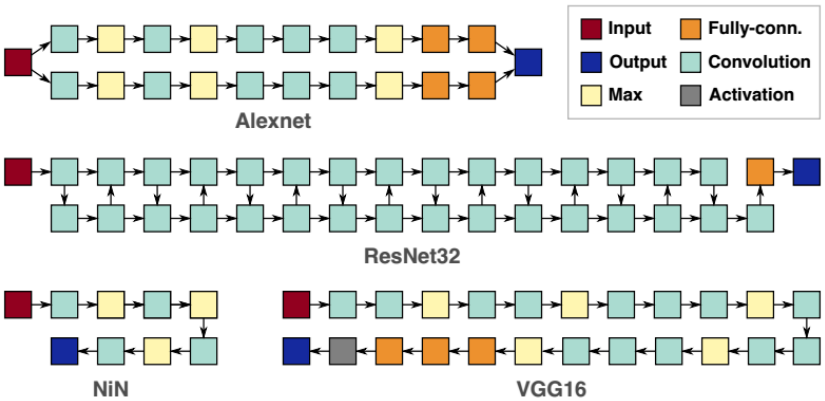

Figure 4.1: Topologies of the benchmark DNN architectures. Source: [25]

Appendix A.2 describes the gRPC communication protocol, which is used as the communication protocol in this thesis. The following steps are per-

formed to incorporate the gRPC protocol in the python programming environment.

- Create a remote procedure in the server-side (fog nodes), which exposes to the client.
- Create a proto file (e.g., message.proto), which contains service methods and protobuf message type definitions for all requests and response messages.
- Generate gRPC classes for python, which is handled by special tools. For python, `grpcio` and `grpcio-tools` handle the class generation functionality.

```
$ pip install grpcio
$ pip install grpcio-tools
$ python -m grpc_tools.protoc -I. --python_out=.
    --grpc_python_out=. message.proto
```

  The above commands generate two python files, e.g., `message_pb2.py` (message class) and `message_pb2_grpc.py` (server and client classes).
- Create the gRPC server and the client in python programming environment.

The client and the server use the classes generated by the gRPC-tools to establish the gRPC channel, the stub methods, and the gRPC server for handling the remote procedure calls.

4.2 Implementation details

This section describes the approaches taken to partition a DNN task and offload the sub-tasks to the fog nodes for DNN computation. The procedure starts with training the model parameters in four state-of-the-art DNN architectures and then implementing the DINA-P and the DINA-O algorithm for partitioning the DNN task and offloading the sub-tasks to the neighboring fog nodes, respectively for DNN inference.

4.2.1 Training models

The models need to be trained initially to perform DNN inference. The BDD100k [39] has twelve categories/classes of objects, i.e., rider, traffic light

(TL), lane, traffic sign (TS), bike, motor, truck, bus, car, drivable area (DA), person and train that can be classified from the images. Image attributes are stored in a JavaScript Object Notation (JSON) file format.

In supervised learning, the output of the prediction related to image classification and speech recognition is compared with the target output to provide the accuracy of the prediction. Usually, one-hot encoding represents the target output. One hot encoding is a process that transforms the object categorical values into binary values for better prediction in machine learning applications. As mentioned earlier, the BDD100k data set images have different attributes, such as weather, scene, timeofday, timestamp, and category. The categories are extracted from the file for each image and transformed into one-hot encoding. For example, consider one image from the data set with the following categories of objects: traffic light, traffic sign, car, drivable area, and line. Table 4.1 represents the categories as one-hot encoding.

Ride	TL	lane	TS	Bike	Motor	Truck	Bus	Car	DA	Person	Train
0	1	1	1	0	0	0	0	1	1	0	0

Table 4.1: Example of one-hot encoding.

The following strategies are taken to train a model.

- The categories are represented as one-hot encoding for each image in the data set.
- Each image needs to be transformed into a suitable format before training based on the DNN architecture. All the images are resized, cropped at center, and normalized. The resize and the normalization values are similar to the Imagenet dataset. As the BDDdataset and the Imagenet lies in a similar domain, the difference of the mean and the standard deviation are inferior. This transformation is done using `torchvision.transforms.Compose` package. Table 4.2 describes the transformation parameters used in different DNN architectures.
- All the state-of-the-art DNN architectures are implemented by the PyTorch except the NiN architecture. `torchvision.models` package is used to load the architecture for training where the model parameters are assigned with random values initially. The NiN architecture is implemented separately.
- The models are trained in GPU for high efficiency. PyTorch provides `torch.cuda` package for CUDA tensor types. PyTorch supports the

Resize (pixels)	256
Normalization	
Mean Std	[0.485, 0.456, 0.406] [0.229, 0.224, 0.225]
Center crop (pixels)	
AlexNet	227
VGG16	224
ResNet34	224
NiN	32

Table 4.2: Transformation parameters.

fast transfer of the computational graph generated in CPU to GPU memory.

- The images from the dataset are trained in mini-batches. The ideal batch size is between 64 and 256. The batch size has a significant impact on the test accuracy. If the batch size is kept small, the number of model parameters update more frequently per epoch.

4.2.2 Implementing DINA-P and DINA-O

Section 3.5 describes the algorithms for partitioning the DNN task and offloading the sub-tasks to neighboring fog nodes. The DINA-P partitions the DNN task, and the DINA-O detects the best match with the swap-matching phase and then offloads the sub-tasks to the best matching fog nodes. After executing the DNN computation in the neighboring nodes, the results are sent back to the user node. The following steps are performed.

- The images of the dataset are used as the DNN tasks. The implementation approach of this thesis is mostly targeting image classification application for self-driving cars. The same approach can be used for other applications as well, such as voice recognition, natural language processing, which require heavy DNN computation. The main idea is to offload the DNN computations to the edge devices which have high hardware configurations than the user devices.

- According to the Eq (3.1) and the Eq (3.2), the utilities of the fog nodes and the user nodes are calculated. At the beginning of the experiment, random utilities are considered for simplicity. After calculating the transmission time (T_{fa}^{tr}) and the execution time (T_{fa}^{exe}) through some experiments, the actual utilities are measured for the user nodes

and the fog nodes. The delay threshold (τ_a) is chosen based on the size of the sub-tasks. The queuing time (T_{fa}^{que} and $\overline{T}_{fa}^{que}$) and the edge delay (δ) are considered zero for the simple network topology. The execution of the DINA-P algorithm requires the utilities of the fog nodes. The number of sub-tasks depends on the number of discovered fog nodes. After the partition, the sub-tasks are offloaded accordingly. For example, [1,3,224,224] represents an image tensor where one indicates the batch-size, three denotes the number of channel in the image (e.g., RGB), the rest are width and height of the image. When the image is fed to the DINA-P algorithm considering the random utilities as 4.0 and 6.0 for the two fog nodes, the algorithm returns [1,3,90,224] and [1,3,136,224], respectively as the partitioning results. The size of the sub-tasks depends on the utilities of the fog nodes.

- The DINA-O algorithm decides where to offload the sub-tasks. After the final matching, the user offloads the sub-tasks to the appropriate fog nodes for execution.

- A distributed strategy is applied for executing a complete DNN task. The input image is divided into sub-tasks by the DINA-P algorithm and offloaded to the matching fog nodes for execution. After the completion of the execution, the results are sent back to the user device, which are merged and partitioned again for offloading.

- The fog nodes execute the state-of-the-art DNN architectures layer-by-layer. Most of the DNN architectures follow a standard structural convention, a stack of convolutional layers followed by a series of fully-connected layers. The model parameters are separated for the individual layers. The parameters are assigned when that particular layer is executing. Figure 4.2 illustrates an overview of the distributed approach.

4.2.3 Implementation approach

The DNN task is partitioned using three different approaches, i.e., adaptive partitioning (DINA-P and DINA-O), random partitioning, and greedy offloading, for the purpose of evaluation.

- **Adaptive partitioning**: The DINA-P algorithm is applied in this partitioning approach. After discovering the neighboring fog nodes and an initial matching satisfying the constraints (see Section 3.5.2), DNN partitioning is performed with DINA-P (Algorithm 1). The sub-tasks

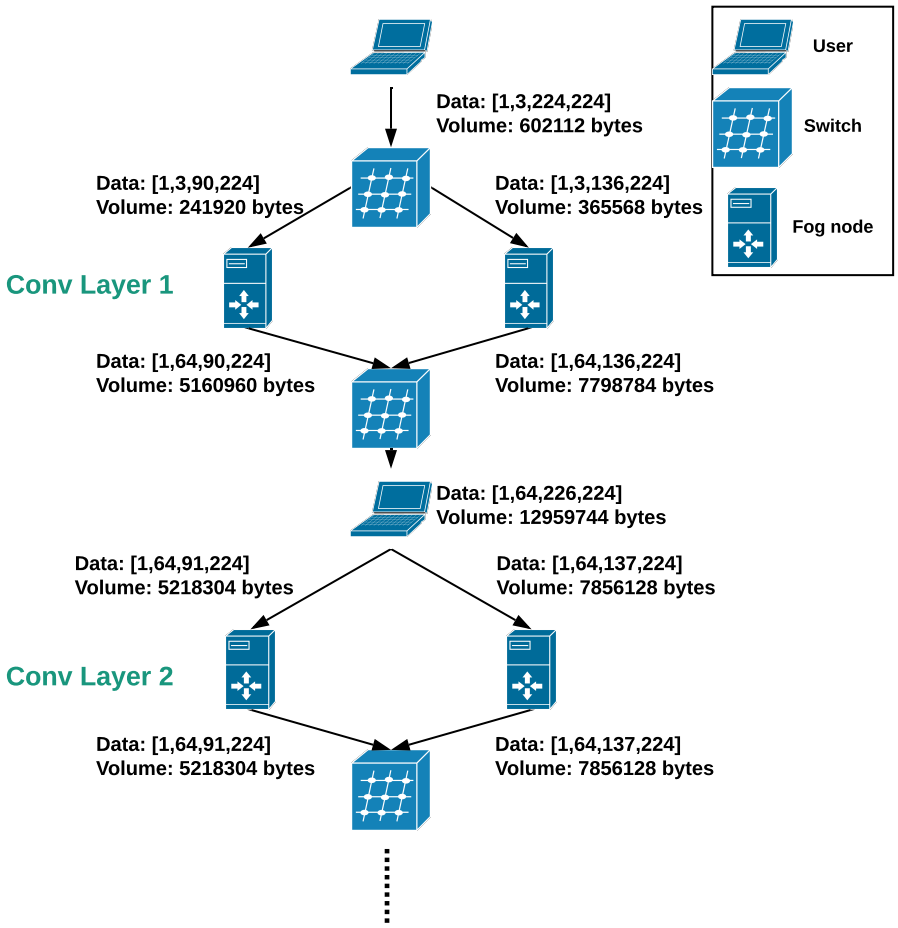

Figure 4.2: An overview of the distributed approach.

are offloaded to the fog nodes applying the DINA-O (Algorithm 2) algorithm after the final matching is obtained.

- **Random partitioning**: The partition and the offloading are executed without considering the utilities and the preference list of the user nodes and fog nodes. The DNN task is divided randomly and offloaded to randomly assigned fog nodes.
- **Greedy offloading**: In the greedy offloading, the whole DNN task offloads to one of the randomly associated fog nodes. The DNN task is not partitioned in this approach.

Chapter 5

Evaluation

Chapter 4 described the implementation details for partitioning a DNN task and offloading the sub-tasks to the associated fog nodes. A constrained test environment is built for evaluating the implementation described in Section 5.1. This thesis conducted an experimental research, which is a quantitative methodological approach. A quantitative research method defines a systematic investigation of observable phenomenon with experiments and statistical, mathematical, or numerical analysis. In this chapter, the experimental results are discussed in detail. Section 5.2 describes the selection of a quantitative dataset. The topological view of the network is discussed in Section 5.3.

5.1 Experimental setup

This thesis utilizes a real hardware platform to depict as the fog nodes and the user nodes. The Nvidia Jetson Nano development kits are small but very powerful devices, used as the fog nodes. It is equipped with a Quad-core ARM A57 processor and Nvidia Maxwell GPU micro-architecture, and is useful for deploying computer vision and deep learning applications. Table 5.1 illustrates the server platform specifications.

Hardware	Specifications
CPU	Quad-core ARM Cortex-A57 1.43 GHz
GPU	Nvidia Maxwell architecture with 128 Nvidia CUDA cores
Memory	4 GB 64-bit LPDDR4

Table 5.1: Server platform specifications.

A Dell Latitude, 7400 model notebook, is used as a user device for ex-

perimental purposes. Compared to modern embedded and portable devices, a notebook is more powerful, but for simplicity, it is used as a dumb terminal. The user device offloads the heavy DNN computation to the fog devices, considering the fog devices' utility values. Table 5.2 illustrates the client platform specifications. A visual illustration of experimental setup consisting of Nvidia Jetson Nano devices and Dell latitude is shown in Figure 5.1.

Hardware	Specifications
Model	Dell Latitude 7400
CPU	Intel Core i5-8365U 1.60GHz
Memory	15.4 GB 64-bit SODIMM DDR4

Table 5.2: Client platform specifications.

Figure 5.1: An illustration of experimental setup.

5.2 Selection of the dataset

The Berkeley Deep Drive dataset (BDD100k) [39] is the largest driving video data set where 120M images are extracted from 100K videos collected from autonomous vehicles. The images are organized and managed, considering diverse geographical, environmental, and weather data to train the DNN

model more accurately. Decision made by an autonomous car while driving depends mostly on how well the model has been trained. To train a model and inference after training, selecting an appropriate data set is crucial for sensitive application areas. BDD100k provides a wide variety of visual driving scenes and ten task facilities, such as image tagging, imitation learning, road object detection, lane detection, semantic segmentation, drivable area segmentation, multi-object detection tracking, instance segmentation, domain adaptation, and multi-object segmentation tracking.

5.3 Network topology

This thesis uses an elementary network topology. The experiment is done in a constrained environment containing two fog devices and one user device. All the devices are connected via the ethernet cables and a switch. Static IP addresses are used by the devices to communicate. The user device and the fog devices act as a client and servers, respectively. A switch is used as a central hub to connect the client and the servers. This simple topology is set for experimental purposes. The thesis implementation works for multiple heterogeneous fog devices and user devices. The network topology is illustrated in Figure 5.2.

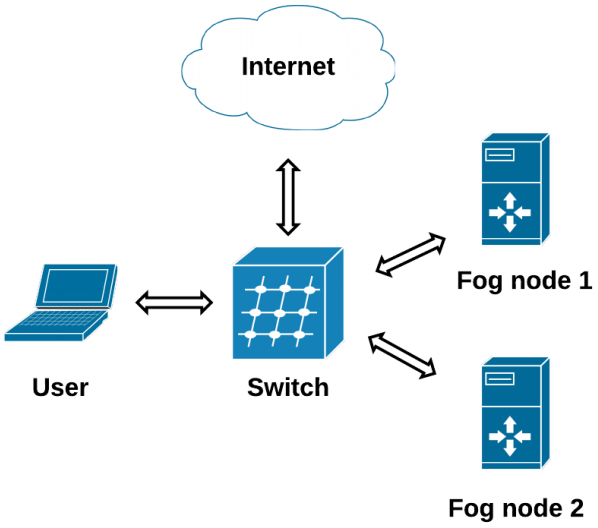

Figure 5.2: A simple network topology.

The communication is bi-directional such that both the user device and the fog devices can send messages alternatively. Appendix A has described two popular communication protocols which provide a bi-directional communication facility. This thesis has experimented with both the communication protocols and found that the gRPC provides better performance than the MQTT with respect to transmission time (see Appendix A.3). The overall implementation is evaluated, running the gRPC as the communication protocol.

5.4 Experimental results

Section 4.2.3 described the implementation approach of this thesis. Based on the network topology, the implementation is evaluated using three approaches discussed below.

Adaptive partitioning (Adaptive): The DNN partition is executed with DINA-P (Algorithm 1) and splits the task into at most two sub-tasks due to the number of fog nodes considered in this experiment. After executing the swap matching phase of the DINA-O (Algorithm 2), the user node associates with the best fog node and offloads the significant portion of the DNN task and the rest is executed in the user node.

Random partitioning (Random): The DNN task is randomly partitioned into at most two sub-tasks and offloaded to the fog nodes simultaneously. This approach does not execute the DINA-P and the DINA-O algorithms.

Greedy offloading (Greedy): A whole DNN task is offloaded to one of the fog nodes without applying the partitioning algorithm.

The evaluation of the implementation is done based on the transmission and the computation time. The observed results suggest that both the transmission and the computation time vary based on the DNN architectures and transmitted data size. Throughout the evaluation, four state-of-the-art DNN architectures (i.e., AlexNet, VGG16, ResNet34, and NiN) are considered.

Total computation time: Figure 5.3 illustrates the comparison of total computation time (T_{fa}) of the four state-of-the-art DNN architectures for three schemes (Adaptive, Random, and Greedy). The adaptive scheme shows better performance comparing the other two schemes except for the NiN ar-

chitecture. As the network grows deeper, the computation time increases. The VGG16 is a very deep neural network which has 16 weighted layers. Thus, it consumes the most computation time in the comparison graph. ResNet34 is also a deep network, but the residual learning helps to reduce the computation time. The best performance is obtained in the case of AlexNet and NiN as both have a linear topology and a small number of layers.

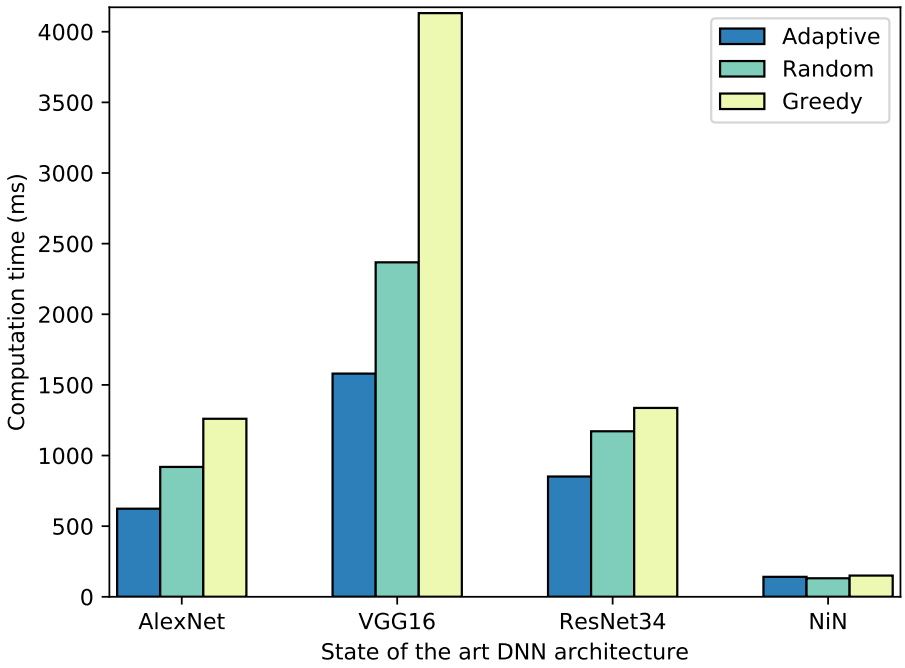

Figure 5.3: Total computation time of the three schemes on four state-of-the-art architectures.

Figure 5.4 demonstrates the improvement of the Adaptive scheme over the Random and Greedy schemes. The figure shows how the adaptive approach outperforms the other two schemes, with improvement between 1.4 to 2.6 times in three DNN benchmarks (AlexNet, VGG16, and ResNet34). In case of NiN architecture, the adaptive approach performs better than the greedy solution but similar to the Random scheme. Appendix B shows the layer-by-layer computation time comparison of three schemes (Adaptive, Random, and Greedy) based on the four benchmarks.

Total transmission time: Figure 5.5 illustrates the comparison of total data transmission time of the four state of the art DNN architectures for three

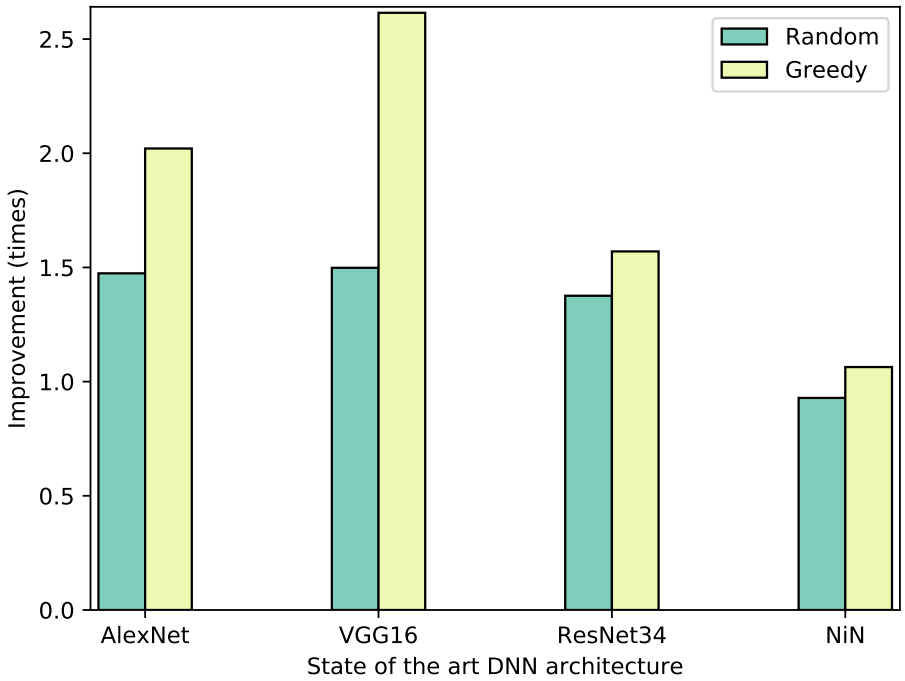

Figure 5.4: Improvement (communication) of adaptive-partitioning against other two schemes.

schemes (Adaptive, Random, and Greedy). In this case also, the adaptive scheme exhibits the best performance. The transmission time remains below 3 seconds in three state of the art architectures (AlexNet, ResNet34, and NiN). The transmission time mostly depends on the size of the transmitted data. In VGG16, the middle layers consist of many input and output channels, which increase the data size. At the beginning and the end of the network, the data size remains considerably lower than the middle stage of the network.

Figure 5.6 demonstrates the improvement of the adaptive scheme over the other two schemes (Random and Greedy). In this case, the adaptive scheme achieves a performance that is 1.1 to 2.9 times more excelling than the Random and the Greedy schemes in three state of the art DNN architectures (VGG16, ResNet34, and NiN). In the case of AlexNet, the Adaptive scheme shows similar performance to the Random scheme but more than two times better performance than the Greedy solution. Appendix B illustrates the layer-by-layer transmission time comparison of three schemes (Adaptive, Random, and Greedy) based on four benchmarks.

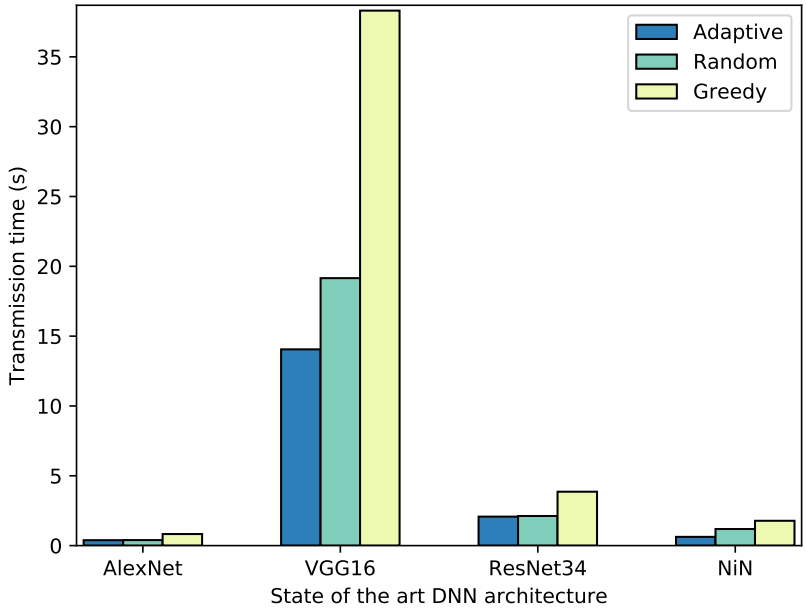

Figure 5.5: Total transmission time of the three schemes based on four state-of-the-art architectures.

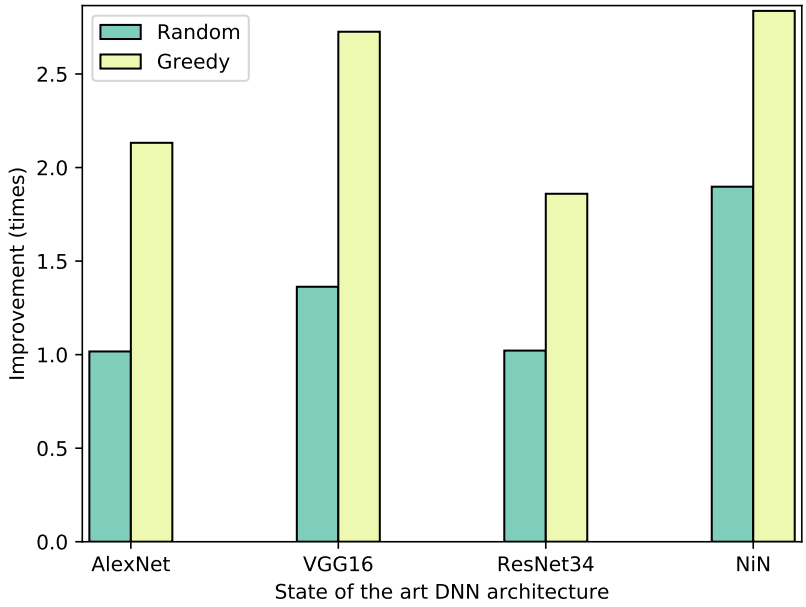

Figure 5.6: Improvement (transmission) of adaptive-partitioning against other two schemes.

Chapter 6

Conclusion

This thesis has provided an implementation of the algorithms for distributed DNN inference acceleration proposed by Mohammed et al. [25]. Specifically, it includes: a fine-grained adaptive partitioning scheme to divide a DNN based on the fog networks; and a distributed algorithm based on swap-matching for offloading DNN inference to fog devices. Partitioning the DNN task allows to employ pre-trained network without any modification by tailoring partitions based on the computation and storage capabilities of devices. The offloading algorithm reduces the total computation and communication time, adapts to the network conditions, and increases the resource utilization in the network.

The implementation has been evaluated on a fog testbed consisting of Nvidia Jetson Nano devices. The experimental results show some key observations. First, communication latency increases in the intermediate layers of the DNN benchmarks due to a large number of in and out channels. Second, most DNN benchmarks are designed by the convolutional layers, followed by the fully-connected layers where the fully-connected layers take more computation time than the convolutional layers. Third, the model loading and initialization in the GPU are time-consuming processes in a layer-by-layer implementation of the DNN architectures. Moreover, the processing time increases due to transferring the generated computational graph in CPU to GPU memory for every layer separately. Finally, both the computation time and the communication latency increase as the network grows deeper.

The evaluation demonstrates that the realized approach of joint partitioning and offloading (i.e., DINA-P and DINA-O) achieves a significant reduction in total execution time and total transmission time compared to other schemes. In particular, it performs 1.4 to 2.6 times and 1.1 to 2.9 times better in terms of computation time and transmission time, respectively.

The work in this thesis can be extended by considering distributed learning instead of DNN inference. Federated learning or collaborative learning is one of the types of distributed learning, where a DNN algorithm is trained across decentralized heterogeneous fog devices, and each device has its local data samples. The federated learning addresses some critical issues (e.g., data privacy, data security, and data access control) as the private sample data of each fog device are not shared with other fog devices throughout the learning process. The federated learning is widely adopted in the application areas where data privacy and security have high priority, such as defense, telecommunications, and pharmaceutics. Another possible future work is evaluating DINA in large-scale networks with several heterogeneous fog nodes and end devices.

Bibliography

[1] Anawar, M. R., Wang, S., Azam Zia, M., Jadoon, A. K., Akram, U., and Raza, S. Fog computing: An overview of big iot data analytics. *Wireless Communications and Mobile Computing 2018* (2018), 1–22.

[2] Bando, K., Kawasaki, R., and Muto, S. Two-sided matching with externalities: A survey. *Journal of the Operations Research Society of Japan 59*, 1 (2016), 35–71.

[3] Bodine-Baron, E., Lee, C., Chong, A., Hassibi, B., and Wierman, A. Peer effects and stability in matching markets. In *International Symposium on Algorithmic Game Theory* (2011), Springer, pp. 117–129.

[4] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR abs/1412.3555* (2014).

[5] Forecast, G. M. D. T. Cisco visual networking index: global mobile data traffic forecast update, 2017–2022.

[6] Frasconi, P., Gori, M., and Sperduti, A. A general framework for adaptive processing of data structures. *IEEE transactions on Neural Networks 9*, 5 (1998), 768–786.

[7] Gale, D., and Shapley, L. S. College admissions and the stability of marriage. *The American Mathematical Monthly 69*, 1 (1962), 9–15.

[8] Goller, C., and Kuchler, A. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN'96)* (1996), vol. 1, IEEE, pp. 347–352.

[9] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y.

Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pp. 2672–2680.

[10] Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., and Shet, V. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *CoRR abs/1312.6082* (2013).

[11] Google. Protocol buffers - google's data interchange format. `https://github.com/protocolbuffers/protobuf`, 2008.

[12] Google. grpc - a high-performance, open source universal rpc framework. `https://grpc.io/`, 2015.

[13] Han, Z., Gu, Y., and Saad, W. *Matching theory for wireless networks*. Springer, 2017.

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

[15] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[16] Hu, C., Bao, W., Wang, D., and Liu, F. Dynamic adaptive dnn surgery for inference acceleration on the edge. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications* (2019), IEEE, pp. 1423–1431.

[17] Hubel, D. H., and Wiesel, T. N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology 160*, 1 (1962), 106–154.

[18] Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., and Tang, L. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News 45*, 1 (2017), 615–629.

[19] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *CoRR abs/1412.6980* (2014).

[20] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

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

[22] Lin, M., Chen, Q., and Yan, S. Network in network. *CoRR abs/1312.4400* (2014).

[23] Mao, J., Chen, X., Nixon, K. W., Krieger, C., and Chen, Y. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017* (2017), IEEE, pp. 1396–1401.

[24] Minsky, M. Logical versus analogical or symbolic versus connectionist or neat versus scruffy. *AI Mag. 12*, 2 (1991), 34–51.

[25] Mohammed, T., Joe-Wong, C., Babbar, R., and Di Francesco, M. Distributed inference acceleration with adaptive dnn partitioning and offloading.

[26] Nair, V., and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)* (2010), pp. 807–814.

[27] OASIS. Message queuing telemetry transport version 5.0. `https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html`, 2019.

[28] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[29] Patterson, J., and Gibson, A. *Deep Learning: A Practitioner's Approach*, 1st ed. O'Reilly Media, Inc., 2017.

[30] Roth, A. E., and Sotomayor, M. A. O. *Two-Sided Matching: A Study in Game-Theoretic Modeling and Analysis*. Econometric Society Monographs. Cambridge University Press, 1990.

[31] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV) 115*, 3 (2015), 211–252.

[32] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2015).

[33] Socher, R., Lin, C. C., Manning, C., and Ng, A. Y. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)* (2011), pp. 129–136.

[34] Sperduti, A., and Starita, A. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks 8*, 3 (1997), 714–735.

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

[36] Srivastava, R. K., Greff, K., and Schmidhuber, J. Highway networks. *CoRR abs/1505.00387* (2015).

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

[38] Teerapittayanon, S., McDanel, B., and Kung, H. T. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017), pp. 328–339.

[39] Yu, F., Xian, W., Chen, Y., Liu, F., Liao, M., Madhavan, V., and Darrell, T. BDD100K: A diverse driving video database with scalable annotation tooling. *CoRR abs/1805.04687* (2018).

[40] Zeiler, M. D., and Fergus, R. Visualizing and understanding convolutional networks. In *European conference on computer vision* (2014), Springer, pp. 818–833.

Appendix A

Communication protocols

This thesis aims to offload the partitioned DNN tasks to the nearest fog devices following their utilities. This requires to co-ordinate actions over a network. In this regard, a communication protocol is needed as a set of rules for sharing information between two or more entities via a physical wired or wireless medium. Two communication protocols, such as publisher-and-subscriber protocol (e.g., MQTT [27]) and client-and-server protocol (e.g., gRPC), can be considered. A short description of both protocols is provided next.

A.1 Message Queuing Telemetry Transport

Message Queuing Telemetry Transport (MQTT) is a lightweight publisher-and-subscriber network protocol. It typically runs over the TCP/IP and supports other ordered, lossless, and bi-directional connections. This protocol has been widely used in the IoT technologies, where small sophisticated devices need to communicate with low energy usage. Instead of communicating with servers, all the devices (i.e., *clients*) send messages to an MQTT broker by publishing and subscribing data on selected topics. MQTT-SN (sensor network) is a variant of the MQTT used in sensor networks with some additional functionalities, such as supporting more protocol suits (e.g., ZigBee, Z.Wave). Figure A.1 overviews the MQTT protocol.

A.1.1 Protocol operations

In MQTT protocol, there are two entities involved, namely, client and broker. Broker acts similarly to server in client-server architecture and responsible for message transmission, filtering between the clients. Clients are devices

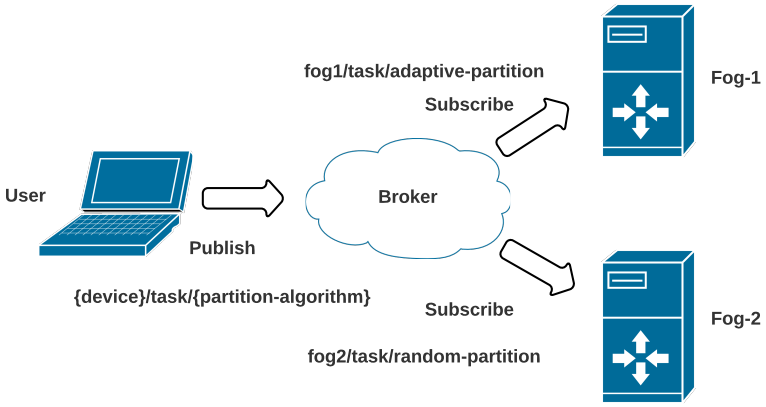

Figure A.1: An overview of MQTT protocol.

that prepare data to send and receive through a physical wired or wireless medium. Clients do not communicate directly with each other, but instead publish or subscribe to topics managed by the broker.

The broker plays an important role in the MQTT protocol: it handles the authentication process of the connected devices on the network, manages message publication, sessions, and subscription. The primary purpose of the broker is to collect and queue the messages from the client published on a topic and deliver the message towards the clients that subscribe to that topic.

Some essential properties are associated with the connection establishment between the client and the broker. To establish a connection, the client sends a CONNECT message. In response, it will receive a CONNACK message from the broker. The clean session property is set to true to start a new session with no message in the queue. Keep alive property keeps the connection open for a particular duration. To maintain connection open, the client periodically sends PINGREQ messages and the broker response with PINGRESP message until the connection expires. MQTT-SN has a feature known as sleep, which tells the broker that the client is going to sleep mode for a certain period by sending a DISCONNECT message specifying the DURATION parameter.

MQTT messages are published to the broker based on specified topics. The broker delivers the message to those recipients who subscribe to the topic. The topic may have different levels that are separated by forward slash, such as `fog1/task/adaptive-partition`. The topic is case sensitive and does not need to be pre-registered at the broker.

Recipients can use a wild card to subscribe to multiple topics that have a similar topic level. There are two wild cards for subscribing topics. The single-level is denoted by a plus sign (+), which replaces one topic level. For instance, `+/task/adaptive-partition` covers all the execution results of the partitioned task by all the fog devices. The multi-level is denoted by a hash sign (#). It replaces multiple topic levels. For example, `fog1/task/#` covers the execution result of adaptive partitioning, random partitioning, and greedy offloading.

The factor that helps the broker to regulate the message queue is known as the MQTT Quality of Service (QoS) level. The QoS level defines the guarantee of message delivery in MQTT. There are four QoS levels.

-1 : QoS -1 refers to fire and forget. It is suitable for non-critical and low power applications. The broker instantly sends the published message on a topic to the subscribed devices and removes the message. QoS -1 does not require establishing an MQTT connection. There is no acknowledgment processes in QoS -1. The sender does not resend the message if the broker deletes the message. QoS -1 focuses more on minimizing message cost and resource usage of constrained devices over reliability.

0 : QoS 0 stands for at most once. It ensures a message reaches the destination no more than once, which refers to best-effort delivery. QoS 0 requires the MQTT connection establishment before sending and receiving messages. As QoS -1, the recipient does not receive any acknowledgments, and the sender does not retry sending the message. Moreover, the broker does not store the published messages.

1 : QoS 1 refers to at least once. QoS 1 guarantees the message is received at least one time by the recipient. A critical message delivery situation takes the opportunity of QoS 1. The sender stores the published message until it receives a PUBACK acknowledgment packet from the recipient. The sender can send the same message multiple times if the recipient can handle duplicates. The broker queues the messages for later delivery if the recipient stays offline.

2 : QoS 2 provides exactly once service, which is best suited for critical or reliable delivery. In comparison with other QoS levels, it provides the most reliable communications with higher overhead. The reliable message delivery only once is ensured by using four request/response flows between the sender and the receiver, i.e., PUBLISH, PUBACK, PUBREL, PUBCOMP.

A.1.2 Evaluation

MQTT has the following advantages.

- **Simplified communication**: An MQTT payload can transport any data (binary and text) as long as the recipient can interpret it. Communicating parties do not need to stay online to carry on the data transmission. With the help of the QoS level, any offline client can receive messages after coming back online. MQTT provides a single connection to a message topic; the publisher only needs to know the broker's IP address, and the subscriber-only needs to know the topic of the published data.

- **Eliminate polling**: MQTT supports instantaneous and push-back delivery process. Therefore, the subscriber clients do not need to poll for new messages after a specified time interval.

- **Reliability and Scalability**: The QoS levels provide data transmission guarantee in different categories. The publisher-and-subscriber model can be scale up easily in an energy-efficient way. If a broker publishes a message on a topic, the subscribed clients receive the message based on the QoS level.

- **Bidirectional messaging**: MQTT provides bidirectional communication facilities. Any client can act as the subscriber and the publisher at the same time.

However, MQTT also has disadvantages.

- **Operation over TCP**: MQTT operates on the top of the Transmission Control Protocol (TCP), which requires more memory and processing power as TCP handshaking is needed to set up a connection between the clients and the broker before exchanging message.

- **Centralized broker**: The broker works as a communication hub in the MQTT protocol. It can be a single point of failure in the network: if it fails due to a shortage of power. The whole communication may disrupt. Moreover, the use of a centralized (single) broker affects the scalability due to additional overhead for each client connected to it.

- **Security**: By default, MQTT does not provide data encryption. Communication can be made secure by implementing Transport Layer Security (TLS)/Secure Sockets Layer (SSL). If TLS/SSL is implemented, additional overhead is introduced.

A.2 gRPC Remote Procedure Calls

gRPC is an open-source Remote Procedure Calls (RPC) framework that provides high performance, language and platform-independent service, bi-directional streaming facility, and integrated authentication with HTTP/2 [12]. RPC is a technique where a client calls a procedure, which may exist in different address spaces (i.e., *server*), similar to a local procedure calling. This technique is appropriate in the client-server protocol. The client calls a remote procedure, resuming the calling environment, to compute an operation by transferring procedure parameters to the server. After execution, the server generates results that are transferred back to the calling environment (i.e., the client's environment) where the execution resumes. The gRPC follows the client-server protocol. A client application can call a remote method on a server application that acts like a local method. The server application is responsible for handling the client's call. gRPC uses Protocol Buffers (Protobuf) [11] for serializing structured data. The protobuf is a language and platform-independent data interchangeable format. Specifically, a proto file defines the structure for the data for serializing. gRPC uses a special plugin, known as `protoc`, to generate code from the proto file for both the client and the server. Figure A.2 overviews the gRPC protocol.

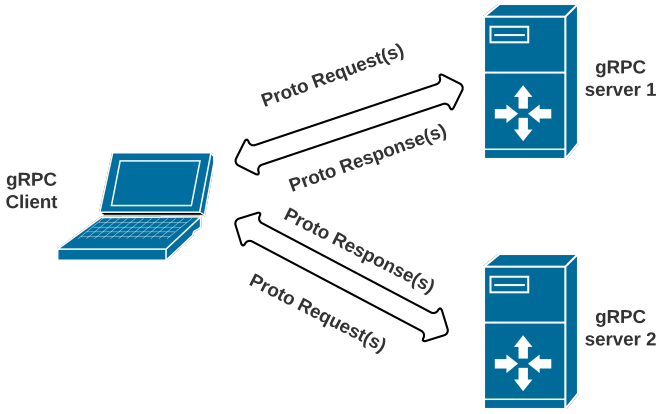

Figure A.2: An overview of gRPC protocol.

A.2.1 Protocol operations

gRPC works similarly to a regular RPC protocol. A service is defined, first specifying the methods with parameters and return types, which the clients will call remotely. An example of a proto file is given below.

```
syntax = "proto3";
message Request {
    string message = 1;
}
message Response {
    string message = 1;
}
service ServerResponse {
    rpc Node(Request) returns (Response) {}
}
```

There are four different service methods used in gRPC.

Unary RPC:
A client sends a single request to the server and gets a single response.

```
rpc Node(Request) returns (Response);
```

When the client calls a remote (stub) method on the server, the server is notified. The client invokes the remote method with its metadata, method name, and parameters. The server, in response, sends its metadata to the client. The client then sends a request message, and the server executes the defined method and populates a response along with status details to the client. If the response status is OK, the client will receive the response.

Server streaming RPC:
A client sends a single request to the server and gets a stream of messages as a response. In the case of streaming methods, gRPC guarantees the ordering of messages within an individual RPC call.

```
rpc Node(Request) returns (stream Response);
```

In this method, instead of a single response, the client receives a stream of messages. After sending all the messages, the server sends the status details and optional trailing metadata to the client.

Client streaming RPC:
A client sends a sequence of messages as a request to the server and gets a single response.

```
rpc Node(stream Request) returns (Response);
```

It works the opposite of server streaming RPC. The client sends a sequence of messages as request and the server response with a single message and the status details. Typically the server responds after receiving all the messages from the client, but it is not mandatory.

Bidirectional streaming RPC:
A client sends a sequence of messages as a request to the server and receives a sequence of messages as a response.

```
rpc Node(stream Request) returns (stream Response);
```

The client and the server stream operate independently in bidirectional streaming RPC. The server can write to received messages one by one, or wait and write after getting all the messages from the client.

Some additional parameters, along with the message, are sent to the server, such as deadlines/timeouts. The client can specify how long it will wait to complete the operation before the RPC is terminated. The client can assign deadlines, which is a fixed point in time or timeouts, which is duration in time based on language-specific APIs. Moreover, either the client or the server can cancel an RPC at any time.

Metadata are sent in key-value pair format where the keys are strings, and the values are strings or binary data. Access to metadata depends on the language that is being used.

gRPC channels help the client to connect with the server on a specified host and port. It is used when creating a client stub. It uses HTTP/2 as a transport protocol.

A.2.2 Evaluation

gRPC has the following advantages.

- **Performance**: Protobuf serializes message payload of the client and the server very efficiently and quickly. It also helps to keep payload small, which is useful for lower bandwidth applications. It uses HTTP/2, which has significant improvement over HTTP 1.X, with respect to framing and compression, duplex streaming, multiplexing over a single Transmission Control Protocol (TCP) connection.

- **Automatic code generation**: The proto file defines the services and the messages. `Protoc` is used to automatically generate a service base class, messages, and a complete client from the proto file without writing the client and response parsing code on our own.
- **Streaming**: gRPC allows bi-directional streaming of messages as part of the supported service methods discussed above.
- **Protocol specification**: gRPC follows strict specifications, which makes it consistent across different languages and platforms.

gRPC uses protobuf for serializing structured data. It encodes the message in binary format. Special tools are required to decode the payload, which is one of the disadvantages of gRPC protocol.

A.3 Protocols comparison

Figure A.3 shows comparison between MQTT and gRPC protocols based on communication latency. For the experiment, the DINA [25] is applied on the VGG16 [32] architecture. The transmission time is measured by executing one complete DNN classification task (image classification). The result depicts that the gRPC protocol outperforms the MQTT protocol by 1.5 times in terms of communication latency.

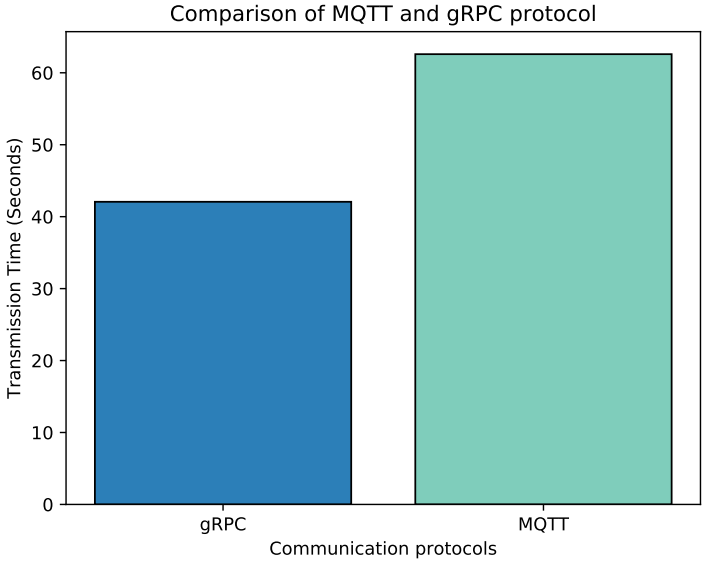

Figure A.3: Comparison of MQTT and gRPC protocols.

Appendix B

Time comparison

Appendix B shows the layer-by-layer computation and transmission time comparison of three schemes (adaptive, random, and greedy) based on four benchmarks.

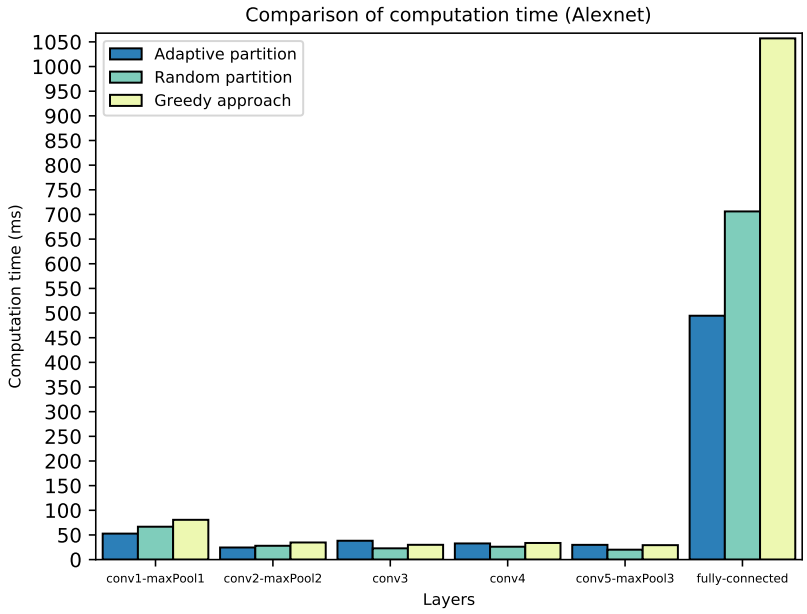

Figure B.1: Comparison of computation time (AlexNet).

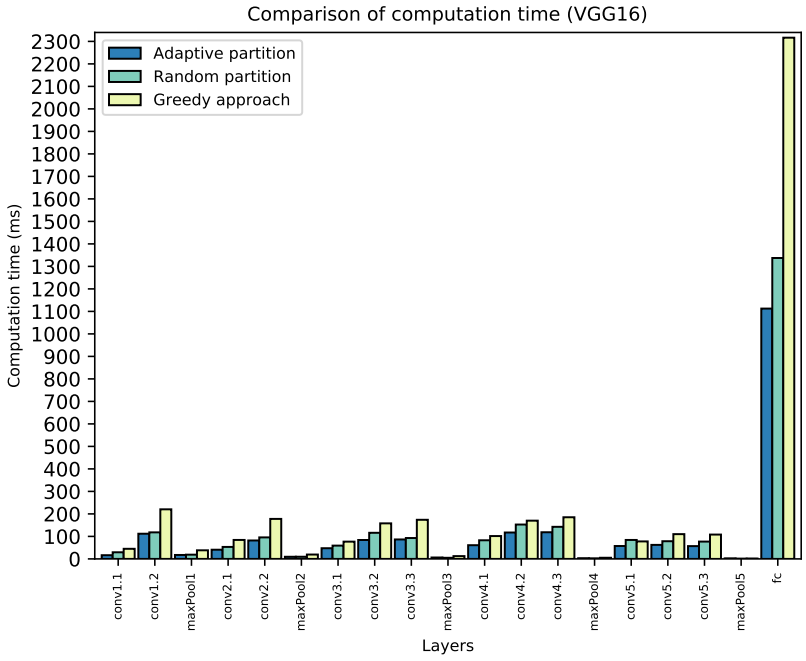

Figure B.2: Comparison of computation time (VGG16).

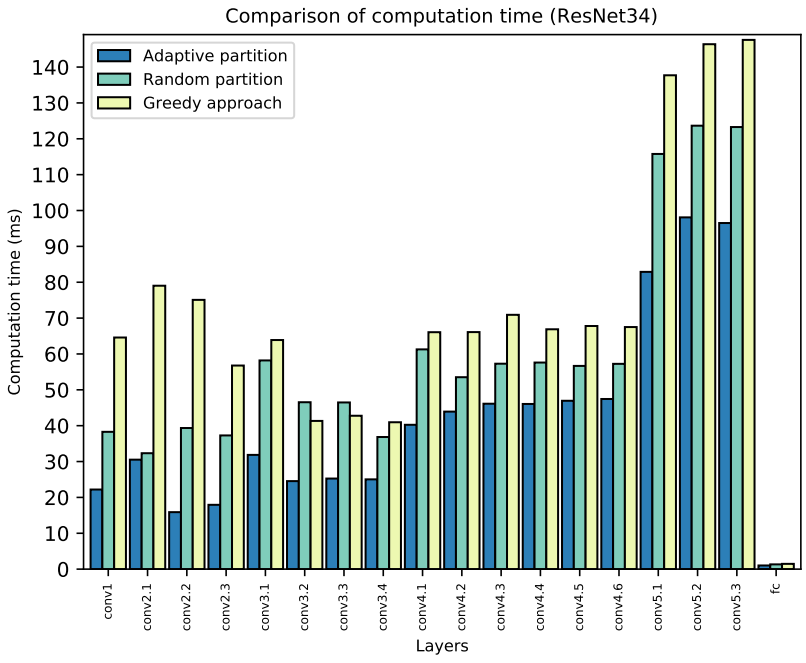

Figure B.3: Comparison of computation time (ResNet34).

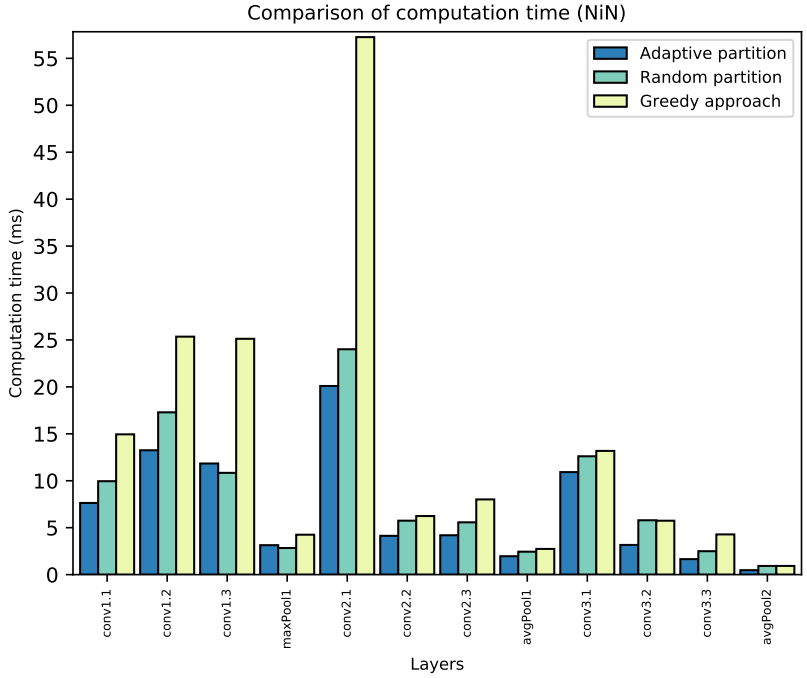

Figure B.4: Comparison of computation time (NiN).

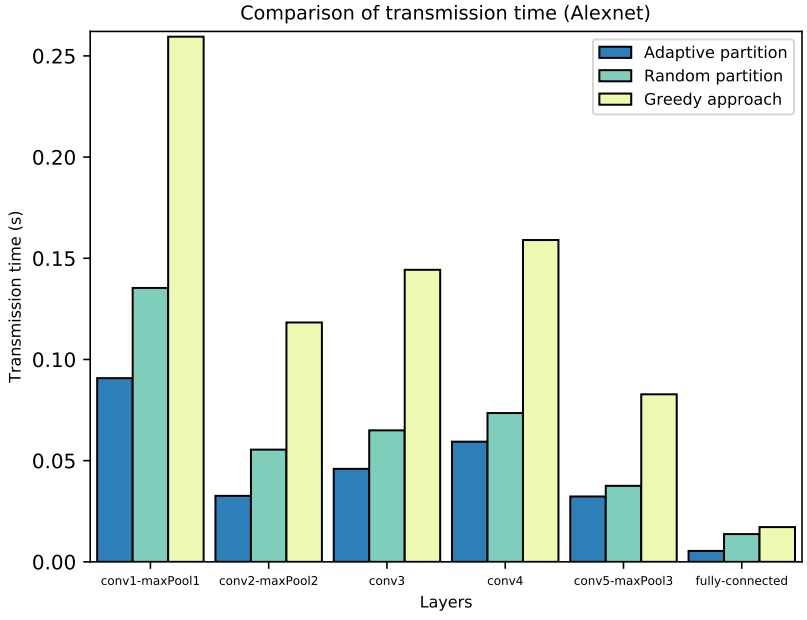

Figure B.5: Comparison of transmission time (AlexNet).

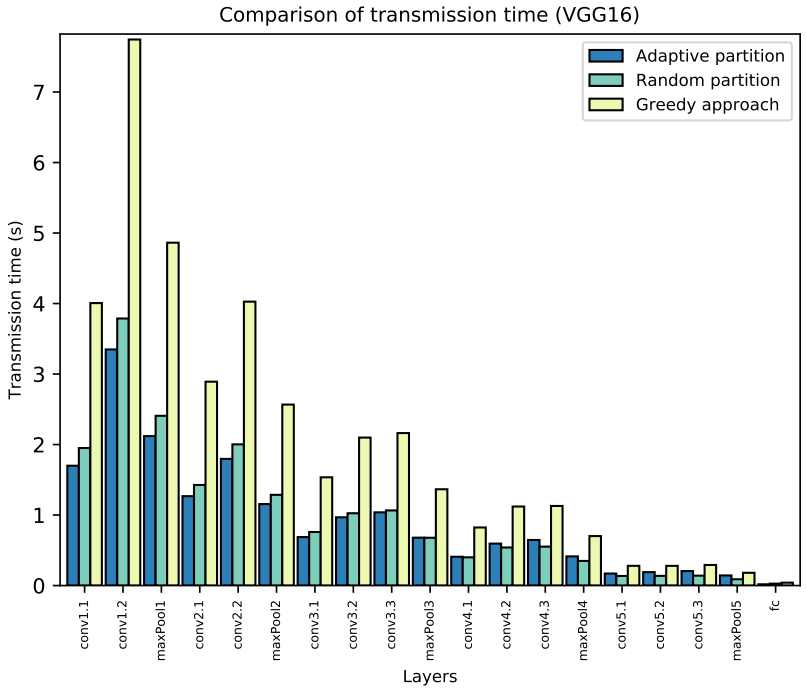

Figure B.6: Comparison of transmission time (VGG16).

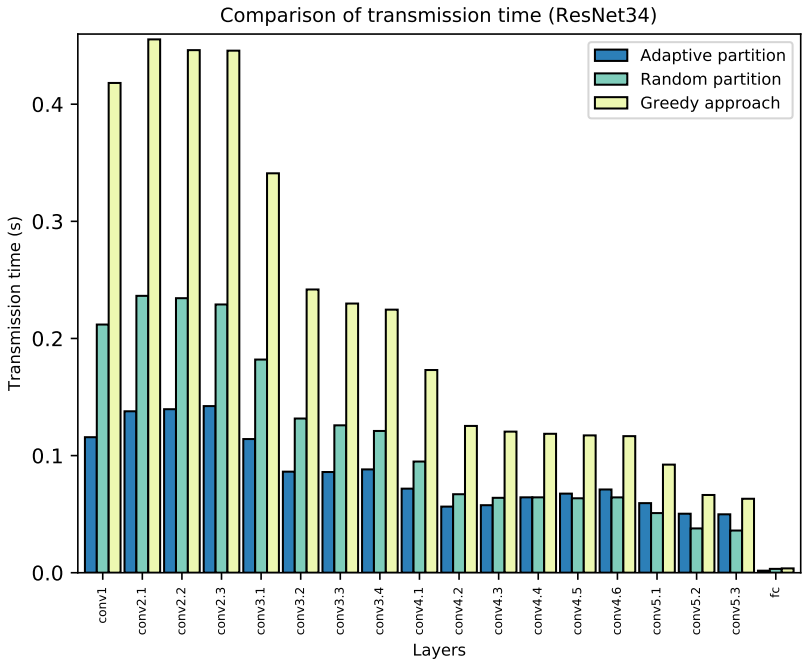

Figure B.7: Comparison of transmission time (ResNet34).

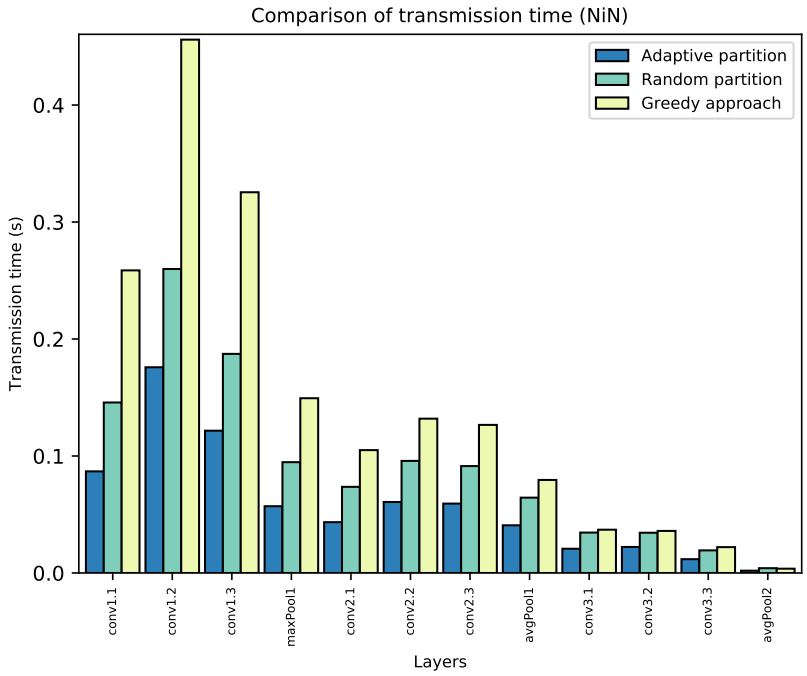

Figure B.8: Comparison of transmission time (NiN).