上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS

论文题目：Deep Neural Network Partitioning in Distributed Computing System

学生姓名:	申继媛
学生学号:	5130309194
专　　业:	计算机科学与技术
指导教师:	蒋力
学院(系):	电子信息与电气工程学院

Deep Neural Network Partitioning in Distributed Computing System

ABSTRACT

In recent years, there is a research trend towards deep neural network training in distributed computing systems. Despite the ubiquitous utilizations of *Deep Neural Networks* (DNN) across various applications, considering communication costs and limited resources, it is generally difficult to deploy DNNs on distributed computing systems in model parallelism style. As a decisive section of distributed DNN training, *deep neural network partitioning* (DNN-Partition) is a global process to partition and allocate deep neural networks to different distributed computing nodes while minimizing their communication costs and keeping network training accuracy and load balance.

Distributed System Researchers have developed amounts of significant distributed computing frameworks such as MXNET etc. in order to exploring distributed deep neural network training. What deserves to be mentioned the most is Parameter Server which achieves efficient communications between machines by figuring out commonly required data for each component. However, traditional approaches only pay attention to definite data communication rules: making changes to their intrinsic communication schemes or designing overall distributed computing frameworks. More clearly, there is not yet a well-established software-level theory to accomplish DNN-Partition in distributed computing systems, and the vast majority of work in distributed DNN Training has focused on system-level framework design. To advance the research of

DNN-Partition in this thesis, we propose to formulate DNN partition as an independent problem so that more statistical and machine learning algorithms could be further investigated in distributed systems.

In this paper, one novel framework including two partition schemes are proposed which manage to optimize DNN-Partition in distributed computing systems (both data and workloads are allocated across the distributed computing systems). Our proposed framework, referred to as ***deep neural network balance graph partition*** (GraphDNN), takes steps to compress dense weight matrix, analyze deep neural network features of weights and deep neural network nodes connections and finally cluster deep neural network nodes according to distributed system partition demand. Since the fully-connected layer and the convolutional layer in deep neural networks maintain distinctive attributes. Two partition schemes included here are designed separately. We are concerned about dense weight matrix processing for fully-connected layers; on the other hand, the continuity of convolution spurs us to rearrange inputs for convolutional layers. We also proposed further dynamic partition optimizations: dynamic pruning where low rank connectivity is pruned during training process; greedy cross-partition fixing where values lie linear distances are masked; and explorations on ReLU function zero results. Experiments showed that GraphDNN can partition already trained DNN models into different distributed computing nodes and alleviate communication costs and memory storage to a great extent while keeping training accuracy and system load balance.

Keywords: Distributed Computing System, Deep Neural Network Partitioning, Distributed Deep Neural Network Training, Dynamic Partition Optimization

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Distributed Computing System

1.1.1 Distributed Computing System

In its most general sense, a *distributed computing system* is an abstract concept which refers to a whole system where individual componential working devices are then simplified as mathematical nodes connected by the underlying networks (in this paper, we refer to them as distributed working nodes or computing nodes) so that (1) they can maintain one shared computation limit and (2) they will need to communicate or say coordinate their actions through passing messages or data. Notice here, the structure for network connecting should be in one-to-one connection style. If done properly, each of these distributed computing nodes can then separately perform themselves from each other like single independent entities. As following descriptions, this paper will then give some general definitions and important properties of distributed computing systems convenient for further discussions and algorithm designs.

A ***distributed computing system*** (DCS), much more as an objectively conceptual definition given here, consists of a finite number of individual computing nodes, as shown in figure 1.1 (a system where four separate computing nodes are involved). For

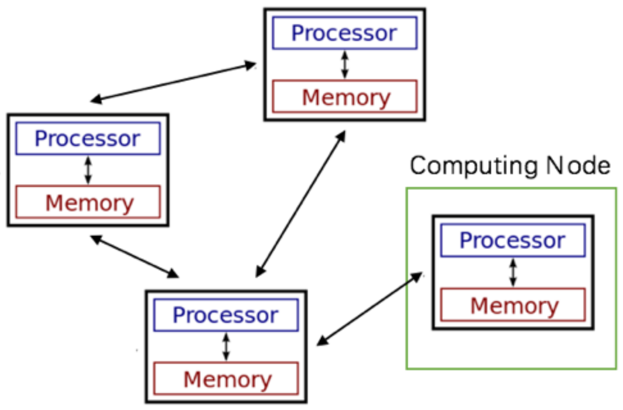

Figure 1.1 Diagram of a Distributed Computing System

users, each computing node in distributed computing system is regarded as a single and individual entity which usually contains two major modules: the processor and the memory, working for computations and storage separately. All of these single computing nodes are connected though their global underlying network for data communication targets, and each computing node can present different computability. Computing nodes here theoretically work as a substitute concept of real computing devices such as the graphics processing unit (GPU) or the mobile devices (such as the mobile phone) etc. Furthermore, there are two important characteristics of these distributed computing systems: the communicating and concurrency of different computing nodes and the independent down failure of each node.

1.1.2 Distributed Deep Neural Network Management

Traditionally, the whole deep neural network training and running stages are performed in only one single device. However, recent work in unsupervised learnings and deep learning researches has shown that being able to train much large network models can facilitate the classification or generation performance to a great extent. Therefore, there is a pressing demand for handling large scale deep neural networks to spread their work load including training and running across these distributed computing systems in pursuing further parallelism and higher performance. Furthermore, along with related dramatic benefits in performance for distributed methods, when applying model parallelism ideas, problems concerning with ***distributed deep neural network management*** are now becoming the core focus as well in both of the academia and the industry: how to appropriately deploy these complex deep neural network models in distributed computing systems; or how to actively enhance the improvements of performance in both of the optimization level and inference level of distributed deep neural networks.

In recent years, more and more researchers began taking efforts to work on such distributed deep neural network management problems in order to minimize inter communication costs while as well as keeping utilizations in leveraging work load of each single computing node accordingly. For instance, many of these large-scale deep neural network researches hammer at model replications. In particular, a software framework called DistBelief ([1], Dean et al. (2012), NIPS.) provides an asynchronously underlying procedure for replicated models and another framework referred as Sandblaster ([2], Dean et al. (2012), NIPS.) to support a variety of distributed batch optimization procedures. What's more, an innovative and attractive theory for this is the famous parameter server framework (Li et al. (2014), Mu Li, Dave Andersen, Alex Smola et al., 2014 OSDI.) ([3], Li et al. (2014), NIPS.). The parameter servers include all partitions of some of these public parameters for all distributed computing components. Workers (these distributed computing nodes in the same system with the parameter servers) communicates directly and simply with these parameter servers, and there is a global scheduler component for each working group to keep noticing the overall computation progress and deploy their work load appropriately. That is the parameter server is referred as a global parameter storage space for maintaining synchronization in distributed computations which shows computations in data parallelism style.

To specifically conclude, current researches on the distributed deep neural network management problem concentrate on developing system-level frameworks based on data parallelism principles. In this thesis, we, from a different stand, take up with model partitioning methods and target at developing a software-level framework for the deployment of deep neural networks in distributed computing systems based on the model parallelism principles.

1.2 Our Results and Organization of Thesis

1.2.1 Main Contribution of This Thesis

Two static partitioning schemes (Section 4.1) are proposed separately in our novel framework called deep neural network balance graph partition (GraphDNN). One partitioning scheme is designed for the fully-connected layer where dense weight matrix mainly counts: it hammers at neural network deep compressing, learning and analyzing features and connections in DNN and finally clustering single nodes as partitions. The other partitioning scheme is based on the continuity feature of convolution layers: the layer compression [sparsity]. Three dynamic partitioning optimizations (Section 5.1) on low rank connectivity pruning, greedy cross-weight fixing and proceeding explorations on the ReLU activation function is further proposed as well. Experimental results in Section.4 and Section.5 demonstrate impressingly great accuracy, communication cost reduction, workload balance and state-of-art deployment performance.

1.2.2 Organization of This Thesis

The rest of this paper is organized as follows. Section 2 reviews related background statistical approaches regarding deep neural network concepts, neural network compression schemes and graph partition algorithms. Then we formulates the Deep Neural Network Partitioning problem in distributed computing systems in Section 3, after which we overview the overall motivation of our work. The proposed static partition framework GraphDNN is discussed in Section 4, followed by further dynamic optimization mechanisms in Section 5. Finally, Section 6 concludes the whole thesis.

Chapter 2 Background Statistical Approaches

Our proposed framework GraphDNN [details in Section 4] develops its performance on the strengths of those deep neural network compression methods and mathematical graph partition algorithms. Therefore, in order to have a more comprehensive understanding of the GraphDNN framework, this chapter specifically explains related background ideas, concepts and algorithms of these background approaches respectively.

2.1 Basics of Deep Neural Network

2.1.1 Basics of Neural Network

Neural Network is then a mathematically computational model, which typically consists of three sorts of abstract layers: *input layer*, *hidden layer* and *output layer*. Each layer is made up of some computing nodes (currently there is no widely acknowledged definitions of neural network available in academia, here we simply try to make some general explanations). As we demonstrated in the following figure 2.1, a *node* is a place where addition and multiplication computations happen: it processes the given input data by combining the input vector from the data with a set of coefficients, or more normally say weights, as denoted on the graph edges (one node deals with one entry in the given vector). After that, these input-weight production pairs are then summed up together, and their addition results are then passed through a node's so-called activation function, to determine whether or not and to what extent this middle-referred signal progresses further through the network to affect the ultimate outcome.

From specifically mathematical respect, in each node, if given a m-dimensional real-valued input vector $\{x_1, x_2, ..., x_m\}$ where x_i represents the i-th input matrix value; and a m-dimensional real-valued weight vector $\{\omega_1, \omega_2, ..., \omega_m\}$ where $\omega_i (i \neq 0)$ measures

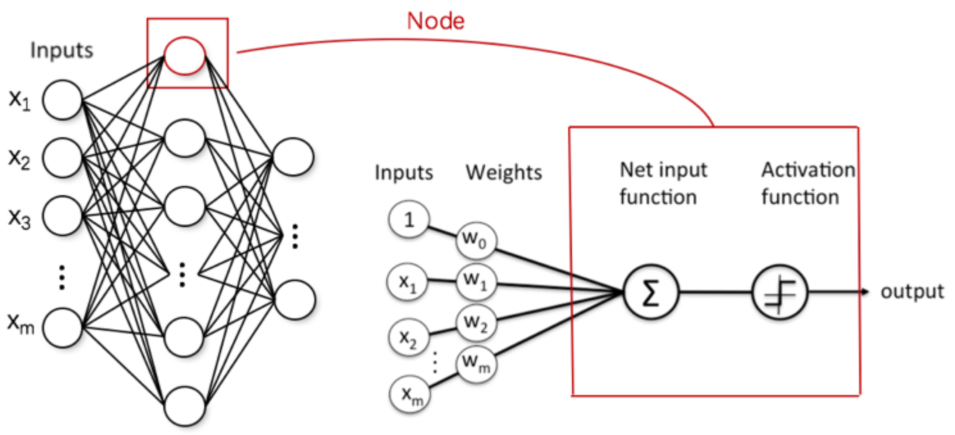

Figure 2.1 Diagram of a Computing Node in a Neural Network

the progress effects of x_i; then the node conducts its computations as

$$y_j = g(\omega_1 x_1 + \omega_2 x_2 + ... + \omega_m x_m + \omega_0) = g(\sum_{i=1}^{m} \omega_i x_i + \omega_0) \tag{2-1}$$

where $g(\centerdot)$ denotes the layer's activation function and ω_0 denotes the layer's bias.

As shown in the above equation (2-1), computations in each layer actually execute some kind of space conversion from the input feature space to the output classification space, where the overall process could be regarded as five atom operations in sequence: dimensional ascend or reduction $(\vec{\omega} \centerdot \vec{x})$; amplification or contraction $(\vec{\omega} \centerdot \vec{x})$; rotation $(\vec{\omega} \centerdot \vec{x})$; translation $(+\omega_0)$ and curve (g).

2.1.2 Deep Neural Network

Single Layer Neural Network (Perceptron) is categorized as those neural networks with only two layers: that is one input layer and one output layer. No hidden layer is available here. As the simplest neural network model configuration, perceptron generally works as a linear classifier that based on a linear predicting function combining a set of weight coefficients with the feature vector: it learns classifications with the dot product of weight values and input data. What's more, the perceptron algorithm also allows for further dynamic online learning, the reason for this lies in that it processes elements in the training set one at a time.

Deep Neural Network (DNN) referred to neural networks with layer configurations as: one input layer, one output layer and $n(n > 1)$ hidden layers. As explained above, the hidden layers can be considered as increasingly complex feature transformations, and the final layer would make use of the most abstract features computed through all former layers. DNN architectures can generate compositional abilities that express the potential of modeling extremely complex data sets with fewer computing units compared to those shallow neural networks, which making DNN architectures usually targeted for non-linear expressions and problems. Although DNNs are typically designed as feedforward networks, considering high performance benefits brought with by the recurrent networks, two sorts of hidden layers are concluded for DNNs:

(1) **Fully-Connected Layer:** All neurons (or here computing nodes) in a fully-connected layer have full connections to all activations (i.e., the activation function applied on input data for computing values) in the previous layer, just as seen in regular neural networks. Computations in the fully-connected layer are exactly as what we detailedly described above: these activations are computed with a direct matrix multiplication followed by a addition compensation with the biased offset.

(2) **Convolutional Layer:** Extremely different from the fully-connected layer, conceptual configurations and parameters of the convolutional layer contain a group of learnable convolutional filters or say the convolutional kernels, which may have a comparably small receptive field, but will travel itself across the entire length of the input data according to its given configurations. During the forward pass, each convolutional filter is convolved across the width and the height dimension of the input data configuration by: First, computing the dot product between the vertex components of its convolutional kernel and those input values; and next producing a corresponding activation map result of that kernel. As a result, the entire network then learns the convolutional kernels that will activate themselves when they detect some specific type of feature at some configuration positions in the input values. Stacking these middle-reach activation map results for all kernels along the length dimension can finally form the full output space of the convolutional layer. Every element in the output space side thus can also be well explained as the output of a neuron that is homologous to some small region in

the input data configuration and shares its parameters with neurons in this same activation map result. What's more, further details about the convolutional layer would be discussed in Section 3 and Section 4 as well.

2.2 Neural Network Compression

2.2.1 Problem Description and Overview

Intricate matrix or vertex computations and intensive memory storage space consumptions can naturally induce corresponding difficulties in deploying deep neural networks in distributed computing systems: a large volume of task problems, for instance, communication costs, memory bandwidth consumptions and storage space utilizations gradually explode during the distributed online training process, which is introduced by enormous configuration parameters, input parameters and the complex architectures of distributed deep neural networks as well.

This obvious obstacle attracts many respective research points. Denil et al. ([4], Denil et al. (2013), NIPS.) showed that surprisingly large weight redundancy existing in scaling deep neural networks: their utilizations of low rank weight matrices proved that much less values could be utilized accurately enough to reconstruct the entirely original deep neural network (redundancy property) without decisive loss of accuracy. This interesting discovery makes deep neural networks a big accomplishment in our computer science field, and it activates further explorations on this deep neural network redundancy: for instance, **Neural Network Compression** focuses on reducing storage space demands of deep neural networks (the total weight count and/or their representation formats) by taking advantages of this redundancy property. Recent work on the neural network compression methods showed important advances. Lei et al. ([5], Ba and Caurana (2013), NIPS.) illustrated that further network trainings (i.e. retraining) on the exact configurations of an already trained one could then compress the corresponding deep neural network to an equally performed but comparably much less deep neural network. Courbariaux et al. ([6], Courbariaux et al. (2015).) also showed that very low demand in storage space is available enough for both running the trained networks and

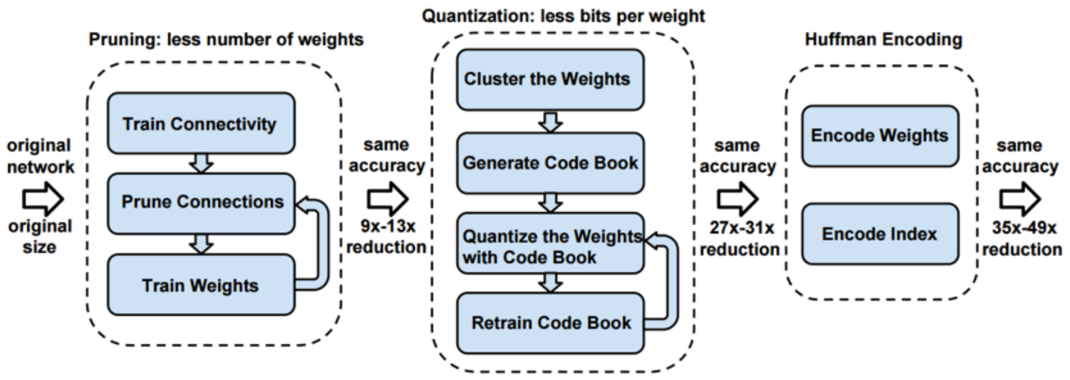

Figure 2.2 Deep Compression Three-Stage Pipeline Diagram

training them. Impressively, Song Han et al. ([7], Han et al. (2015), ICLR) recently figured out their 'deep compression', a three-stage compression framework, which then achieved around 40x storage space decrease; around 3.5x running speedup and around 5x energy consumption efficiency for compressing neural networks. As we will utilize deep compression to compress the deep neural networks as a step in our GraphDNN, next paragraphs will have some discussions on detailed descriptions about the algorithm deep compression proposed in [7](Han et al. (2015).).

2.2.2 Deep Compression

As shown in figure 3.1, deep compression consists of three stages considering both the number of total weight synapses, the exact weight values and their respective representation formats: pruning, quantization and Huffman encoding. Overview steps are: First, we prune the deep neural network by setting a reasonable (normally a small value) weight threshold and accordingly removing those potentially redundant connections (normally we will refer to those synapses with smaller weight values than our predefined threshold). Next, remaining weight values are then quantized (i.e. cluster these weight values under the principle - less representation bits per weight value) to ensure a less space code book demanded with weight values and biases. Finally, in taking advantages of the biased network weight distribution, Huffman Encoding provides a more simplified value representation. (Note that deep compression does not really influence the deep neural network accuracy.) Specifically discussing: given the Training Dataset and the Network Model, we first learn the deep neural network connectivity by normal training, its weight matrix is generated in this way.

Network Pruning: [connectivity cut]

Prune those connections with comparably small weight values, i.e., directly cut off the network connections whose absolute weight values are smaller than the pre-stipulated threshold or more accurately a ratio of the total number of weight values. After that, the pruned deep neural network should be retrained to learn this new network connectivity based on the remaining trained weight values. If given a prune ratio r, the prune compression step will produce a compression with exactly the compression rate of: r. Tricks can also be utilized for further optimization regarding memory storage space: for instance, the sparse weight structure (the codebook and indices) is stored in Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) format.

Quantization: [Less space for weight value presentation]

The pruned network is further compressed by specifying each weight with less bits. We quantize the effective weights and share these quantization value clusters in multiple connections - the K-Means Clustering Algorithm: we partition n connections into k clusters in which each connection belongs to the cluster that maintains the nearest (or smallest) mean with it. After clustering, the network retraining process is utilized here again to refine those global weight values. If given k clusters, for a neural network with n connections and each connection is represented by b-bit numbers, the quantization step will finally produce an effective rate of around: $nb/nlog_2k + kb$. More tricks can also be utilized here for further optimizations regarding memory storage space of the code book: for instance, encoding all the positions with index differences as well as paddings instead of exactly their absolute values.

Huffman Encoding: [Further less space for presentation]

Different representation bits are deployed according to the occurrence frequency of respective weight values, i.e. weight values with a high occurrence frequency would then be encoded with less representation bits; those weight values with a comparably low occurrence frequency would then be denoted with more representation bits. The Huffman encoding step, as an exactly mathematical optimization stage, can further reduce storage demand of a code book for deep compression. In the following paragraphs, discussions and experiments will focus more on the inter-node communication costs and regardless of this encoding part.

To specifically conclude, the deep compression algorithm compresses the whole neural network with less connectivity to an impressing extent according to the paper [7](Han et al. (2015)). What's more, the algorithm is clear and effective, and the compressed weight matrix can be directly utilized for computations and retraining.

2.3 Graph Partition

2.3.1 Problem Description and Overview

Consider a Graph $G = (V, E)$ here, with possibly edges weights and vertices weights, denoted as (W_V, W_E), and possibly coordinates for vertices (e.g. for meshes); we want to partition G into k pieces such that (1) Node weights are balances across every different partitions; (2) The overall arithmetic sum of the weights of those cut edges is minimized. For this, the important special case is when $k = 2$. Such sorts of partition problems have been discussed in literature as bicriteria-approximation or resource augmentation approaches. A common extension is tothe hyper-graph where one edge can connect more than two vertices. Actually when it is applied to the deep neural network mathematical model, hyper-graph partitioning matches.

Graph Partition is a typically and mathematically NP-Hard problem, which means that there are no perfect methods or mechanisms as its solutions in this field, and all of those practically exhibited graph partition solutions are based on heuristics. Generally, there are two broad categories of these partitioning methods as: the coordinate ideas and the coordinate-free ideas.

Typical coordinate ideas are based on graphs represented in regular meshes or general plots. For instance, the Recursive Coordinate Bisection ([8], Williams (1991), CPE.) simply makes choices on a cutting hyperplane which is parallel to a coordinate plot for partitioning; the Inertia Bisection ([9], Leland and Hendrickson (1994), SHPC.) then optimizes this kind of cutting hyperplane based on the vertex density, which maintains restricted to hyperplanes principles where nothing basically changes; the Random Cycles ([10], Arora et al. (2009), J. ACM.) then makes stereographic projection for graphs, which randomly takes some finite great three-dimensional circles for projected graph, undoes those stereographic projection and then converts the circle to a separator (choose the best cycle of these random ones).

On the other hand, coordinate-free ideas really count in this thesis as well. For instance, the Breadth-First-Search firstly picks up a start vertex (might start from several different vertices), labels all nodes by their distance from the start vertex and partitions the graph according to these computed distances; the Greedy Refinement starts with a preprocessing w strategy and refines this deployment method by the purely greedy strategy where well-known algorithms are Kernighan-Lin ([11], Kernighan and Lin (1970), J. Bell Sytem.) and Fiduccia-Mattheyses; the Spectral Partition then relies on those independent features of the entire graph where a deployment strategy is then derived from the spectrum of the corresponding adjacency matrix; Multi-layer Partition then works by applying one or more stages where a wide variety of partitioning and refinement strategies can be utilized. As we quickly comparing the above three partitioning techniques: Breadth-First Search is simple but may not provide great partitions; Kernighan-Lin is a good corrector and if given one reasonable partition is given, it can perform well; Spectral Method produces good partitions, but it can be comparably slow in processing speed stands.

In order to get deeper understandings on the graph partition problems, the following subsection will make some introductory discussions on the very significant algorithms which is optimized and applied in our framework: the Spectral Partition mechanism.

2.3.2 Spectral Partition

Given a graph $G = (V, E)$ with its adjacency matrix A, where an element A_{ij} implies an edge between the node i and the node j, and its degree matrix D, which is a diagonal matrix, where each diagonal entry of a row i, d_{ii}, represents the node degree of node i. The Laplacian matrix L is defined as the difference of degree matrix and adjacency matrix, i.e. in the formulation L is computed as $L = D - A$. Now, a ratio-cut partition for the graph G is defined as a partition of V into disjoint U, and W, under the minimization of the following ratio:

$$\frac{|E(G)\bigcap(U \times W)|}{|U| \cdot |W|} \tag{2-2}$$

which means the total number of edges that actually cross this partition to the total number of pairs of vertices that could support such edges.

Fiedler eigenvalue and eigenvector The second smallest eigenvalue, λ_2, of L reaches the optimally minimized cost of ratio-cut partition. Naturally, the eigenvalue V_2 corresponding to λ_2 is referred as the Fiedler vector which makes two partitions with each element signs. What's more, further repeating the intuitively equal partition can actually split the original graph into desired multiple components, e.g. k.

The Spectral Partition algorithm brings about three great advantages here: Firstly, the global computation property (does not rely on specific local points, and also can relieve the local minimum problem or the local maximum problem); Secondly, the partition performance (can produce good partitions); Thirdly, the computation convenience (directly compute eigenvalue and vectors will refer to partition results).

Chapter 3 Deep Neural Network Partitioning

3.1 Deep Neural Network Partitioning

Based on the distributed deep neural network management (defined in Section 1.1.2), ***deep neural network partitioning (DNN-Partition)*** takes efforts to enforce software-level partition algorithm design to deploy complex deep neural networks in distributed computing systems while satisfying the work load balance in this model parallelism as well as the communication cost minimization target.

Given a distributed computing system which comprises k independent computing nodes denoted as $\{\Psi_1, \Psi_2, ..., \Psi_k\}$ with corresponding computability denoted as $\{\varphi_1, \varphi_2, ..., \varphi_k\}$. Given a deep neural network, denoted as $\mathbb{C}$, trained on this distributed computing system with model parallelism, DNN-Partition then explores the problem that $\mathbb{C}$ need to be split into k corresponding partitions under the requirements of:

► **Work Load Balance** Partitions in $\mathbb{C}$, denoted as $\{\mathbb{C}_1, \mathbb{C}_2, ..., \mathbb{C}_k\}$, have respective computing node counts, denoted as $\{\mu_1, \mu_2, ..., \mu_k\}$. Work load balance is promised only when the prerequisite conditions here such as $\mu_1 : \mu_2 : ... : \mu_k = \varphi_1 : \varphi_2 : ... : \varphi_k$ or $\mu_1 : \mu_2 : ... : \mu_k \approx \varphi_1 : \varphi_2 : ... : \varphi_k$ is achieved. Put an explanation in a GPU platform here, then the computability could be the number of multiple addition operation, the computing node could be the number of threads.

► **Communication Cost** The exact format space of the transmitted data is measured as communication cost during each process. Total communication costs would be an additional result as $\sum x_{format}$.

After the deep neural network is well deployed, both the inference process and the retraining process can reveal that communication costs between computing nodes can bring out significant performance degradations, which makes distributed deep neural network impractical in real applications. As mentioned in Section 1, researches on DistBelief and parameter server etc. choose further system-level algorithms for data storage and transmission. Their solutions aimed at providing novel strategies for useful data without any consideration of specific deep neural network. DNN-Partition problems then stands on a totally different angle of view. Instead of considering system configurations and implementations, DNN-Partition takes the more software-level stands: to find algorithms for directly splitting the deep neural network into k corresponding partitions. Therefore, in this paper, deep neural network compression mechanisms and partition algorithms are of big interests.

3.2 Motivation

The gap between large computing workloads of deep neural networks and limited computing resources of mobile devices adversely impact user experience and inspired some research works to fill the gap. As explained in chapter 1, different from former researches on system architecture strategies, this project desire to consider distributed deep neural network training in a software style.

In addition, as mentioned earlier in this chapter, many studies, in client-server paradigm or local distributed computing system, have been performed to reduce the computing workloads of deep neural networks, such as model compression. However, we note that computation ability as well as the storage space shown by each single entity cannot be promised. It means that even through compression, the original workloads of deep neural networks have been saved greatly, amounts of distributed computing systems still cannot afford themselves for these computation tasks. Therefore, we consider to first compress original networks and then deploy the network among all single computing entities in the distributed computing system.

Furthermore, as we expressing the deep neural networks in the mathematic format of matrices, it is natural for us to introduce graph partition methods (or matrix partition methods) to statically partition a deep neural network. Next, we will have detailed discussions about our static framework and dynamic optimizations.

Chapter 4 DNN Static Partitioning Algorithm Design

The static partitioning component designed for deploying deep neural networks in distributed computing system is introduced in this chapter in detail, including all deep neural network compression processes and the deployment stages before distributed online training in separate computing work nodes. Static partitioning algorithm, as we define here, is suitable enough for deciding the deep neural network deployment in distributed computing system. The designed algorithms can achieve great effects in both efficiency, conciseness and clearness. And experiments later in this chapter also show that our static algorithms for deploying deep neural networks realize both impressing reduction in inter-node communication costs and largely less storage space demand.

4.1 GraphDNN: DNN Static Partitioning Algorithm Design

After our experiment results shown in figure 4.1, it is significant to mention that a distinct property of two different DNN layers are explored as: the convolutional layers contribute to the majority (e.g. 86.5% to 97.8%) of computational time; while the fully-connected layers mainly (e.g. more than 87.1%) cost total memory storage space for its model parameters. In consideration of such property difference exhibiting in their main cost fields, our proposed framework GraphDNN then separately designs deployment methods for the fully-connected layers and the convolutional layers, and it aims at

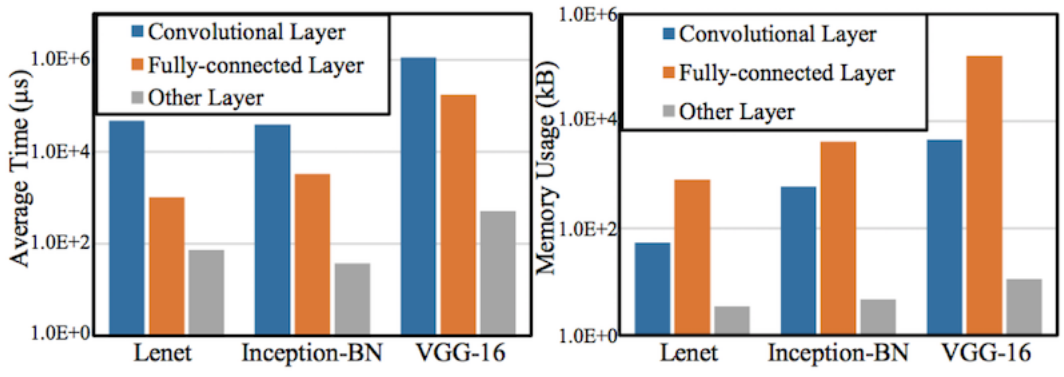

Figure 4.1 Average computing time and memory usage of the layers in DNNs

keeping fewer network connections in the fully-connected layers and keeping the exact exhibition of the convolutional layers in every distributed computing work node.

4.1.1 Fully-Connected Layer

Each neuron in a fully-connected layer has full connections to all activations from all neurons in the previous layer, as seen in regular fully-connected neural networks. Their activations can therefore be computed with a matrix multiplication operation followed by a biased offset (Also see chapter 3 of this thesis for more detailed descriptions about fully-connected layers). In fact, the fully-connected layer can provide the desired performance, or results, enough for some general classifications. However, normally when it comes to deep neural networks, the large amount of network weight values companied with a great amount of fully-connected layers makes it extremely cost-ineffective in the communication. Considering such dense weight matrices, the designed algorithm hammers at weight matrix rarefaction and network partition.

GraphDNN, as algorithm 1 clearly clarifies, first applies Song Han's deep compression [as detailedly described in Section 2.1.2] to these fully-connected layers, which then returns the desired weight matrix in a sparser format with roughly equal performance (only some minor degradations acceptable in accuracy) compared to the original dense weight matrix. Basically, deep compression separately processes each fully-connected layer: first, prune all these weight with its weight smaller than a specific threshold or a ratio of the total connection count [the specific threshold or definite ratio configuration can affect network test accuracy, as illustrated by experimental results in this chapter]; then, quantize the remaining weight values (k-means clustering) and store those shared weight values and their relative positions to every corresponding synapse connection in the codebook; finally, utilize the Huffman encoding to encode all these particular values including weight values and positions under the constraints of the corresponding entropy of all remaining data. In short, after the network compression stage been applied to deep neural networks, we can now get a much sparser weight matrix for each fully-connected layer which thus largely reduces the storage space demands, saves a great amount of connecting synapses, improves average computing efficiency of further deployment algorithms and also significantly contributes to communication cost reductions between distributed computing work nodes.

Now that we get the whole deep neural network with its corresponding sparse weight matrix, the next step is to partition this deep neural network in distributed computing systems. As detailed discussions about the deep neural network partitioning problem in Chapter 2, GraphDNN focuses on satisfying work load balance in this model parallelism as well as communication cost minimizations. In this paper, work load balance is evaluated by neural network capacity, that is the number of nodes and the computability of corresponding computing node; communication cost is then evaluated by weight value formats of cross-partition synapses and the number of total cross synapses.

Our partitioning framework deploys deep neural network based on the spectral factorization in every single layer ([12], Hespanha (2004).). Speaking of one fully-connected layer, all of its input nodes, output nodes and their connections are regarded as nodes and edges in its corresponding weighted and undirected graph to mathematically modeling (both input neurons and output neurons correspond to nodes; connecting synapses correspond to edges), and formerly trained weight values to the edge weights. In this way, we can then transfer deep neural network deployment ideas to mathematical graph partition problems, as what have exactly been thoroughly introduced in section 2.3. Therefore, as illustrated above, spectral clustering techniques are aiming at minimizing between-cluster similarities (inter-partition communication costs). Here we re-define the similarity in spectral clustering technique simply as the number of between-cluster connecting synapses.

In GraphDNN, our spectral partitioning algorithm is proposed to deploy weighted connecting synapses in distributed computing work nodes. As shown in figure 4.2, to specifically conclude, in order to obtain the exact partition matrix Φ of size $r_{input} \times c_{output}$ for this deployment method, we formulate the network partition problem as same

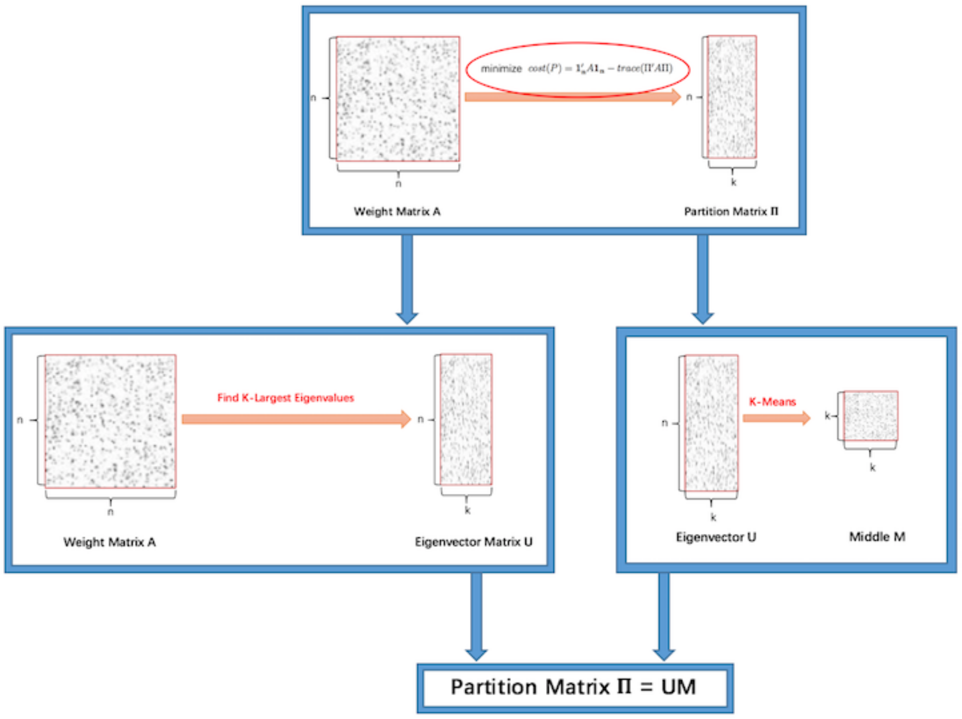

Figure 4.2 GraphDNN: Static Partitions after Deep compression

as mathematically solving the following dynamic problem:

$$
\begin{aligned}
maximize \quad & trace(\Phi' A \Phi) \\
subject\,to \quad & \\
& \Phi[i][j] \in \{0,1\}\ for\ \forall i,j \\
& |\Phi| = \sqrt{n} \\
& \Phi'\Phi \leqslant kI_k \\
& \Phi\ is\ orthogonal
\end{aligned}
\tag{4-1}
$$

First, normalize the input sparse weight matrix $A_{r\times c}$ to a target matrix satisfying the following two prerequisites: (1) a matrix with a same row length and column length (Because spectral clustering techniques require accurate computations of both eigenvalues and eigenvectors which needs square input matrix) (2) the matrix is doubly stochastic (Notice here, negative weight values in the matrix should be first transferred to their absolute values to satisfy the doubly stochastic matrix format, and it is not only benefit for the algorithm further proceeding but also better for later inter-node communication cost evaluations). Then after this step, we should have a output: the normalized matrix $A_{t\times t}$ where $t = r + c$.

Second, perform Singular Value Decomposition on the normalized matrix $A_{t\times t}$ to get its eigenvectors and eigenvalues. (Since we need to find the largest k eigenvalues, I recommend for the algorithm proposed by C. Yang et al. which is implicit and flexible to large-scale computations.) Furthermore, because for doubly stochastic matrices, all eigenvalues are real, and their values are smaller than or equal to 1 when compared, with one of them exactly equals to 1, then the solution of the above dynamic programming problem (4-1) is always smaller than or equal to 1. Let $D_{k\times k} := \lambda_1, ..., \lambda_k$ be the largest k eigenvalues of A, and $U_{n\times k} := \vec{u_1}, ..., \vec{u_k}$ be their corresponding eigenvectors. Suppose we have partition matrix $\Phi := UM$ where matrix $M \in \mathbb{R}_{k\times k}$ exits. Then we can naturally induce that

$$\Phi'\Phi = M'U'UM = M'M \tag{4-2}$$

and as well as

$$trace(\Phi' A\Phi) = trace(M'U'AUM) = trace(M'DM) \tag{4-3}$$

Therefore, in this way, the above dynamic programming problem (4-1) can be bounded by $D = I$ and Φ is the k-partition matrix that we are looking for. Then, we can further compute the partition matrix Φ under the constraint equation as

$$\Phi := argmin_{\overline{\Phi}}||\overline{\Phi} - UM|| \tag{4-4}$$

Third, apply the k-means clustering algorithm in order to figure out the targeted partition matrix Φ. Suppose rows of the eigenvectors $U_{n\times k}$ are clustered by k orthogonal row vectors $\vec{m_1}, ..., \vec{m_k}$. That is the computing matrix M can be computed by simply clustering row vectors of the matrix U. Up to here, both the eigenvector matrix U and the computing matrix M are well figured out in this way. Therefore, we can then com-

```
Algorithm 1: GraphDNN: Static Section for Fully-Connected Layer
    Input: The original dense weight matrix $A'$ of size $r \times c$; $k$ distributed
        computing work nodes respectively $d_1, ..., d_k$.
    Output: Partition matrix $\Phi$ of size $r \times c$, with $0, ..., k-1$ as each element
        representing corresponding partition set.
1   Perform the Deep Compression algorithm including pruning, quantization and
      Huffman encoding on the target deep neural network. After this stage, the target
      weight matrix is compressed to the sparse matrix $A$ of size $r \times c$ for network
      configuration with basically equal performance;
2   Normalize the nonnegative symmetric weight matrix $A_{r\times c}$ to satisfying this two
      conditions: (1) with a same row length and column length (2) is doubly
      stochastic, i.e., the arithmetic sum of all elemental values equals to one, or say
      $\sum_{v \in V} w(v, \overline{v})$. Let $D_{k\times k} := \lambda_1, ..., lambda_k$ be the largest $k$ eigenvalues of
      weight matrix $A$, and $U_{n\times k} := \vec{u_1}, ..., \vec{u_k}$ be their corresponding eigenvectors;
3   Compute the eigenvectors matrix $U$ and the eigenvalues matrix $D$ (all $k$ largest
      eigenvalues) in consideration of two formulation constraints: $U'U = I_{K\times k}$ and
      $AU = UD$. For this, computing $k$ largest eigenvalues and eigenvectors can be
      simply reached out by the eigs function which is completely available by both
      the software matlab and octave with the parameter as la for largest setting;
4   Cluster the row vectors of eigenvectors $U$ by the k-means clustering algorithm.
      For this, here we randomly choose $k$ observations as initialization instead of
      selecting observations in a linear order to reduce unnecessary calculating work;
5   Obtain the partition matrix $\Phi_{r\times c}$ by associating each cluster with one of the
      vectors in the canonical basis of $\mathbb{R}^k$ and replacing each row vector of $U$ by the
      basis vector associated with its cluster.
```

pute the partition matrix Φ according to the formulation (4-4) above: associate each cluster with one of the vectors in the canonical basis of $\mathbb{R}^k$ and replace each row of U by the basis vector associated with its cluster.

In conclusion, the static partitioning component of our GraphDNN framework for the fully-connected layers operates with the deep neural network mainly through two significant stages: first a network retraining process and then a matrix operation process. The entire procedure will finally return the corresponding deployment method of a given input deep neural network with its weight matrix A' of size $r \times c$ by a generated partition matrix $\Phi_{r\times c}$ and the substitute weight matrix $A_{r\times c}$ with merely some minor concessions made to the accuracy of the original deep neural network.

4.1.2 Convolutional Layer

As we already know, neurons in a convolutional layer, which present already very sparse weight matrix, are extremely different from those ones present in a fully-connected layer. Similar to what we did for fully-connected layers, GraphDNN also processes each single convolutional layer separately. Therefore speaking of one given convolutional layer, GraphDNN here chooses to only make network compressions on every single convolutional layer, and when it comes to static deployment solutions for working nodes in distributed computing systems, each convolutional layer is copied among every different computing work node instead of partitioning the convolutional layer into finite balanced network blocks as the way we did for fully-connected layer to (1) make the best of the convolutional layer's computation ability (i.e. every work node maintains one entire copy of the convolutional layers); (2) take efforts to reduce as much inter-node communication costs.

Algorithm 2: GraphDNN: Static Section for Convolutional Layer

```
Input: The original dense weight matrix $A'$ of size $r \times c$; $k$ distributed
       computing work nodes respectively $d_1, ..., d_k$.
Output: $k$ pieces of compressed convolutional kernel, with $0, ..., k-1$.
1 Perform Neural Network Compression (CP-Decomposition, Quantization and
    Deep Compression are all well suitable for applying) and then it will return the
    compressed configuration of convolutional kernels $A$;
2 Copy compressed convolutional kernels $k$ pieces and match them to the
    distributed computing system by piece-node pairs.
```

Currently multiple compression strategies for convolutional layers have been explored by various scholars, and some of them are efficient and widespread: First, Vadim et al. ([13], Lebedev et al. (2014), ICLR.) proposed the fine-tuned CP-decomposition mechanism: given one convolutional layer, the 4D convolutional kernel is computed to the arithmetic sum of a small number of rank-one tensors by the low-rank CP-decomposition algorithm; then in this way the original convolutional layer is replaced by a sequence of four different convolutional layers with above decomposed small kernels; after this configuration transferring, a general fine-tuning idea is utilized in the dynamic training process (normally it only changes the network values in backward process). Second, Jiaxiang Wu and Cong Leng et al. ([14], Wu et al. (2016), CVPR.) illustrated a novel framework called Quantized-CNN for model compression and simultaneous accelerations in mobile devices (this method seems like the second stage - weight quantization of deep compression). Third, as we already discussed before, the Deep Compression methods proposed by Song Han et al. can provide convolutional layer compression as well.

To specifically conclude, the static partitioning component of our GraphDNN framework for convolutional layers operates with the deep neural network also mainly through two stages: first a network retraining process and then a matrix operation process. The entire process will return k copies of compressed convolutional kernels.

It is worth to mention that another idea about the convolutional layer design here, as the above algorithm 3 clearly clarifies, which is regarding operations to make changes to the convolutional layer input matrix format. Conventional partition schemes about convolutional layers comply to the structural symmetry. That is, generally, input neurons in convolutional layers are partitioned along two-dimensional space. However, such a two-dimensional partition may not be suitable for distributed computing systems: because convolutional kernels are small and it can result in much larger costs for activating than communicating, in this way there is no need to activate these cross connections for such communication demands. Therefore, the idea is designed to partition the input neurons along the longer edge of the input matrix for each individual computing node in order to produce less overlapped parts required to transfer.

The designed method, as shown in algorithm 3, for convolutional layer partition explores through the input structure. For instance, in the first step, one 2x2-dimensional kernel is transferred to a 4x1-dimensional substitute; next, layer workloads are deployed linearly to distributed computing work nodes accordingly. Such a partition strategy can bring effective results for applications where connection activation really counts.

Algorithm 3: Another Design for Convolutional Layer

```
Input: The original dense weight matrix $A'$ of size $r \times c$; $k$ distributed
       computing work nodes respectively $d_1, ..., d_k$.
Output: $k$ pieces of compressed convolutional kernel, with $0, ..., k-1$.
1 Perform Neural Network Compression (CP-Decomposition, Quantization and
    Deep Compression are all well suitable for applying) and then it will return the
    compressed configuration of convolutional kernels $A$;
2 if $r > c$ then
3     $A$ is divided by rows;
4 else
5     $A$ is divided by columns;
6 end
7 Then one-dimensional bound of $A$ is linearly indexed according to $k$.
```

4.2 Experiments

4.2.1 Investigations on Configuration

The MNIST Database

The MNIST (Modified National Institute of Standards and Technology database) handwritten digit database, as the subset of a larger database NIST, has been provided for various different applications especially in academia, for instance, training image processing neural networks; working as the standard database for testing some generally theoretical algorithms that are also related to classifications; or training networks for digit recognitions etc. The database is so popular that when talking about neural networks, researches usually begin with this database. To get an overall understanding

Table 4.1 Some Methods on MNIST

Type	Classifier	Error rate (%)
Linear classifier	Pairwise linear classifier	7.6
Non-linear classifier	40 PCA + quadratic classifier	3.3
Boosted stumps	Product of stumps on Haar	0.87
Support vector machine	Virtual svm, deg-9 poly	0.56
Convolutional neural network	Committee of 5 CNNs	0.21

Table 4.2 Summary of database MNIST

Name	Role	Size (bytes)
Train-images-idx2-ubyte	Training set images	9912422
Train-labels-idx1-ubyte	Training set labels	28881
T10k-images-idx3-ubyte	Test set images	1648877
T10k-labels-idx1-ubyte	Test set labels	4542

about the database, table 4.1 here shows some of the machine learning methods used on the database and their error rates, by the type of classifier.

Furthermore, the MNIST database totally consists of 60,000 training samples and 10,000 testing samples. Half of the MNIST training data and its testing data are taken as subset from the NIST training data; and the other half of the MNIST training data and its testing data are taken as subset from the NIST testing data. We can then refer to these file parameters as shown in the above table 4.2.

The ImageNet

Compared to the MNIST as a handwritten digit recognition database, the ImageNet works as an image database for visual object recognitions as well. Up until 2016, around over ten million images have been hand-annotated (it can denote the existence of an object, whether or not there is the object) where at least one millions ones are along with corresponding bounding boxes (it can then provide those visible components of the corresponding object in the image). Since 2010, the ImageNet project opens annual software contests for competitions in visual tasks, for instance, classifying, detecting and recognizing objects and senses. Among these competitions, the AlexNet in 2012 and the VGG-16 in 2014 are really widespread. Table 4.3 demonstrates experiments over the ImageNet ILSVRC-2012 dataset. For following compression configurations, it is worth to mention that convolutional layers are quantized with eight bits, and fully-connected layers are comparably quantized with six bits, and we do not take experiments on further Huffman encoding stage.

Configurations

Experiments about the static strategy of partition framework GraphDNN includes (1) the network compression effects including three different network models and the configurations; (2) the model partitioning performance, i.e. reduction extent of communication costs with its accuracy and balance. The next subsection will make some analysis about experiment results separately.

4.2.2 Explorations on Static Partition

▶ Network Compression

This part will discuss the compression experiment effects based on three tables:

1. Table 4.3 exhibits the overall compression effects of three network models: LeNet-300-100, AlexNet and VGG-16 respectively, where LeNet on MNIST database and the other two networks on the ImageNet. From observations of experimental results shown in the table 4.3, all of these three network models here exhibit their stable compression effects.

In the table, ’Most Error’ records the largest error among all layers; ’Parameters’ records the exact file size of the network model; ’Compress Rate’ records the ratio of the compressed parameter file size and the original one.

From these data, we can observe around 29x to 36x reduction benefits regarding the network parameters. Under even the descent tendency in ’most error’, such as 0.02% by LeNet and 0.29% by VGG-16, the parameter file size can reach impressing compression demands. These compression operations can improve the network performance by largely saving corresponding model storage space without significant loss of the original neural network accuracy. This compression effect brought by deep compression is

Table 4.3 Overall Compression Effects

Network	Most Error (%)	Parameters	Compress Rate
LeNet-300-100 original	1.64	1070 KB	＼
LeNet-300-100 compressed	1.62	33 KB	32x
AlexNet original	42.78	240MB	＼
AlexNet compressed	42.77	7.52MB	29x
VGG-16 original	31.50	552MB	＼
VGG-16 compressed	31.21	15.3MB	36x

Table 4.4 Compression Results of LeNet-300-100

Layer	Weight size(k)	Weight (%; P)	Weight bits (P+Q)	Compress rate(%; P+Q)
1	945	8	6	2.9
2	121	9	6	3.8
3	5	26	6	15.7
total	1070	8(12x)	6	3.1(32x)

the base on which we built further partition and deployment experiments.

2. Furthermore, table 4.4 illustrates detailed statistics about the Lenet-300-100 compression process. Lenet-300-100 a frequently used deep neural network which is composed of three fully-connected layers with its specific model configurations as: the first input layer (as 784 nodes fully connected with 300 nodes); the second hidden layer (as 300 nodes fully connected with 100 nodes); and the third output layer (as 100 nodes fully connected with 10 nodes).

In the table, Layer: each fully-connected layer in the network is ranked as one to three correspondingly in sequences; P: represents pruning; Q: represents quantization; weight size: denotes the original weight file size of corresponding layer; weight(%;P): represents the ratio between the exact weight file size after the pruning stage and the original weight file size; weight bits (P+Q): represents the representation bit number of weight values; compress rate(%; P+Q): represents the ratio between the exact weight file size after the pruning and quantization stage and the original weight file size.

Table 4.4 expresses the detailed benefits given by the compression process on LeNet-300-100. The pruning stage brings 12x benefits, i.e. the weight file is compressed to its 8%; furthermore, the quantization stage brings 32x benefits, i.e. the weight file is com-

Table 4.5 Compression Configurations of LeNet-300-100

Ratio	0.95	0.90	0.85	0.80	0.75	0.70
Accuracy	0.7715	0.8722	0.9062	0.9126	0.9334	0.9406

pressed to its 3.1%. In Song's paper, they also illustrate that pruning and quantization can already achieve exactly significant performance when they are combined together.

3. Table 4.5 shows the specific effect when we apply deep compression to LeNet-300-100 with different compression rate. In the table, 'Ratio' represents the compression rate at which we will prune the neural network; 'Accuracy' represents the network accuracy after compression. As we can observe, first, accuracy decreases with the increase of compression extent. Second, my implementation of deep compression without the third Huffman encoding stage is acceptable even the compression ratio reaches 0.90 here.These compressed files, from the compression ratio 0.70 to 0.90, are further utilized in following partition experiments.

▶ GraphDNN Partition

This part discusses static partition experiment effects based on two figures and a table:

Configurations of all next experiments mainly consists of: 1) the network model as Lenet-300-100; 2) original weights are generated under the end condition, i.e. the end accuracy, as 0.01; 3) compressed weights are generated under the end condition, i.e. the end accuracy, as 0.001; 4) the activation function is figured as the sigmoid function. The accuracy for original generated weight is finished as 0.9647 (actually the neural network can perform much better performance on this database, here for our experimental convenience, the end condition is set as 0.01 correspondingly).

What's more, not only the static partition experiments, but also an experiment for the baseline solution is required here. The baseline solution experiment is conducted over those purely original weight values, i.e. we utilize the intuitive equal partition on original full-weight network for comparison purpose in partition performance. The original intuitive partition mechanism, i.e., directly vertically partition the neural net-

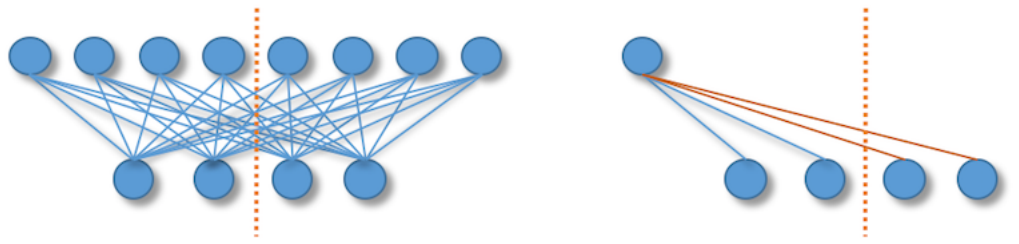

Figure 4.3 Baseline Partition Mechanism

work in an even principle. As shown in Figure 4.3, we give the detailed description as: take the bisection for instance, we divide the fully-connected layer vertically in the middle of the network and separate it as the desired finite number of equally network components. All synapses that cross the orange dash line here are in different partitions. It is then naturally to derive the equally vertical baseline for three and four nodes etc.

1. Figure 4.4 shows static partition performance in view of the changing in compression rate. It exhibits three subgraphs on different number of distributed computing entities separately, and each subgraph represents the tendency based on compression ratio.

In the figure, 'overall reduce', the blue square dash line, represents the reduction ratio between the static partition and compared experiment; 'the largest layer reduce', the yellow square dash line, represents the largest reduction ratio throughout the network layers; 'accuracy change', the green triangle dash line, represents the corresponding absolute accuracy; 'memory requirement/node', the rose circle dash line, represents the ratio between the final file memory requirement and the original file memory requirement of one single computing node.

As figure 4.4 illustrates, there are four observations can be referred as:

First, considering on the overall reduction trend. We can naturally induce the decreasing tendency with the growing tendency of compression rate, that is: the network gets sparser, then the cost reduction effects can get better. This observation is intuitive, because sparser networks normally maintain fewer connecting synapses, which will introduce better reduction when applied a same partition mechanism.

Second, considering on the largest layer reduction trend. Even though the LeNet-

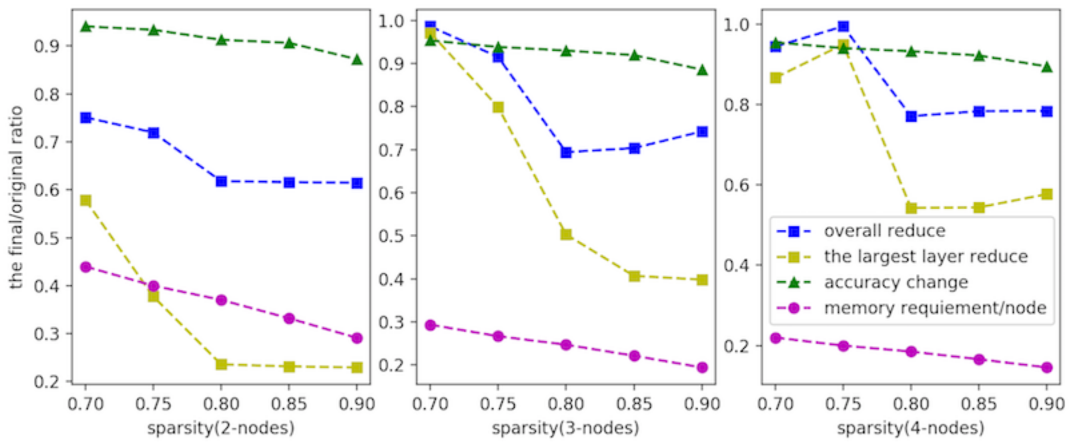

Figure 4.4 Final/Original Ratios at different worker nodes

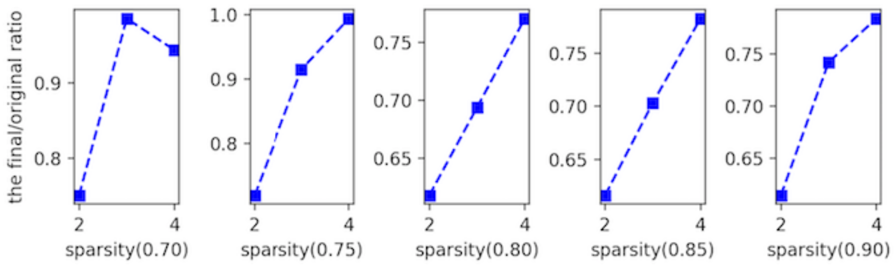

Figure 4.5 Final/Original Reduction Ratios at different compression ratio

300-100 network model is not much deep, the reduction effects are already impressing enough: for instance, here the largest reduction of two working nodes is 0.229 in ratio compared to original communication costs with only little accuracy loss; and the overall reduction of two working nodes is 0.6147.

Third, considering on the accuracy trend. As that of compressed weight results, the network accuracy will decrease when the compression rate is larger, and the absolute values are roughly same. It also expresses that the partition of network do not have crucial influence on the network accuracy.

Fourth, considering on the memory requirement/node trend. The memory reduction is shown by the model parameter file, mainly the weights. From the memory tendency, we can observe that (1) in one node count situation, memory requirement will decrease with the increase in network sparsity; (2) when number of node is getting larger, the memory requirement of each node will gets smaller.

Table 4.6 Workloads of Individual Node at CR0.80

layer	2-nodes		3-nodes			4-nodes			
1	542	542	362	361	362	271	271	271	271
2	200	200	133	133	134	100	100	100	100
3	55	55	36	37	37	28	27	27	28

2. Figure 4.4 shows the static partition reduction effect in view of the changing in node number. It exhibits five subgraphs on different compression rates separately, and each subgraph represents the corresponding reduction tendency.

In the figure, ’overall reduce’, the blue square dash line, represents the reduction ratio between the static partition and compared experiment. Considering five subgraphs in figure 4.5 (same as comparing the three subgraphs in figure 4.4 together), overall reduction performance tendency can be figured out as: when there are less working nodes, partition strategy brings more obvious effects as for the reduction ratio. For instance, in two nodes, overall reduce at 0.70 to 0.90 are 0.7510, 0.7194, 0.6179, 0.6159, 0.6147 while for three nodes, they are 0.9859, 0.9148, 0.6939, 0.7034, 0.7420 correspondingly.

3. Table 4.6 shows the network node deployment situations in individual computing nodes after the static partition mechanism at the compression rate 0.80 (CR0.80). In this table, each column represents a single device: ’layer’ records the corresponding layer in the LeNet model; ’2-nodes’, ’3-nodes’ and ’4-nodes’ represents different settings in computing device number; each entry in the table represents the respective node amount of each device. Notice here that current experiments are conducted only on devices with a common computation ability. Results in table 4.6 demonstrate that current partition algorithm gives good balance for devices with same computation ability.

To specifically conclude here, as shown in above simulation experiments, the static partition framework, GraphDNN, for deploying deep neural networks can satisfy our initial goals and achieve large reductions for inter-node communication costs while keeping work load balance among distributed computing nodes. To get better deployment strategies, the method normally prefer to choose the sparsest parameters under the accuracy limitation, and then apply further spectral partition mechanism.

Chapter 5 DNN Dynamic Partitioning Optimization Algorithm Design

The dynamic partitioning optimization component designed for deploying deep neural network in distributed computing systems is introduced in this chapter in detail including all distributed retraining methods that we can think of currently. Designed dynamic retraining optimization algorithms also achieve great effects in all of efficiency, conciseness and clearness. And the optimization experiments in this chapter show that our dynamic partitioning optimization algorithms for deploying deep neural networks in general distributed computing systems realize good reductions as well in regarding of inter-node communication costs.

5.1 GraphDNN: DNN Dynamic Partitioning Optimization Algorithm Design

Dynamic retraining algorithms here are designed for dynamic optimizations of the previous static deployment methods of deep neural networks in distributed computing systems which is illustrated in Chapter 4. Two separate schemes here, dynamic pruning and greedy cross-weight fixing, are proposed to fine-tune our static network partition methods during the distributed network retraining process. Both of these schemes are general enough, and they can be utilized as normal dynamic partitioning optimization strategies for all related cross-partition problems. Furthermore, we also make some explorations on the probability shown by the ReLU activation function and try to conduct predictions on the performance of network nodes accordingly.

5.1.1 Dynamic Pruning

Considering that fewer cross-partition synapses may contribute to less inter-node communication costs, the straight-forward dynamic optimization for cutting off some of these cross-partition synapses is the dynamic pruning method, as algorithm 4 clarifies:

Algorithm 4: Dynamic Retraining Optimization1: Dynamic Pruning

```
Input: Based on the static algorithms in Chapter.4, the target DNN has already
       been well partitioned with its three available parameter vectors: partition
       matrix, mask matrix and weight matrix. Let fully-connected layer with
       weight matrix of size $r_{input} \times c_{output}$; $t$ be the threshold setting of
       Dynamic Pruning.
Output: Weight Matrix $W'_{-[1\times(r\times c)]}$ and Partition Matrix $P'_{-[1\times(r\times c)]}$.
1  while $iteration < MaxIter$ or $error > EndCondition$ do
2      Let $P_{-[1\times(r\times c)]}$ be the partition matrix, element value equals to one if the edge
         is a cross-partition edge, zero otherwise; Let $M_{-[1\times(r\times c)]}$ be the mask
         matrix, element value equals to zero if the edge is cut off, one otherwise;
         Let $W_{-[1\times(r\times c)]}$ be the weight matrix, each element represents
         corresponding edge weight;
3      while $in\ Forward\ Process$ do
4          for $each\ cross - partition\ edge\ i\ with\ P[i] = 1$ do
5              if $W[i] < t$ then
6                  $P[i] = 0, M[i] = 0, W[i] = 0;$
7              end
8          end
9          do general Forward computations;
10     end
11     while $in\ Backward\ Process$ do
12         for $each\ cross - partition\ edge\ i\ with\ P[i] = 1$ do
13             if $W[i] < t$ then
14                 $P[i] = 0, M[i] = 0, W[i] = 0;$
15             end
16         end
17         do general Backward computations;
18     end
19 end
```

in both forward and backward retraining processes, we prune all cross-partition synapses whose absolute weight values are less than the predefined threshold (configured by a pre-stipulated value or a specific ratio of the total count of connecting synapses). Since each time after backward learning process all the unhidden weight values of the network are adjusted by those middle computed deltas, here dynamic pruning always keeps an generally basic control on the network weight values during the whole retraining process and it can thus introduce some effective reductions for inter-node communication costs of partitions by excluding meaningless connecting synapses (here considering those synapses with low weight values) as well as maintaining with no significant bad influence on accuracy regarding the original accuracy of the deep neural network.

To specifically conclude, Dynamic Pruning generally lies in a same principle as the static pruning stage of previous GraphDNN: conduct pruning operations on connecting synapses with small weight values to reduce meaningless data transmissions. The difference is that the static pruning performs only once in the beginning (no further control of those minimum weight values for later training processes); while the dynamic pruning holds its minimum weight value standard through the whole retraining stages, which is much more promised for the neural networks.

5.1.2 Greedy Cross-Weight Fixing

As clarified, dynamic pruning can actually prune cross-partition connections with very small absolute weight values. To compensate for this pruning strategy, some other components of the neural network should increase their weight values, and they may also lie as cross-partition synapses. Inspired by this compensation process, we hope more cross-partition synapses could be cut out and synapses in a same partition could compensate these changes. On the other hand, we cannot cut off all cross-partition synapses since it would definitely damage the accuracy of the deep neural network (classification accuracy cannot be secured because only in-consecutive and irregular parts of input are given). In this way, we then propose a new dynamic optimization -

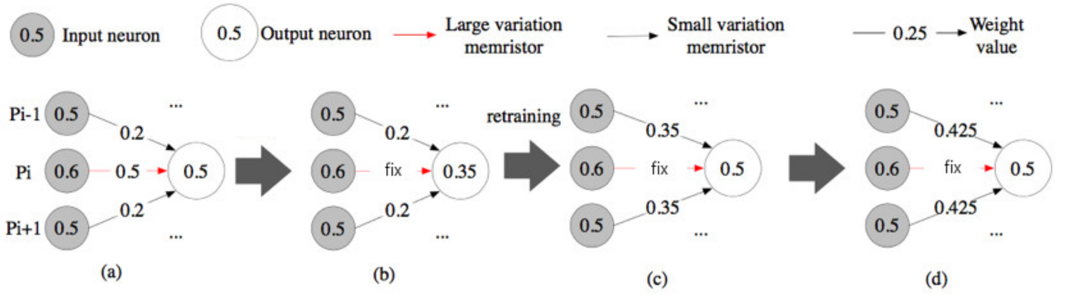

Figure 5.1 Weight changing process: (a) initial weights; (b) fixing the weight connection; (c) after retraining; (d) in the next iteration.

Greedy Cross-Weight Fixing.

5.1.2.1 Background Learning Process

Importantly, in order to fully understand our optimization, we should first understand the basic learning mechanism of neural networks: how the nearby network part will react when we change the value of a synapse and fix it. Descriptions about this is discussed in the following two processes: the changing process and the fixing process.

1) The Changing Process

The back-propagation method which utilizes the gradient descent method is adopted to fine-tune and fix those cross-partition connection synapses. For instance, given a simple one-fully-connected-layer neural network, we will now explain how those weights surrounding the fixed weight change during the training phase. Principles in multiple layers can be referred accordingly. During the back-propagation process, the significance of the gradient descent - how much the weight value changes - associated with the neuron is determined by its input values from multiple previous neurons. Therefore, when we cut off (i.e. fix as zero) a cross-partition synapse with weight W_{ij} connecting an input neuron i to j, an error can occur between the expected output value of j and the real output value of j; this error is induced by the miscalculation of i and W_{ij}. Fortunately, if other input neurons (e.g., $i-1$, $i+1$) are correlated with i, this error can be compensated by $i-1$ and $i+1$ in the back-propagation process: as the fixed weight W_{ij} is cut off in the back-propagation process, the gradient descent algorithm will instead change the weight synapse connecting j to $i-1$ and $i+1$ to minimize the error.

Figure 5.1 demonstrates the above changing process. As shown in Figure 5.1(a), a group of neighboring input neurons P_i, P_{i-1}, P_{i+1} have similar values. With the gradient descent algorithm, the associated weights also have similar values.

[Note: why they would have similar values? Because, generally, as well as in the following experiments, the values of the input samples, for a common neural-network based application, are highly correlated. For example, in a visual application, the input data of the neural networks are continuous because the image pixel at a nearby region have similar values. It is worth noticing that the partitioning algorithm is designed to partition the network work loads for its computation demands; the original network topology and values still maintain the original meanings.]

From descriptions in the figure, the weight W_{P_iO} belongs the dataset of which we find elements are as far as from each other; thus we cut off the connection synapse and fix its weight value, as shown in Figure 5.1(b), which causes an increased error in the value of the output neuron. When this neural network is trained again, the local gradient value of W_{P_iO}, due to the fixed W_{P_iO}, has to be embodied in $W_{P_{i-1}O}$ and $W_{P_{i+1}O}$ instead, so that the value of the output neuron bounces back to the original one, as shown in Figure 5.1(c). This process continues, as shown in Figure 5.1(d), until the accuracy change reaches our limit, for instance, 2% in reduction.

2) The Fixing Process

This process describes how we will make decisions on fixing during the dynamic procedure. The overall principle is that each time the fixing process prefers to select and fix those cross-partition synapses that keep themselves as far as much from each other. The above Changing Process describes when one connecting synapse is fixed, how its surrounding synapses perform weight value adjustments to compensate for this weight change. In this part, the Fixing Process will focus on how to select those cross-partition synapse to be pruned within pre-stipulated accuracy limitations:

There are two design aspects for next connection weight fixing considerations: First, also as what is explained in the above Changing Process, the continuity property shown by the input data (which can be image pixels or video frames) maintains in the distributed computing environment as well. Second, weight compensations are applied by some surrounding connecting synapses of the fixed network synapse. Inspired by k-means clustering algorithm, this paper tries to make the fixing operations on network connections which hold the most far distances. This idea is apparently intuitive enough for k-means applications where nodes distances are already measured by their synapse weights. When it comes to synapse distances in the neural network, we should take the fully-connected layer structure into consideration: as figure 5.1 illustrates, changes introduced by W_{P_iO}, has to be embodied in $W_{P_{i-1}O}$ and $W_{P_{i+1}O}$. Then we think output nodes are the key measure for synapse distances in neural networks.

5.1.2.2 Our Algorithm: Greedy Cross-Weight Fixing

Now we define synapse distances in fully-connected layers as: given a fully-connected layer with i input nodes and j output nodes, P_k represents the k_{th} input node, Q_k represents the k_{th} output node, $W_{P_mQ_n}$ represents the weight value of the synapse connecting the m_{th} input node and the n_{th} output node. The definition of synapse distances can be expressed as the equation:

$$\begin{aligned}
&Dist_{ab} \\
&= Dist_{P_uQ_v - P_mQ_n} \\
&= |(i-u) + (n-v-1) \times i + m| \\
&= |(n-v) \times i + m - u|
\end{aligned} \tag{5-1}$$

i.e. Distance of Synapse a (connecting P_m and Q_n) and Synapse b (connecting P_u and Q_v) equals to the absolute value of result $(i-u) + (n-v-1) \times i + m = (n-v) \times i + m - u$.

Keep that the distance of different synapses as the fixing measurement and the prestipulated accuracy reduction as the end retraining standard. Both of the above Changing Process and the Fixing Process then can constitute the following Greedy Fixing Optimization.

Furthermore, if you would like to combine the dynamic pruning process and the greedy cross-weight fixing process during the retraining, there are two points should be followed: First, the implementing sequence of 'dynamic pruning-then-greedy cross-weight fixing' should be assured each iteration. Because dynamic pruning processes the small block of synapses. If the dynamic pruning is performed after greedy fixing, some of synapse pruning operations in fixing become meaningless. Second, you should keep one redundant cycle each time after one greedy weight fixing operation, which is utilized to justify the values of the weight matrix.

Algorithm 5: Dynamic Retraining Optimization2: Greedy Cross-Weight Fixing

Input: Let $P_{-[1\times(r\times c)]}$ be the partition matrix, element value equals to one if the edge is a cross-partition edge, zero otherwise; Let $M_{-[1\times(r\times c)]}$ be the mask matrix, element value equals to zero if the edge is cut off, one otherwise; Let $W_{-[1\times(r\times c)]}$ be the weight matrix, each element represents corresponding edge weight; Let $Dist_{ab}$ be the distance of edge a and edge b; k be the desired number of cross-partition edges to be cut off each iteration; ct records the total count of current cross-partition connecting edges.

Output: Weight Matrix $W'_{-[1\times(r\times c)]}$ and Partition Matrix $P'_{-[1\times(r\times c)]}$.

```
1  while $\delta < EndCondition$ do
2      for $i\ in\ all\ r \times c\ connectingedges$ do
3          if $P\_[i] = 1$ then
4              $ct+ = 1$;
5          end
6          for $i\ in\ all\ r \times c\ connectingedges$ do
7              if $P\_[i] = 1\ and\ tmp = ct/k$ then
8                  $M\_[i] = 0, W\_[i] = 0, P\_[i] = 0$;
9              end
10         end
11     end
12 end
```

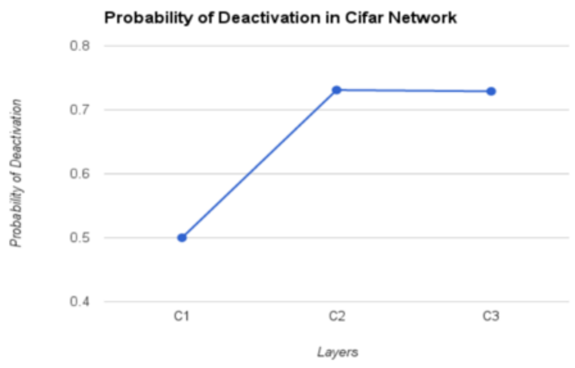

Figure 5.2 Zero Probability of CIFAR-10 DNN

5.1.3 Explorations on ReLU Activation

In the context of neural network, activation functions, work as nonlinear roles, are introduced to compensate for the limited expressing ability of linear models. The unit (i.e., the node) that employs the rectifier function is then referred as rectified linear unit. The rectifier is an activation function defined as $f(x) = max(0, x)$, where x is the input for a neuron. From the expression, we can understand that when input value of the neuron is less than zero, output should be zero, i.e., this neuron is not activated; the neuron is activated only when its input value is larger than zero. This activation function has various advantages: (1) Biological Plausibility: one-sided; (2) Sparsity: in a randomly initialized network, only about half of hidden layer units are actually activated during the inference process (neurons with output values larger than zero); (3) Gradient Propagation: there is no gradient vanishing problems or gradient exploding problems; (4) Efficient Computation: there is no other computations like multiplications or additions but only a simple comparison; (5) Scale-Invariant: as the expression $max(0, ax) = a \cdot max(0, x)$ demonstrated.

In considerations of exploring the ReLU activation function, we proposed two optimization probabilities during the dynamic optimization: On the one hand, in the neuron-level optimization, as the exhibited sparsity property, for randomly pre-generated weights, there is 50% neurons are actually not activated during inference. For instance, as shown in Figure 5.2, nearly 50% to 73% neurons are not activated in the CIFAR-10 DNN network model. What's more, it also demonstrates that this sparsity of deactivation can increase along with the depth increase of the deep neural network. If these deactivated neurons can be dynamically controlled before, communication costs for data transmission on these nodes then can be largely relieved. On the other hand, in the layer-level optimization, cross-partition communication costs are only related with the answer of the problem if there is a transmission but not the transmitted data itself (i.e., even the target data is zero for the cross-partition synapse, communication also need to be done, which actually does not have any influence on the cross-partition communication costs). However, in this way, if here some cross-partition synapses transmit the data with a value zero, we could then take actions to simplify the cross-partition communication whether by some suitable data compressing methods or some data format designing algorithms.

Neuron-level Prediction

There are several observations of the ReLU activation function: First, based on randomly generated weight values, the rectifier will only activate about 50% neurons (the other half is dead then). Second, ReLU activation has no relations with what the exact inter-value of computations, which means that the actual value (the dot product of corresponding inputs and weights) does not have influences on the output value of this neuron. If the neuron will finally produce negative product results, after activation the unit will then generate zero as its output. Considering this, if we can predict whether one neuron will be activated next or not activated, we can then save corresponding computations and of course the communication cost when it is unnecessary to have a computation and transmission during distributed computing nodes.

Recently, Sanjay Ganapathy et al. ([15], Ganapathy et al. (2017), CVPR.) proposed Saturation Prediction and Early Termination (SPET) to predict whether one neuron will finally be activated based on partial computation results of inputs and weights. They set 50% as the saturation interval, and one threshold measuring partial sum of weight-input product to control the prediction. Such predictions can be done with ReLU for retraining in distributed computing systems to reduce meaningless communications through cross-partition connections.

Layer-level Data Compression or Data Format

Specifically, for zero transmissions generated by the ReLU activation function, there are some researches inspire us to further process them: The so-called Efficient Inference Engine (EIE), proposed by Song Han et al. ([1], Dean et al. (2012), ISCA.), is the first accelerator to explore how to utilize dynamic sparsity of activation functions for conducting computation costs, and it achieves another 3x speed acceleration in total designed system by simply skipping those zero activation units. Therefore, we can have two different schemes available under zero activations: First, we can just take actions as what EIE did - ignore those neurons with zero activations and keep those cross-partition connections in silence; Second, instead of the absolute ignoring, we will also make these cross-partition communications happen, but under this situation zero values should be transmitted together under a streamline and pre-defined data transmission mechanism. Both of the above two schemes should be combined with the previous neuron-level prediction optimization.

1) **Zero Skipping**

With the help of the neuron-level prediction, we can now predict whether the output value of this neuron is zero. If the neuron generates zero as its output, we can simply skip the corresponding communications of this connection and do not transmit anything through this synapse. In this way, the original communication costs of these skipped connections are now with a value zero. No cross-partition communication costs will then be consumed instead. More directly, before each cross-partition data transmission process in one layer, we can set up the initial product value of all neurons as zero, then if there is no data transmission is performed from cross-partition synapses, they will correctly maintain with value zero; on the other hand, those changes introduced from other neurons in different computing nodes can be suitably applied to these neurons as well. However, the goal of our project is to reduce the communication costs instead of not transmitting given data, therefore this paper further explores optimizations on this.

2) **Data Transmission Format**

Although through ignoring and skipping zero values the data transmissions can indeed allow large reductions in cross-partition communication costs because of the sparsity property of the ReLU activation function, at the same time this intuitive operation actually largely changes the nature training framework as well (i.e., some more parameters for control purpose should be introduced to make decisions on each nodes if there are no data transmissions available for further steps). Therefore, here we try to explore the dynamic retraining optimizations on the ReLU zero values through the data transmission format of these across-partition synapses: the expression format and the transmission format.

For the expression format aspect, weight values are normally exhibited in the float type or the double type, as what the following simulation experiments reveal, which means every weight value will then take up about four bytes or eight bytes (it is inevitable and the basic demand both in the storage space aspect and in the transmission aspect) as expressions. Even for the zero value situation, it also takes up about four bytes or eight bytes for the cross-partition data transmission. Considering this kind of wasting, this scheme tries to transmit zero values through cross-partition synapses in a type with a comparably smaller storage format, for example the unsigned integer type (which only takes one byte).

For the transmission format aspect, since there are a number of outputs may not be activated and they will then produce results with value zero, when we desire to transmit all these zeros, there can still remain much connection wake-up time even though the data is expressed in a smaller storage format, take one byte as an example. When compared to the time for transmit a one-byte data by the cross-partition synapse, the wake-up time of the connection really counts. Therefore, here multiple zeros are compacted together and transmitted in one communication round.

In conclusion, these explorations on the ReLU activation function mainly discover some possible dynamic optimization opportunities through its sparsity property. The above mechanisms focus on how to deal with value zeros: one way is to simple skip transmit these zeros; the other way is to transmit them together and each zero with a smaller size compression format. Apparently, all of above three optimization methods (i.e., Dynamic Pruning; Greedy Cross-Weight Fixing; Explorations on ReLU activation function) can be combined together during the whole distributed retraining process.

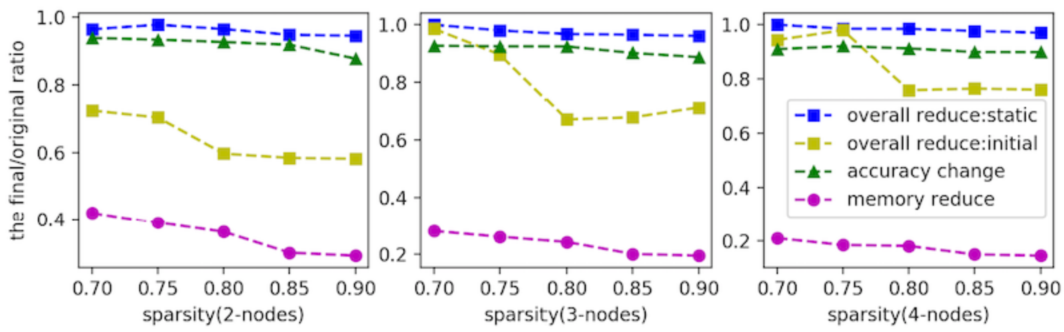

Figure 5.3 Dynamic1: Final/Original Ratios at different worker nodes

5.2 Experiments

As mentioned in chapter 4, it has been shown that our static partition framework can well deploy the deep neural network. However, the reduction benefits can further improved during the retraining process. Besides the static partition mechanisms as before, in this part we explore dynamic optimizations based on the experiments as mentioned before, i.e. deployment of LeNet-300-100 on the MNIST database.

5.2.1 Explorations on Dynamic Pruning

This subsection describes the experiment results of the first optimization mechanism, dynamic pruning, by comparing the output values with former static partition experiments and intuitive experiments concerning communication cost reductions and memory storage space reductions. Then we will discuss dynamic pruning experiment statistics through the figure as followings.

1. Figure 5.3 shows dynamic pruning performance in view of changing in compression rate. It exhibits three subgraphs on different number of distributed computing entities separately, and each subgraph represents the tendency based on compression ratio.

In the figure, ’overall reduce:static’, the blue square dash line, represents the reduction ratio between dynamic pruning and the static partition; ’overall reduce::initial’, the yellow square dash line, represents the reduction ratio between dynamic pruning and the compared experiment; ’accuracy change’, the green triangle dash line, represents

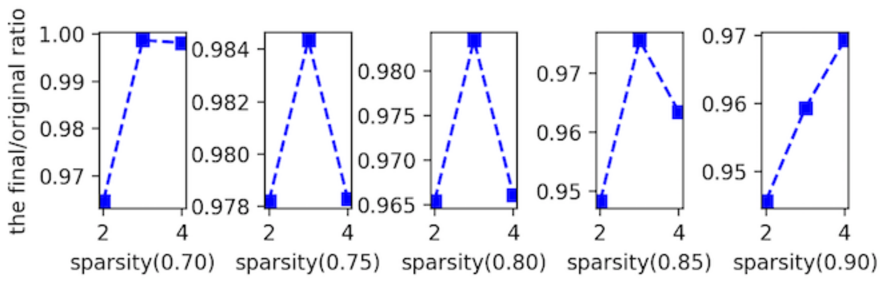

Figure 5.4 Dynamic1: 'overall reduce:static' at different compression rate

the corresponding absolute accuracy; 'memory requirement/node', the rose circle dash line, represents the ratio between the final file memory requirement and the original file memory requirement of one single computing node.

As figure 5.3 illustrates, considering on the overall reduction trend, both of comparison with static and initial ones. Dynamic pruning can produce 94.57% to 98.60% reductions compared to the static partition methods when applied to two independent working nodes. There are two observations:

First, it reveals that the reduction ratio offered by dynamic pruning is not too large, i.e. reduction rates in different compression rates generally have around 0.01 difference. This is not hard to understand, since the first point of network compression stage is to prune those small weight values, which means there is much less space reserve for optimization like this.

Second, it shows that along with the descent of our compression ratio, dynamic pruning shows its tendency to decrease in the reduction effects. This is not hard to understand, since a larger compression ratio makes the network sparser, which means there can be more weights whose absolute values maintain large difference. That is for a sparse network, there can be more very large values and very small values, i.e. in this way they can reserve more space for improving the cost reduction by pruning small-value weights.

2. Figure 5.4 shows dynamic pruning performance in view of changing in number of individual working node. It exhibits five subgraphs on different compression rate separately, and each subgraph represents the tendency based on node count. The blue dash line records the reduction ratio compared to the former static partition experiment.

As figure 5.4 demonstrates, there is no obvious principle can be concluded from the overall tendency, and the difference expressed by different node number setting is really small, around 0.01. Therefore, dynamic pruning performs a same effect when applied to systems with different number of nodes.

5.2.2 Explorations on Greedy Cross-Weight Fixing

This subsection describes the experiment results of greedy strategy combined with the intuitive dynamic pruning one and organizes the optimization effects by comparing them with former static statistics and the dynamic pruning only one.

1. Figure 5.5 shows greedy fixing performance in view of changing in compression rate. It exhibits three subgraphs on different number of distributed computing entities separately, and each subgraph represents the tendency based on compression ratio.

In the figure, 'overall reduce:dynamic1', the green circle dash line, represents the reduction ratio between greedy fixing and dynamic pruning; 'overall reduce:static', the blue rhombus dash line, represents the reduction ratio between greedy fixing and static partition; 'overall reduce:initial', the orange triangle dash line, represents the reduction ratio between greedy fixing and the compared experiment; 'largest reduce:initial', the yellow squares, represents the largest reduction ratio throughout the network layers between greedy fixing and the compared experiment; 'accuracy change', the pink triangle line, represents the corresponding absolute accuracy.

As figure 5.5 illustrates, there are two observations:

First, the greedy fixing strategy can further more improve the performance to a more 70% to 90% extent for two working nodes. Because the greedy fixing here focuses on all weight values that are as far as much from each other instead of caring only about those weighted connections with only small values, such a greedy cross-partition fixing strategy can then introduce us with less connections for the original network partition. What's more, considering the largest reduction ratio, 0.0647 occurs in the two nodes at 0.8 compression rate, which is much less than what we considered at static state, 0.229.

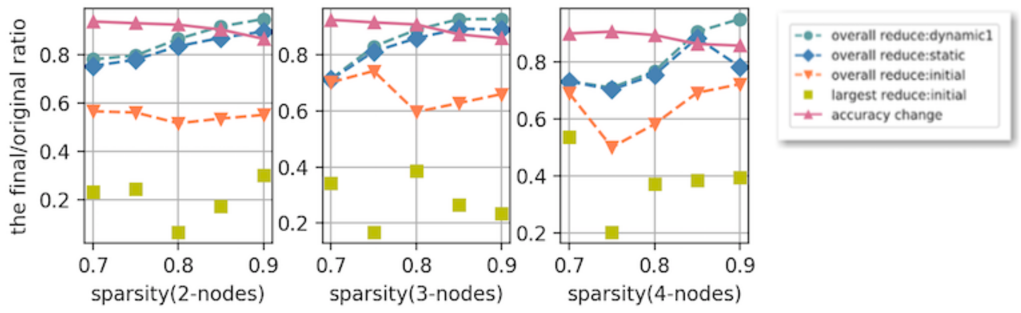

Figure 5.5 Dynamic2: Final/Original Ratios at different worker nodes

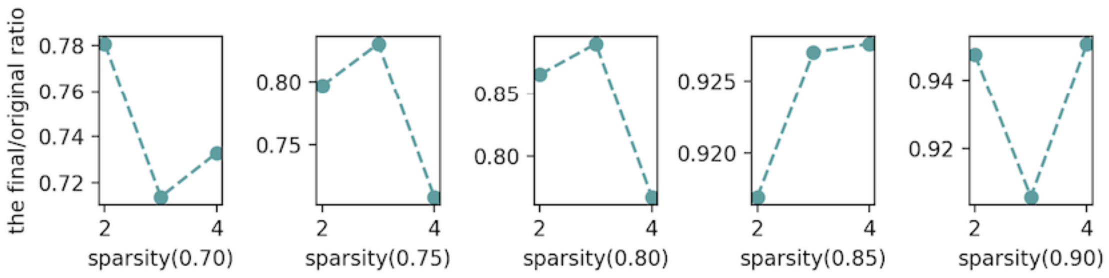

Figure 5.6 Dynamic2: overall reduce:dynamic1' at different compression rate

It reveals the effectiveness of greedy fixing.

Second, the fixing method contributes significant improvements to the reduction effects of sparser network. That is greedy fixing can contribute more benefits when the network is not that sparse. This is not hard to understand, since when the network is sparser, there are less connections available for fixing, or say, it will quickly find its end condition when fix some of its connections.

2. Figure 5.6 shows greedy fixing performance in view of changing in number of individual working node. It exhibits five subgraphs on different compression rate separately, and each subgraph represents the tendency based on node count. The green dash line records the reduction ratio compared to the former dynamic pruning experiment.

As figure 5.6 demonstrates, same as the dynamic pruning optimization, there is no obvious principle can be concluded from the overall tendency, and the difference expressed by different node number setting is really small, around 0.01. Therefore, greedy fixing performs an almost same effect when applied to systems with different number of nodes.

Table 5.1 Dynamic3: Final/Dynamic2 Ratios

reduction	0.90	0.85	0.80	0.75	0.70
2-nodes	0.7933	0.7837	0.7906	0.8123	0.7995
3-nodes	0.8544	0.8643	0.8672	0.8539	0.8511
4-nodes	0.8789	0.8755	0.8697	0.8706	0.8813

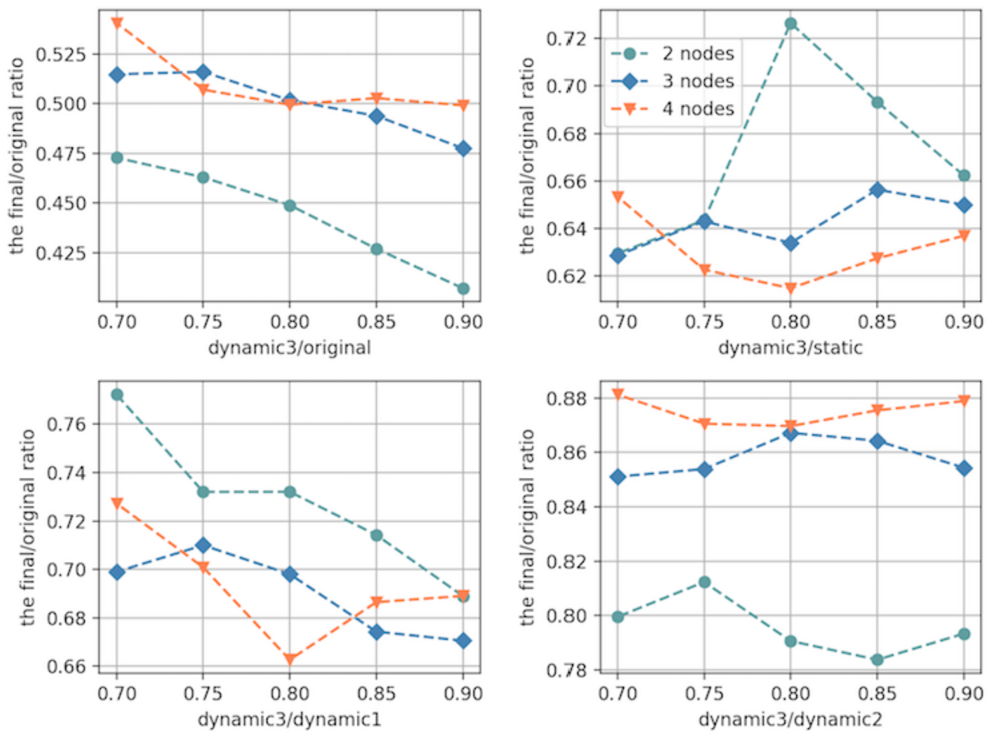

Figure 5.7 Dynamic3: Final/Original Ratios at different worker nodes

5.2.3 Explorations on ReLU Activation Function

This subsection describes the experiment results of the proposed optimization ideas about explorations on the ReLU activation function. Considering that zero skipping is irrelative with our concentration about figuring out the balance of inter-node communication costs and individual working node computability, we conduct experiments focusing on making changes to data transmission formats when there is zeros. Then we will discuss experiment statistics through the figure and the table as followings.

1. Table 5.1 shows specific reduction ratio between relu and greedy fixing.

2. Figure 5.7 shows the reduction benefits of zero explorations by comparing the effect with the original, the static, the dynamic pruning and the greedy algorithm. In the figure, '2 nodes', the green circle dash line, represents the corresponding reduction ratio of the system with two single individual computing nodes; '3 nodes', the blue rhombus dash line, represents that of three computing nodes; and '4 nodes', the orange triangle dash line, then represents that of four computing nodes.

As you can notice from overall tendency, there are several observations as followings:

First, the ReLU activation function generally performs better accuracy and compression than the sigmoid one which we utilized in former experiments as well as when regarding those partitioning optimizations. This is already rectified in many corresponding researches, here we simply point it out.

Second, there is no obvious tendency on the connectivity of network sparsity and zero format, i.e. no corresponding relationship exists between the sparsity and the optimization extent. This is not hard to understand, since this optimization method is focusing on the specific activation, and it has no relation with the sparsity of the object network which then make an overall effect for networks in different compression rates. Therefore, the reduction benefits brought by the dynamic 3 method are generally similar among all compression rate for the neural network.

To specifically conclude, in this chapter we propose three dynamic optimizations for dynamic deployment: dynamic pruning, greedy cross-weight fixing and communication format on zeros. All of the optimizations showed effective reductions on communication costs without any influence on the workloads of each computing node and merely slight decrease in the network accuracy. And we think the algorithms above can be portable to satisfy related optimization demands.

Chapter 6 Conclusion

6.1 Main Contribution

In this paper, there are mainly two contributions: First, one novel framework, GraphDNN, is proposed here which manages to deploy complex deep neural networks among independent devices in distributed computing systems pursuing computation work load balance and communication costs reductions based on the overall management. GraphDNN can achieve the portability, efficiency and simplicity performance. Second, three different dynamic optimizations are then explored to further cut off cross-partition connections during the online training process: dynamic pruning, greedy cross-weight fixing and explorations on the ReLU activation function. And all of these three mechanisms are exactly general that can be applied for similar problems as well.

The Deep Neural Network Balance Graph Partition framework processes deployments on the fully-connected layers and convolutional layers in two different schemes. For fully-connected layers, it takes the first step to compress the layer's dense weight matrix; after that it partitions the target deep neural networks into these distributed computing nodes by spectral computing methods. For convolutional layers, it first takes network compressions on the layer's kernel and parameters; then convolutional layer is copied among these different computing nodes. Another idea about the input data format of convolutional layers is also introduced in detail in Chapter 4.

Online training optimizations further show three possible developments on the above deployment strategy. Dynamic pruning directly keeps control on minimum weight threshold during the training process; Greedy cross-weight fixing each time linearly fixes finite number of the weights under the limitation of overall network accuracy; explorations on ReLU activation function utilize its sparsity property, and they take actions to cut off these zeros or transmit them together with a smaller size data format. With the help of dynamic optimizations during network online training, the deployment mechanisms can thus present more competitive performance.

6.2 Future Work

This paper conducts researches and explores methods on theoretically taking deployments for deep neural networks among distributed computing systems. Considering the actual environment of distributed computing systems, there are some devices may dump or there are some new devices to add in during their working state, and in all of these situations, generally referred as edge computing problems, it is really interesting for us to determine how to partition the network suitably for tolerating such sudden events in distributed computing systems.

REFERENCE

[1] DEAN J, CORRADO G, MONGA R, et al. Large Scale Distributed Deep Networks[C]//Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States. Harrahs and Harveys, Lake Tahoe: Curran Associates, Inc., 2012:1232–1240. http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.

[2] LI M, ANDERSEN D G, PARK J W, et al. Scaling Distributed Machine Learning with the Parameter Server[C]//11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014. Broomfield, CO: USENIX Association, 2014:583–598. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[3] LI M, ANDERSEN D G, SMOLA A J, et al. Communication Efficient Distributed Machine Learning with the Parameter Server[C]//Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada. Palais des Congrès de Montréal, Montréal CANADA: Curran Associates, Inc., 2014:19–27. https://www.cs.cmu.edu/~muli/file/parameter_server_nips14.pdf.

[4] DENIL M, SHAKIBI B, DINH L, et al. Predicting Parameters in Deep Learning[C]//Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States. Harrahs and Harveys, Lake Tahoe: Curran Associates, Inc., 2013:2148–2156. http://papers.nips.cc/paper/5025-predicting-parameters-in-deep-learning.

[5] BA L J, CAURANA R. Do Deep Nets Really Need to be Deep?[J]. CoRR, 2013, abs/1312.6184. http://dblp.uni-trier.de/db/journals/corr/corr1312.html#BaC13.

[6] COURBARIAUX M, DAVID J P, BENGIO Y. Training deep neural networks with low precision multiplications[C]//International Conference on Learning Representations. arXiv:1412.7024: arXiv, 20153.

[7] HAN S, MAO H, DALLY W J. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding[J]. CoRR, 2015, abs/1510.00149. `http://arxiv.org/abs/1510.00149`.

[8] WILLIAMS R D. Performance of dynamic load balancing algorithms for unstructured mesh calculations[J]. Concurrency - Practice and Experience, 1991, 3(5):457–481. `https://doi.org/10.1002/cpe.4330030502`.

[9] LELAND R, HENDRICKSON B. An Empirical Study of Static Load Balancing Algorithms[C]//In Proceedings of the Scalable High Performance Computer Conference. United States: National Nuclear Security Administration, 19941:682–685.

[10] ARORA S, RAO S, VAZIRANI U V. Expander flows, geometric embeddings and graph partitioning[J]. J. ACM, 2009, 56(2):5:1–5:37. `http://doi.acm.org/10.1145/1502793.1502794`.

[11] KERNIGHAN B W, LIN S. An Efficient Heuristic Procedure for Partitioning Graphs[J]. Bell Sys. Tech. J., 1970, 49(2):291–308.

[12] HESPANHA J P. An efficient MATLAB algorithm for graph partitioning[R]. United States: [s.n.] , 2004.

[13] LEBEDEV V, GANIN Y, RAKHUBA M, et al. Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition[J]. CoRR, 2014, abs/1412.6553. `http://arxiv.org/abs/1412.6553`.

[14] WU J, LENG C, WANG Y, et al. Quantized Convolutional Neural Networks for Mobile Devices[C]//2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. Las Vegas, NV, USA: IEEE, 2016:4820–4828. `https://doi.org/10.1109/CVPR.2016.521`.

[15] GANAPATHY S, VENKATARAMANI S, RAVINDRAN B, et al. DyVEDeep: Dynamic Variable Effort Deep Neural Networks[J]. CoRR, 2017, abs/1704.01137. `http://arxiv.org/abs/1704.01137`.

Appendix

Static Partition Formulations

▶ Initialization

Given fully-connected layer α with its compressed weight matrix $W_{r\times c}$, where r denotes its input node number, c denotes its output node number. The k-partition method is then transferred as: for weighted graph $G = (V, E)$ with its weight matrix as $A_{n\times n}$, where $n = r + c$. The objective is to satisfy (1) partition the graph to $P = V_1, V_2, ..., V_k$; (2) node count in each partition is less or equal to $\lceil n/k \rceil$; (3) $cost(P) = \sum_{i\neq j}\sum_{v\in V_i, \overline{v}\in V_j} cost(v, \overline{v})$ is minimized.

▶ Formulation

Define the $n \times k$ matrix as $\Pi = [\pi_{vj}]$. It is a k-partition matrix if satisfying (1) $\pi_{vj} \in 0, 1, \forall v, j$; (2) Π is orthogonal; (3) $trace(\Pi'\Pi) = n$. According to the Hespanha's proposition, matrix Π is figured out iff every row vector of Π is a vector of the canonical basic of $\mathbb{R}^k$. Define k-partition vector $d_{n\times 1}$ related to Π.

Lemma .1. *Partition P is in one-to-one corresponding relationship with Π.*

For one side:

$$\pi_{vj} = \begin{cases} 0 & v \notin V_j \\ 1 & v \in V_j \end{cases} \qquad \forall v \in V,\ j \in 1, 2, .., k.$$

For the other side:

$$V_j := v \in V : \pi_{vj} = 1, \qquad \forall j \in 1, 2, ..., k.$$

Therefore, the relation of P and Π can be expressed as

$$\Pi'\Pi = diag[|V_1|, |V_2|, ..., |V_k|]$$

Lemma .2. *The 'communication costs' of partition P can be expressed as:*

$$cost(P) = \mathbf{1_n'}A\mathbf{1_n} - trace(\Pi' A \Pi)$$

With above considerations, we can reach the formulation (4-1) in chapter 4, i.e.,

$$\begin{aligned} maximize \quad & trace(\Pi' A \Pi) \\ subject\,to \quad & \\ & \Pi[i][j] \in \{0,1\}\; for\; \forall i,j \\ & trace(\Pi) = \sqrt{n} \\ & \Pi'\Phi \leqslant kI_k \\ & \Pi\; is\; orthogonal \end{aligned} \tag{1}$$

Because A is a doubly stochastic matrix, all eigenvalues are real numbers, and they are all smaller than or equal to one. Therefore, the dynamic programming problem above $trace(\Pi' A \Pi) = \sum_{i=1}^{k} \pi_i' A \pi_i \leq \sum_{i=1}^{k} \pi_i' \pi_i = trace(\Pi'\Pi) = n$.

Let $U := [\lambda_1 = 1 \geq \lambda_2 \geq ... \geq \lambda_k]$ be the largest k eigenvalues; Let $D := [u_1, u_2, ..., u_k]$ be the corresponding k eigenvectors. Then $U'U = I_{k\times k}$ and $AU = UD$. Suppose for some matrix $Z \in \mathbb{R}^{k\times k}$, $\Pi := UZ$. Then

$$\Pi'\Pi = Z'U'UZ = Z'Z$$

and

$$trace(\Pi' A \Pi) = trace(Z'U'AUZ) = trace(Z'DZ).$$

► **The clustering process**

Suppose that the n rows of the $n \times k$ matrix U are clustered around k orthogonal row vectors $z_1, z_2, ..., z_k$. Let n_i be the total number of rows clustered around z_i. Define

$$Z := \begin{bmatrix} z_1 \\ z_2 \\ \cdots \\ z_k \end{bmatrix}^{-1} = [\frac{z_1'}{||z_1||^2} \quad \frac{z_2'}{||z_2||^2} \quad \cdots \quad \frac{z_k'}{||z_k||^2}] \tag{2}$$

the vth row of $\overline{U} := UZ$ is given by

$$\overline{u}_v = [\frac{< u_v, z_1 >}{||z_1||^2} \quad \frac{< u_v, z_2 >}{||z_2||^2} \quad \cdots \quad \frac{< u_v, z_k >}{||z_k||^2}] \tag{3}$$

where u_v represents the vth row of U. Since u_v is close to one of the v_i and all z_i are orthogonal, therefore $\overline{u_v}$ will be close to one of the vectors in the canonical basic of $\mathbb{R}^k$. According to the Hespanha's proposition, UZ is then close to a k-partition matrix.

Furthermore, because there are n_k rows close to z_k, the number of entries close to one in the jth column of $\overline{U}$ is approximately n_k. Therefore, $\Pi'\Pi \approx diag[n_1, n_2, ..., n_k]$. The computation of Z can then be regarded as a clustering algorithm whose goal is to determine orthogonal vectors $z_1, z_2, ..., z_k$ around which the rows of U are clustered, with no cluster larger than $\lceil n/k \rceil$. Then the k-means clustering algorithm can be applied to obtain matrix Π by associating each cluster with one of the vectors in the canonical basis of $\mathbb{R}^k$ and replacing each row of U by the basic vector associated with its cluster.

► Hespanha's proposition

• A $n \times k$ matrix Π is a k-partition matrix iff every row of Π is a vector of the canonical basic of $\mathbb{R}^k$.

(1) on the one side: each row of Π should have only one value one because otherwise, if there were two ones in columns π_i and π_j, then $\pi_i'\pi_j \geq 1$ and Π would not be orthogonal. On the other hand, every row of Π should have at least a one because for a matrix whose entries are in $\{0, 1\}$, its squared Frobenius norm is exactly the total number of entries equal to one. Since every row can have at most a one, every row should actually have a one to get its Frobenius norm equals to n. Therefore, if Π is a k-partition matrix then every row of Π is a vector of the canonical basic of $\mathbb{R}^k$.

(2) on the other side: suppose that each row of Π is a vector of the canonical basic of $\mathbb{R}^k$. Then all entries of Π belong to the set $\{0, 1\}$ and its Frobenius norm equals to n since its squared Frobenius norm is equal to the number of entries equal to one (one per row). Moreover, Π is orthogonal because if π_i and π_j are two distinct columns of Π then each row of these vector should have at least one zero and therefore $\pi_i'\pi_j = 0, \ \forall i \neq j$.

Acknowledgements

First and foremost, I'm incredibly grateful to Prof. Li Jiang for mentoring me over the past two years. Without your earnest assistance and conscientious involvement in every step during the learning process, this thesis would have never been reached. Thank you very much for your invariable support and understanding over these past two years. Your advice and great help truly inspired me.

I would like to show my great gratitude to Prof. Xiaoyao Liang for enthusiastic discussions and mentoring for the passing two years. Your sincere advice on researches deeply influenced my values and helped me to become a more positive student, which is of significant meanings throughout my whole life. What's more, your assistance and understanding are really important.

This thesis wouldn't have been possible without the committee. All of your objective and sincere advice and suggestions are really important and helpful for this thesis. Thank you very much for your efforts and contributions.

Finally, I would like to express my huge gratitude to my family and friends. This thesis would be nothing without all of you. To my parents, for giving me the incredible love and support during endless trips between the library and the lab, and to my laboratory mates: Lerong Chen, Mamixa and Nicolas. You guys are amazing.