FAST AND EFFICIENT MODEL PARALLELISM FOR DEEP CONVOLUTIONAL NEURAL NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Burak Eserol
August 2019

Fast and Efficient Model Parallelism for Deep Convolutional Neural Networks
By Burak Eserol
August 2019

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Muhammet Mustafa Özdal(Advisor)

Cevdet Aykanat(Co-Advisor)

Hamdi Dibeklioğlu

Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

FAST AND EFFICIENT MODEL PARALLELISM FOR DEEP CONVOLUTIONAL NEURAL NETWORKS

Burak Eserol
M.S. in Computer Engineering
Advisor: Muhammet Mustafa Özdal
Co-Advisor: Cevdet Aykanat
August 2019

Convolutional Neural Networks (CNNs) have become very popular and successful in recent years. Increasing the depth and number of parameters of CNNs has crucial importance on this success. However, it is hard to fit deep convolutional neural networks into a single machine's memory and it takes a very long time to train these deep convolutional neural networks. There are two parallelism methods to solve this problem: data parallelism and model parallelism.

In data parallelism, the neural network model is replicated among different machines and data is partitioned among them. Each replica trains its data and communicates parameters and their gradients with other replicas. This process results in a huge communication volume in data parallelism, which slows down the training and convergence of the deep neural network. In model parallelism, a deep neural network model is partitioned among different machines and trained in a pipelined manner. However, it requires a human expert to partition the network and it is hard to obtain low communication volume as well as a low computational load balance ratio by using known partitioning methods.

In this thesis, a new model parallelism method called hypergraph partitioned model parallelism is proposed. It does not require a human expert to partition the network and obtains a better computational load balance ratio along with better communication volume compared to the existing model parallelism techniques. Besides, the proposed method also reduces the communication volume overhead in data parallelism by $\sim 93\%$. Finally, it is also shown that distributing a deep neural network using the proposed hypergraph partitioned model rather than the existing parallelism methods causes the network to converge faster to the target accuracy.

Keywords: Parallel and Distributed Deep Learning, Convolutional Neural Networks, Model Parallelism, Data Parallelism.

ÖZET

DERİN KONVOLÜSYONEL SİNİR AĞLARI İÇİN HIZLI VE VERİMLİ MODEL PARALELLEŞTİRMESİ

Burak Eserol
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Muhammet Mustafa Özdal
İkinci Tez Danışmanı: Cevdet Aykanat
Ağustos 2019

Konvolüsyonel sinir ağları son yıllarda çok popüler ve başarılı bir hale geldiler. Konvolüsyonel sinir ağlarının bu başarıyı elde etmesinde derinlikleri ve içerdikleri parametre sayıları önemli bir faktördür. Fakat, derin konvolüsyonel sinir ağlarını tek bir makinenin hafızasına sığdırmak zor bir hale gelmiştir ve bu sinir ağlarını eğitmek çok uzun süreler gerektirir. Bu problemi çözmek için iki adet paralelleştirme yöntemi mevcuttur: veri paralelleştirmesi ve model paralelleştirmesi.

Veri paralelleştirmesinde sinir ağları modeli bir çok farklı makineye kopyalanmaktadır ve veri bu makineler arasında bölüntülenmektedir. Her bir kopya, kendisine atanmış veriyi eğitir ve modelin parametrelerini ve parametrelerin değişimlerini diğer kopyalara gönderir. Bu süreç veri paralelleştirmesinde çok büyük bir iletişim yoğunluğuna sebep olur. Bu yoğunluk eğitim sürecini yavaşlatır ve derin sinir ağlarının sonuca yakınsamasını geciktirir. Model paralelleştirmesinde ise derin bir sinir ağı modeli farklı makinelere bölüntülenmektedir ve her bir bölüntü peşi sıra şekilde çalışmaktadır. Fakat, modelin nasıl bölüntüleneceğine karar vermek için bir uzman kişi gereklidir ve bu bölüntüleme işleminde var olan bölüntüleme yöntemlerini kullanarak düşük iletişim yoğunluğu ile birlikte düşük iş dengesizliği oranı elde etmek zordur.

Bu tezde yeni bir model paralelleştirme yöntemi olan hipergrafik bölüntülenmiş model paralelleştirme önerilmiştir. Bu yöntem bölüntüleme işlemi için uzman bir kişi gerektirmez ve var olan model paralelleştirme yöntemlerine göre daha iyi iş dengesizliği oranı ile birlikte daha iyi iletişim yoğunluğu elde etmektedir. Ek olarak, bu yeni önerilen yöntem veri paralelleştirme yönteminde ortaya çıkan iletişim yoğunluğunu $\sim$ %93 oranında azaltmaktadır. Son olarak ise önerilen

yöntemin var olan paralelleştirme yöntemlerinden daha hızlı bir şekilde sonuca yakınsadığı gösterilmiştir.

Anahtar sözcükler: Paralel ve Dağınık Derin Öğrenme, Konvolüsyonel Sinir Ağları, Model Paralelleştirmesi, Veri Paralelleştirmesi.

Acknowledgement

First of all, I would especially like to thank Assoc. Prof. Muhammet Mustafa Özdal and Prof. Dr. Cevdet Aykanat for giving me a chance to work under their supervision at Bilkent University. I would not be able to improve myself in such a great way without their guidance.

Special thanks to Dr. Hamdi Dibeklioğlu and Assoc. Prof. Süleyman Tosun for accepting to be jury members and their valuable feedback to improve this study.

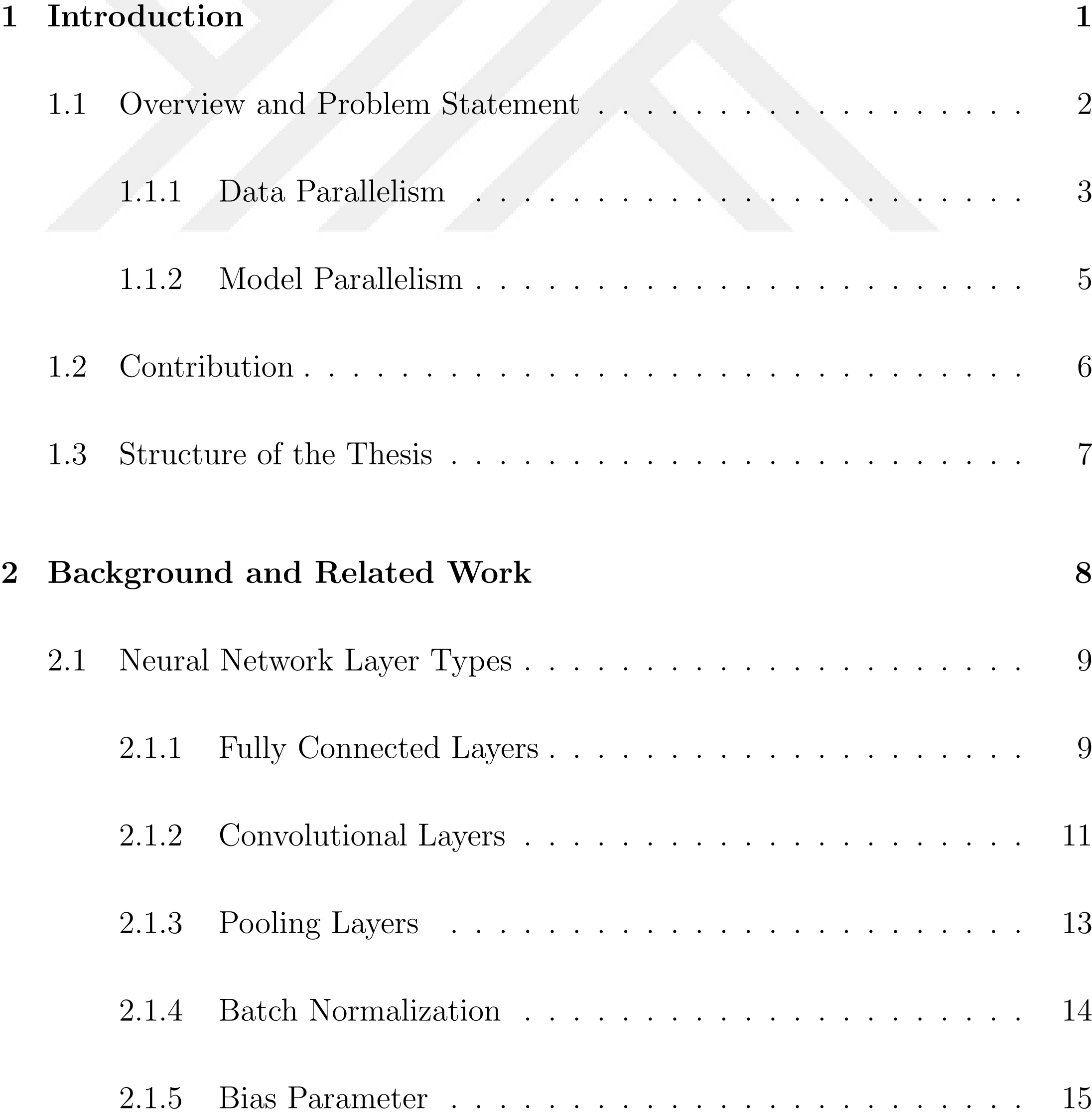
I owe my deepest gratitude to my family, especially to my mother Nevin Eserol, my father Raşit Eserol, and my brother Berk Eserol for their love, support, sacrifice, and encouragement.

I would also like to thank Barış Sesli and Onur Ünal for their friendship and encouragement during this study.

Finally, my thanks to Zeynep İdil Yel for supporting me with her love and friendship.

This work is supported by Intel and the Intel Labs vLab Program.

Contents



1 Introduction **1**

1.1 Overview and Problem Statement 2

1.1.1 Data Parallelism 3

1.1.2 Model Parallelism 5

1.2 Contribution 6

1.3 Structure of the Thesis 7

2 Background and Related Work **8**

2.1 Neural Network Layer Types 9

2.1.1 Fully Connected Layers 9

2.1.2 Convolutional Layers 11

2.1.3 Pooling Layers 13

2.1.4 Batch Normalization 14

2.1.5 Bias Parameter 15

2.2 Activation Functions and Non-Linearity 15
2.2.1 Sigmoid 15
2.2.2 Hyperbolic Tangent 16
2.2.3 Rectified Linear Unit 17
2.3 Training of Deep Neural Networks 18
2.3.1 Loss Functions 18
2.3.2 Regularization Methods 22
2.4 Convolutional Neural Networks 24
2.4.1 LeNet 24
2.4.2 AlexNet 25
2.4.3 VGG 26
2.4.4 Residual Network 27
2.5 Hypergraph Partitioning 27
2.6 Parallelism Models 28
2.6.1 Data Parallelism 29
2.6.2 Model Parallelism 30

3 Modeling Communication and Computation 33

3.1 Graph Representation 33
3.2 Modelling Communication 35

3.3 Fine-grain Hypergraph Representation 38

3.4 Coarse-grain Hypergraph Representation 39

3.5 Hypergraph Partitioning Details 41

4 Asynchronous Pipelined Model Parallelism 45

4.1 Collocating Gradient Operations 45

4.2 Threaded Training of Model Parallelism 47

5 Experiment Results 50

5.1 Computational Load Balance and Communication Volume Analysis 50

5.1.1 Layer-wise Partitioning 51

5.1.2 Filter-wise Horizontal Naive Partitioning 53

5.1.3 Filter-wise Incremental Naive Partitioning 54

5.1.4 Channel-wise Incremental Naive Partitioning 55

5.1.5 Coarse-grain Hypergraph Partitioning 57

5.1.6 Fine-grain Hypergraph Partitioning 58

5.1.7 Data Parallelism 59

5.1.8 Hypergraph Partitioning Based Model Parallelism vs Data Parallelism 60

5.2 Run Time Analysis 61

5.2.1 Single Processor 62

5.2.2 Model Parallelism 63

5.2.3 Data Parallelism 63

5.2.4 Hypergraph Partitioning Based Model Parallelism vs Data Parallelism 65

5.3 Accuracy and Loss Convergence Results 66

6 Conclusion **69**

6.1 Future Work 70

A Hypergraph Partitioning Statistics **77**

List of Figures

1.1 Number of Layers and Associative Error Rates for Different Deep Neural Networks in ImageNet Challange [13] 2

1.2 Synchronous and Asynchronous Data Parallelism 4

1.3 Model Parallelism [7] 5

2.1 Visualization of Fully Connected Layers 10

2.2 Visualization of Convolution Operation 12

2.3 Visualization of Convolutional Layer 13

2.4 Visualization of Max Pooling Layer 14

2.5 Sigmoid Function 16

2.6 Hyperbolic Tangent Function 17

2.7 Rectified Linear Unit Function and Variations 17

2.8 Gradient Descent Visualization 20

2.9 Dropout Visualization 23

2.10 Skip Connection [13] 27

3.1 Fine and Coarse-grain CNNs Graph Representations 34

3.2 Fine-grain and Coarse-grain Hypergraph Representations of the CNNs 36

3.3 Fine-grain Hypergraph 38

3.4 Coarse-grain Hypergraph 40

3.5 Sample Input File for Patoh 43

4.1 Model Parallelism Working Schema 46

4.2 Model Parallelism Working Schema - Illustration 46

4.3 Model Parallelism with Collocated Gradients Working Schema . . 47

4.4 Model Parallelism with Collocated Gradients Working Schema - Illustration 47

4.5 Model Parallelism with Collocated Gradients Working Schema . . 48

4.6 Threaded Model Parallelism with Collocated Gradients Working Schema 48

4.7 Threaded Model Parallelism with Collocated Gradients Working Schema - Illustration 49

5.1 Node Specifications 67

5.2 VGG16 Training Accuracy and Loss 67

List of Tables

2.1 LeNet Structure 25
2.2 AlexNet Structure 25
2.3 VGG16 Structure 26

5.1 Layer-wise Partitioning Computational Load Balance Analysis . . 52
5.2 Layer-wise Partitioning Communication Volume Analysis 52
5.3 Filter-wise Horizontal Naive Partitioning Computational Load Balance Analysis 53
5.4 Filter-wise Horizontal Naive Partitioning Communication Volume Analysis 53
5.5 Filter-wise Incremental Naive Partitioning Computational Load Balance Analysis 54
5.6 Filter-wise Incremental Naive Partitioning Communication Volume Analysis 54
5.7 Channel-wise Incremental Naive Partitioning Computational Load Balance Analysis 56

5.8 Channel-wise Incremental Naive Partitioning Communication Volume Analysis . . . 56

5.9 Coarse-grain Hypergraph Partitioning Computational Load Balance Analysis . . . 57

5.10 Coarse-grain Hypergraph Partitioning Communication Volume Analysis . . . 57

5.11 Fine-grain Hypergraph Partitioning Computational Load Balance Analysis . . . 59

5.12 Fine-grain Hypergraph Partitioning Communication Volume Analysis . . . 59

5.13 VGG16 Memory and Parameter Analysis . . . 60

5.14 VGG16 Communication Volume Comparison for 1 Epoch . . . 62

5.15 VGG16 Per Epoch Run Time Analysis of Different Parallelism Methods in Seconds . . . 65

A.1 Coarse-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.03 . . . 77

A.2 Coarse-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.15 . . . 77

A.3 Coarse-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.2 . . . 78

A.4 Fine-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.03 . . . 78

A.5 Fine-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.15 . . . 78

A.6 Fine-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.2 . . . 79

A.7 Coarse-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.03 . . . 79

A.8 Coarse-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.15 . . . 79

A.9 Coarse-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.2 . . . 80

A.10 Fine-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.03 . . . 80

A.11 Fine-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.15 . . . 80

A.12 Fine-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.2 . . . 81

Chapter 1

Introduction

Neural networks process the given information through layers that contain a large number of neurons similar to the human brain. Neurons in a layer of the network get information from the neurons of the previous layer. Activated neurons based on the given information pass their output to the next neuron. This communication process of neurons results in different actions at the end of the network. Influenced by the human brain, neural networks have been studied by scientists for a long period of time. There are many research works about different types of neural networks from 1940s [29, 14] until today [31, 34]. Although neural networks are an old idea, they have become popular in the last decade. The popularity of neural networks comes from its success in various areas under the field called Deep Learning. Deep Learning is a sub-field of Machine Learning that enables neural networks to learn representations of the given data examples. These learned representations help neural networks to perform different tasks. Until now, many different tasks have been achieved successfully by Deep Learning such as image classification [21, 37], visual recognition [35, 45, 32], language understanding [8], disease detection [10] and many more.

Another reason behind the success of Deep Learning is technological advances. Today's neural networks with millions of parameters require high computational power. This computational power was not available for a long period of time.

With the help of technological advances, processing limitations for neural networks started to vanish. However, there are still processing limitations for deep neural networks that consist of a high number of layers and millions of parameters. Even today, it could take weeks to train a deep neural network, and high-tech companies are still trying to overcome processing limitations by working on specialized chips [4]. In this work, a new solution is proposed to overcome processing limitations by presenting a new way to distribute a deep neural network among different processors.

1.1 Overview and Problem Statement

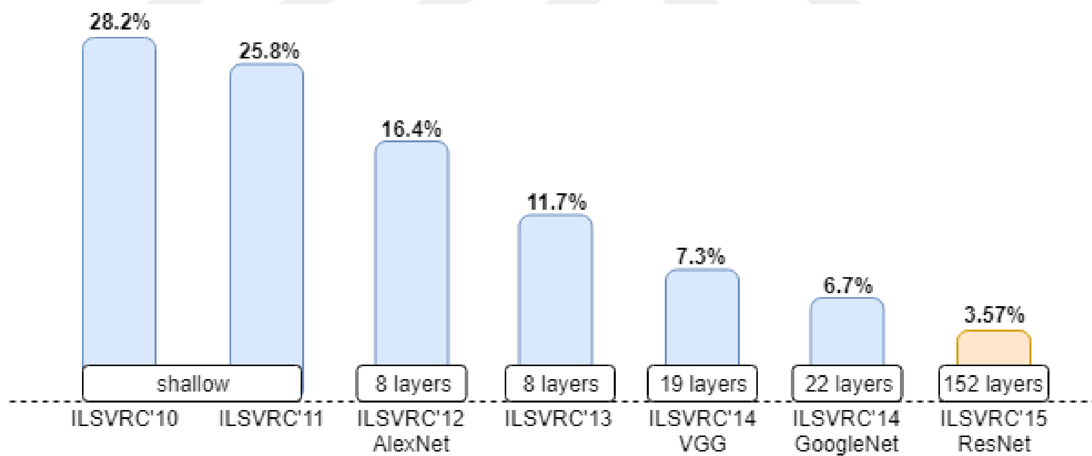


Figure 1.1: Number of Layers and Associative Error Rates for Different Deep Neural Networks in ImageNet Challange [13]

The depth of the deep neural networks is an important parameter that can affect the final accuracy of the network. Throughout the years, increasing the depth of the neural networks resulted in better accuracies. One good example of this is The ImageNet Large Scale Visual Recognition Challenge (ILSRVC) [33]. This challenge evaluates different deep neural network algorithms for object detection and image classification. Throughout the years, different deep neural networks achieved lower error rates. In Figure 1.1, the number of layers in different deep

neural network algorithms are shown. The height of the bars represents the error rate of the given CNN architecture. In the early years of the competition, shallow networks are used that contain a low number of layers such as 8. Throughout the years, the number of layers is increased in the different deep neural networks and achieved better results and reached 152 layers with ResNet [13]. However, increasing the number of layers also requires much more computational power and memory space. Alexnet [21] from ILSVRC'12 consists of 8 layers and 62 millions of parameters. VGG16 [35] consists of 16 layers and 138 millions of parameters. Deep neural networks with a large number of parameters may not fit into a single machine's memory. One possible solution is to reduce the batch size so that the amount of data that is passing from the network together is low. Another solution is distributed training when the amount of data is huge or the model has a very large number of parameters. There are two distributed training methods: data parallelism and model parallelism.

1.1.1 Data Parallelism

Data parallelism is a widely used parallelism technique for distributed training. In data parallelism, the neural network model is replicated among different processors. Each processor has the same copy of the model as shown in Figure 1.2. In every iteration in the training process, each worker uses a different part of the input data and computes gradients for the model. Each gradient calculated in the workers is sent to another device which is called the parameter server. The parameter server also stores the same copy of the model and applies gradients to the model to get updated weights. Updated weights at the parameter server are then broadcast to each worker and a new iteration of the training continues. It is also possible not to use the parameter server as an additional device. In this technique of data parallelism, parameters are distributed among processors where the model is also replicated. Thus, each processor is responsible for calculating the gradients of a subset of parameters and sending them to other replicas.

Besides, there are two different ways of applying data parallelism. In synchronous

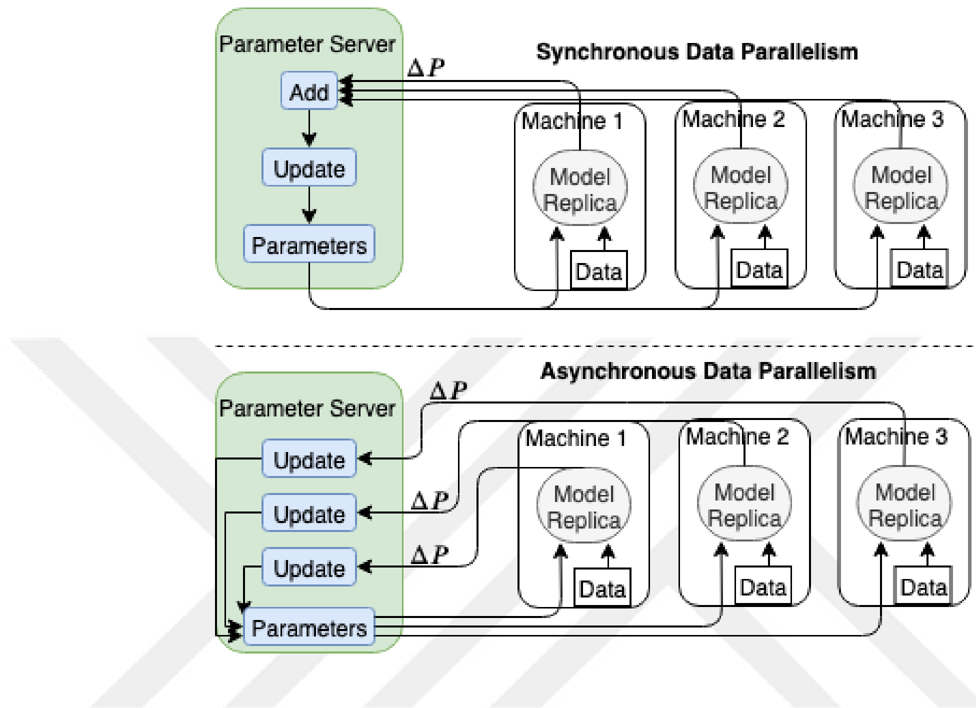


Figure 1.2: Synchronous and Asynchronous Data Parallelism

data parallelism, all of the workers wait for each other to compute gradients and send them to the parameter server. Updated weights are sent back to workers and they continue the training process in a synchronous way when all of the replicas obtain the updated weights. Computation in this method is completely deterministic and convergence can be better as each mini-batch from input data is trained using updated waits. However, each worker needs to wait for other workers to finish to send gradients to the parameter server. Therefore, it may take a long time to train the neural network. In contrast, in asynchronous data parallelism, workers don't wait for each other to send gradients to the parameter server. Therefore, it is a faster method while there can be a convergence problem as weights used by workers are not up-to-date all the time.

The most important drawback of data parallelism is the communication overhead. All of the weights are needed to be communicated for each worker after each iteration. Also, the parameter server needs to send updated weights to the workers before each iteration. The amount of communication in data parallelism is huge and it increases as we increase the number of workers. Therefore, it is only useful

when we have a small model with a huge amount of data.

1.1.2 Model Parallelism

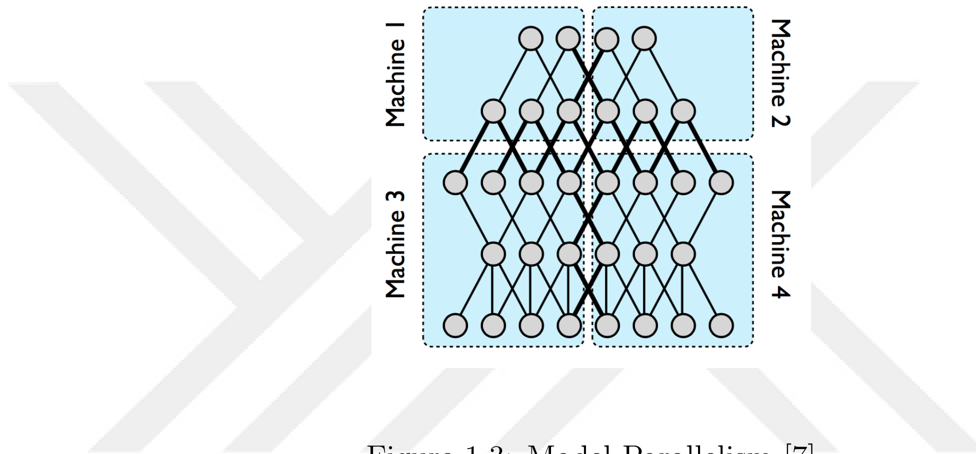


Figure 1.3: Model Parallelism [7]

In model parallelism, different processors are responsible for computation associated with different parts of the network. Each layer of the neural network can be assigned to a different processor or each half of a layer can be assigned to a different processor. During the forward pass phase of the training, each processor is responsible for the computations for its part. After the computation is done in a processor, activations are communicated to the responsible processor that contains the connected nodes to these activations. During backward pass, gradients of the weights are communicated among processors so that each processor can update its weights. Model parallelism is rarely examined in the literature and the distribution of the model to multiple processors constitutes a challenging problem. Keeping the computational load balance among processors with low communication overhead is hard to achieve in model parallelism. Therefore, most of the research is done on data parallelism.

In this work, a solution for the distribution problem of model parallelism is proposed. In the proposed solution, a deep neural network is represented as a hypergraph. Then, a hypergraph partitioning method is proposed for distributing the

network to the processors. In the proposed method, the partitioning constraint encodes the computational load balance among processors whereas the partitioning objective encodes the minimization of total communication volume among processors.

1.2 Contribution

This thesis proposes a new model parallelism technique for deep convolutional neural networks. The proposed model parallelism technique contains the following contributions:

- A new idea of representing a deep convolutional neural network as different hypergraph models. For each different hypergraph model, communication patterns are shown for different atomic task definitions.
- It does not require a human expert to decide how to partition the deep convolutional neural networks. Hypergraph partitioning automatically partitions the network so that resulting partition has low computational load balance and communication volume.
- Besides the known model parallelism partitioning method where a group of layers is distributed among different processing units, it proposes new partitioning methods called fine-grain hypergraph and coarse-grain hypergraph. Those proposed new methods achieve better computational load balance and communication volume than known methods.
- It proposes a threaded asynchronous model parallelism training to solve the staleness problem of model parallelism.
- It reduces the communication volume that exists in data parallelism by ∼93% on average.
- It speeds ups the training process by ∼3x using 4 processors compared to the single processor training.

- It obtains faster convergence than any data parallelism technique.

1.3 Structure of the Thesis

The structure of the rest of this thesis is as follows. Chapter 2 gives background information about deep neural networks, how they are trained, and hypergraph partitioning as well as summarizes the related works. In Chapter 3, information about how to represent a deep convolutional neural network as a hypergraph is given. Computational load balance and communication volume analysis of different parallelism techniques for distributed deep convolutional neural network models is given in this chapter. Then, Chapter 4 provides information about how to run asynchronous pipelined model parallelism and run time analysis of different parallelism techniques for distributed deep convolutional neural network methods. Next, Chapter 5 presents the accuracy and loss of convergence results of different methods. Finally, Chapter 6 concludes the thesis and gives information about possible future works.

Chapter 2

Background and Related Work

Deep Neural Networks consist of layers where each layer contains neurons. Neurons in the layers are composed of weights and biases. When a neuron receives an input, it applies multiply and add operations to the input using its weights and bias. The way those multiplication and addition is applied is based on the type of the layer. Besides, the type of the layers differs in terms of not only operations but also neuron connectivity patterns. For each different type of network, there might be a different connectivity pattern for neurons in successive layers.

When a neuron produces its output, it may apply a non-linearity to its output before passing it to the next neuron. To apply a non-linearity, layers use activation functions. One purpose of using activation functions is to deactivate some of the connections in the network if that specific information is not important for the final decision of the network. Another purpose is to increase the importance of the connections that contain important information for the final decision of the network. Activation functions are one of the most important parts of the neural networks. They introduce non-linearity, thus neural network learns and makes sense of complicated and complex functional mappings.

After one iteration of the given data through the network, the prediction of the network is obtained. The training procedure for a network consists of multiple

iterations. After each iteration of the training, weight, and bias values are updated in the layers so that they can learn information from the given data. The updating procedure of weights and biases is called back-propagation. The main purpose of back-propagation is to update parameters in the network so that the predicted output of the network is close to the expected output. If the predicted output of the network is far from the expected output, then there will be high loss value based on the type of loss function. Loss value is the metric that shows how far the network is from predicting accurate results. There could be different loss functions based on the data and the problem.

General working principles of a deep neural network and important factors are given above. The principles and important factors above may work differently for different types of deep neural networks. There are many different types of deep neural networks such as Recurrent Neural Networks, Long-Short Term Memory Networks, Multilayer Perceptrons, Convolutional Neural Networks and more. The working principles of deep neural networks given above will be explained in detail in the next sections. Also, there will be more detailed information about Convolutional Neural Networks (CNNs) and how they work as algorithms described in this work are more focused on CNNs.

2.1 Neural Network Layer Types

2.1.1 Fully Connected Layers

Fully connected layers apply linear transformations to the given input vector before applying activation functions and have full connection to the neurons in the previous layer. All of the neurons in the fully connected layer are multiplied by neurons in the previous layer and if there is a bias in the fully connected layer, it is added to the resulting linear transformation. We can formulate the operation done by a fully connected layer as follows. Let us say the weights of the fully connected layer are denoted by W and have an input dimension of k. Then we

can formulate the operation done by a fully connected layer with respect to the given input x as follows.

$$y = W \times x + b$$
$$y_i = \sum_{j=1}^{k}(W_{ij} \times x_j) + b_i$$

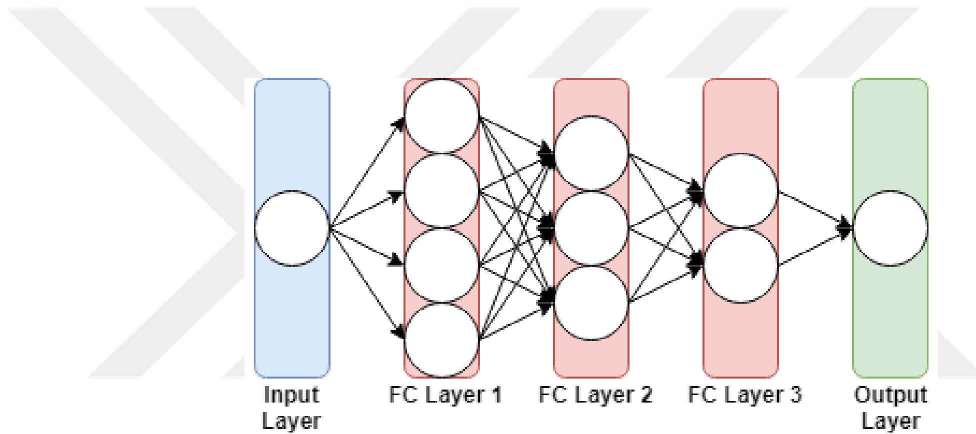


Figure 2.1: Visualization of Fully Connected Layers

In Figure 2.1, connectivity pattern of fully connected layers is shown. The fully connected layers with red color are also called hidden layers. Hidden layers 1, 2, and 3 in the representation are fully connected layers and neurons in those layers apply a linear transformation to the neurons in the previous layer. For hidden layer 1, the input dimension is 1 and the output dimension is 4 as it includes 4 neurons. Similarly, the input dimension of hidden layer 2 is 4 since the input for hidden layer 2 contains 4 neurons. The output layer is also fully connected. This is because it is connected to all of the neurons from the previous layer and outputs 1-dimensional result.

In convolutional neural networks, fully connected layers are usually used at the end of the network after a series of convolutional layers. Convolutional layers which will be discussed next are good at subtracting local information from the data and connecting that information with fully connected layers at the end of

the network is good for generalizing the local information and obtaining a result with the given number of neurons.

2.1.2 Convolutional Layers

Convolutional layers are used widely in deep neural networks. They are the core information extractors, and most of the computations done in deep neural networks are done on these layers. A simple convolutional layer is composed of a set of filters and a bias parameter. Filters are used to extract local information from the input such as images. At the beginning of the training, the weights of filters are set randomly. During the training process, those filters are updated and their weights are computed (learned) so that they can detect features from the input images. Extracted features from first convolutional layers are low-level information such as edges, and extracted features from later convolutional layers are high-level information such as the wheel of a car.

As data flows through the network which is also called forward-pass, the filter of convolutional layers is slid across the width and height of the input volume. During this sliding process, the dot product of the filter and the current slide of the input is calculated. Resulting dot products are also called activation maps or feature maps which represent the response of the filter at every spatial position. Eventually, the network will learn to activate specific filters as it detects extractable information on the image.

As shown in Figure 2.2, let us say we have an image of size [32x32x3], where the first and second dimensions represent width and height, and the third dimension represents color channel dimension of the image such as RGB (Red-Green-Blue). If we a have filter with dimension [4x4] (width and height), each neuron in the convolutional layer will have a weight tensor with dimension [4x4x3]. The last dimension is 3 because input has 3 channels and a different filter is needed for each channel of the input. Furthermore, if the convolutional layer consists of 32 neurons, the dimension of the layer becomes [32x4x4x3], where each parameter is trainable.

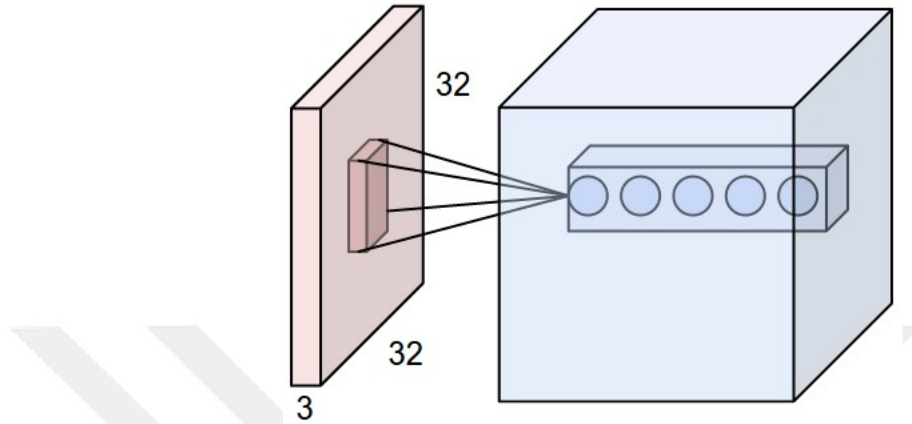


Figure 2.2: Visualization of Convolution Operation

As mentioned earlier, the filter is slid across the input image and the output dimension is dependent on the sliding length, which is also called stride. When stride is set to 1, filter slides on one pixel at a time for each dot product operation. When the stride is set to n, filter slides on n pixels after each dot product operation. Increasing the stride value will result in smaller output volumes as the filter skips a higher number of pixels. Another parameter for the convolutional operation is called padding. This parameter pads zeros around the border of the input to preserve the spatial size of the input volume so that the input and output width and height are the same. Both stride and padding are hyperparameters for convolutional layers, and they need to be optimized based on the problem.

The sliding of convolutional filters can be applied to different dimensions based on the input. 1-dimensional (1D) convolution implies that sliding of the filter is done on 1-axis, which can be time. The output of 1D convolutional is a 1D array. 2D convolution implies that sliding of the filter is done on 2-axes (x,y), and outputs 2D matrix. 3D convolutional implies that sliding of the filter is done on 3-axes (x,y,z) and output has 3D volume. In most of the convolutional layers, 2D convolutions are applied on 3D data as there is a filter for the third dimension. On 3D data, the 2D filter is slid across the x and y-axis. Therefore, the output is also a 2D matrix with 3D volume.

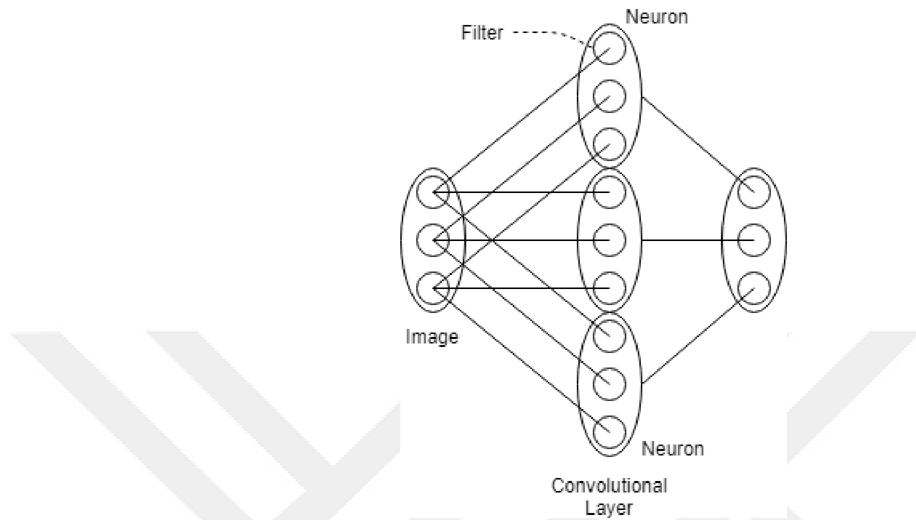


Figure 2.3: Visualization of Convolutional Layer

As shown in Figure 2.3, the connectivity pattern of convolutional layers also differs from fully-connected layers. Convolutional layers are locally connected layers. If the input image dimension is [32x32x3], which contains 3 color channels, and a neuron inside the convolutional layer contains [4x4] filters, there will be a different filter for each channel in the total dimension of [4x4x3]. Each [4x4] filter inside a neuron is convolved with a single channel of the input image. Therefore, there is a local connectivity pattern in convolutional layers, instead of connecting each filter with each channel of input which is seen at fully connected layers.

2.1.3 Pooling Layers

Pooling layers are also widely used in convolutional neural networks. The main purpose of using pooling layers is to reduce the spatial dimension of the feature maps. Thus, the amount of data that is transferred between layers will be less and there will be less amount of communication. While pooling layers reduce the dimension of the feature maps, they also extract meaning and powerful information from small slides of the data.

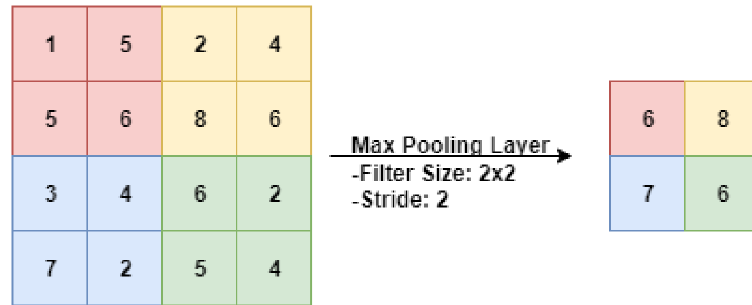


Figure 2.4: Visualization of Max Pooling Layer

There are several types of pooling layers such as max pooling, average pooling, and sum pooling. Max pooling layer is the one that is generally used in convolutional neural networks to reduce the dimension of the feature maps. In Figure 2.4, max-pooling layer is visualized. Let's say we have a max-pooling layer with a filter size of [2x2] and stride of 2. Then, the maximum value of each [2x2] part of the feature map is taken. Since we have stride value as 2, the filter is slid 2 pixels after each pooling operation on both axes. In Figure 2.4, input size [4x4] is reduced to [2x2] after the max-pooling layer and only important features are extracted while reducing the dimension. Similarly, sum pooling applies summation operation to the slices of input instead of extracting maximum value. Besides, since average pooling takes an average of each slice, it can be considered as a smoothing operation.

2.1.4 Batch Normalization

The batch normalization layer adds a normalization step between layers. Although batch normalization layers are recently proposed [17], they have become very popular and they are widely used in deep neural networks. This layer reduces internal covariate shift in neural networks and applies zero mean unit variance normalization by subtracting mean from the output of a layer and dividing into standard deviation. Batch normalization layers enable the use of high learning rates while avoiding gradient saturation which happens in activation functions

such as tanh and sigmoid (will be discussed in the next sections).

2.1.5 Bias Parameter

Bias parameters are the values added to the dot product of input and weights of the layers. These parameters allow activation functions to shift to the left or right so that data can fit better. Bias parameters do not have connections to the input data and the previous neurons. They only affect the output of layers.

2.2 Activation Functions and Non-Linearity

Activation functions are one of the crucial operations for deep neural networks to learn representations and patterns from the input. Their main purpose is to introduce non-linear complex function mappings between input and output. Activation functions apply non-linearity to the output of a layer so that the next layer only gets important features from the previous layer. It is necessary to use activation functions in deep neural networks. This is because, without activation functions, a neural network becomes just a linear function with a limited learning power. Different types of activation functions can be used for different purposes. All of the different functions have different advantages and disadvantages that we should be aware of while designing a deep neural network. Bias parameters are the values added to the dot product of input and weights of the layers. These parameters allow activation functions to shift to the left or right so that data can fit better. Bias parameters do not have connections to the input data and the previous neurons. They only affect the output of layers.

2.2.1 Sigmoid

The sigmoid function maps the given input to a range between 0 and 1 as seen in Figure 2.5. Therefore, it can be a good choice for the layers that predict

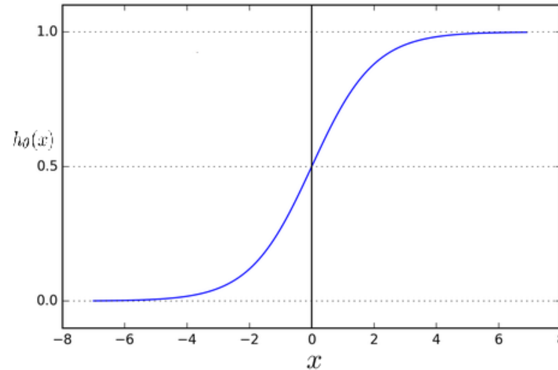


Figure 2.5: Sigmoid Function

probability since probability also ranges between 0 and 1,

$$h_\theta(x) = \frac{1}{1+e^{-x}}$$

Since the sigmoid function is differentiable, we can find it's slope and use it in gradient descent during the training phase, which will be discussed in the next section. The sigmoid function may have several disadvantages. The sigmoid function may suffer from vanishing gradient problem. As gradients flow back through the network, gradients for early layers may vanish. As earlier layers are important for extracting information from the input, the learning process may slow down. Besides, since sigmoid function outputs between 0 and 1, strongly negative inputs become zero. This may also cause the training process to get stuck. This is because deep neural networks use activations of the layers to calculate parameter gradients and this can result in model parameters that are updated less regularly than we would like.

2.2.2 Hyperbolic Tangent

Unlike sigmoid function, hyperbolic tangent outputs between -1 and 1 as shown in Figure 2.6. Therefore, there will be a difference between the mapping of negative values and strongly negative values. Also, only near-zero values will be

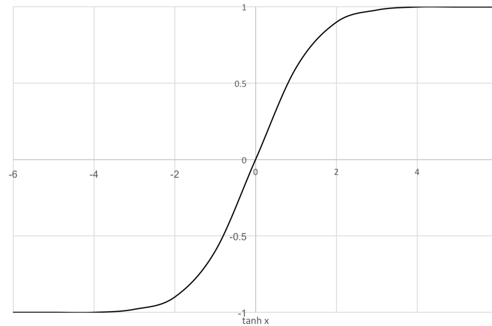


Figure 2.6: Hyperbolic Tangent Function

mapped to zero instead of all negative values. Thus, getting stuck in the training process is less likely to happen. Hyperbolic tangent can also be considered as a scaled version of the sigmoid function. Although it can find a solution for getting stuck during training, the hyperbolic tangent function also suffers from vanishing gradient problem as gradient may become very small for earlier layers

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1,$$
$$tanh(x) = 2 * h_\theta(2x) - 1.$$

2.2.3 Rectified Linear Unit

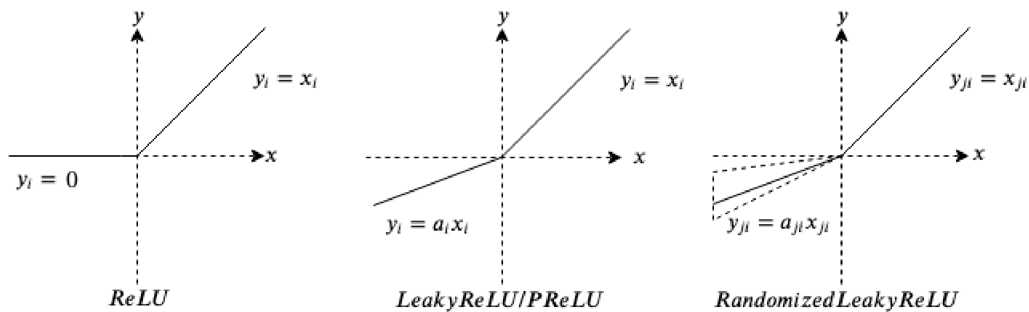


Figure 2.7: Rectified Linear Unit Function and Variations

Rectified Linear Unit (ReLU) is another activation function and a simpler one. ReLU outputs a maximum value of 0 and the value of the input as shown in Figure 2.7.

$$y(x) = max(0, x)$$

ReLU is good in terms of differentiating between positive and strongly positive outputs. This shows which neuron is highly important for the given input. Similar to sigmoid, negative outputs become 0 in ReLU and gradients will be zero for those connections. Those connections will stop responding to the learning process. This problem is also referred to as the Dying ReLU problem [27]. There are different types of ReLU functions in order to solve this problem. In Figure 2.7, Leaky ReLU is shown where negative outputs of a layer are represented as smaller negative values. The main idea of variations of ReLU function is to make gradients non-zero for negative activations [41].

ReLU is the most used activation function in deep neural networks. It also increases the training speed due to its simple equation. However, this does not mean that we should use ReLU as an activation function all the time. The activation function should be chosen based on the characteristics of the network. It is also possible to create custom activation functions.

2.3 Training of Deep Neural Networks

2.3.1 Loss Functions

Loss functions are used to learn how far the network predictions are from the truth. Based on the loss value, weights inside the network are updated so that better results can be predicted. At the beginning of the training, the weights of the neural network are set randomly. At first, we expect the network to perform badly as it has random weights. Eventually, based on the loss function, weights

are updated so that it can perform better predictions. In other words, besides predicting accurate results, the main purpose of the neural network is to minimize the loss function. This process is the most significant part of training a neural network.

Based on the network's purpose, different types of loss functions can be used. One of the basic loss functions is the mean square error.

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

During the training, each data in the training dataset is passed from the network. In the equation above, y denotes the true value for the given data, and $\hat{y}$ denotes the prediction of the network for the given data. The square of the difference between the true and the predicted value for each data gives us the mean square error for the network. Neural networks can be trained using multiple data samples at the same time, where data samples are referred to as batch. For a batch, the average loss is calculated by summing the loss for all data samples and dividing by the number of samples. The loss values calculated for a batch are used to update the weights of the network so that there will be more accurate predictions.

Various loss functions can be used in neural networks based on the network's purpose. Some of them are basic and have similar ideas to each other such as mean square error, L2 error (where we don't divide the sum of losses into n), mean absolute error, mean absolute percentage error, etc. Those loss functions are generally used in regression problems where the network predicts a real number like the age of a person or the value of a stock. There is also another problem called classification. In the classification problem, the network predicts a class from a discrete number of possible classes such as predicting a dog among dog and cat images. The most frequently used loss function for the classification problem is cross-entropy. For the binary classification task, where there are 2 possible labels that the network can predict, binary cross-entropy loss is defined as follows:

$$Loss(y, \hat{y}) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

Cross entropy measures the divergence between two different probability distributions. High cross-entropy value means that there is a big difference between two distributions and it should be minimized more. For the classification problems where there are more than 2 possible classes, categorical cross-entropy loss is used. Several more loss functions can be used based on the problem and it is possible to create custom loss functions.

Optimization algorithms calculate the gradients of the weight by taking the partial derivative of the weight with respect to loss. Gradient value for a weight shows how much it should change to obtain lower loss values. In Figure 2.8, visualization of gradient descent can be seen. There is a parameter called the learning rate or learning step. In each iteration of gradient descent, the main aim is to find the global minimum point that minimizes the loss function. Big learning step values can cause reaching the global minimum earlier, while also there is a risk of never reaching the global minimum. For a small learning rate, there is a risk of getting in the local minimum. Therefore, the learning rate should be chosen carefully based on the problem. There are variants of gradient descent algorithms such as batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.

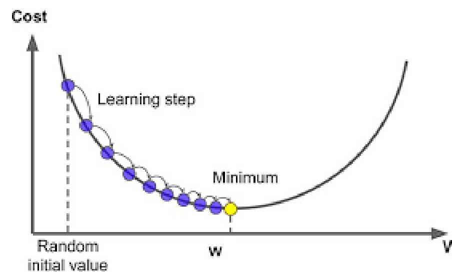


Figure 2.8: Gradient Descent Visualization

Batch Gradient Descent

Previously, it is shown that loss functions are used to calculate the performance of the neural network in terms of accurate predictions. In the batch gradient descent, cost (the result of the loss function) is calculated for all of the data in the training set for once. Then, batch gradient descent computes the gradient of the cost for each weight and bias parameter in the network. As one update is done for all the data in the training set, the batch gradient descent algorithm is slow and may cause memory problems for networks with a large number of parameters. Batch gradient descent is guaranteed to find the global minimum for convex surfaces and the local minimum for non-convex surfaces.

Stochastic Gradient Descent

Unlike Batch Gradient Descent, Stochastic Gradient Descent (SGD) updates weights for each data in the training set. Therefore, SGD is a much faster algorithm. However, since SGD updates the network for each data in the training set, there is a high variance between different updates.

Mini-Batch Gradient Descent

Mini-batch Gradient Descent takes advantage of the previously explained algorithms. It performs an update on parameters for each mini-batch of the training set. The mini-batch contains several data points from the training set and the size of a mini-batch can be set. Mini-batch Gradient Descent reduces the variance between updates from SGD. It is widely used in deep neural networks and the batch size is an important parameter that is needed to be optimized.

Optimization Algorithms

Several different optimization algorithms apply gradient descent. The most frequently used one is called Adaptive Moment Estimation (Adam)[19] optimizer. Adam computes a different learning rate for each parameter inside the neural network. This allows the algorithm to take different size steps during gradient descent. There are several other algorithms for optimization such as Adagrad [9], Adadelta [42] etc. All of the optimization algorithms apply different rules for updating the learning rate during the training phase.

2.3.2 Regularization Methods

One of the biggest problems in deep neural networks is over-fitting. Over-fitting means that the neural network memorized the data instead of learning it. Memorizing the training set does not mean making good predictions for the test set. Therefore, different regularization methods are applied to avoid the over-fitting problem in deep neural networks. Regularization methods make various changes during the training phase so that the neural network learns representations of the data better.

L2 and L1 Regularization

As discussed earlier, there is a loss function in the neural network that outputs the cost of a current prediction. L2 and L1 regularization methods [22] update the cost by adding regularization terms to it. Therefore, the network needs to minimize both the loss function and the regularization term. Network weights with L2 and L1 regularization are not likely to over-fit.

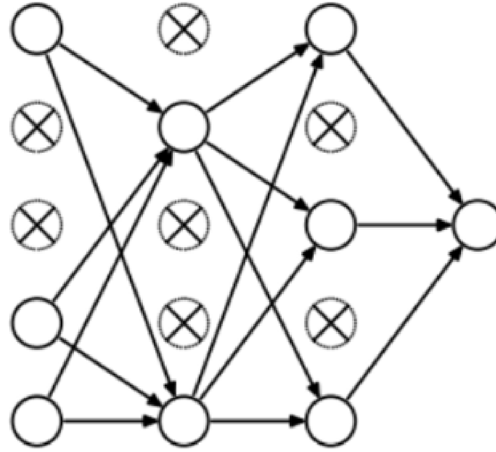


Figure 2.9: Dropout Visualization

Dropout

Dropout [36] is one of the most widely used regularization techniques in deep neural networks. As shown in Figure 2.9, it randomly selects a node from a layer and removes all of its incoming and outgoing connections at every iteration. Therefore, in every iteration during the training phase, the neural network has a different set of nodes and different outputs from layers. The dropout rate is a parameter that needs to be set before the training. It is the probability of a node being selected to be a dropout from the network for that iteration. This makes it hard to memorize the data for the neural network.

Data Augmentation and Early Stopping

Instead of making changes inside the neural network, we can make changes in the training data. Data augmentation reduces over-fitting by increasing the size of the training set. To increase the size, we can create new training sets where original data is shifted, flipped, rotated, or noise added. After these changes, it will be hard for the neural network to memorize the training set. However, still, there

will be a chance for the network with a large number of parameters to memorize new training data. Another method to avoid overfitting is early stopping. We can stop the training process earlier in case of different conditions. One condition could be getting similar loss values for consecutive iterations. Another condition could be getting higher loss values for the validation set for a given number of iterations. There can be many possible early stopping conditions that we can use to prevent the neural network to continue training before it memorizes the training set.

2.4 Convolutional Neural Networks

Our work mainly focuses on Convolutional Neural Networks (CNNs) to improve distributed training performance. CNNs are built using the modules that are described in the previous sections. CNNs are built using convolutional layers followed by pooling layers and non-linearities. To produce outputs, fully connected layers are used at the end of the network. Throughout the years, different CNN models are proposed to achieve better accuracy and smaller loss values. In this section, brief information about some of the well-known CNNs will be given and it will be possible to see advances in Convolutional Neural Networks through the years.

2.4.1 LeNet

One of the first successful CNN architectures is LeNet [23]. In Table 2.1, structure of LeNet is shown. There is a sequence of three layers, which are a convolutional layer, the pooling layer, and non-linearity. This same sequence is still widely used in today's CNNs. As non-linearity functions, the sigmoid and hyperbolic tangent is used in LeNet. To subsample the feature maps, average pooling layers are used. At the end of the network, fully connected layers are used to produce the output. LeNet can be considered as an inspiration for the CNNs that are developed after it.

Layer		Filters	Size	Kernel Size	Stride	Activation
Input	Image	1	32x32	-	-	-
1	Convolution	6	28x28	5x5	1	tanh
2	Avg. Pooling	6	14x14	2x2	2	tanh
3	Convolution	16	10x10	5x5	1	tanh
4	Avg. Pooling	16	5x5	2x2	2	tanh
5	Convolution	120	1x1	5x5	1	tanh
6	FC	84	84	-	-	tanh
Output	FC	10	10	-	-	softmax

Table 2.1: LeNet Structure

2.4.2 AlexNet

Layer		Filters	Size	Kernel Size	Stride	Activation
Input	Image	1	227x227x3	-	-	-
1	Convolution	96	55x55x96	11x11	4	relu
2	Max Pooling	96	27x27x96	3x3	2	relu
3	Convolution	256	27x27x256	5x5	1	relu
4	Max Pooling	256	13x13x256	3x3	2	relu
5	Convolution	384	13x13x384	3x3	1	relu
6	Convolution	384	13x13x384	3x3	1	relu
7	Convolution	256	13x13x256	3x3	1	relu
8	Max Pooling	256	6x6x256	3x3	2	relu
9	Flatten	9216	9216	-	-	relu
10	FC	4096	4096	-	-	relu
11	FC	4096	4096	-	-	relu
Output	FC	1000	1000	-	-	softmax

Table 2.2: AlexNet Structure

In 2012, AlexNet is developed by Alex Krizhevsky [21]. It is a much deeper and wider version of LeNet and became famous after winning ILSRVC ImageNet Challenge 2012 [33]. Since AlexNet is a wider and deeper version of LeNet, it is capable of learning features from more complex objects. Besides the size of the network, there are several other differences in AlexNet compared to the LeNet. The Rectified Linear Unit is used as a non-linearity function. To avoid overfitting, dropout is used at the fully connected layers and max-pooling layers are used instead of average pooling layers.

2.4.3 VGG

Layer		Filters	Size	Kernel Size	Stride	Activation
Input	Image	1	224x224x3	-	-	-
1	Convolution	64	224x224x64	3x3	1	relu
2	Convolution	64	224x224x64	3x3	1	relu
3	Max Pooling	64	112x112x64	3x3	2	relu
4	Convolution	128	112x112x128	3x3	1	relu
5	Convolution	128	112x112x128	3x3	1	relu
6	Max Pooling	128	56x56x128	3x3	2	relu
7	Convolution	256	56x56x256	3x3	1	relu
8	Convolution	256	56x56x256	3x3	1	relu
9	Max Pooling	256	28x28x256	3x3	2	relu
10	Convolution	512	28x28x512	3x3	1	relu
11	Convolution	512	28x28x512	3x3	1	relu
12	Convolution	512	28x28x512	3x3	1	relu
13	Max Pooling	512	14x14x512	3x3	2	relu
14	Convolution	512	14x14x512	3x3	1	relu
15	Convolution	512	14x14x512	3x3	1	relu
16	Convolution	512	14x14x512	3x3	1	relu
17	Max Pooling	512	7x7x512	3x3	2	relu
18	Flatten	25088	25088	-	-	relu
19	FC	4096	4096	-	-	relu
20	FC	4096	4096	-	-	relu
Output	FC	1000	1000	-	-	softmax

Table 2.3: VGG16 Structure

VGG was one of the most successful and deepest CNN of the time it was proposed [35]. VGG16 contains 16 layers, where 13 of them are convolutional (unlike AlexNet, 3x3 filters are used in each) and 3 of them are fully connected layers. The success of VGG showed that the depth of the neural network is an important factor that affects the final performance. However, besides better accuracy and error results, increasing the depth of the network comes with its drawbacks. The number of parameters in VGG is almost 140 million and the total memory required to pass one image forward is almost 93MB. Thus, it is hard to train VGG on a single machine. This is because, if we want to use a batch size of 128 (which is commonly used), the required memory is 11.625 GB and it is not possible to run the network on platforms with RAM less than 11.625 GB. As

described in the problem statement section, our main focus is to run networks on multiple machines efficiently, where they do not fit into a single machine's memory. Therefore, VGG is used in experiments that will be discussed later.

2.4.4 Residual Network

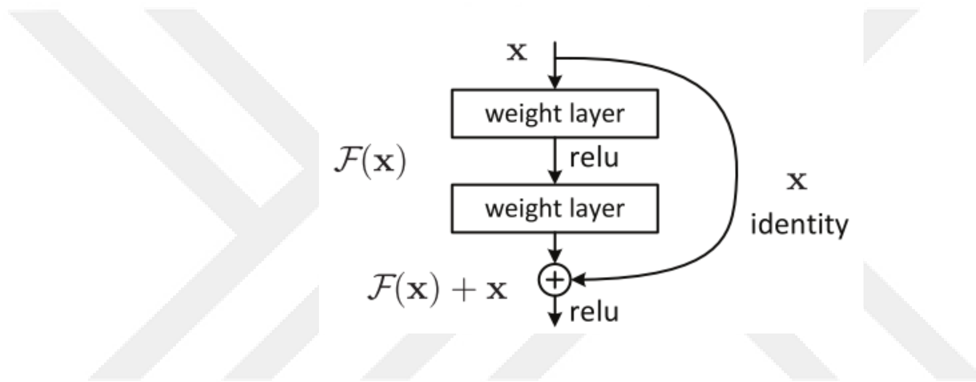


Figure 2.10: Skip Connection [13]

Residual Network [13] (ResNet) is the winner network of ILSRVC ImageNet Challenge in 2015 [33]. The main contribution of ResNet is to keep the number of parameters low while increasing the depth of the network. They have done this using skip connection as shown in Figure 2.10. A skip connection is used to bypass the input to the next layers. Besides, fully connected layers are not used in ResNet, which is the main reason for ResNet to have a lower number of parameters.

2.5 Hypergraph Partitioning

A graph consists of vertices and edges. Edge in a graph connects a pair of vertices. A hypergraph is a generalization of the graph where a hyperedge connects possibly more than two vertices. A hypergraph $H = (V, N)$ is defined as a set V of vertices and a set N of nets or hyperedges. Every $n \in N$ connects a subset of vertices, i.e., $n \subseteq V$. Weights and costs can be assigned to vertices and nets respectively.

Weight of a vertex v is denoted as $w(v)$ and cost of a net n is denoted as $cost(n)$. Given hypergraph $H = (V, N), \{V_1, ..., V_k\}$ is called a K-way partition of vertex set V. A K-way vertex partition of H is said to satisfy the balancing constraint if $W_k \leq W_{avg}(1+\epsilon)$ for $k = 1, ..., K$. W_k denotes the weight of a part V_k. W_{avg} is the average part weight and ϵ represents the predetermined maximum allowable imbalance ratio. Weight of a part V_k is obtained as:

$$W_k = \sum_{v \epsilon V_k} w(v)$$

In a partition of H, a net that connects at least one vertex in a part is said to connect that part. Connectivity $\lambda(n)$ of a net n denotes the number of parts connected by n. A net is said to be cut if it connects more than one part (i.e., $\lambda(n) > 1$) and uncut otherwise. The partitioning objective is to minimize the cutsize defined over the cut nets. There are various definitions and two relevant ones are the cut-net and connectivity metrics:

$$cutsize_{cutnet} = \sum_{n \epsilon N_{cut}} cost(n) \qquad cutsize_{con} = \sum_{n \epsilon N_{cut}} \lambda(n) cost(n)$$

In the cut-net metric, each cut net n incurs $cost(n)$ to the cutsize, whereas in the connectivity metric, each cut net incurs $\lambda(n)cost(n)$ to the cutsize.

2.6 Parallelism Models

In distributed deep neural network training, there are two main approaches: data parallelism and model parallelism. Most of the works in the literature focus on data parallelism as it is easier to implement and analyze than model parallelism. Information about some of the works for both parallelism models will be given in this section.

2.6.1 Data Parallelism

In data parallelism, the same neural network model is replicated among different processors and data is partitioned among them so that each replica uses a different part of the data. Those processors need to communicate with each other to send gradients to each other and to continue the training process. Data parallelism does not solve the problem of the network not fitting into a single machine's memory, because all of the replicas still contain the complete network. Therefore, data parallelism does not meet the requirements of training large deep neural networks [2, 3, 7, 39, 40].

In terms of communication patterns, there are different works in the literature. In [11], the All-Reduce communication pattern is used in the backpropagation phase. Before starting backpropagation, each processor computes its gradients. Then, the All-Reduce communication pattern reduces the gradients and ditributes the results to all processors. In the early distributed machine learning techniques, similar communication primitives were widely used between processors. Later, another and widely used technique in data parallelism called parameter server [25, 6, 2, 7, 5, 15] has become very popular and widely studied. As stated earlier in the Introduction section, in this method all of the worker processors send their gradients to the parameter server. Afterward, worker processors get updated weights from the parameter server before starting the forward pass of the next batch.

Another important factor to consider while using data parallelism is the synchronization of worker processors after training of every mini-batch of data. It is possible to synchronize the worker processors so that each worker waits for all others to finish computing their gradients. When all of the gradients are summed and applied to the weights, each worker processor starts to train a new mini-batch with the same updated weights. This synchronization is also known as Bulk Synchronous Parallelism (BSP) [38]. The problem with BSP data parallelism is that worker processors may stay idle for a long period because of synchronization and

communication overheads, which become a bigger problem as computation becomes faster with technological advances. There are some other works [43] that try to solve the problem of processors being idle by overlapping communication and computation, which can affect the convergence performance negatively.

Another approach is called Asynchronous Parallel (ASP) data parallelism. In this approach, worker processors do not wait for other processors to compute their gradients in ASP data parallelism [44]. This reduces idle processor times and leads to more efficient processor utilization compared to BSP. However, since processors do not wait for other processors' gradients being applied to their weights, ASP may result in stale weights.

Other works apply different synchronization techniques [15] such as Stale Synchronous Parallel (SSP), which is a combination of BSP and ASP. In SSP, there is a control mechanism where fast iterated processors check the network at every iteration for possible communications, and slow processors only check every S iterations for fewer network accesses to catch up the training process.

2.6.2 Model Parallelism

In model parallelism, a neural network model is partitioned across different processors so that each processor is responsible for the training of different parts of the neural network model. Works in the literature about model parallelism [16, 26, 24, 18] showed that model parallelism can achieve faster training times than data parallelism. There are several properties of model parallelism that makes it hard to implement and results in less amount of research in literature than data parallelism.

During the training of deep neural networks, each layer should wait for the previous layer to get its output. This means that when we partition the neural network among different processors, each processor has to wait for the previous processor to continue to the training process. When a processor computes its output (feature maps or activation maps) it becomes idle until the iteration of the next

mini-batch. Therefore, in some of the works [20, 7, 2], model parallelism is only applied when a neural network does not fit into a single machine's memory, but not to improve the efficiency of training performance. In this thesis, this problem of model parallelism is solved by using threaded asynchronous model parallelism technique which will be discussed in detail in later chapters.

Another problem with model parallelism is to decide on how to partition a deep neural network among multiple processors. This decision process is left to a human expert so that he/she should decide on which parts of the network should reside in which processor. It is a hard problem for a human expert to solve as a deep neural network contains millions of parameters and there exist hundreds of communications. In [16], a "giant" convolutional neural network is trained that contains 557 million of parameters. In this work, it is stated that it is hard to partition a convolutional neural network model so that it achieves both low computational load balance and low communication volume. Therefore, their way of partitioning (layer-wise) causes a convolutional neural network training to waste time because of imbalanced work overhead. Another work [30] tried to solve this problem using another deep learning technique called Reinforcement Learning. However, this is also a complicated, time and resource-consuming process. In this thesis, this problem of model parallelism is solved by using hypergraph partitioning which decides automatically on the partitioning of a deep neural network model.

It is also possible to combine model parallelism and data parallelism in a single deep neural network. In [20], it is shown that most of the computation is done on convolutional layers, and most of the parameters are contained in fully connected layers. Therefore, it is suggested that data parallelism should be used on convolutional layers, while model parallelism should be used on fully connected layers. Although it is a good way of partitioning a deep neural network among different processors, it still requires a human expert to decide on the partitioning process. In [12], a neural network model is partitioned in a layer-wise manner among different processors. For the parts with low computational load, data parallelism is used to improve computational load balance. Data parallelism is used at the parts where there is a low computational load compared to the others pars after

layer-wise partitioning. When a deep neural network is partitioned in a layer-wise manner, it is almost inevitable to have imbalanced computational load between different processors. In this thesis, the problem of model parallelism is solved by using coarse-grain and fine-grain hypergraph partitioning of deep neural networks so that the computational load balance ratio of different processors is close to 1, while keeping communication volume low.

Chapter 3

Modeling Communication and Computation

In this chapter, graph and hypergraph representations of CNNs will be given for different levels of granularity. The graph model of a CNN is straightforward and shows the order of the operations inside a CNN. However, the graph model does not accurately capture the communication pattern inside the network. We propose the hypergraph representations of the CNNs, and we show that these hypergraph models more accurately capture the communication pattern.

3.1 Graph Representation

A CNN can be represented as a multistage graph. A multistage graph is a directed graph where vertices can be divided into a set of stages in such a way that all edges connect vertices belonging to two successive stages. The input layer and the intermediate feature maps can be represented as vertices, and tasks between those vertices such as convolution operation, pooling, nonlinearity, etc. can be modeled as edges between vertices. The important thing to decide on while

representing a convolutional neural network is granularity. There can be fine-grain representation where there are small but many operations, or there can be coarse grain representations where there are big but fewer operations.

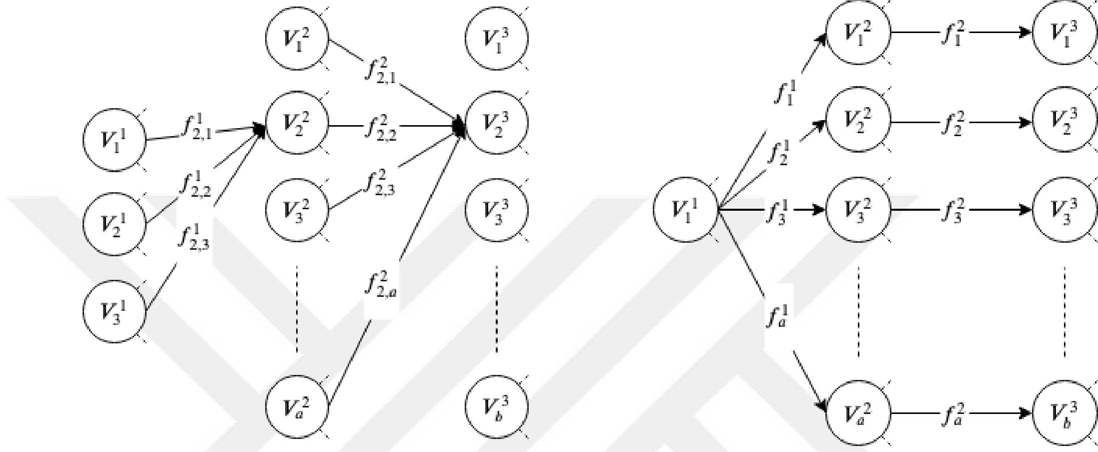


Figure 3.1: Fine and Coarse-grain CNNs Graph Representations

Let's say that we have a convolutional neural network with an input layer and two convolutional layers that contain a and b number of filters. This convolutional neural network can be represented as a graph shown in Figure 3.1. In the fine-grain representation, each edge is a convolution operation of each channel of a filter and each vertex is a feature map which is the reduced result of convolution done by connected edges. This means that for each vertex, outgoing edges represent convolution operations on that feature map, and the incoming edges represent the reduction operation. V_j^i denotes the feature map on the ith layer obtained using j^{th} filter. f_{jk}^i denotes the kth channel of the jth filter on the ith layer. If on i^{th} layer there are n numbers of feature maps, then the formula to obtain a feature map on $(i+1)^{th}$ layer is as follows,

$$V_j^{i+1} = \sum_{k=1}^{n} V_j^i \circledast f_{jk}^i,$$

where $\circledast$ denotes the convolution operation. In the coarse grain representation, the atomic task definition is different from the fine-grain representation. This time, instead of applying the convolution operation using each channel of a filter, convolution is done using the filter itself. This means that instead of applying

the 2D convolution on 2D data using a 2D filter, this time 2D convolution is applied on 3-dimensional data using 3-dimensional filter. In this representation, V_j^i represents the feature map on the ith layer obtained using j^{th} filter. f_j^i denotes the jth filter on the ith layer. Then the formula to obtain feature map on the $(i+1)^{th}$ layer is as follows,

$$V_j^{i+1} = V_j^i \circledast f_j^i,$$

where $\circledast$ denotes the convolution operation. The main difference between the two graph representations is the definition of the atomic task as stated earlier. By taking the CNN graph representations into account, we designed hypergraph representations for both fine and coarse grain representations.

3.2 Modelling Communication

Our hypergraph representations of a CNN are shown in Figure 3.2. In these representations, we changed the definitions of vertices and edges compared to the graph representation. A vertex in our hypergraph representation (except for the vertices in the input layer) denotes the convolution operation and the other layer operations such as activation or pooling if any. A vertex can also represent the reduction operation of the results of the convolution operation of multiple filter channels. Also, a net in our hypergraph models represents the communication between vertices (tasks). This means that dependency of the operations inside the consecutive layers is modeled. Therefore, representing tasks as a vertex and connecting them using hyperedges accurately models the communication inside the network. This modeling allows us to partition the vertices (tasks) to different processors while considering their dependency on each other and the communication volume.

Let's say that we have a CNN with an input layer and two convolutional layers that contain a and b number of filters respectively. This CNN can be represented

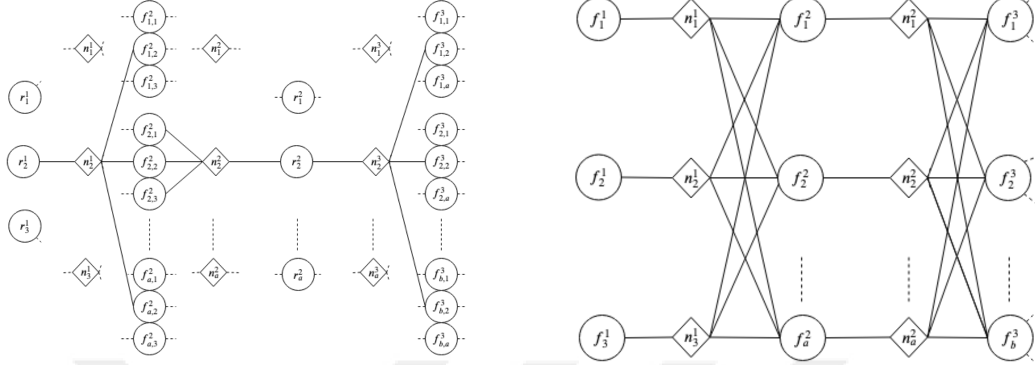


Figure 3.2: Fine-grain and Coarse-grain Hypergraph Representations of the CNNs

as a hypergraph as shown in Figure 3.2. In the fine-grain representation, except input layer, $f^i_{j,k}$ denotes the kth channel of the jth filter on the ith layer and r^i_j denotes the reduced result of jth filter on the ith layer. There are two types of vertex definitions in the fine-grain hypergraph representation. Vertices $f^i_{j,k}$ denote the convolution operation of a channel inside the filter. Vertices r^i_j denote the reduction operation of the feature maps, which sums the results obtained from previous convolutional operations. These vertices also apply pooling and non-linearity operations. Also, there are two types of nets in our fine grain representation, where one is to model the communication pattern of convolution operations, and the other one is to denote the reduction operation for feature maps. In the fine-grain hypergraph, the pins of nets corresponding to the convolution operations are defined as follows:

$$pins(n^i_j) = \{r^i_j\} \cup \{f^{(i+1)}_{j,\forall k}\}$$

In the fine-grain hypergraph, the pins of nets that correspond to the reduction operation are defined as follows:

$$pins(n^i_j) = \{f^i_{j,\forall k}\} \cup \{r^{(i+1)}_j\}$$

The definition of an atomic task is different in Figure 3.2 for fine and coarse grain hypergraphs. In the fine-grain hypergraph representation, each vertex represents

a 2D convolution operation on 2D data using a 2D filter. In the coarse grain hypergraph representation, each vertex represents 2D convolution operation on 3-dimensional data using 3-dimensional filter. In the coarse grain hypergraph, the pins of nets are defined as follows:

$$pins(n_j^i) = \{f_j^i\} \cup \{f_{\forall j}^{i+1}\}$$

Given hypergraph representations above, a partition Π is obtained such that $\Pi = \{V_1, V_2, ..., V_k\}$. This partition is decoded as follows: without loss of generality all vertices (the tasks) are represented by vertices of the part V_k assigned to processor P_k for $i = 1, 2, ..., k$. For partitioning, there are two important components used which are constraint and objective function. Partitioning constraint is to maintain the balance such that:

$$W(V_k) = \frac{W_{avg}}{k}(1 + \epsilon)$$

As an objective function, the connectivity metric is used. For given hypergraph representations, *connectivity* $\lambda(n) - 1$ denotes the number of communications occurring in the given partition. However, communication overhead is not necessarily determined by the number of communications. It is typically determined by the volume of communication. Therefore, the objective function of partitioning to minimize the following cost function:

$$cutsize_{con} = \sum_{n \epsilon N_{cut}} (\lambda(n) - 1) cost(n)$$

More detailed information about hypergraph representations of well-known convolutional neural networks and their performance for different partitions will be given in the next sections.

3.3 Fine-grain Hypergraph Representation

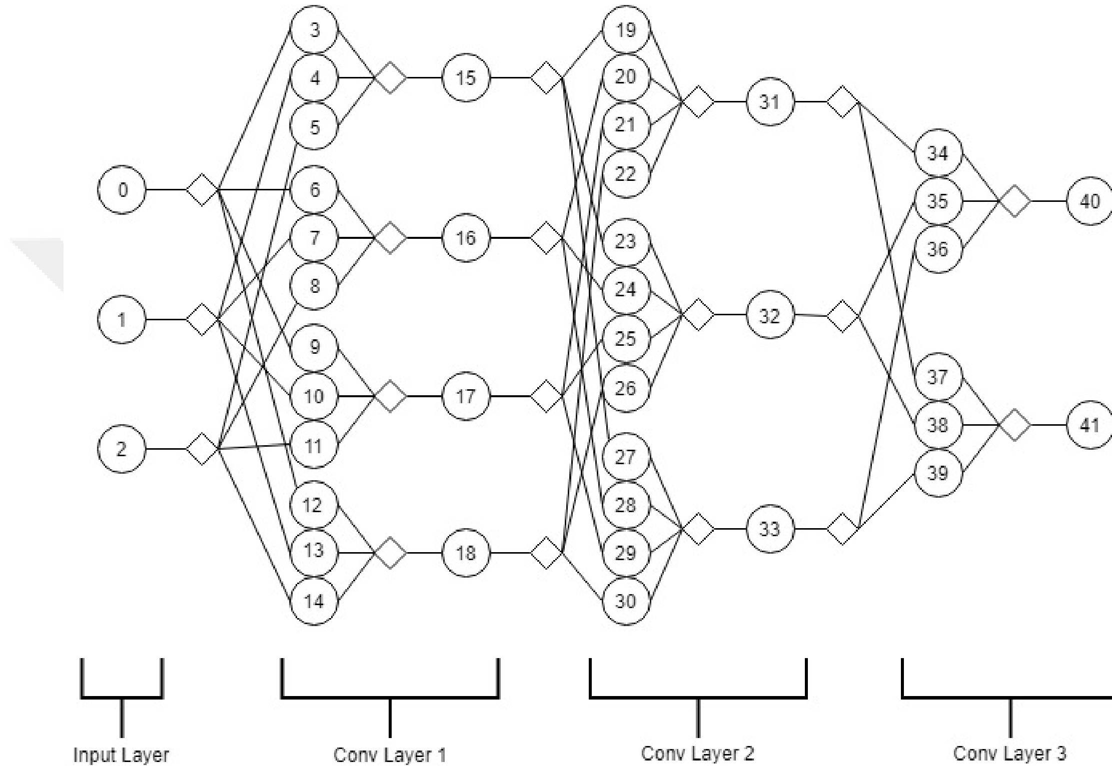


Figure 3.3: Fine-grain Hypergraph

In the fine-grain hypergraph representation of the CNNs, each channel inside of a filter is represented as a vertex. The fine-grain hypergraph model of the CNN is represented in Figure 3.3 with three convolutional layers that contain 4, 3, and 2 filters respectively after the input layer. Each vertex corresponds to a 2D filter and the definition of the atomic task in a fine-grain hypergraph is given in the previous section.

There are different operations inside the fine-grain hypergraph as described in the previous section. It is needed to sum the convolution results of the channels inside the same filters to get the feature maps. For example, the results of the convolutional operation between vertex 0 and 3, 1 and 4, 2 and 5 are reduced to one feature map and communicated (if necessary) to the vertex 15. If there exist operations such as pooling, and activation, they are done in the vertex 15.

Therefore, vertex 15 corresponds to the output feature map of the first filter in convolutional layer 1. Then, the output feature maps of each layer are fed into the next layer as an input.

The number of channels inside a filter is determined by the number of filters in the previous layer. For example, in the first convolutional layer, each filter is composed of three channels, corresponds to RGB. In the second convolutional layer, each filter is composed of four channels. Because there are four filters in the first convolutional layer and thus there are four feature maps that are used in the second convolutional layer.

3.4 Coarse-grain Hypergraph Representation

In the coarse-grain hypergraph, the definition of the atomic task is bigger and each filter inside a layer is represented as a vertex. A net connects filters from consecutive layers. To visualize, let's consider the same network presented for the fine-grain hypergraph representation, where the first layer is the input layer and there are 3 consecutive convolutional layers with the number of filters 4, 3, and 2 respectively.

In Figure 3.4, the same CNN architecture in Figure 3.3 is modeled as coarse-grain hypergraph. Each filter (vertex) is numbered from 3 to 11 and nets are represented as diamonds. Except for the input layer, the definition of all of the vertices is the same. This is because all of the operations such as convolution, reduction, pooling, and activation is done within a vertex.

Vertices 3, 4, 5, and 15 in Figure 3.3, are represented as vertex 3 in the coarse-grain representation. If the input of the network has 3 channels, such as RGB images, then the channel size of the vertices 3, 4, 5, and 6 is also 3. Since the number of filters in convolutional layer 1 is 4, the channel size of the vertices 7, 8, and 9 is also 4. Each vertex in the layers performs a 2D convolution operation on the vertex from the previous layer that it is connected to. To exemplify,

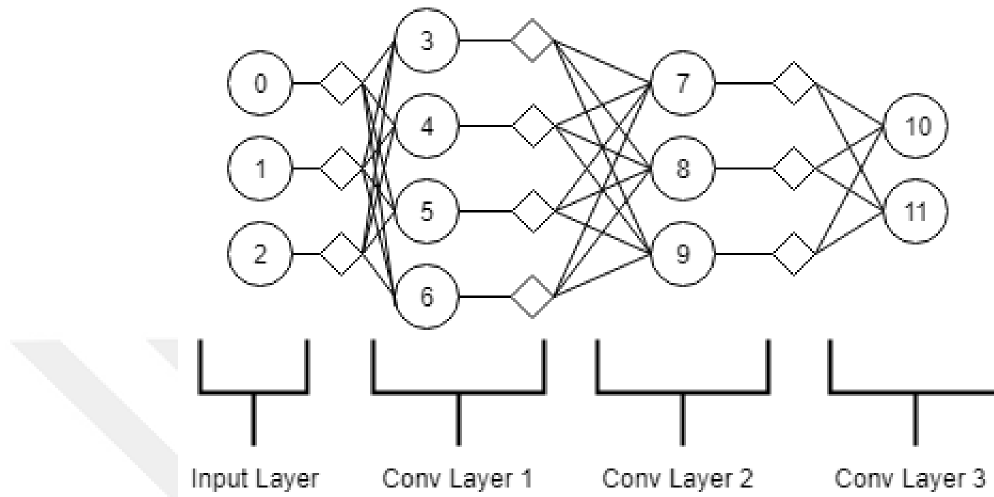


Figure 3.4: Coarse-grain Hypergraph

the vertices 3, 4, 5, and 6 contain a 3D filter and perform separate convolution operation on input layer on axis x and y. Non-linearity operation is applied after each convolutional operation. If there is a pooling layer between convolutional layers, pooling operation is applied to the feature maps before communicating the result to the vertices in the next layer. This procedure also reduces the communication volume.

All of the filters are connected to each vertex from the previous layer. This is because all of the feature maps that are produced from previous layers are required for each vertex inside the current layer. For example, feature maps that are obtained from the vertices 3, 4, 5, and 6 are used in vertex 7. Therefore, there is a net that connects vertices 3, 4, 5, 6 and vertex 7, which implies these vertices are dependent on each other, and there have to be communication if they reside on different processors. Definition of the atomic task is finer grain than++ the convolution of a layer and it is convolution of a single filter. Thus, this reduces the communication volume between layers if any of the vertices reside on different processors. A hypergraph partitioning algorithm is aware of the fact that consecutive layers are dependent on each other. Therefore, it is unlikely to partition a single layer among many processors. Even if there is a processor difference between consecutive layers, there will be a communication of feature

map that is produced from a single vertex. Therefore, instead of communicating all of the filters inside the layer, there will be a communication of a single feature map which also reduces communication overhead and also provides a better computational load balance.

A profiler run is performed to set the vertex weights. After profiler, based on the total run time of the layer during the forward and backward pass, and the number of floating-point operations is used to set the weight of a vertex. The net weights are set based on the communication volumes. Therefore, the hypergraph partitioning algorithm will try to allocate vertices on the same part that are connected with a high weighted net (net with a high cost). Otherwise, there will be a large amount of communication between pins.

Fine-grain hypergraph representation is better in terms of having lower computational load balance ratio and communication volume. However, coarse-grain hypergraph representation could be better in terms of computational performance, because Tensorflow is highly optimized for coarser operations. Therefore, instead of doing smaller operations many times, it is usually better to have less but coarser operations to get the benefit of the optimized Tensorflow structure. Besides, the Tensorflow implementation of the fine-grain hypergraph representation requires a high level of optimization, because there are millions of connections and many possible partition scenarios. Implementation of the coarse-grain representation is a lot easier and still achieves good performance, which will be discussed in the next sections.

3.5 Hypergraph Partitioning Details

Graph representation of the CNNs does not model the communication volume exactly, because feature maps and products of the operations are represented as vertices. Also, operations inside the network are represented as edges, which only model the relation between operations and their products. Our hypergraph representations model the communication volume that exists on CNN. This is

because the relationship between operations and their dependence is shown using hyperedges that can connect more than two operations. Partitioning our hypergraph representations results in a well-defined partition of the operations while considering their relation and communication pattern.

Without representing a CNN as a graph or hypergraph, it can be distributed easily using layer by layer partitioning. What is meant by layer by layer partitioning is that placing groups of succesive layers among the different processors. For example, for VGG16 with 16 layers, the first 8 layers can be placed into one processor, and the last 8 layers can be placed into the other processor. This kind of partitioning can cause work balance problems as work that is done by each layer differs from each other. Especially for the number of processors that is greater than 2, there will be a large work imbalance between partitions with large communication overhead.

To solve the given problem about model parallelism above, an algorithm is designed to represent a deep convolutional neural network as a hypergraph and then partition it. The purpose of hypergraph partitioning is to partition the vertex set into different disjoint blocks while minimizing an objective function as described in the previous sections.

To partition the hypergraph, the hypergraph partitioning tool Patoh [1] is used. To be able to use Patoh for hypergraph partitioning, it is necessary to prepare a hypergraph file that is acceptable by the tool. A sample input file consists of information about the hypergraph such as zero-indexed vertices, number of vertices, number of nets, number of pins, weights of the net, and weights of vertices.

A sample input file for Patoh is shown in Figure 3.5 by using the convolutional neural network described in Figure 3.4. To create a more complete convolutional neural network, three fully connected layers are also added to the sample file where the last one is with a softmax activation function. Each net in the same layer has the same cost, as the amount of communication is the same for all vertices in a layer. Besides, each vertex in the same layer has also the same

```
0 13 12 38 3
1 0 1 2 3 4
2 1 5 6 7
2 2 5 6 7
2 3 5 6 7
2 4 5 6 7
3 5 8 9
3 6 8 9
3 7 8 9
4 8 10
4 9 10
5 10 11
6 11 12
1 2 2 2 2 3 3 3 4 4 5 6 7
```

Figure 3.5: Sample Input File for Patoh

weight, as the amount of the work that is done for those vertices is the same. In the sample input file, let's assume that weights of nets and vertices are increasing by one at each layer.

The first line of the input file gives information about the hypergraph. The first integer can be 0 or 1 denoting the start index of the vertices. The second integer is the number of vertices, the third integer is the number of nets, the fourth integer is the number of pins. The last integer is to show Patoh that weights of both nets and vertices are given. It is also possible to give an input file with only net costs or vertex weights or none of them. In our case, the last integer is 3 as we are providing weights of both nets and vertices. Lines after the first line except the last one contain information about each net. The first integer of each line is the weight of the net, and the rest are vertex indices that are connected through this net. Finally, the last line contains weights of each vertex in the hypergraph.

Given an input file, Patoh partitions the hypergraph. After partitioning is done, Patoh returns an array which contains the partition index for each vertex in the hypergraph. Given the resulting partitioning array, Tensorflow [28] implementation is done that creates a convolutional neural network by distributing each part of the partition among different processors and creates a trainable TensorFlow

graph. From the resulting partitioning array, it is possible to compute computational load balance and communication volume between partitions and compare them between different methods.

Chapter 4

Asynchronous Pipelined Model Parallelism

In traditional model parallelism, where each worker is responsible for different parts of the network, there is an underutilization of the compute resources. This is because the worker computes and passes activation maps to the next worker and then becomes idle until the next iteration. This type of model parallelism is still useful when a network does not fit in a single machine's memory, yet many of the processors stay idle for a long period throughout the training. To achieve fast and efficient training using model parallelism, all of the workers should be working throughout the training.

4.1 Collocating Gradient Operations

To achieve efficient training, first problem is that each processor should perform backpropagation of parameters only it is responsible for. By default, the back-propagation of all of the parameters is done by one processor only. This results in longer wait times for other processors which are not responsible for backpropagation. Also, all of the activation maps should be communicated to the processor

that is responsible for backpropagation so that it can compute all of the gradients. Then, all of the computed gradients are needed to be communicated back to the processors. This means that, when one processor is responsible for all of the backpropagation process, besides processors being idle, also communication volume increases.

Figure 4.1: Model Parallelism Working Schema

Figure 4.2: Model Parallelism Working Schema - Illustration

Figure 4.1 shows the basic model parallelism explained above for one iteration. In Figure 4.2, basic model parallelism is illustrated for two iterations. When processor 1 completes the forward pass of a batch, it sends activation maps to the processor 2 which is responsible for backpropagation operation. While processor 2 is busy with backpropagation, processor 1 stays idle. When processor 2 finishes backpropagation and sends gradients to processor 1, the same process starts for a new iteration.

To solve the described problem, gradient operations are collocated so that each processor is responsible for the backpropagation of its parameters only. Although this does not solve the processors' staying idle problem during the training, it

reduces the communication volume and it is necessary for efficient and fast training.

Figure 4.3: Model Parallelism with Collocated Gradients Working Schema

Figure 4.4: Model Parallelism with Collocated Gradients Working Schema - Illustration

Figure 4.3 shows the basic model parallelism with collocated gradients operations for one iteration and it is also illustrated in Figure 4.4 for two iterations. This time, processors are responsible for the backpropagation of parameters they are responsible for.

4.2 Threaded Training of Model Parallelism

Efforts so far did not affect the run time of a training process. A deep neural network model is distributed to different processors for both forward pass and backpropagation operations. However, still, each processor waits for other processors to continue the training process. To solve this issue, threads are used so that whenever a processor becomes idle, it fetches the next batch of the data and

starts to process. This kind of working schema is used in one of the recent works [12] as well, where model parallelism is studied.

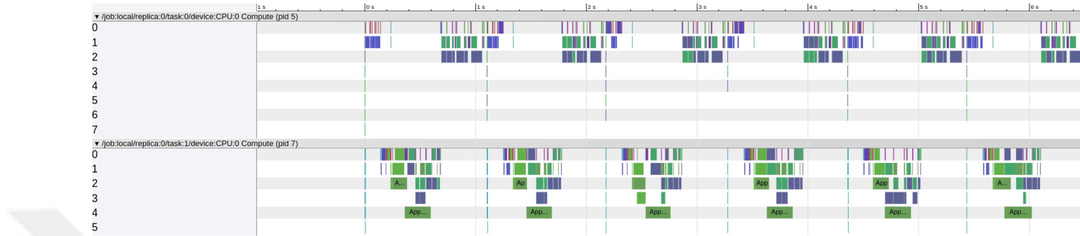


Figure 4.5: Model Parallelism with Collocated Gradients Working Schema

Figure 4.6: Threaded Model Parallelism with Collocated Gradients Working Schema

Figure 4.5 shows the training process without using threads. As seen in the Figure 4.5, there are many time slots where processors stay idle. In Figure 4.6, threads are used and whenever a processor becomes idle, it fetches the next batch and continues to the training process. Thus, the training process becomes fully utilized and each processor is applying operations for parameters only it is assigned for. This process is also illustrated in Figure 4.7.

Considering the computational load balance and communication volume, partitioning results are obtained from PatoH. To obtain partitioning results, a deep convolutional neural network is modeled as fine-grain and coarse-grain hypergraphs. To summarize the work that is done so far, we obtained a hypergraph

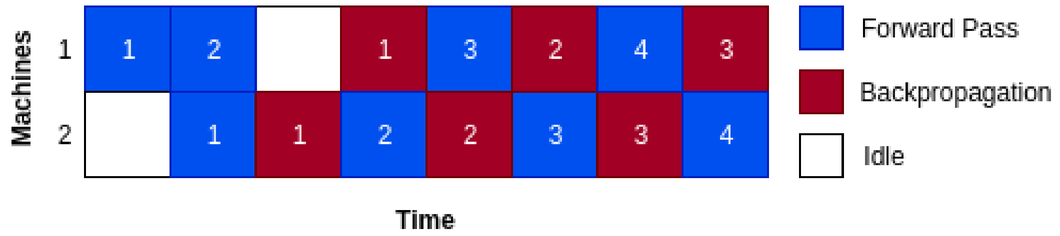


Figure 4.7: Threaded Model Parallelism with Collocated Gradients Working Schema - Illustration

model of deep CNN, distributed the network among different processors by taking consideration of computational load balance and communication volume, and fully utilized the run time by using asynchronous pipelined model parallelism.

Chapter 5

Experiment Results

In this chapter, different parallelism models on VGG16 network will be analyzed and compared based on the computational load balance, communication volume, run time performance, and convergence performance in reaching the target accuracy and loss.

5.1 Computational Load Balance and Communication Volume Analysis

With the properties described above, an input file has been created for the VGG16 network for both coarse-grain hypergraph and fine-grain hypergraph. Besides those, several naive partitioning algorithms are also implemented to show that hypergraph partitioning is better than naive methods in terms of computational load balance and communication volume between partitions.

To model the vertex weights, a profiler run is executed on a single processor using batch size of 32. Then for each layer inside the VGG16 network, a weight is calculated using the time passed for a forward pass and backpropagation and also using the number of floating-point operations. Since the input size and the

number of filters are different for each layer, the weight of layers also differs from each other.

Communication volumes are also calculated for batch size of 32 and value for communication analysis on tables is in terms of megabytes (MB). Each layer needs to communicate its output to the next layer and the volume of this communication increases linearly with batch size. Therefore, it is important to place consecutive layers into the same partition so that there will be less amount of communication. It is also important to partition layers or filters so that each partition has a similar amount of computational load.

In this experiment, SKL (Skylake Xeon) nodes are used where nodes have 112 logical cores each. The Xeon nodes are connected by Intel Omni-Path (Intel OPA) 100 Series interconnect. All nodes are connected via 10Gb Ethernet. More information about each node(processor) can be seen in Figure 5.1.

5.1.1 Layer-wise Partitioning

Layer-wise partitioning is to partition a deep neural network by distributing layers among different processors. Layer-wise partitioning is widely used in the literature and easy to implement. It requires a human expert to decide on which layers should be assigned to different processors. To illustrate, it is possible to apply layer-wise partitioning on a 16 layer VGG16 network by placing the first 8 layers to one partition, and the last 8 layers to the other partition. However, this is one of the very bad examples of layer-wise partitioning as the work that is done by the first 8 layers is much more than the work that is done by the last 8 layers. Therefore, there will be a computational load balance problem, and the second processor will have to wait for long time for the first processor to finish its job.

Considering the above problem, layer-wise partitioning is applied to the VGG16 network where partition points are set recursively to obtain the lowest computational load balance (maximum work balance / average work balance) ratio. Then, for each number of processors N, computational load balance analysis is done in

Table 5.1, and communication volume analysis is done in Table 5.2.

Layer-wise	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	22,757	20,244	21,500	1.1	1.0	51,308
N=4	12,827	8,497	10,750	1.5	1.1	51,308
N=8	7,253	2,677	5,375	2.7	1.3	58,024
N=16	6,624	60	2,687	110.4	2.4	105,984

Table 5.1: Layer-wise Partitioning Computational Load Balance Analysis

Layer-wise	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg comm
N=2	24.5	24.5	12.2	2.0
N=4	85.7	49.0	21.4	2.2
N=8	131.6	49.0	16.4	2.9
N=16	273.5	98.0	17.0	5.7

Table 5.2: Layer-wise Partitioning Communication Volume Analysis

Although computational load balance is not a problem for small values of N, computational load balance starts to become a problem as we increase the number of partitions. This is because the work that is done for a layer could be much more than the work of any other layer. When N is equal to 16, this means that each layer is assigned to a different partition. A huge difference in the amount of work for different layers can be seen in Table 5.1 where N is equal to 16.

The main problem of layer-wise partitioning is the definition of the atomic task, which is a convolution of a layer. Since a layer is not partitioned also within itself, it is not easy to obtain good partitions. There are also other problems such as it requires a human expert to decide on where to partition the network, and the number of partitions is bounded by the number of layers. To solve the problems of layer-wise partitioning, a deep neural network is modeled as different hypergraphs where an atomic task is defined as the convolution of a filter or convolution of a channel. Thus, a deep neural network becomes easier to be partitioned that results in better computational load balance and communication volume.

5.1.2 Filter-wise Horizontal Naive Partitioning

As described previously, a deep neural network is represented as a hypergraph for coarse-grain partitioning. As a naive partitioning approach, horizontal partitioning can be used. What is meant by horizontal naive partitioning is that for each value of N, the network is partitioned horizontally so that each partition has the same amount of work. In short, filters inside each layer is partitioned among N different parts equally.

Filter-wise	Max. Work	Min. Work	Avg. Work	Max./Min.	Max./avg.	Max.*N
N=2	21,500	21,500	21,500	1.0	1.0	43,001
N=4	10,750	10,750	10,750	1.0	1.0	43,001
N=8	5,375	5,375	5,375	1.0	1.0	43,001
N=16	2,687	2,687	2,687	1.0	1.0	43,001
N=32	1,343	1,343	1,343	1.0	1.0	43,001
N=64	671	671	671	1.0	1.0	43,001
N=128	335	335	335	1.0	1.0	43,001

Table 5.3: Filter-wise Horizontal Naive Partitioning Computational Load Balance Analysis

Filter-wise	Comm. Vol	Max. Comm.	Avg. Comm.	Max./Avg. comm
N=2	484	98	242	0.4
N=4	1,144	294	286	1.0
N=8	2,307	686	288	2.3
N=16	4,556	1,470	284	5.1
N=32	9,016	3,038	281	10.7
N=64	17,916	6,174	279	22.0
N=128	35,707	12,446	278	44.6

Table 5.4: Filter-wise Horizontal Naive Partitioning Communication Volume Analysis

From the computational load balance analysis in Table 5.3, it can be seen that filter-wise horizontal naive partitioning is much better in terms of computational load balance. This is because each part has the same amount of work. However, as it can be seen in Table 5.4, partitioning a network horizontally results in a huge amount of communication volume. When a network is partitioned horizontally, there is communication at each consecutive layer as filters of those layers always reside in different partitions. As the number of parts increases, the amount

of communication volume also increases due to the increase in communication between successive layers.

5.1.3 Filter-wise Incremental Naive Partitioning

In the filter-wise incremental naive partitioning, by starting from the first layer, filters of the layers are assigned to a partition until total work of that partition exceeds average work (total work / N). When a partition's total work exceeds average work, the next filters are assigned to a new partition until that partition's total work exceeds average work.

Filter-wise	Max. Work	Min. Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	21,524	21,495	21,510	1.0	1.0	43,049
N=4	10,767	10,732	10,755	1.0	1.0	43,069
N=8	5,419	5,289	5,377	1.0	1.0	43,352
N=16	2,720	2,535	2,688	1.0	1.0	43,528
N=32	1,362	1,161	1,344	1.1	1.0	43,597
N=64	919	144	682	6.3	1.3	57,957
N=128	715	144	352	4.9	2.0	87,317

Table 5.5: Filter-wise Incremental Naive Partitioning Computational Load Balance Analysis

Filter-wise	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg comm
N=2	42	24.5	21.3	1.1
N=4	117	49.0	29.3	1.6
N=8	372	98.0	46.5	2.1
N=16	667	196.0	41.6	4.7
N=32	1,294	490.0	40.4	12.1
N=64	2,212	882.0	35.1	25.1
N=128	4,091	1568.0	33.5	46.7

Table 5.6: Filter-wise Incremental Naive Partitioning Communication Volume Analysis

Similar to the previous partitioning algorithm, in the filter-wise incremental naive partitioning, computational load imbalance is expected to be low as all of the partitions' work close to each other. As shown in Table 5.5, computational load balance for different N values is low until N becomes higher values such as 64

and 128. Therefore, considering the work balance, this method results in a good partition.

When we compare the communication volume analysis of this method in Table 5.6 with the layer-wise partitioning method in Table 5.2, it can be seen that this method does not result in better communication volumes. As stated previously, work that is done by each layer can be very different from each other. Therefore, applying incremental naive partitioning can result in many different partitions between successive layers which results in higher communication volumes. However, the incremental naive partitioning algorithm obtains better computational load imbalance ratios than layer-wise partitioning. The reason for this is that the definition of an atomic task is smaller in naive partitioning and it is easier to obtain lower computational load imbalance ratios.

The filter-wise incremental naive partitioning algorithm obtains much better communication volume for all values of N compared to the horizontal naive partitioning algorithm. The main reason behind this is that there is not communication at each successive layer in the incremental naive algorithm. Although incremental naive algorithm obtains slightly worse computational load imbalance ratios compared to the horizontal algorithm, values of computational load imbalance ratio are very close to the 1. Therefore, the incremental naive partitioning algorithm becomes better than the horizontal naive partitioning algorithm for having a better combination of communication volumes and computational load imbalance ratios.

5.1.4 Channel-wise Incremental Naive Partitioning

The same incremental naive partitioning algorithm is applied to fine-grain hypergraph. The result of horizontal naive partitioning for fine-grain and coarse-grain hypergraph is the same. As expected, the channel-wise incremental naive partitioning algorithm also results in similar computational load balance and communication volume results with the filter-wise incremental naive partitioning algorithm. This is because the work of the layers is the same for both hypergraph

models. Thus, applying a naive partitioning algorithm on these methods results in similar results.

Channel-wise	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	21,507	21,231	21,369	1.0	1.0	43,015
N=4	10,753	10,477	10,684	1.0	1.0	43,015
N=8	5,378	5,099	5,342	1.0	1.0	43,027
N=16	2,689	2,410	2,671	1.1	1.0	43,030
N=32	1,345	1,061	1,335	1.2	1.0	43,060
N=64	915	144	667	6.3	1.3	58,621
N=128	576	144	336	4.0	1.7	73,226

Table 5.7: Channel-wise Incremental Naive Partitioning Computational Load Balance Analysis

Channel-wise	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg comm
N=2	42	24.5	21.4	1.1
N=4	118	49.0	29.5	1.6
N=8	373	98.0	46.6	2.1
N=16	663	196.0	41.4	4.7
N=32	1,275	490.0	39.8	12.2
N=64	2,371	980.0	37.0	26.4
N=128	4,471	1960.0	35.2	55.6

Table 5.8: Channel-wise Incremental Naive Partitioning Communication Volume Analysis

There can be numerous algorithms that can be applied to both hypergraph models. However, these algorithms should take consideration of nets which denote communication patterns between different vertices. Otherwise, it will be hard to obtain good computational load balance and communication volume at the same time. For this purpose, a hypergraph partitioning tool Patoh [1] is used as it tries to minimize both communication volume and computational load imbalance by taking consideration of nets and communication patterns. In the next sections, results obtained from Patoh for fine-grain and coarse-grain hypergraphs are presented.

5.1.5 Coarse-grain Hypergraph Partitioning

As described in previous sections, the VGG16 network is modeled as a coarse-grain hypergraph and it is partitioned using Patoh. Resulting computational load balance analysis and communication volume analysis can be seen in Table 5.9 and Table 5.10. There is a parameter that can be tuned in Patoh which is the final imbalance that denotes the imbalance ratio of the final partition. The tables below are obtained using the default final imbalance ratio which is 0.1. For several other results that are obtained using different final imbalance ratios, please see Appendix A.

Coarse-grain	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	21,825	21,195	21,510	1.0	1.0	43,650
N=4	11,123	10,189	10,755	1.0	1.0	44,494
N=8	5,740	5,012	5,377	1.1	1.0	45,924
N=16	3,177	2,544	2,688	1.2	1.1	50,838
N=32	1,691	1,242	1,344	1.3	1.2	54,113
N=64	951	620	672	1.5	1.4	60,897
N=128	623	144	336	4.3	1.8	79,823

Table 5.9: Coarse-grain Hypergraph Partitioning Computational Load Balance Analysis

Coarse-grain	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg comm
N=2	27	24.5	13.7	1.7
N=4	61	24.5	15.3	1.5
N=8	298	98.0	37.3	2.6
N=16	661	196.0	41.3	4.7
N=32	1,300	490.0	40.6	12.0
N=64	2,388	980.0	37.3	26.2
N=128	4,604	1,960.0	35.9	54.4

Table 5.10: Coarse-grain Hypergraph Partitioning Communication Volume Analysis

When computational load balance and communication volume metrics of coarse-grain hypergraph partitioning is compared to the both of the filter-wise naive partitioning methods given above, it can be seen that hypergraph partitioning results in a better combination of computational load balance and communication volume. Although both naive methods are targeting perfect computational load

balance and achieve better computational load balance ratios, it is important to have low computational load imbalance while keeping communication volume as low as possible. In this manner, it can be said that coarse-grain hypergraph partitioning obtains better results.

It is also important to compare coarse-grain hypergraph partitioning with layer-wise partitioning. This is because layer-wise partitioning is the model parallelism method that is used most of the time. In terms of communication volume, both of the methods achieve a better result than others at various values of N. In terms of computational load balance, for each value of N, the coarse-grain hypergraph partitioning algorithm achieves low computational load imbalances.

5.1.6 Fine-grain Hypergraph Partitioning

As described earlier, VGG16 network is transformed into fine-grain hypergraph and its partitioning results obtained using Patoh can be seen in Table 5.11 and Table 5.12. Similar to the coarse-grain version, the fine-grain hypergraph model also achieves a better combination of computational load balance and communication volume compared to the naive methods. When we compare two different hypergraph methods, the fine-grain hypergraph achieves better computational load balance ratios than the coarse-grain hypergraph, as its definition of an atomic task is smaller. Therefore, it is easier to partition. In terms of communication volume, both of the methods achieve better results than the others for various values of N. One disadvantage of the fine-grain hypergraph model is it is harder to implement than the coarse-grain hypergraph model using deep learning frameworks. That is because implementing the convolution of each channel requires much more optimization than the convolution of each filter.

Fine-grain	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	21,761	20,978	21,369	1.0	1.0	43,522
N=4	11,262	10,165	10,684	1.1	1.0	45,049
N=8	5,516	5,171	5,342	1.0	1.0	44,129
N=16	2,807	2,563	2,671	1.0	1.0	44,926
N=32	1,552	1,105	1,335	1.4	1.1	49,681
N=64	690	624	667	1.1	1.0	44,160
N=128	546	144	333	3.7	1.6	69,888

Table 5.11: Fine-grain Hypergraph Partitioning Computational Load Balance Analysis

Fine-grain	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg comm
N=2	27.5	24.5	13.7	1.7
N=4	56.3	24.5	14.0	1.7
N=8	307.3	98.0	38.4	2.5
N=16	549.2	196.0	31.8	6.1
N=32	916.8	197.5	28.6	6.8
N=64	1,340.1	275.6	20.9	13.1
N=128	1,964.0	444.0	15.3	28.9

Table 5.12: Fine-grain Hypergraph Partitioning Communication Volume Analysis

5.1.7 Data Parallelism

As described in the problem statement section, data parallelism is the other and the most used way of distributing deep neural networks. It is not meaningful to analyze data parallelism in terms of computational load balance as all of the replicas contain the same network model. However, we can analyze data parallelism in terms of communication volume and compare it with communication volumes obtained from hypergraph partitioning algorithms.

Table 5.13 shows output size, memory consumption, and number of parameters for each layer in VGG16. Output size shows the number of activations in every layer. During the training of a neural network, activations of early layers are also kept in the memory as they will be needed during the backpropagation phase. Parameters show the number of parameters the network contains. These parameters are updated using their gradients during backpropagation. Therefore, for deep networks such as VGG16, there should be enough amount of memory

Layer	Output Size	Memory	Parameters
INPUT	[224x224x3]	224*224*3 = 150K	0
CONV1	[224x224x64]	224*224*64 = 3.2M	3*3*3*64 = 1,728
CONV2	[224x224x64]	224*224*64 = 3.2M	3*3*64*64 = 36,864
POOL1	[112x112x64]	112*112*64 = 800K	0
CONV3	[112x112x128]	112*112*128 = 1.6M	3*3*64*128 = 73,728
CONV4	[112x112x128]	112*112*128 = 1.6M	3*3*128*128 = 147,456
POOL2	[56x56x128]	56*56*128 = 400K	0
CONV5	[56x56x256]	56*56*256 = 800K	3*3*128*256 = 294,912
CONV6	[56x56x256]	56*56*256 = 800K	3*3*256*256 = 589,824
CONV7	[56x56x256]	56*56*256 = 800K	3*3*256*256 = 589,824
POOL3	[28x28x256]	28*28*256 = 200K	0
CONV8	[28x28x512]	28*28*512 = 400K	3*3*256*512 = 1,179,648
CONV9	[28x28x512]	28*28*512 = 400K	3*3*512*512 = 2,359,296
CONV10	[28x28x512]	28*28*512 = 400K	3*3*512*512 = 2,359,296
POOL4	[14x14x512]	14*14*512 = 100K	0
CONV11	[14x14x512]	14*14*512 = 100K	3*3*512*512 = 2,359,296
CONV12	[14x14x512]	14*14*512 = 100K	3*3*512*512 = 2,359,296
CONV13	[14x14x512]	14*14*512 = 100K	3*3*512*512 = 2,359,296
POOL5	[7x7x512]	7*7*512 = 25K	0
FC1	[1x1x4096]	4096	7*7*512*4096 = 102,760,448
FC2	[1x1x4096]	4096	4096*4096 = 16,777,216
FC3	[1x1x1000]	1000	4096*1000 = 4,096,000
Total		93 MB / image	138,357,544 (with biases)

Table 5.13: VGG16 Memory and Parameter Analysis

to keep the parameters. Besides, batch size should be small so that all of the memory consumed by the network should fit into RAM.

5.1.8 Hypergraph Partitioning Based Model Parallelism vs Data Parallelism

In data parallelism, all of the parameters are needed to be communicated between processors after every iteration. The amount of communication for the total number of parameters is 527.792 MB for the VGG16 network given above as there are 138,357,544 parameters and each parameter is 32 bits. If there are n processors and collective communication primitives are used instead of parameter server, the

formula for communication volume becomes $2*(n-1)*thenumberofparameters$. To simplify, if we denote the number of parameters as p, the amount of the communication for data parallelism after each iteration is $2*(n-1)*p$. This shows that the amount of communication is related to the number of processors and the size of the parameters. Besides, since the number of iterations decreases as batch size increases, the amount of communication is also related to the batch size for one epoch. However, increasing the batch size can also lead to late convergence of the network and may not be possible as it is hard to fit a large network with big batch size into a single machine's memory.

To compare communication volume between data parallelism and hypergraph partitioning model parallelism, let's say we have data of 64000 images with shape [224x224x3] each. In Table 5.10, communication volume for different values of N with batch size 32 for only forward pass is given. Those communication volume values needed to be doubled for complete training procedures with forward pass and backpropagation. Then, we can compare the communication volume for different distributed neural network methods for 1 epoch using the VGG16 network.

In Table 5.14, communication volume comparison is done for data parallelism and hypergraph partitioning model parallelism using the experiment setup given above. For all number of processors and batch size values, hypergraph partitioning obtain better communication volumes than data parallelism. The proposed method in this thesis decreases the communication volume that exists in data parallelism by 93.315% on average.

5.2 Run Time Analysis

In the previous section, different methods for distributed deep neural network training are analyzed in terms of computational load balance and communication volume. In this section, the run time of the aforementioned methods will be analyzed based on different batch sizes, and different number of processors. In

Comm. Volume	Data Parallelism	Hypergraph Partitioning	Comm. Reduce
N = 2, BS = 16	4,123 GB	107 GB	97%
N = 2, BS = 32	2,061 GB	107 GB	94%
N = 2, BS = 64	1,030 GB	107 GB	89%
N = 4, BS = 16	12,370 GB	239 GB	98%
N = 4, BS = 32	6,185 GB	239 GB	96%
N = 4, BS = 64	3,092 GB	239 GB	92%
N = 8, BS = 16	28,863 GB	1167 GB	95%
N = 8, BS = 32	14,431 GB	1167 GB	91%
N = 8, BS = 64	7,215 GB	1167 GB	83%
Average			93%

Table 5.14: VGG16 Communication Volume Comparison for 1 Epoch

these experiments, a subset of data from the Imagenet dataset is used with 10240 images and run time for completing 10 epochs are measured in terms of seconds.

In this experiment, SKL (Skylake Xeon) nodes are used where nodes have 112 logical cores each. The Xeon nodes are connected by Intel Omni-Path (Intel OPA) 100 Series interconnect. All nodes are connected via 10Gb Ethernet. More information about each node(processor) can be seen in Figure 5.1.

5.2.1 Single Processor

In Table 5.15, a single processor is used to train VGG16 network for 10 epochs with 10240 images. Increasing batch size also decreases the run time as there will be less number of backpropagation stage and using higher batch sizes does not increase run time linearly as hardware is more optimized for larger operations. For example, doubling the batch size does not double the run time of forward pass and backpropagation. Although increasing batch size decreases the overall run time, it negatively affects convergence. When the batch size is high, calculated gradients become less representative for the given batch as there are data for many different classes in a single batch. Therefore, it takes much more time for the network to converge to a global minimum using a big batch size.

5.2.2 Model Parallelism

There are various types of model parallelism methods as described earlier. In this section, these different methods will be compared based on run time.

As expected, layer-wise parallelism achieves faster run time results than the single processor and run time results are shown in Table 5.15.

As shown in Table 5.15, the incremental naive partitioning algorithm on the coarse-grain hypergraph model achieves slower run time results than layer-wise partitioning. Both of the methods just focus on computational load balance without taking into consideration of the communication patterns inside a deep neural network. This shows us that partitioning a network using incremental naive method does not lead to better run time results.

Table 5.15 shows the run time results of the coarse-grain hypergraph partitioning. When we compare it with an incremental naive method, hypergraph partitioning obtains far better results for most of the N and batch size values. This shows that even if a deep neural network is converted into a hypergraph, communication patterns should be taken into consideration while partitioning it. When we compare hypergraph partitioning run time results with layer-wise partitioning run time results, it is seen that hypergraph partitioning obtains faster results again for most of the cases. Therefore, during accuracy and loss convergence performance tests between data parallelism and model parallelism, the coarse-grain hypergraph partitioned model parallelism will be used.

5.2.3 Data Parallelism

In this section, run time analysis of different data parallelism techniques will be done using the same setup used for model parallelism. As stated earlier, running 10 epochs faster does not mean reaching to the target accuracy faster. Therefore, it is not appropriate to say whether model parallelism is better than data parallelism in terms of epoch run times alone. Each parallelism technique

will be compared within itself with various changes.

The difference between asynchronous and synchronous data parallelism is described earlier. There are several parameters that we can change in data parallelism such as the number of parameter servers and parameter distribution algorithm. In one technique in data parallelism, we can set 1 parameter server so that each worker sends its updates to that parameter server and receives updated weights from there also. In another technique, instead of using a parameter server, each worker can behave like a small parameter server. This means that we can partition the parameters between different workers so that each worker is responsible for the update of the subset of parameters. While distributing parameters we can take consideration of different distribution algorithms. The first algorithm performs in a round-robin manner. Each parameter inside a network is assigned to workers iteratively. However, this also can lead to an imbalanced distribution of parameters as each parameter's weight is different from each other. The second algorithm performs greedily. Each parameter is assigned to the worker which has the minimum amount of assignments so far. Thus, more balanced parameter distribution is obtained between workers.

In Table 5.15, it can be seen that synchronous data parallelism achieves faster run times when the parameter server is distributed among workers. This shows that in synchronous data parallelism, where each worker waits for each other to complete updates, the distributed parameter server reduces the communication time between workers and achieves faster results. The same observation can not be made for asynchronous data parallelism as none of the workers wait for each other. Since we observed that the greedy distribution of parameters leads to better results than round-robin distribution, Table 5.15 provides the results obtained by greedy distribution for results using a distributed parameter server.

Run Time Analysis		BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=1	Single Processor	9405	9536	6265	5420	5150	4842
N=2	Layer-wise	5431	4718	3831	3560	3293	3079
	Filter-wise	5506	5002	4392	3557	3503	3301
	Coarse-grain Hypergraph	5630	4754	4012	3527	3276	2862
	Async. DP (1PS)	10256	6736	4758	3700	3106	2705
	Sync. DP (1PS)	10731	7231	5068	3938	3244	2790
	Async. DP (Dist. PS)	9757	6704	4833	3797	3214	2737
	Sync. DP (Dist. PS)	10140	6906	4979	3875	3176	2934
N=4	Layer-wise	3263	2801	2478	2209	2100	1949
	Filter-wise	3856	3407	2915	2672	2529	2313
	Coarse-grain Hypergraph	3306	2662	2050	1971	1814	1682
	Async. DP (1PS)	5361	3411	2366	1868	1569	1351
	Sync. DP (1PS)	6657	4181	2796	2120	1679	1510
	Async. DP (Dist. PS)	5007	3371	2421	1864	1567	1367
	Sync. DP (Dist. PS)	5898	3855	2628	2040	1684	1431

Table 5.15: VGG16 Per Epoch Run Time Analysis of Different Parallelism Methods in Seconds

5.2.4 Hypergraph Partitioning Based Model Parallelism vs Data Parallelism

It is possible to compare model parallelism which is applied using coarse-grain hypergraph partitioning and data parallelism in terms of run times using Table 5.15. For small batch size values such as 16, 32, 64, model parallelism achieves faster run times for one epoch. This is because the number of backpropagation increases when batch size is small and thus the number of communication for data parallelism increases. Therefore, data parallelism becomes slower than the

model parallelism for small batch sizes. For bigger batch size values such as 128, 256, 512, data parallelism becomes faster than model parallelism. Because, the number of backpropagation and amount of communication decreases for data parallelism, while the amount of communication stays same for model parallelism. However, having statistically efficient weights during training is more important than the fast run time of one epoch. What is meant by statistically efficient weights is that during the training of the neural network, always having the most up to date weights is important to achieve fast convergence. Therefore, in the next section, the convergence performance of model parallelism with coarse-grain hypergraph partitioning will be compared with data parallelism.

5.3 Accuracy and Loss Convergence Results

So far, different distributed deep neural network training methods are analyzed based on computational load balance, communication volume and epoch run time. In this section, all of these methods will be analyzed based on convergence performance, which is the most important run-time metric. In this experiment, 50 subclasses of the imagenet dataset are used to train a VGG16 network using different distributed neural network training methods. The reason for using a subset of imagenet is that a training epoch of the VGG16 network takes approximately 30 hours on a single processor using CPU. When we consider that there should be hundreds of epochs to reach the benchmark accuracy of VGG16, training of the network will take months. Additionally, reducing the number of classes in the dataset causes VGG16 to overfit. To prevent the network from overfitting, some regularization methods can be applied without changing the neural network structure. However, these regularization methods can be in addition to any of the methods. Therefore, without changing network structure and applying regularization methods, training accuracy and loss values are reported in this section.

In this experiment, SKL (Skylake Xeon) nodes are used where nodes have 112 logical cores each. The Xeon nodes are connected by Intel Omni-Path (Intel

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                112
On-line CPU(s) list:   0-111
Thread(s) per core:    2
Core(s) per socket:    28
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz
Stepping:              7
CPU MHz:               999.975
CPU max MHz:           4000.0000
CPU min MHz:           1000.0000
BogoMIPS:              5400.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              39424K
NUMA node0 CPU(s):     0-27,56-83
NUMA node1 CPU(s):     28-55,84-111
```

Figure 5.1: Node Specifications

OPA) 100 Series interconnect. All nodes are connected via 10Gb Ethernet. More information about each node(processor) can be seen in Figure 5.1. Also, run time analysis that was previously presented in Section 5.2 also obtained using the same cluster.

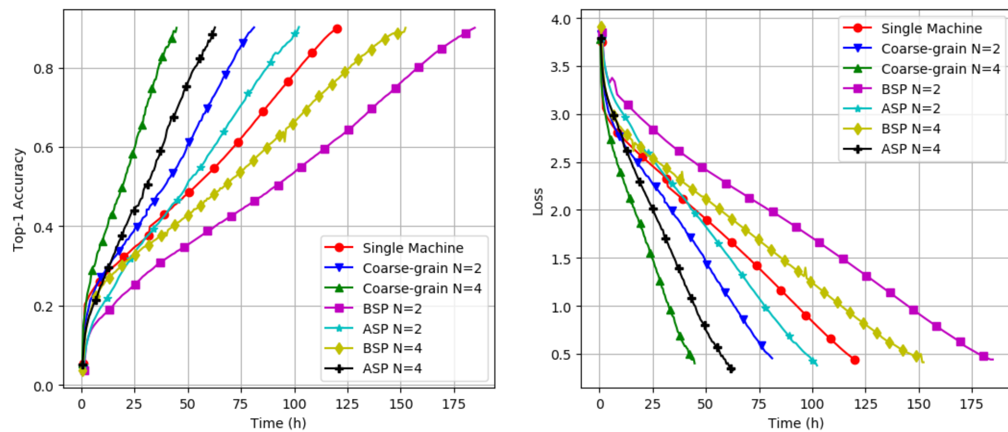


Figure 5.2: VGG16 Training Accuracy and Loss

In Figure 5.2, change in the training accuracy and loss are shown until accuracy exceeds the value of 90%. It takes almost 122 hours for a single processor to achieve the target accuracy, while the coarse-grain hypergraph model parallelism

achieves the target accuracy in almost 44 hours. This shows that the coarse-grain hypergraph model parallelism speeds up the training process ∼3x times using 4 processor. When different data parallelism techniques are compared, asynchronous data parallelism achieved faster convergence than synchronous data parallelism in this experiment setup, as also stated by other works [2, 15, 25]. It is also possible to compare the hypergraph partitioning based model parallelism with data parallelism techniques. When N is equal to 2, hypergraph partitioning model parallelism achieves ∼1.2x times faster convergence compared to the asynchronous data parallelism, and ∼2.25x times faster convergence compared to the synchronous data parallelism. When N is equal to 4, hypergraph partitioning model parallelism achieves ∼1.5x times faster convergence compared to the asynchronous data parallelism, and ∼3.5x times faster convergence compared to the synchronous data parallelism. This shows that for each value of N in Figure 5.2, hypergraph partitioning model parallelism achieves faster convergence than any other data parallelism technique. The performance gap between these two methods also increases as the value of N increases.

Chapter 6

Conclusion

Deep learning has become very popular because of its success in various fields. These successes have been mainly owing to the model's capacity to learn and represent complex functions. Besides, it has been shown that increasing the size of the models has a big impact on the performance. However, increasing the size of the model also causes it to not fit into a single machine's memory and training takes a very long time. There are two main efforts to solve this problem: model parallelism and data parallelism.

It is shown that the current model parallelism techniques have limitations, such as layerwise partitioning which causes high computational load imbalance and communication volume. In this thesis, new model parallelism techniques based on coarse-grain hypergraph and fine-grain hypergraph models are proposed to solve the limitations of traditional model parallelism. The first problem of traditional model parallelism is that it requires a human expert to decide on where to partition layers. Secondly, as long as a model is partitioned by placing a group of layers among different parts, having a high computational load imbalance is inevitable as work that is done by each layer is very different from each other. Besides the high computational load imbalance, the current model parallelism methods may also cause high communication volume. In this thesis, we proposed new model parallelism techniques to automatically partition the network

based on work that is done by each filter or channel and communication pattern inside the network. Thus, the proposed methods in this thesis obtain a better computational load imbalance while keeping communication volume low.

Data parallelism also has its limitations. The first one is that if the model does not fit into a single machine's memory, it also won't fit into the memory of replicas that are used in data parallelism. Therefore, the batch size has to be reduced to be able to take advantage of data parallelism. Also, since the size of the model is huge and there are millions of parameters, the amount of communication that is done in data parallelism is huge and time consuming. The model parallelism techniques proposed in this work reduce the communication volume of data parallelism by $\sim$93%. Finally, it is also shown that proposed methods reach the target accuracy faster than any data parallelism technique during training.

6.1 Future Work

Several future works can improve the results shown in this thesis.

- The same hypergraph partitioning model parallelism based algorithms should be applied on a cluster of GPUs. New algorithms proposed in this thesis are mainly bound by the computation power. If GPUs are used, those algorithms will perform better. Data parallelism techniques will have communication volume as the bottleneck even if computational power is increased. Therefore, the performance gap between the hypergraph partitioning model parallelism and data parallelism could become larger for experiments on a cluster of GPUs. Also, to report test accuracy and loss convergence performance results on a complete Imagenet dataset, GPU usage is a necessity. This is because it takes a very long time to train a deep neural network using the Imagenet dataset with a CPU cluster.
- Implementation of a fine-grain hypergraph is not presented in this thesis and it should be implemented. However, this implementation requires too

much effort on optimization as there are millions of connections and communications.

- Besides coarse-grain and fine-grain hypergraph models, there should be different hypergraph models to represent a convolutional neural network. For these new hypergraph models, the definition of atomic tasks should be different based on the problem.

- Hypergraph models of different types of convolutional neural networks should be done such as inception modules in [37] or residual connections in [13].

- Besides convolutional neural networks, there should be hypergraph models for deep neural networks that contain different types of layers and connection patterns such as LSTMs.

Bibliography

[1] Ümit Çatalyürek and Cevdet Aykanat. “Patoh (partitioning tool for hypergraphs)”. In: *Encyclopedia of Parallel Computing* (2011), pp. 1479–1487.

[2] Trishul Chilimbi et al. “Project adam: Building an efficient and scalable deep learning training system”. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 571–582.

[3] Adam Coates et al. “Deep learning with COTS HPC systems”. In: *International conference on machine learning*. 2013, pp. 1337–1345.

[4] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*. Tech. rep. WP-08608-001_v1.1. Aug. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[5] Henggang Cui et al. “Exploiting iterative-ness for parallel ML computations”. In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, pp. 1–14.

[6] Henggang Cui et al. “Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 2016, p. 4.

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

[8] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[9] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[10] Andre Esteva et al. "Dermatologist-level classification of skin cancer with deep neural networks". In: *Nature* 542.7639 (2017), p. 115.

[11] Priya Goyal et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour". In: *arXiv preprint arXiv:1706.02677* (2017).

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

[13] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[14] Donald O Hebb et al. *The organization of behavior*. 1949.

[15] Qirong Ho et al. "More effective distributed ml via a stale synchronous parallel parameter server". In: *Advances in neural information processing systems*. 2013, pp. 1223–1231.

[16] Yanping Huang et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism". In: *arXiv preprint arXiv:1811.06965* (2018).

[17] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[18] Jin Kyu Kim et al. "STRADS: a distributed framework for scheduled model parallel machine learning". In: *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 2016, p. 5.

[19] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[20] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *arXiv preprint arXiv:1404.5997* (2014).

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[22] Anders Krogh and John A Hertz. "A simple weight decay can improve generalization". In: *Advances in neural information processing systems*. 1992, pp. 950–957.

[23] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[24] Seunghak Lee et al. "On model parallelization and scheduling strategies for distributed machine learning". In: *Advances in neural information processing systems*. 2014, pp. 2834–2842.

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

[26] Yucheng Low et al. "Distributed GraphLab: a framework for machine learning and data mining in the cloud". In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 716–727.

[27] Lu Lu et al. "Dying ReLU and Initialization: Theory and Numerical Examples". In: *arXiv preprint arXiv:1903.06733* (2019).

[28] Martıén Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `http://tensorflow.org/`.

[29] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[30] Azalia Mirhoseini et al. "Device placement optimization with reinforcement learning". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 2430–2439.

[31] German I Parisi et al. "Continual lifelong learning with neural networks: A review". In: *Neural Networks* (2019).

[32] Esteban Real et al. "Regularized evolution for image classifier architecture search". In: *arXiv preprint arXiv:1802.01548* (2018).

[33] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[34] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117.

[35] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[36] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[37] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[38] Leslie G Valiant. “A bridging model for parallel computation”. In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[39] Minjie Wang et al. “Minerva: A scalable and highly efficient training platform for deep learning”. In: *NIPS Workshop, Distributed Machine Learning and Matrix Computations*. 2014.

[40] Ren Wu et al. “Deep image: Scaling up image recognition”. In: *arXiv preprint arXiv:1501.02876* (2015).

[41] Bing Xu et al. “Empirical evaluation of rectified activations in convolutional network”. In: *arXiv preprint arXiv:1505.00853* (2015).

[42] Matthew D Zeiler. “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (2012).

[43] Hao Zhang et al. “Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on {GPU} Clusters”. In: *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 2017, pp. 181–193.

[44] Martin Zinkevich et al. “Parallelized stochastic gradient descent”. In: *Advances in neural information processing systems*. 2010, pp. 2595–2603.

[45] Barret Zoph et al. “Learning transferable architectures for scalable image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710.

Appendix A

Hypergraph Partitioning Statistics

Coarse-grain	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	21,825	21,195	21,510	1.0	1.0	43,650
N=4	11,123	10,576	10,755	1.0	1.0	44,494
N=8	5,844	5,237	5,377	1.1	1.0	46,752
N=16	3,177	2,587	2,688	1.2	1.1	50,838
N=32	1,659	1,242	1,344	1.3	1.2	53,100
N=64	939	621	672	1.5	1.3	60,117
N=128	584	78	336	7.4	1.7	74,800

Table A.1: Coarse-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.03

Coarse-grain	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	21,825	21,195	21,510	1.0	1.0	43,650
N=4	11,280	9,914	10,755	1.1	1.0	45,121
N=8	5,654	4,844	5,377	1.1	1.0	45,238
N=16	3,177	2,403	2,688	1.3	1.1	50,838
N=32	1,714	1,252	1,344	1.3	1.2	54,874
N=64	960	612	672	1.5	1.4	61,454
N=128	599	144	336	4.1	1.7	76,743

Table A.2: Coarse-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.15

Coarse-grain	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	21,825	21,195	21,510	1.0	1.0	43,650
N=4	11,545	9,649	10,755	1.1	1.0	46,183
N=8	5,818	4,807	5,377	1.2	1.0	46,548
N=16	2,951	2,328	2,688	1.2	1.0	47,220
N=32	1,643	1,234	1,344	1.3	1.2	52,578
N=64	995	606	672	1.6	1.4	63,741
N=128	606	144	336	4.2	1.8	77,634

Table A.3: Coarse-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.2

Fine-grain	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	22,004	20,734	21,369	1.0	1.0	44,009
N=4	11,004	10,386	10,684	1.0	1.0	44,019
N=8	5,449	5,236	5,342	1.0	1.0	43,597
N=16	2,709	2,619	2,671	1.0	1.0	43,354
N=32	1,817	851	1,335	2.1	1.3	58,156
N=64	690	638	667	1.0	1.0	44,160
N=128	546	144	333	3.7	1.6	69,888

Table A.4: Fine-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.03

Fine-grain	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	23,107	19,631	21,369	1.1	1.0	46,215
N=4	11,665	10,097	10,684	1.1	1.0	46,663
N=8	5,793	4,880	5,342	1.1	1.0	46,351
N=16	2,787	2,526	2,671	1.1	1.0	44,606
N=32	1,409	1,267	1,335	1.1	1.0	45,092
N=64	700	598	667	1.1	1.0	44,863
N=128	546	144	333	3.7	1.6	69,888

Table A.5: Fine-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.15

Fine-grain	Max Work	Min Work	Avg. Work	Max/Min	Max/avg	Max*N
N=2	22,708	20,031	21,369	1.1	1.0	45,416
N=4	11,012	10,353	10,684	1.0	1.0	44,048
N=8	5,604	5,104	5,342	1.0	1.0	44,833
N=16	2,802	2,573	2,671	1.0	1.0	44,833
N=32	1,463	1,061	1,335	1.3	1.0	46,821
N=64	711	595	667	1.1	1.0	45,532
N=128	546	144	333	3.7	1.6	69,888

Table A.6: Fine-grain Hypergraph Partitioning Work Imbalance Analysis with Final Imbalance = 0.2

Coarse-grain	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg Comm.
N=2	27.5	24.5	13.7	1.7
N=4	63.4	24.5	15.8	1.5
N=8	303.1	98.0	37.8	2.5
N=16	659.5	196.0	41.2	4.7
N=32	1,302.1	490.0	40.6	12.0
N=64	2,477.8	1,078.0	38.7	27.8
N=128	4,634.2	1,960.0	36.2	54.1

Table A.7: Coarse-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.03

Coarse-grain	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg Comm.
N=2	27.5	24.5	13.7	1.7
N=4	59.8	24.5	14.9	1.6
N=8	293.1	98.0	36.6	2.6
N=16	655.7	196.0	40.9	4.7
N=32	1,288.9	490.0	40.2	12.1
N=64	2,430.8	980.0	37.9	25.8
N=128	4,582.7	1,960.0	35.8	54.7

Table A.8: Coarse-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.15

Coarse-grain	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg Comm.
N=2	27.5	24.5	13.7	1.7
N=4	58.4	24.5	14.6	1.6
N=8	282.4	98.0	35.3	2.7
N=16	629.9	196.0	39.3	4.9
N=32	1,284.0	392.0	40.1	9.7
N=64	2,479.5	980.0	38.7	25.2
N=128	4,510.6	1,960.0	35.2	55.6

Table A.9: Coarse-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.2

Fine-grain	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg Comm.
N=2	26.6	12.2	13.3	0.9
N=4	61.2	24.5	15.3	1.5
N=8	288.9	98.0	36.1	2.7
N=16	540.0	196.0	33.4	5.8
N=32	812.9	196.0	22.2	7.7
N=64	1,344.0	298.5	20.3	14.6
N=128	2,114.1	457.8	16.5	27.7

Table A.10: Fine-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.03

Fine-grain	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg Comm.
N=2	15.3	12.2	7.6	1.6
N=4	61.0	24.5	15.2	1.6
N=8	260.0	98.0	31.7	3.0
N=16	426.2	196.0	35.0	5.5
N=32	891.2	196.0	27.4	7.1
N=64	1,382.8	290.9	21.0	13.8
N=128	1,971.1	418.0	15.3	27.1

Table A.11: Fine-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.15

Fine-grain	Comm. Vol	Max Comm.	Avg Comm.	Max/Avg Comm.
N=2	24.5	24.5	12.2	2.0
N=4	72.8	24.5	13.7	1.7
N=8	303.2	98.0	37.9	2.5
N=16	532.6	196.0	33.1	5.9
N=32	851.4	196.0	26.2	7.4
N=64	1,410.2	272.5	21.5	12.6
N=128	2,123.8	526.7	16.5	31.7

Table A.12: Fine-grain Hypergraph Partitioning Communication Volume Analysis with Final Imbalance = 0.2