Title

Fast and Accurate Machine Learning on Distributed Systems and Supercomputers

Author

You, Yang

Publication Date

2020

Peer reviewed|Thesis/dissertation

Fast and Accurate Machine Learning on Distributed Systems and Supercomputers

by

Yang You

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Demmel, Chair
Professor Kurt Keutzer
Professor Ming Gu

Summer 2020

Fast and Accurate Machine Learning on Distributed Systems and Supercomputers

Abstract

Fast and Accurate Machine Learning on Distributed Systems and Supercomputers

by

Yang You

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor James Demmel, Chair

The past ten years have seen tremendous growth in the volume of data in Deep Learning (DL) applications. As a result, the long training time of Deep Neural Networks (DNNs) has become a bottleneck for Machine Learning (ML) developers and researchers. For example, it takes 29 hours to finish 90-epoch ImageNet/ResNet-50 training on eight P100 GPUs. It takes 81 hours to finish BERT pre-training on 16 v3 TPU chips. This thesis is focused on fast and accurate ML training. Although production teams want to fully utilize supercomputers to speed up the training process, the traditional optimizers fail to scale to thousands of processors. In this thesis, we design a series of fundamental optimization algorithms to extract more parallelism for DL systems. Our algorithms are powering state-of-the-art distributed systems at Google, Intel, Tencent, NVIDIA, and so on. The focus of this thesis is bridging the gap between High Performance Computing (HPC) and ML.

There was a huge gap between HPC and ML in 2017. On the one hand, we had powerful supercomputers that could execute 2×10^{17} floating point operations per second. On the other hand, we could not even make full use of 1% of this computational power to train a state-of-the-art machine learning model. The reason is that supercomputers need an extremely high parallelism to reach their peak performance. However, the high parallelism led to a bad convergence for ML optimizers. To solve this problem, my co-authors and I proposed the LARS optimizer, LAMB optimizer, and CA-SVM framework. These new methods enable ML training to scale to thousands of processors without losing accuracy. In the past three years, we observed that the training time of ResNet-50 dropped from 29 hours to 67.1 seconds. In fact, all the state-of-the-art ImageNet training speed records were made possible by LARS since December of 2017. LARS became an industry metric in MLPerf v0.6. Moreover, our approach is faster than existing solvers even without supercomputers. If we fix the training budget (e.g. 1 hour on 1 GPU), our optimizer can achieve a higher accuracy than state-of-the-art baselines.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

I would like to extend my deepest gratitude to my advisor Prof. James Demmel (Jim). The completion of my dissertation would not have been possible without the support and nurturing of Jim. I attended my first academic conference in Boston (IPDPS 2013). Jim was the keynote speaker of that conference. I was impressed by Jim's great personality and leadership in our area. That is when I knew I wanted to be a PhD and Jim's student. Dr. Jason Riedy (Jim's former student) later wrote an email to introduce me to Jim. I feel that I am very fortunate that I could become Jim's student. Under Jim's supervision, I finished 13 papers in the last five years. Jim spent hours to improve each of my paper drafts. Jim is a very nice person. He gave me lots of freedom and necessary guidance. I am grateful for the time he spent reviewing my career objectives and recommending strategies for achieving them. Undoubtedly, the most influential person in my academic career has been Jim. My later life will be profoundly influenced by Jim. I also want to thank Prof. Katherine Yelick (Kathy), who is co-leading the BeBOP group with Jim. Kathy hosted my Prelim exam and gave me lots of advice in the last five years.

I gratefully acknowledge the help of Profs. Ming Gu and Kurt Keutzer. Kurt was the chair of my Qual exam committee. Kurt and I have some common research interests and I attended many of his group meetings. I learned a lot from him by working with his group. Ming and I both come from China so Ming gave me a lot of useful advice on life and career development. I'd also like to extend my gratitude to Prof. Cho-Jui Hsieh. I met Cho at Jim's 60-th birthday party. Cho is an expert in machine learning, which is the key topic of my PhD thesis. Cho truly gave me lots of great advice in the last five years. I cannot leave Berkeley without mentioning Prof. William Kahan, who was Jim's advisor, and advised me on a research project. I respect him a lot because he is still doing research at the age of 87.

Before I came to Berkeley, I worked as a master student with Profs. Haohuan Fu and Guangwen Yang at Tsinghua University. I worked as an exchange student with Profs. David Bader and Rich Vuduc at George Tech. I want to thank them for their help.

I had several internships in Industry. I want to thank my mentors Jonathan Hseu and Jing Li (Google), Boris Ginsburg (NVIDIA), Yuxiong He (Microsoft), Rajesh Bordawekar and David Kung (IBM). I am excited to see that my research can be applied in real-world industry applications. I also want to thank my colleagues and fellow students: Benjamin Brock, David Bruns-Smith, Erin Carson, Orianna DeMasi, Aditya Devarakonda, Grace Dinh, Michael Driscoll, Marquita Ellis, Evangelos Georganas, Amir Gholami, Qijing Jenny Huang, Nicholas Knight, Penporn Koanantakool, Igor Kozachenko, Maryam Mehri Dehnavi, Samyam Rajbhandari, Rebecca Roelofs, Guanhua Wang, Xin Wang, Bichen Wu, and Xiangyu Yue. Specifically, I want to mention Ronghang Hu, who is working with Prof. Trevor Darrell. I am indebted to Ronghang for encouragement and help.

I would also like to thank my coauthors of work on which this thesis is based: Srinadh Bhojanapalli, Aydın Buluç, Kenneth Czechowski, Inderjit S Dhillon, Igor Gitman, Sanjiv Kumar, Ji Liu, Sashank Reddi, Le Song, Xiaodan Song, Chris Ying, Hsiang-Fu Yu, and Zhao Zhang. I really appreciate their time and efforts in our collaborations.

Last but not least, I want to thank my mom Yanlei Huang, my dad Zhiqiang You, my elder brother Peng You, and my wife Shiyue Liang. I would like to express my deepest appreciation to my family. My success would not have been possible without the unconditional support of my family. I will be always there for you.

Chapter 1

Introduction

1.1 Motivation

The past several years have seen tremendous growth in the volume of data in Deep Learning (DL) applications. As a result, the long training time of Deep Neural Networks (DNNs) has become a bottleneck for Machine Learning (ML) researchers. For example, it takes 29 hours to finish 90-epoch ImageNet/ResNet-50 training on eight P100 GPUs. It takes 81 hours to finish BERT pre-training on 16 v3 TPU chips. My research is focused on fast and accurate ML. Although production teams want to fully utilize supercomputers to speed up the training process, the traditional optimizers fail to scale to thousands of processors. In this thesis, my co-authors and I design a series of fundamental optimization algorithms to extract more parallelism for DL systems. Our algorithms are powering state-of-the-art distributed systems at Google, Intel, Tencent, NVIDIA, and so on (see Table 1.1). The focus of this thesis is bridging the gap between High Performance Computing (HPC) and ML.

There was a huge gap between HPC and ML in 2017. On the one hand, we had powerful supercomputers that could execute 2×10^{17} floating point operations per second. On the other hand, we could not even make full use of 1% of this computational power to train a DNN. The reason is that supercomputers need an extremely high parallelism to reach their peak performance. However, the high parallelism led to a bad convergence for DNN optimizers. To solve this problem, we proposed the LARS optimizer (You, Gitman, and Ginsburg, 2017), LAMB optimizer (You, J. Li, et al., 2019), and LEGW (You, Hseu, et al., 2019). These new methods enable DNN training to scale to thousands of processors without losing accuracy. In the past three years, we observed that the training time of ResNet-50 dropped from 29 hours to 67.1 seconds. In fact, all the state-of-the-art ImageNet training speed records were made possible by LARS since December of 2017 (Table 1.1). The speedup comes from the fact that researchers can scale the training to a larger batch size, and so use larger scale, parallelism, without losing accuracy in a fixed number of epochs. For example, researchers scaled the batch size of ImageNet training from 256 (K. He et al., 2016) to 1K (Krizhevsky, 2014), 5K (Mu Li, 2017), 8K (Goyal et al., 2017), 16K (You, Z. Zhang, Demmel, et al., 2017), 32K (Ying et al., 2018), and 64K (Sameer Kumar et al., 2019). LARS became an industry

metric in MLPerf v0.6 (Mattson et al., 2019). Moreover, our approach is faster than existing solvers even without supercomputers. If we fix the training budget (e.g. 1 hour on 1 GPU), our optimizer can achieve a higher accuracy (You, J. Li, et al., 2019).

Table 1.1: ImageNet/ResNet-50 Training Speed Records.

Teams	Date	Accuracy	Time	Optimizer
K. He et al., 2016	12/10/2015	75.3%	29h	Momentum
Goyal et al., 2017	06/08/2017	76.3%	65m	Momentum
You, Z. Zhang, Demmel, et al., 2017	11/02/2017	75.3%	48m	LARS
You, Z. Zhang, Demmel, et al., 2017	11/07/2017	75.3%	31m	LARS
Akiba, Suzuki, and Fukuda, 2017	11/12/2017	74.9%	15m	RMSprop
You, Z. Zhang, Demmel, et al., 2017	12/07/2017	74.9%	14m	LARS
X. Jia et al., 2018	07/30/2018	75.8%	6.6m	LARS
Mikami et al., 2018	11/14/2018	75.0%	3.7m	LARS
Ying et al., 2018	11/16/2018	76.3%	2.2m	LARS
Yamazaki et al., 2019	03/29/2019	75.1%	1.25m	LARS
Sameer Kumar et al., 2019	10/02/2019	75.9%	67.1s	LARS

1.2 Structure of this Thesis

We give an overview of the ideas in this thesis. The key contribution of this thesis is **"increasing parallelism while changing algorithms as needed to maintain efficiency by avoiding communication"**. To do so, we need to maintain accuracy by modifying the algorithms and/or hyper-parameters as needed, or using different approximate solutions. The following sections and chapters will embody this idea in different algorithms. In summary, this thesis includes three main parts:

- **[Part 1]** Chapter 2 covers the details of communication-avoiding SVM. Chapter 3 covers the details of communication-efficient KRR. Chapter 4 covers the details of asynchronous greedy coordinate descent (Asy-GCD). Section 1.3 acts as a brief introduction for Chapters 2, 3, and 4.
- **[Part 2]** Chapter 5 covers the details of speeding up ImageNet training on supercomputers. It will introduce both the algorithm design and the trade-off in communication/computation. Section 1.4 acts as a brief introduction for Chapter 5.

- **[Part 3]** Chapter 6 covers the details of reducing BERT-training time from three days to 76 minutes, which is built on top of the techniques like large-batch optimization and hyper-parameter auto-tuning. Section 1.5 acts as a brief introduction for Chapter 6.

These three parts are closely related to each other. We start from convex optimization in Part 1 and dive into non-convex optimization in Part 2. Since non-convex optimization is much more difficult than convex optimization, we present more details in Part 3. Chapter 7 presents the concluding remarks.

1.3 Fast and Scalable Convex Optimization Methods for Machine Learning (ML)

Kernel methods like Support Vector Machines (SVM) and Kernel Ridge Regression (KRR) scale poorly on distributed systems because the cost of communication dominates the arithmetic. My co-authors and I designed several communication-avoiding approximate algorithms that outperformed state-of-the-art methods. This study won the **Best Paper Award of IPDPS 2015** (You, Demmel, Kenneth Czechowski, et al., 2015).

Some early machine learning applications that require high-performance training are Kernel methods like non-linear SVM and KRR. Before the Deep Learning era, SVM and KRR were some of the state-of-the-art techniques used in a wide range of applications. On shared-memory systems, although the basic SVM algorithm features limit the performance and efficiency on the many-core architectures that we evaluated, my MIC-SVM (You, S. L. Song, et al., 2014; You, Fu, et al., 2015) still achieves 4.4-84x and 18-47x speedups against LIBSVM on Intel MIC and Ivy Bridge CPUs respectively (with the techniques like Adaptive Heuristic and Multi-level Parallelism). To further improve SVM's efficiency on shared-memory systems, we proposed the Asynchronous parallel Greedy Coordinate Descent (Asy-GCD) algorithm. At each iteration, workers asynchronously conduct greedy coordinate descent updates on a block of variables. We analyzed the theoretical behavior of Asy-GCD and proved a linear convergence rate. Asy-GCD (You, Lian, et al., 2016) is a general optimization algorithm for minimizing a smooth function with bounded constraints.

On distributed-memory systems, SVM often suffers from significant communication overheads. In our study, we considered the problem of how to design communication-efficient approximate versions of parallel SVM. Prior to our study, the parallel isoefficiency (A. Y. Grama, Anshul Gupta, and V. Kumar, 1993) of a state-of-the-art implementation scaled as $W = \Omega(P^3)$, where W is the problem size and P the number of processors; this scaling is worse than even one-dimensional block row dense matrix vector multiplication, which has $W = \Omega(P^2)$. My study (You, Demmel, Kenneth Czechowski, et al., 2015; You, Demmel, Kent Czechowski, L. Song, and Rich Vuduc, 2016) considers a series of algorithmic refinements, leading ultimately to a Communication-Avoiding SVM (CA-SVM) method that improves the isoefficiency to nearly $W = \Omega(P)$. We evaluate these methods on 96 to 1536 processors, and show average speedups of $3-16\times$ ($7\times$ on average) over previous distributed SVM, and a 95% weak-scaling efficiency on six real-world datasets, with almost no loss in overall classification

accuracy. For KRR, my communication-efficient approach (You, Demmel, Cho-Jui Hsieh, et al., 2018) is able to reduce the computation cost from $\Theta(n^3/P)$ to $\Theta((n/P)^3)$ where n is the number of data samples. We discuss these results in more detail in Chapters 2, 3, and 4.

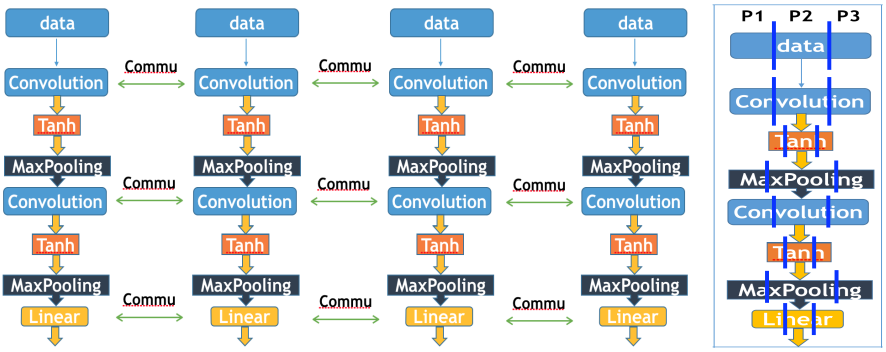
1.4 Large-Scale Optimization for Deep Learning (DL)

Existing optimization methods fail to extract enough parallelism to fully utilize supercomputers. I designed a large-batch algorithm to help the system to scale to thousands of processors. Our fast ImageNet training work won the **Best Paper Award of ICPP 2018** (You, Z. Zhang, Cho-Jui Hsieh, et al., 2018).

HPC and supercomputing techniques are becoming increasingly popular among Internet tech giants. Amazon AWS provides an elastic and scalable cloud infrastructure to run HPC applications[1]. Google released its first 100-petaFlop supercomputer, named the TPU Pod[2]. Facebook made a submission on the Top500 supercomputer list[3]. Microsoft is heavily investing in supercomputing techniques[4]. Modern supercomputers require an extremely high parallelism to reach their peak performance. However, where does the parallelism come from for DL? There are two kinds of parallelism for neural network training: data parallelism and model parallelism (J. Dean et al., 2012a). Let us assume P denotes the number of nodes. Model parallelism partitions the neural network into P pieces whereas data parallelism replicates the neural network itself in each node and divides the data across the P nodes. Figure 1.1 illustrate the ideas of data parallelism and model parallelism. Due to the data dependency between different layers in forward propagation and backward propagation, developers cannot efficiently parallelize across different layers except for pipelining (Huang et al., 2018). Thus, developers usually need to parallelize within each layer to implement model parallelism. In this way, a wider neural network provides higher parallelism. However, given the same number of parameters, a deep model typically achieves better results than a wide model (Eldan and Shamir, 2016). Thus, modern neural networks are deep rather than wide. For example, a typical layer of BERT (state-of-the-art NLP model) is a 1024-by-1024 matrix. The widest layer of BERT is a 1024-by-4096 matrix. If we disable data parallelism (i.e. set batch size as one), we can not make full use of the computational power of even one GPU or CPU chip to accelerate a 1024x1024x1024 matrix multiply.

For applications with an extremely wide model or a large single sample, model parallelism can work with data parallelism (e.g. Shazeer et al., 2018). Assume we have P nodes in this situation, and we partition these P nodes into G groups (e.g. G=256, P=1024). We use model-parallelism within each group and data-parallelism across different groups. Data parallelism dominates in most of our applications (i.e. $G \gg P/G$). Thus, I study the data-parallel DL method, which includes synchronous and asynchronous approaches. My study shows that the synchronous approach can be much faster and more stable than the asynchronous

[1]https://d1.awsstatic.com/HPC2019/Amazon-HyperionTechSpotlight-190329.FINAL-FINAL.pdf
[2]https://techcrunch.com/2019/05/07/googles-newest-cloud-tpu-pods-feature-over-1000-tpus
[3]https://www.top500.org/site/50701
[4]https://thenextweb.com/artificial-intelligence/2019/07/23/openai-microsoft-azure-ai/



1.1.1 1.1.2

Figure 1.1: Methods of parallelism. 1.1.1 is data parallelism on 4 machines. 1.1.2 is model parallelism on 3 machines. Let us assume P denotes the number of nodes, Model parallelism partitions the neural network into P pieces whereas data parallelism replicates the neural network itself in each node.

approach (You, Buluç, and Demmel, 2017). Our recent successes are based on a synchronous data-parallel approach. To scale this approach to more processors, we need to increase the batch size. The motivation behind this is that increasing the batch size as we increase the number of processors can keep the per-processor work constant, which avoids reducing the singe processor efficiency.

However, large-batch training is difficult because it introduces a generalization gap, i.e. increasing batch size often leads to a significant loss in test accuracy. Achieving the same accuracy as the baseline (e.g. batch size = 256) is the most challenging task for large-batch training. By controlling the learning rate (LR) during the training process, people can efficiently use large-batch training in some notable applications. Goyal et al., 2017 used linear scaling rule and warmup scheme in ImageNet training with ResNet-50. However, existing methods do not perform well when we increase the batch size beyond 1024 for some models like AlexNet. Also, the approach of Goyal et al., 2017 suffers from accuracy loss when we scale the batch size beyond 8K. Brute-force search or tuning of the hyper-parameters can not solve this problem. To further scale the batch size of existing methods and enable large-batch training to general models and datasets, we need a new optimizer. Our optimizer (You, Gitman, and Ginsburg, 2017), Layer-wise Adaptive Rate Scaling (LARS), was proposed to solve this problem. LARS uses different LRs for different layers based on the runtime information: the norm of the weights ($||w||$) and the norm of the gradients ($||\nabla w||$). The reason behind this technique is that we observe that the trust ratio ($||w||/||\nabla w||$) varies significantly for different layers. The optimal LR of a large-ratio layer may lead to the divergence of a small-ratio layer. To improve the generalization performance, we also add layer-wise weight decay to LARS. Our original LARS (You, Gitman, and Ginsburg, 2017) was simulated on a single-node system. People were sceptical about its robustness on a real distributed system. Therefore,

we designed a distributed-memory LARS (You, Z. Zhang, Cho-Jui Hsieh, et al., 2018; You, Z. Zhang, Cho-Jui Hsieh, et al., 2019), which helped us successfully scale the batch size to 32K without losing accuracy and finish the ImageNet/ResNet training in 14 minutes on a regular CPU cluster (without using any accelerators like GPUs or TPUs). All the ImageNet training speed records used LARS since December of 2017 (Table 1.1). We discuss these results in more detail in Chapter 5.

1.5 General Optimization and Auto-Tuning for DL

Huge models with a large number of parameters and hyper-parameters to tune bring an unprecedented challenge to auto-tuning optimization. I designed the LAMB (Layer-wise Adaptive Moments for Batch training) optimizer that minimizes the users' tuning efforts and achieves state-of-the-art accuracy on a diverse set of applications (You, J. Li, et al., 2020). This project was mentioned in Google's production release[5]. LAMB helped fast.ai to scale their transformer model to 128 GPUs[6]. LAMB helped Google establish new state-of-the-art results on GLUE, RACE, and SQuAD benchmarks (Z. Lan et al., 2019).

To motivate LAMB, we note that LARS performs well on a wide range of applications; however, one exception is BERT (Devlin et al., 2018), which is a huge DL language model with 300 million parameters. It builds on top of deep bidirectional transformers for language understanding. The baseline of BERT used the AdamW (Adam with weight decay) optimizer. Previous large-batch training techniques and AdamW perform poorly when we scale to an extremely large batch size (e.g. 16K). BERT training is very time-consuming (around 3 days on 16 TPU-v3 chips). To scale up the BERT batch size, I proposed the LAMB optimizer (You, J. Li, et al., 2020). LAMB supports adaptive element-wise updating and accurate layer-wise correction. LAMB is a general optimizer that works for both small and large batches. Users only need to input the initial LR and no other hyper-parameters. By using LAMB, we are able to scale the batch size of BERT to 64K without losing accuracy. Our study is the first work to reduce the BERT training time from 3 days to 76 minutes. We provide a convergence analysis for both LARS and LAMB to achieve a stationary point in non-convex settings (You, J. Li, et al., 2020). It is well known that the convergence rate of SGD depends on the maximum of all the Lipschitz constants (Ghadimi and G. Lan, 2013a). Our study shows that the convergence rate of LARS/LAMB only depends on the average of all the Lipschitz constants. That means LARS/LAMB can converge faster than SGD in large-batch setting.

Before our study, no optimizer could perform well in all the major DL tasks. For example, all the state-of-the-art ImageNet classification results are achieved by Momentum SGD, which significantly outperforms the adaptive optimizers like Adam. On the other hand, the Adam optimizer can achieve a higher performance than Momentum SGD in BERT training. To the best of our knowledge, LAMB is the first solver that can achieve state-of-the-art accuracy for both ImageNet training and BERT training (You, J. Li, et al., 2020). Even

[5]https://cloud.google.com/blog/products/ai-machine-learning/googles-scalable-supercomputers-for-machine-learning-cloud-tpu-pods-are-now-publicly-available-in-beta

[6]https://medium.com/@yaroslavvb/scaling-transformer-xl-to-128-gpus-d21875961c5d

using a small batch size, LAMB can still outperform current optimizers. In practice, different applications usually need different batch sizes because they use different hardware resources. LARS or SGD optimizer requires users to re-tune the hyper-parameters when they change the batch size. Re-tuning the hyper-parameters is a non-trivial overhead for a production team (especially at runtime). To solve this problem, I proposed the LEGW (Linear Epoch Gradual Warmup) approach (You, Hseu, et al., 2019). We also provide theoretical analysis to support LEGW. On top of LARS, LEGW enables users to scale the batch size from 256 to 65536 without tuning any hyper-parameters. LEGW can also automatically pick the learning rate for LAMB. The key difference between LAMB and LEGW is that LAMB is a general optimizer while LEGW is a hyper-parameter tuning framework. We discuss these results in more detail in Chapter 6.

Chapter 2

Communication-Efficient Support Vector Machines

2.1 Introduction

This chapter concerns the development of communication-efficient algorithms and implementations of kernel support vector machines (SVMs). The kernel SVM is a state-of-the-art algorithm for statistical nonlinear classification problems (Corinna Cortes and Vladimir Vapnik, 1995), with numerous practical applications (Joachims, 1998b; Tay and L. Cao, 2001; Leslie, Eskin, and Noble, 2002). However, the method's training phase greatly limits its scalability on large-scale systems. For instance, the most popular kernel SVM training algorithm, Sequential Minimal Optimization (SMO), has very little locality and low arithmetic intensity; we have observed that it might spend as much as 70% of its execution time on network communication on modern HPC systems.

Intuitively, there are two reasons for SMO's poor scaling behavior (You, Demmel, Kent Czechowski, L. Song, and Richard Vuduc, 2015). The first reason is that the innermost loop is like a large sparse-matrix-sparse-vector multiply, whose parallel isoefficiency function[1] scales like $W = \Omega(P^2)$. The second reason is that SMO is an iterative algorithm, where the number of iterations scales with the problem size. When combined, these two reasons result in an isoefficiency of $W = \Omega(P^3)$, meaning the method can only effectively use $\sqrt[3]{W}$ processors (refer to Section 5.4.2 of A. Grama, 2003 for W and P).

In this chapter, we first evaluate distributed memory implementations of three state-of-the-art SVM training algorithms: SMO (John C Platt, 1999), Cascade SVM (Graf et al., 2004), and Divide-and-Conquer SVM or DC-SVM (Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon, 2013). Our implementations of the latter two are the first-of-their-kind for distributed

[1] In parallel computing, the isoefficiency function denotes the growth rate of the problem size W required to keep the efficiency fixed as the number of processors P increases. A small isoefficiency function means that a small increase in the problem size is enough for an efficient utilization of an increasing number of processors, suggesting that the algorithm scales very well. However, a large isoefficiency function indicates a poorly scalable algorithm.

memory systems, as far as we know. We then optimize these methods through a series of techniques including: (1) developing a Divide-and-Conquer Filter (DC-Filter) method, which combines Cascade SVM with DC-SVM to balance accuracy and performance; (2) designing a Clustering-Partition SVM (CP-SVM) to improve the parallelism, reduce the communication, and improve accuracy relative to DC-Filter; and (3) designing 3 versions of a Communication-Efficient SVM or CE-SVM (BKM-SVM, FCFS-SVM, CA-SVM) that achieves load-balance and significantly reduces the amount of inter-node communication. Our contributions are:

(1) We convert a communication-intensive algorithm to an embarrassingly-parallel algorithm by significantly reducing the amount of inter-node communication.

(2) CE-SVM achieves significant speedups over the original algorithm with only small losses in accuracy on our test sets. In this way, we manage to balance the speedup and accuracy.

(3) We optimize the state-of-the-art training algorithms step-by-step, which both points out the problems of the existing approaches and suggests possible solutions.

For example, FCFS-SVM achieves 2-13× (6× on average) speedups over distributed SMO algorithm with comparable accuracies. The accuracy losses range from none to 1.1% (0.47% on average). According to previous work by others, such accuracy losses may be regarded as small and are likely to be tolerable in practical applications. FCFS-SVM improves the weak scaling efficiency from 7.9% to 39.4% when we increase the number of processors from 96 to 1536 on NERSC's Edison system (NERSC, 2016).

This chapter is based on a joint work with James Demmel, Kent Czechowski, Le Song, and Rich Vuduc. It was published as a journal paper entitled *Design and implementation of a communication-optimal classifier for distributed kernel support vector machines* (You, Demmel, Kent Czechowski, L. Song, and Rich Vuduc, 2016).

2.2 Background and Related Work

SVMs have two major phases: training and prediction. The training phase builds the model from a labeled input data set, which the prediction phase uses to classify new data. The training phase is the main limiter to scaling, both with respect to increasing the training set size and increasing the number of processors. By contrast, prediction is embarrassingly parallel and fairly "cheap" per data point. Therefore, we focus on training, just like prior papers on SVM-acceleration (John C Platt, 1999; L. J. Cao and Keerthi, 2006; Graf et al., 2004).

In terms of potential training algorithms, there are many options. In this chapter, we focus on a class of algorithms we will call *partitioned SMO algorithms*. These algorithms work essentially by partitioning the data set, building kernel SVM models for each partition using SMO as a building block, and then combining the models to derive a single final model. In addition, they estimate model parameters using iterative methods. We focus on two exemplars of this class, *Cascade SVM* (§ 2.2) and *Divide-and-Conquer SVM* (§ 2.2). We briefly survey alternative methods in § 2.2. Our primary reason for excluding them in this study is that they

use very different approaches that are both complex to reproduce and that do not permit the same kind of head-to-head comparisons as we wish to consider here.

SVM Training and Prediction

We focus on two-class (binary-class) kernel SVMs, where each data point has a binary label that we wish to predict. Multi-class (3 or more classes) SVMs may be implemented as several independent binary-class SVMs; a multi-class SVM can be easily processed in parallel once its constituent binary-class SVMs are available. The training data in an SVM consists of m samples, where each sample is a pair (X_i, y_i) and $i \in \{1, 2, ..., m\}$. X_i is the i-th training sample, represented as a vector of features. Each y_i is the i-th sample's label; in the binary case, each y_i has one of two possible values, $\{-1, 1\}$. Mathematically, the kernel SVM training is typically carried out in its dual formulation where a set of coefficients α_i (called Lagrange multipliers), with each α_i associated with a sample (X_i, y_i), are found by solving the following linearly-constrained convex Quadratic Programming (QP) problem, eqns. (2.1–2.2):

$$\text{Maximize: } F(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j K_{i,j} \tag{2.1}$$

$$\text{Subject to: } \sum_{i=1}^{m} \alpha_i y_i = 0 \text{ and } 0 \leq \alpha_i \leq C, \forall i \in \{1, 2, ..., m\} \tag{2.2}$$

Here, C is a regularization constant that attempts to balance generality and accuracy; and $K_{i,j}$ denotes the value of a kernel function evaluated at a pair of samples, X_i and X_j. (Typical kernels appear in Table 2.1.) The value C is chosen by the user.

The training produces the vector of Lagrange multipliers, $[\alpha_1, \alpha_2, ..., \alpha_m]$. The predicted label for a new sample, $\hat{X}$, is computed by evaluating eqn. (2.3),

$$\hat{y} = \sum_{i=1}^{m} \alpha_i y_i K(\hat{X}, X_i) \tag{2.3}$$

In effect, α in eqn. (2.3) is the model learned during training. One goal of SVM training is to produce a compact model, that is, one whose α coefficients are *sparse* or mostly zero. The set of samples with non-zero α_i are called the *support vectors*. Observe that only the samples with non-zero Lagrange multipliers ($\alpha_i \neq 0$) can have an effect on the prediction result.

It is worth noting that $K(X_i, X_j)$ is (re)computed on demand by all algorithms that use it, as opposed to computing all values once and storing them. The reason is that the kernel matrix needs $O(m^2)$ memory, which is prohibitive for real-world applications because m is usually much larger than the sample dimension. For example, a 357 MB dataset (520,000 $\times$ 90 matrix) (Bertin-Mahieux et al., 2011) would generate a 2000 GB kernel matrix. To clarify the notation, $K(X_i, X_j)$ means the kernel function that computes the kernel value of X_i and X_j. $K_{i,j}$ means the value at i-th row and j-column of kernel matrix K, so we have $K_{i,j} = K(X_i, X_j)$.

Table 2.1: Standard Kernel Functions

Linear	$K(X_i, X_j) = {X_i}^\top X_j$
Polynomial	$K(X_i, X_j) = (a{X_i}^\top X_j + r)^d$
Gaussian	$K(X_i, X_j) = \exp(-\gamma \|\|X_i - X_j\|\|^2)$
Sigmoid	$K(X_i, X_j) = \tanh(a{X_i}^\top X_j + r)$

Sequential Minimal Optimization (SMO)

The most widely used kernel SVM training algorithm is Platt's *Sequential Minimal Optimization* (*SMO*) algorithm (John C Platt, 1999). It is the basis for popular SVM libraries and tools, including LIBSVM (C.-C. Chang and C.-J. Lin, 2011) and GPUSVM (Catanzaro, Sundaram, and Keutzer, 2008). The overall structure of the SMO algorithm appears in Alg. 1. In essence, it iteratively evaluates the following formulae:

$$f_i = \sum_{j=1}^{m} \alpha_j y_j K(X_i, X_j) - y_i \tag{2.4}$$

$$\hat{f}_i = f_i + \Delta\alpha_{high} y_{high} K_{high,i} + \Delta\alpha_{low} y_{low} K_{low,i} \tag{2.5}$$

$$\Delta\alpha_{low} = \frac{y_{low}(b_{high} - b_{low})}{K_{high,high} + K_{low,low} - 2K_{high,low}} \tag{2.6}$$

$$\Delta\alpha_{high} = -y_{low} y_{high} \Delta\alpha_{low} \tag{2.7}$$

For a detailed performance bottleneck analysis of SMO, see You et al. (You, Shuaiwen Song, et al., 2014). The most salient observations we can make are that (a) the dominant update rule is eqn. (2.4), which is a matrix-vector multiply (with kernel); and (b) the number of iterations necessary for convergence will tend to scale with the number of samples, m.

All of the algorithmic improvements in this chapter start essentially from SMO. In particular, we adopt the approach of Cao et al. (L. J. Cao and Keerthi, 2006), who designed a parallel SMO implementation for distributed memory systems. As far as we know, it is the best distributed SMO implementation so far. The basic idea is to partition the data among nodes and launch a big distributed SVM across those nodes. This means all the nodes share one model during the training phase. Their implementation fits within a map-reduce framework. The two-level ("local" and "global") map-reduce strategy of Catanzaro et al. can significantly reduce the amount of communication (Catanzaro, Sundaram, and Keutzer, 2008). However, Catanzaro et al. target single-node (single-GPU) systems, whereas we focus on distributed memory scaling.

Cascade SVM

Cascade SVM is a multi-layer approach designed with distributed systems in mind (Graf et al., 2004). As Fig. 2.1 illustrates, its basic idea is to divide the SVM problem into P smaller SVM sub-problems, and then use a kind of "reduction tree" to re-combine these smaller SVM

Algorithm 1 Sequential Minimal Optimization (SMO)

Input the samples X_i and labels y_i, $\forall i \in \{1, 2, ..., m\}$.
$\alpha_i = 0$, $f_i = -y_i$, $\forall i \in \{1, 2, ..., m\}$.
$b_{high} = -1$, $high = \min\{i : y_i = 1\}$
$b_{low} = 1$, $low = \min\{i : y_i = -1\}$.
Update α_{high} and α_{low} according to Equations (2.6) and (2.7).
Update f_i according to Equation (2.5), $\forall i \in \{1, 2, ..., m\}$
$I_{high} = \{i : 0 < \alpha_i < C \vee y_i > 0, \alpha_i = 0 \vee y_i < 0, \alpha_i = C\}$
$I_{low} = \{i : 0 < \alpha_i < C \vee y_i > 0, \alpha_i = C \vee y_i < 0, \alpha_i = 0\}$
$high = \arg\min\{f_i : i \in I_{high}\}$
$low = \arg\max\{f_i : i \in I_{low}\}$
$b_{high} = \min\{f_i : i \in I_{high}\}, b_{low} = \max\{f_i : i \in I_{low}\}$
Update α_{high} and α_{low} according to Equations (2.6) and (2.7).
If $b_{low} > b_{high}$, then go to Step 6.

models into a single result. The subproblems and combining steps could in principle use any SVM training method, though in this chapter we consider those that use SMO. A Cascade SVM system with P computing nodes has $\log(P) + 1$ layers. In the same way, the whole training dataset (TD) is divided into P smaller parts ($TD_1, TD_2, ..., TD_P$), each of which is processed by one sub-SVM. The training process selects certain samples (with non-zero Lagrange multiplier, i.e. α_i) out of all the samples. The set of support vectors, SV, is a subset of the training dataset ($SV_i \subseteq TD_i, i \in \{1, 2, ..., P\}$). Each sub-SVM can generate its own SV. For Cascade, only the SV will be passed from the current layer to next layer. The α_i of each support vector will also be passed to the next layer to provide a good initialization for the next layer, which can significantly reduce the iterations for convergence. On the next layer, any two consecutive SV sets (SV_i and SV_{i+1}) will be combined into a new sub-training dataset. In this way, there is only one sub-SVM on the $(\log(P) + 1)$-st layer.

Divide-and-Conquer SVM (DC-SVM)

DC-SVM is similar to Cascade SVM (Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon, 2013). However, it differs in two ways: (1) Cascade SVM partitions the training dataset evenly on the first layer, while DC-SVM uses K-means clustering to partition the dataset; and (2) Cascade SVM only passes the set of support vectors from one layer to the next, whereas DC-SVM passes *all* of the training dataset from layer to layer. At the last layer of DC-SVM, a single SVM operates on the whole training dataset.

K-means clustering: since K-means clustering is a critical sub-step for DC-SVM, we review it here. The objective of K-means clustering is to partition a dataset TD into $k \in Z^+$ sub-datasets ($TD_1, TD_2, ..., TD_k$), using a notion of proximity based on Euclidean distance (Forgy, 1965). The value of k is chosen by the user. Each sub-dataset has a center ($CT_1, CT_2, ..., CT_k$). The center has the same structure as a sample (i.e. n-dimensional vector). Sample X will belong to TD_i if CT_i is the closest data center to X. In this work, k is set to

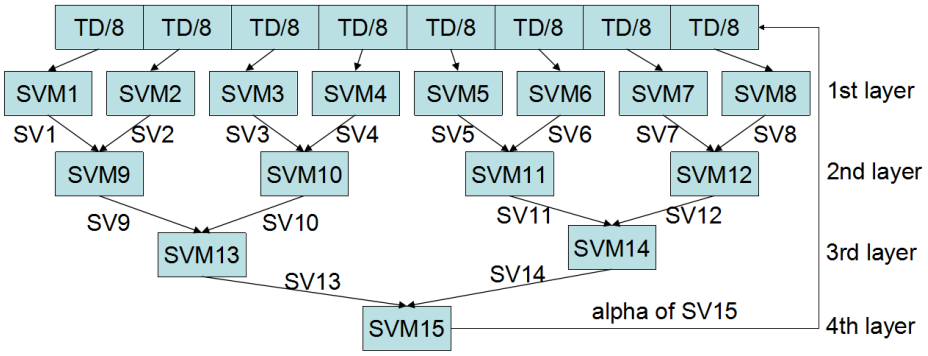


Figure 2.1: This figure is an illustration of Cascade SVM (Graf et al., 2004). Different layers have to be processed sequentially, i.e. layer $i+1$ can only be processed after layer i has been finished. The tasks in the same level can be processed concurrently. If the result at the bottom layer is not good enough, the user can distribute all the support vectors (SV15 in the figure) to all the nodes and re-do the whole pass from the top layer and to the bottom layer. However, for most applications, the result will not become better after another Cascade pass. One pass is enough in most cases.

be the number of processors. A naive version of K-means clustering appears in Alg. 2.

Algorithm 2 Naive K-means Clustering

Input the training samples X_i, $i \in \{1, 2, ..., m\}$.
Initialize data center $CT_1, CT_2, ..., CT_k$ randomly.
set $\delta = 0$
For every i, set $c^i = argmin_j ||X_i - CT_j||$.
If c^i has been changed, $\delta = \delta + 1$
For every j, set $CT_j = \frac{\sum_{i=1}^{m} 1\{c^i=j\} X_i}{\sum_{i=1}^{m} 1\{c^i=j\}}$, $j \in \{1, 2, ..., k\}$.
If $\delta/m >$ threshold, then go to Step 3.

Other methods

There are other potential algorithms for SVMs. One method uses matrix factorization of the kernel matrix K (E. Y. Chang, 2011). Another class of methods relies on solving the QP problem using an iteration structure that considers more than two points at a time (G. Wu et al., 2006; Zanni, Serafini, and Zanghirati, 2006). Additionally, there are other optimizations for serial approach (Joachims, 1999; John C Platt, 1999; Fan, P.-H. Chen, and C.-J. Lin, 2005) or parallel approach on shared memory systems (Catanzaro, Sundaram, and Keutzer, 2008; T.-K. Lin and Chien, 2010). All of these approaches are hard to compare "head-to-head"

Table 2.2: Terms for Performance Modelling

m; n; P	# samples; # features per sample; # nodes or processes
T_1; T_p	serial run time; parallel run time
t_s; t_w	latency time for communication; per-word transfer time
V_k	# SVs in k_{th} Cascade layer, $V_1 = m$
L_k	maximal # iters of all nodes in k_{th} Cascade layer
P_k	# processes in k_{th} Cascade layer
W; T_o	problem size in flops; parallel overhead ($T_o = PT_p - W$)
s; I; k	# SVs; # SVM iters; # K-means iters

against the partitioned SMO schemes this chapter considers, so we leave such comparisons for future work.

2.3 Re-Design Divide-and-Conquer Method

Performance Modeling for Existing Methods

In this section, we will do performance modeling for the three related methods mentioned in Section 2.2. The related terms are in Table 2.2 and the proofs can be found in (You, Demmel, Kent Czechowski, L. Song, and Richard Vuduc, 2015). To evaluate the scalability, we refer to Iso-efficiency function (Section 5.4.2 of A. Grama, 2003), shown in Equation (2.8) where E ($E = T_1/(pT_p)$) is the desired scaling efficiency (Specifically, $T_1 = t_c W$ where t_c is the time per flop. In this chapter, to make it simple, we normalize so that $t_c = 1$. In the same way, t_s and t_w in Table 2.2 actually are ratios of communication time to flop time). T_o is the overall overhead, T_o^{comm} is the communication overhead, and T_o^{comp} is the computation overhead. The minimum problem size W can usually be obtained as a function of P by algebraic manipulations. This function dictates the growth rate of W required to keep the efficiency fixed as P increases. For example, the Iso-efficiency function of 1D Mat-Vec-Mul is $W = \Omega(P^2)$, and it is $W = \Omega(P)$ for 2D Mat-Vec-Mul (Section 8.1 of (A. Grama, 2003), $W = n^2$ where n is the matrix dimension for Mat-Vec-Mul). Mat-Vec-Mul is more scalable with 2-D partitioning because it can deliver the same efficiency on more processors with 2-D partitioning ($P = O(W)$) than with 1-D partitioning ($P = O(\sqrt{W})$).

$$W = \frac{E}{1-E} T_o = \frac{E}{1-E}(T_o^{comm} + T_o^{comp}) \tag{2.8}$$

Distributed SMO (Dis-SMO)

Our Dis-SMO implementation is based on the idea of Cao's paper, we also include Catanzaro's improvements in the code. The serial runtime (T_1) of a SMO iteration is $2mn$ and its parallel runtime (T_p) per iteration is in Equation (2.9). Based on the terms in Table 2.2, the parallel overhead (T_o) can be obtained in Equation (2.10). The scaling model is in Table 2.4. This model is based on single-iteration SMO. However, the model of the completely converged SMO algorithm will be worse (i.e. the lower bound will be larger) because the number of iterations is proportional to the number of samples (Table 2.3). This will furthermore jeopardize the scalability for large-scale computation.

Table 2.3: The number of iterations with different number of samples, epsilon and forest are the test datasets

Samples	10k	20k	40k	80k	160k	320k
Iters (epsilon)	4682	8488	15065	26598	49048	90320
Iters (forest)	3057	6172	11495	22001	47892	103404

$$T_p = 14logPt_s + [2nlogP + 4P^2]t_w + \frac{2mn + 4m}{P} + 2P + n \tag{2.9}$$

$$T_o = 14PlogPt_s + [2nPlogP + 4P^3]t_w + 4m + 2P^2 + nP \tag{2.10}$$

Cascade and DC-SVM

The communication and computation Iso-efficiency functions of Cascade are in Equation (2.11) and Equation (2.12) respectively. Since V_{1+logP} is the number of support vectors of the whole system, we can get that $V_{1+logP} = \Theta(m)$. On the other hand, the number of training samples can not be less than the number of nodes (i.e. $m = \Omega(P)$), because we can not keep all P nodes busy. That is $V_{1+logP} = \Omega(P)$. Therefore, after substituting V_{1+logP} by $\Omega(P)$ in Equation (2.11), we obtain that the lower bound of communication Iso-efficiency function $W = \Omega(P^3)$. Because we can not predict the number of support vectors and the number of iterations on each level (i.e. V_{k-1} and L_k in Equation (2.12)) beforehand, we can only get the upper bound for the computation Iso-efficiency function (Table 2.4). For DC-SVM, since the K-means time is significantly less than the SVM time (Tables 2.9 to 2.14), we ignore the effect of K-means on the whole system performance. Therefore, we get the Iso-efficiency function of DC-SVM by replacing V_k of Cascade with m (Table 2.4).

$$W^{cascade,comm} = \Theta((\sum_{k=2}^{logP} n2^k V_k) + P^2 V_{1+logP}) \tag{2.11}$$

$$W^{cascade,comp} = \Theta(n(\sum_{k=2}^{1+logP} L_k V_{k-1} 2^k - 2Im)) \tag{2.12}$$

Table 2.4: Scaling Comparison for Iso-efficiency Function

Method	Communication	Computation
1D Mat-Vec-Mul	$W = \Omega(P^2)$	$W = \Theta(P^2)$
2D Mat-Vec-Mul	$W = \Omega(P)$	$W = \Theta(P)$
Distributed-SMO	$W = \Omega(P^3)$	$W = \Omega(P^2)$
Cascade	$W = \Omega(P^3)$	$W = O(\sum_{k=1}^{logP} nL_kV_{k-1}2^k)$
DC-SVM	$W = \Omega(P^3)$	$W = O(\sum_{k=1}^{logP} nL_km2^k)$

We compare with Mat-Vec-Mul, which is a typical communication-intensive kernel. Actually, the scalability of these three methods are even worse than 1D Mat-Vec-Mul, which means we need to design a new algorithm to scale up SVM on future exascale computing systems. Our scaling results in Section 2.5 are in line with our analysis.

DC-Filter: Combination of Cascade and DC-SVM

From our experimental results, we observe that Cascade is faster than Dis-SMO. However, the classification accuracy of Cascade is worse. DC-SVM can obtain a higher classification accuracy. Nevertheless, the algorithm becomes extremely slow (Tables 2.9 to 2.14). The reason is that DC-SVM has to pass all the samples layer-by-layer, and this significantly increases the communication overhead. In addition, more data on each node means the processors have to do more on-chip communication and computation. Therefore, our first design is to combine Cascade with DC-SVM. We refer to this approach as Divide-and-Conquer Filter (DC-Filter).

Like DC-SVM, we apply K-means in DC-Filter to get a better data partition, which can help to get a good classification accuracy (Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon, 2013). It is worth noting that K-means itself does not significantly increase the computation and communication overhead (Tables 2.9 to 2.14). For example, K-means converges in 7 loops and only costs less than 0.1% of the total runtime for processing the ijcnn dataset. However, we need to redistribute the data after K-means, which may increase the communication overhead. On the other hand, we apply the filter function of Cascade in the combined approach. On each layer, only the support vectors rather than all the training samples will be sent to next layer, which is like a filter since SV is a subset of the original training dataset. The Lagrange multiplier of each support vector will be sent with it to give a good initialization for next layer, which can reduce the number of iterations for convergence (Graf et al., 2004). In our experiments, the speed and accuracy of DC- Filter fall in between Cascade and DC-SVM, or perform better than both of them. DC-Filter is a compromise between these two existing approaches, which is our first attempt to balance the accuracy and the speedup.

2.4 Communication-Efficient Design

CP-SVM: Clustering-Partition SVM

The node management for Cascade, DC-SVM, and DC-Filter are actually similar to each other (i.e. Fig. 2.1). Table 2.5 provides the detailed profiling result of a toy Cascade example to show how they work. We can observe that only 27% (5.49/20.1) of the total time is spent on the top layer, which makes full use of all the nodes. In fact, almost half (9.69/20.1) of the total time is spent on the bottom layer, which only uses one node. In this situation, the Cascade-like approach does not perform well because the parallelism in most of the algorithm is extremely low. The weighted average number of nodes used is only 3.3 (obtained by Equation (2.13)) for the example in Table 2.5. However, the system actually occupies 8 nodes for the whole runtime. Specifically, the parallelism is decreasing by a factor of 2 layer-by-layer. For some datasets (e.g. Table 2.10), the lower level can be fast and converge within $\Theta(1)$ iterations. For other datasets (e.g. Table 2.5), the lower level is extremely slow and becomes the bottleneck of the runtime performance. Therefore, we need to redesign the algorithm again to make it highly parallel and make full use of all the computing nodes.

$$\frac{\sum_{l=1}^{1+logP}((time\_of\_layer\_l) \times (\#nodes\_of\_layer\_l))}{\sum_{l=1}^{1+logP}(time\_of\_layer\_l)} \tag{2.13}$$

The analysis in this section is based on the Gaussian kernel with $\gamma > 0$ because it is the most widely used case (Catanzaro, Sundaram, and Keutzer, 2008). Other cases can work in the same way with different implementations. For any two training samples, their kernel function value is close to zero ($exp\{-\gamma||X_i - X_j||^2\} \to 0$) when they are far away from each other in Euclidean distance ($||X_i - X_j||^2 \to \infty$). Therefore, for a given sample $\hat{X}$, only the support vectors close to $\hat{X}$ can have an effect on the prediction result (Equation (2.3)) in the classification process. Based on this idea, we can divide the training dataset into P parts $(TD_1, TD_2, ..., TD_P)$. We use K-means to divide the initial dataset since K-means clustering is based on Euclidean distance. After K-means clustering, each sub-dataset will get its data center $(CT_1, CT_2, ..., CT_P)$. Then we launch P independent support vector machines $(SVM_1, SVM_2, ..., SVM_P)$ to process these P sub-datasets, which is like the top layer of the DC-Filter algorithm.

After the training process, each sub-SVM will generate its own model file $(MF_1, MF_2, ..., MF_P)$. We can use these model files independently for classification. For a given sample $\hat{X}$, if its closest data center (in Euclidean distance) is CT_i, we will only use MF_i to make a prediction for $\hat{X}$ because the support vectors in other model files have little impact on the classification result. Fig. 2.2 is the general flow of CP-SVM. CP-SVM is highly parallel because all the sub-problems are independent of each other. The communication overhead of CP-SVM is from K-means clustering and data distribution. CP-SVM generally is faster than the previous algorithms and its accuracy is closer to the SMO algorithm (Tables 2.9 to 2.14). However, in terms of scalability and speed, it is still not good enough. It is worth noting that K-means itself does not significantly increase the computation and communication

Table 2.5: Profile of 8-node & 4-layer Cascade for a subset of ijcnn dataset

level 1^{st}								
node rank	1	2	3	4	5	6	7	8
samples	6000	6000	6000	6000	6000	6000	6000	6000
time: 5.49s	4.87	4.92	4.90	4.68	5.12	5.10	5.49	4.71
iter: 6168	5648	5712	5666	5415	5936	5904	6168	5453
SVs: 5532	746	715	717	718	686	707	721	699
level 2^{nd}								
node rank	1		3		5		7	
samples	1461		1435		1393		1420	
time: 1.58s	1.58		1.50		1.35		1.45	
iter: 7485	7485		7211		6713		7035	
SVs: 5050	1292		1263		1256		1239	
level 3^{rd}								
node rank	1				5			
samples	2555				2495			
time: 3.34s	3.34				3.30			
iter: 9081	8975				9081			
SVs: 4699	2388				2311			
level 4^{th}								
node rank	1							
samples	4699							
time: 9.69s	9.69							
iter: 14052	14052							
SVs: 4475	4475							

overhead. However, we need to redistribute the data after K-means, which may increase the communication overhead.

Communication-Efficient SVM

Based on the profiling result in Fig. 2.6, we can observe that CP-SVM is not well load-balanced. The reason is that the partitioning by K-means is irregular and imbalanced. For example, processor 2 in Fig. 2.6 has to handle 35,137 samples while processor 7 only needs to process 9,685 samples. Therefore, we need to replace K-means with a better partitioning algorithm that balance the load while maintaining accuracy. We design three versions of balanced partitioning algorithms and use them to build the communication-efficient algorithms.

First Come First Served (FCFS) SVM

The goal of FCFS is to assign an equal number of samples m/P to each processor, where each sample is assigned to the processor with the closest center that has not already been assigned m/P samples. Centers are the locations of the first particles randomly chosen and assigned to each processor. (Other choices of centers are imaginable, such as doing K-means;

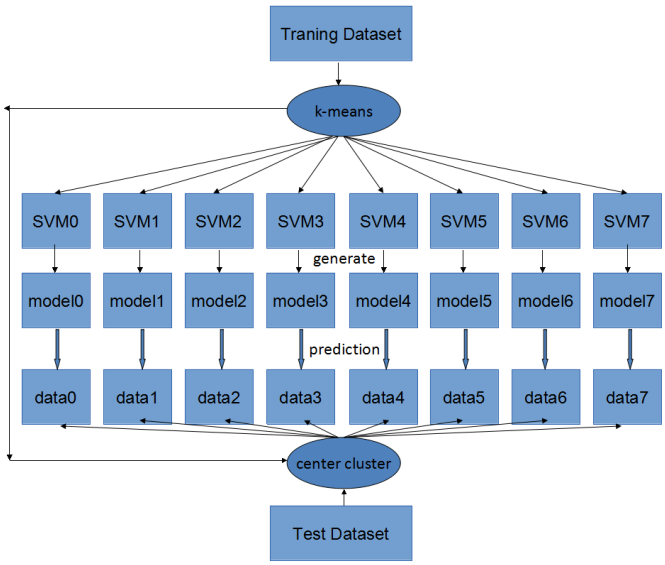


Figure 2.2: General Flow for CP-SVM. In the training part, different SVMs process its own dataset independently. In the classification part, different models can make the prediction independently.

this is the BKM algorithm below.) The detailed FCFS partitioning method is in Algorithm 3. Lines 1-4 of Algorithm 3 is the initiation phase: we randomly pick P samples from the dataset as the initial data centers. Lines 5-15 find the center for each sample. Lines 7-13 find the best under-loaded center for the i-th sample. Lines 16-22 recompute the data center by averaging all the samples assigned to each center. Recomputing the centers by averaging is optional because it will not necessarily make the results better. Fig. 2.3 is an example of Algorithm 3. From Fig. 2.4 we can observe that FCFS can partition the dataset in a balanced way. After FCFS partitioning, all the nodes have the same number of samples. Then the algorithm framework is the same as CP-SVM.

Balanced K-means (BKM) SVM

As mentioned above, the objective of BKM partitioning algorithm is to make the number of samples on each node close to m/P (a machine node corresponds to a data center) based on Euclidean distance. The basic idea of this algorithm is to slightly rearrange the results of the original K-means algorithm. We will keep moving samples from the over-loaded centers to under-loaded centers till they are balanced. The balanced K-means partitioning method is detailed in Algorithm 4. Lines 1-4 of Algorithm 4 compute the K-means clustering of all the inputs. In lines 6-8, we calculate the Euclidean distance distance between every sample and every center: $dist[i][j]$ is the Euclidean distance between i-th sample and j-th center. The variable *balanced* is the number of samples every center should have in the load-balanced

Algorithm 3 First Come First Served Partitioning

Input:

SA[i] is the i-th sample
m is the number of samples
P is the number of clusters (processes)

Output:

MB[i] is the closest center to i-th sample
CT[i] is the center of i-th cluster
CS[i] is the size of i-th cluster

```
Randomly pick P samples from m samples (RS[1:P]) for i ∈ 1 : P do
|  CT[i] = RS[i]  CS[i] = 0
end
balanced = m/P for i ∈ 1 : m do
|  mindis = inf  minind = 0 for j ∈ 1 : P do
|  |  dist = EuclideanDistance(SA[i], CT[j]) if dist < mindis and CS[j] < balanced then
|  |  |  mindis = dist  minind = j
|  |  end
|  end
|  CS[minind]++  MB[i] = minind
end
for i ∈ 1 : P do
|  CT[i] = 0
end
for i ∈ 1 : m do
|  j = MB[i]  CT[j] += SA[i]
end
for i ∈ 1 : P do
|  CT[i] = CT[i] / CS[i]
end
```

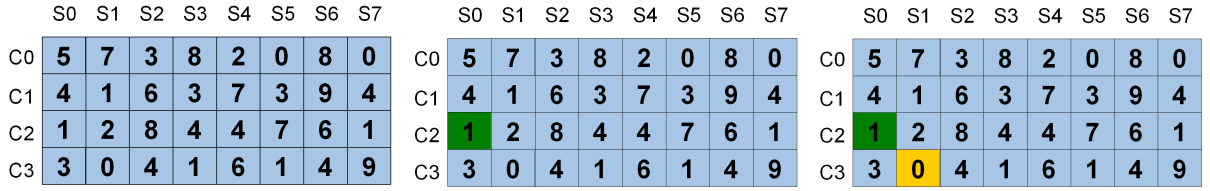
situation. After the K-means clustering, some centers will have more than *balanced* samples. In lines 9-26, the algorithm will move some samples from the over-loaded centers to the under-loaded centers. For a given over-loaded center, we will find the farthest sample (lines 13-16). The id of the farthest sample is *maxind*. In lines 17-23, we find the closest under-loaded center to sample *maxind*. In lines 24-26, we move sample *maxind* from its over-loaded center to the best under-loaded center. In lines 27-33, we recompute the data center by averaging the all the samples in a certain center. Recomputing the centers by averaging is optional. Fig. 2.5 is an example of Algorithm 4. After the BKM algorithm is finished and the load-balance is achieved, the algorithm framework is the same as CP-SVM.

Algorithm 4 Balanced K-means Partitioning

Input:
SA[i] is the i-th sample
m is the number of samples
P is the number of clusters (processes)

Output:
MB[i] is the closest center to i-th sample
CT[i] is the center of i-th cluster
CS[i] is the size of i-th cluster

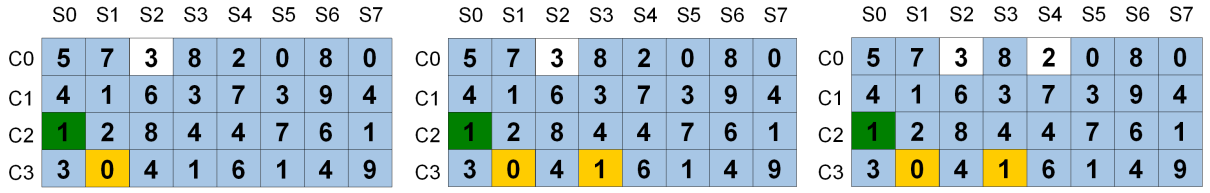
```
Randomly pick P samples from m samples (RS[1:P])  for i ∈ 1 : P do
  | CT[i] = RS[i]
end
do kmeans clustering on all input data  balanced = m/P  for i ∈ 1 : m do
  | for j ∈ 1 : P do
  |   | dist[i][j] = EuclideanDistance(SA[i], CT[j])
  | end
end
for j ∈ 1 : P do
  | while CS[j] > balanced do
  |   | maxdist = 0  maxind = 0  for i ∈ 1 : m do
  |   |   | if dist[i][j]>maxdist and MB[i]==j then
  |   |   |   | maxdist = dist[i][j]  maxind = i
  |   |   | end
  |   | end
  |   | mindist = inf  minind = j  for k ∈ 1 : P do
  |   |   | if dist[maxind][k]<mindist then
  |   |   |   | if CS[k]<balanced then
  |   |   |   |   | mindist = dist[maxind][k]  minind = k
  |   |   |   | end
  |   |   | end
  |   | end
  |   | MB[maxind] = minind  CS[j]=CS[j]-1  CS[minind]=CS[minind]+1
  | end
end
for i ∈ 1 : P do
  | CT[i] = 0
end
for i ∈ 1 : m do
  | j = MB[i]  CT[j] += SA[i]
end
for i ∈ 1 : P do
  | CT[i] = CT[i] / CS[i]
end
```



2.3.1 We have 8 samples (S0-S7) and want to distribute them to 4 centers (C0-C3). In the load balanced situation, each center has 2 samples.

2.3.2 The closest center to S0 is C2 (1<3<4<5). Since C2 is under-loaded, we move S0 to C2. After this, C2 is still under-loaded.

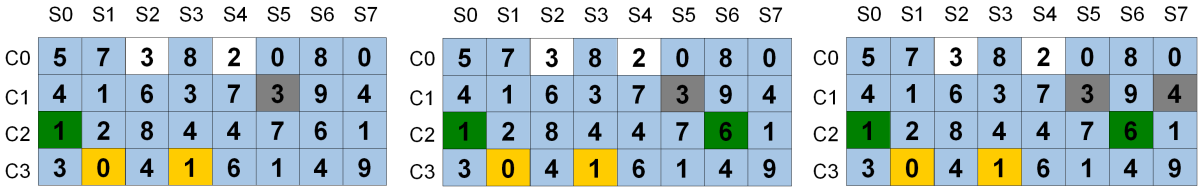
2.3.3 The closest center to S1 is C3 (0<1<2<7). Since C3 is under-loaded, we move S1 to C3. After this, C3 is still under-loaded.



2.3.4 The closest center to S2 is C0 (3<4<6<8). Since C0 is under-loaded, we move S2 to C0. After this, C0 is still under-loaded.

2.3.5 The closest center to S3 is C3 (1<3<4<8). Since C3 is under-loaded, we move S3 to C3. After this, C3 is balanced.

2.3.6 The closest center to S4 is C0 (2<4<6<7). Since C0 is under-loaded, we move S4 to C0. After this, C0 is balanced.



2.3.7 The closest center to S5 is C0 (0<1<3<7). Since C0 and C3 are balanced, we move S5 to C1. After this, C3 is balanced.

2.3.8 The closest center to S6 is C3 (4<6<8<9). Since C3 is balanced, we move S6 to C2. After this, C2 is balanced.

2.3.9 Since only C1 is under-loaded, we move S7 to C1, which is the third choice. After this, all the centers are balanced.

Figure 2.3: This is an example of First Come First Served (FCFS) partitioning algorithm. Each figure is a distance matrix, which is referred as *dist*. For example, *dist*[i][j] is the distance between i-th center and j-th sample. The color of the matrix in the first figure is the original color. If *dist*[i][j] has a different color than the original one, then it means that j-th sample belongs to i-th center.

Communication-Avoiding SVM (CA-SVM)

For CA-SVM, the basic idea is to randomly divide the original training dataset into P parts $(TD_1, TD_2, ..., TD_P)$ evenly. After partitioning, each sub-dataset will generate its own data center $(CT_1, CT_2, ..., CT_P)$. For TD_i $(i \in \{1, 2, ..., P\})$, its data center (i.e. CT_i) is the average of all the samples on node i. Then we launch P independent support vector

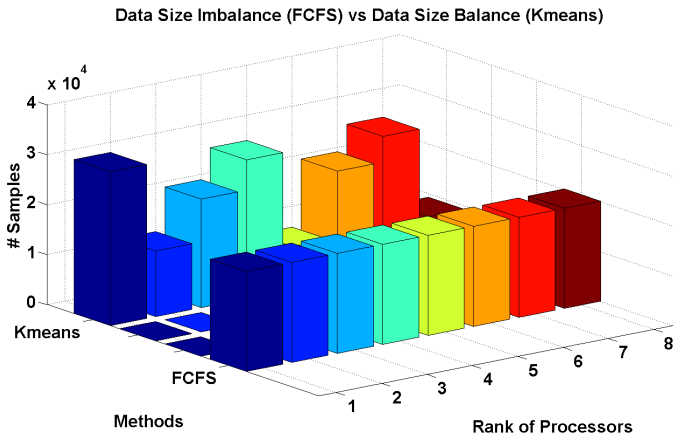


Figure 2.4: The figure shows that the partitioning by K-means is imbalanced while the partitioning by FCFS is balanced. Specifically, each node has exactly 20,000 samples after FCFS partitioning. The test dataset is *face* with 160,000 samples (361 features per sample). 8 nodes are used in this test.

machines $(SVM_1, SVM_2, ..., SVM_P)$ to process these P sub-datasets in parallel. After the training process, each sub-SVM will generate its own model file $(MF_1, MF_2, ..., MF_P)$. Like CP-SVM, we can use these model files independently for classification. For any unknown sample $(\hat{X})$, if its closest data center is CT_i, we will only use MF_i to make prediction for $\hat{X}$. The communication overheads of CP-SVM and BKM-SVM are from the data transfer and distribution in K-means like partitioning algorithm. The communication overhead of FCFS-SVM is from the FCFS clustering method. In this new method, we replace the K-means variants or FCFS with a no-communication partition. Thus, we can also directly refer it as CA-SVM (Communication-Avoiding SVM). However, this assumes that originally the dataset is distributed to all the nodes. To give a fair comparison, we implement two versions of CA-SVM. **casvm1** means that we put the initial dataset on just one node, which needs communication to distribute the dataset to different nodes. **casvm2** means that we put the initial dataset on different nodes, which needs no communication (Fig. 2.8). All the results of CA-SVM in Section 2.5 are based on casvm2. CA-SVM may lose accuracy because evenly-randomly dividing does not get the best partitioning in terms of Euclidean distance. However, the results in Tables 2.9 to 2.14 show that it achieves significant speedup with comparable results.

The framework of CA-SVM is shown in Algorithm 5. The prediction process may need a little communication. However, both the data centers and test samples are pretty small compared with the training samples. Also, the overhead of single variable reduce operation is very low. This communication will not bring about significant overhead. On the other hand, the majority of SVM time is spent on the training process. Like previous work (e.g. SMO, Cascade, DC-SVM), the focus of this chapter is on optimizing the training process.

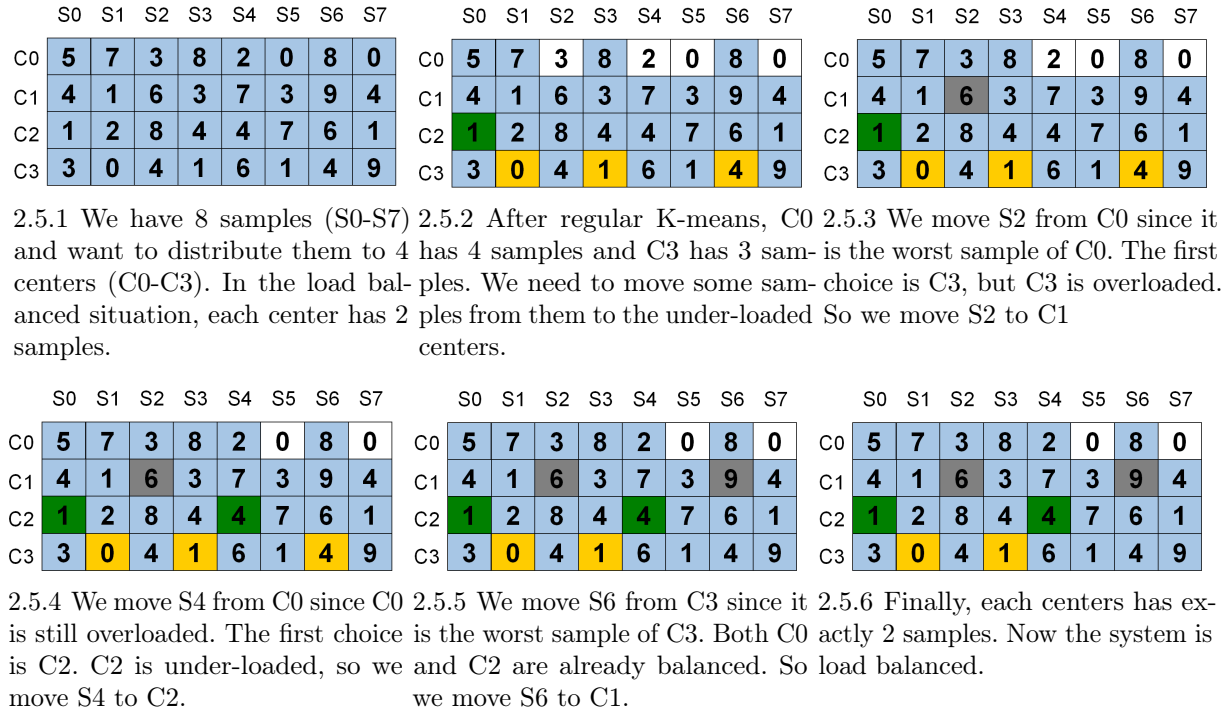


Figure 2.5: This is an example of Balanced K-means partitioning algorithm. Each figure is a distance matrix, which is referred as *dist*. For example, *dist*[i][j] is the distance between i-th center and j-th sample. The color in the first figure is the original color. If *dist*[i][j] has a different color than the original one, then it means that j-th sample belongs to i-th center.

Initial Data Distribution

The major communication overhead of CP-SVM or BKM-SVM are from three parts: (1) The distributed K-means-like clustering algorithm. (2) Before K-means, if we do not use parallel IO, we read the data from root node, then distribute the data to all the nodes; if we use the parallel IO, each node reads m/p samples. Then the algorithm does a gather operation to make the root node have all the data. (3) After the clustering part, the root node gets the redistribution information and distributes the data to all the nodes.

For the parallel IO version of part (2), the reason why we have to gather all the data to the root node is that we use the CSR (Compressed Row Storage) format to store the data to reduce the redundant memory requirement. If we do not use the CSR data format, we can not process high-dimensional data sets like webspam (Webb, Caverlee, and Pu, 2006) in Table 2.8. Because of the CSR format implementation, we must get the global row index and data index on a single node.

For CA-SVM (casvm2 implementation) we use both parallel IO and CSR input format by assuming the sparse input matrix has been prepartitioned into P disjoint row blocks,

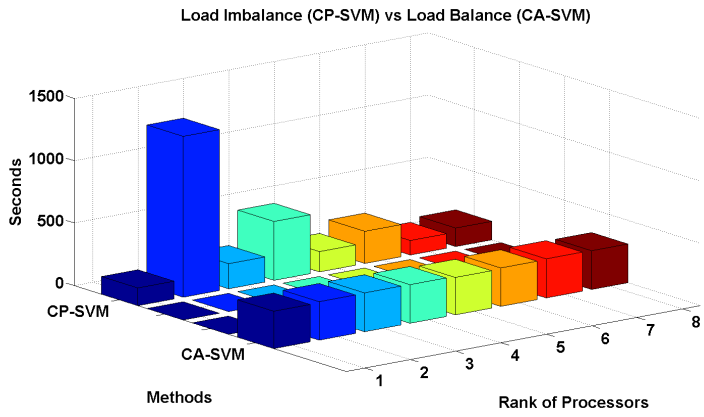


Figure 2.6: The figure shows that CP-SVM is load imbalanced while CA-SVM is load-balanced. The test dataset is *epsilon* with 128,000 samples (2,000 nnz per sample). 8 nodes are used in this test.

each in CSR format. The sub-problems of CA-SVM are independent of each other. Each sub-problem generates its data center (CT_1, CT_2, CT_3, ..., CT_p) and its own model ($model_1$, $model_2$, $model_3$, ..., $model_p$). For an unknown test sample $\hat{x}$, each node will get a copy of $\hat{x}$. Each node computes the distance between $\hat{x}$ and its data center. Let us use $dist_1$, $dist_2$, $dist_3$, ..., $dist_p$ to represent the distances. If $dist_i$ is the smallest one, then we will use $model_i$ to make prediction for $\hat{x}$. It is only necessary to do a reduction operation. In a large datacenter, we expect the user's data to be distributed across different nodes. Considering the load balance issue, we also assume the data should be distributed in a nearly balanced way. Note that even if it is not balanced, CA-SVM can still work (in a slightly inefficient way). On the other hand, from Fig. 2.8, we can observe that the performance of casvm1 (serial IO) is close to the performance of casvm2 (parallel IO).

Algorithm 5 CA-SVM (casvm2 in Fig. 2.8)

Training Process (no communication):
0: $i \in \{1, 2, ..., m/P\}, j \in \{1, 2, ..., P\}$
1: For node N_j, input the samples X_i and labels y_i.
2: For node N_j, get its data center CT_j.
3: For node N_j, launch a SVM training process SVM_j.
4: For node N_j, save the model file of converged SVM_j as MF_j.
Prediction Process (little communication):
1: On j-th node, $d_j = \text{dist}(\hat{X}, CT_j)$
2: Global reduce: $id = argmin_j(d_j)$
3 If rank == id, then use MF_j to make prediction for $\hat{X}$

Communication Pattern

Communication Modeling

We only give the results because the space is limited, the detailed proofs are in (You, Demmel, Kent Czechowski, L. Song, and Richard Vuduc, 2015). The formulas for communication volume are in Table 2.6. The experimental results in the table are based on the dense ijcnn dataset on 8 Hopper nodes. The terms used in the formulas are in Table 2.2. We can use the formulas to predict the communication volume for a given method. For example, for ijcnn dataset, m is 48,000, n is 13, and s is 4474. We can predict the communication volume of Cascade is about $3 \times (48000 \times 13 + 48000 + 4474 \times 13) \times 4\text{B} = 8.4\text{MB}$. Our experimental result is 8.41MB, which means the prediction for Cascade is very close to the actual volume.

Table 2.6: Modeling of Communication Volume based on a subset of ijcnn (Prokhorov, 2001)

Method	Formula	Prediction	Test
Dis-SMO	$\Theta(26IP + 2Pm + 4mn)$	36MB	34MB
Cascade	$O(3mn + 3m + 3sn)$	8.4MB	8.4MB
DC-SVM	$\Theta(9mn + 12m + 2kPn)$	24MB	29MB
DC-Filter	$O(6mn + 7m + 3sn + 2kPn)$	16.2MB	18MB
CP-SVM	$\Theta(6mn + 7m + 2kPn)$	15.6MB	17MB
CA-SVM	0	0MB	0MB

Point-to-Point profiling

Fig. 2.7 shows the communication patterns of these six approaches for a subset of ijcnn. To improve the efficiency of communication, we use as many collective communications as possible because a single collective operation is more efficient than multiple send/receive operations. Due to the communications of K-means, DC-Filter and CP-SVM have to transfer more data than Cascade. However, from Table 2.7 we can observe that CP-SVM is more efficient than Cascade since the volume of communication per operation is higher.

Ratio of Communication to Computation

Fig. 2.8 shows the communication and computation time for different methods applied to a subset of ijcnn. From Fig. 2.8 we can observe that our algorithms significantly reduce the volume of communication and the ratio of communication to computation. This is important since the existing supercomputers (Dongarra, 2014) are generally more suitable for computation-intensive than communication-intensive applications. Besides, less communication can greatly reduce the power consumption (Demmel et al., 2013). Table 2.6 shows that the communication volumes of DC-Filter and CP-SVM are similar. However, Fig. 2.8

shows that there is a big difference between DC-Filter communication time and CP-SVM time. The reason is that the communication of CP-SVM can be done by collective operations (e.g. Scatter) but DC-Filter has some point-to-point communications (e.g. Send/Recv) on the lower levels (Fig. 2.1).

Table 2.7: Efficiency of Communication based on a subset of IJCNN (Prokhorov, 2001)

Method	Volume	Comm Operations	Volume/Operation
Dis-SMO	34MB	335,186	101B
Cascade	8MB	56	150,200B
DC-SVM	29MB	80	360,734B
DC-Filter	18MB	80	220,449B
CP-SVM	17MB	24	709,644B
CA-SVM	0MB	0	N/A

2.5 Experimental Results and Analysis

The test datasets in our experiments are shown in Table 2.8, and they are from real-world applications. Some of the datasets are sparse, we use CSR format in our implementation for all the datasets. We use MPI for distributed processing, OpenMP for multi-threading, and Intel Intrinsics for SIMD parallelism. To give a fair comparison, all the methods in this chapter are based on the same shared-memory SMO implementation (You, Shuaiwen Song, et al., 2014). The K-means partitioning in DC-SVM, DC-Filter, CP-SVM, and BKM are distributed versions, which achieved the same partitioning result and comparable performance with Liao's implementation (Liao, 2013). Our experiments are conducted on NERSC Hopper and Edison systems (NERSC, 2016).

Speedup and Accuracy

From Tables 2.9 to 2.14, we observe that CA-SVM can achieve 3× - 16× (7× on average) speedups over distributed SMO algorithm with comparable accuracies. The Init time includes the partition time like K-means, and the Training time includes the redistribution and the SVM training processes. For Cascade, DC-SVM, and DC-Filter, the training process includes the level-by-level (point-to-point) communications. The accuracy loss ranges from none to 3.6% (1.3% on average). According to previous work (E. Y. Chang, 2011), the accuracy loss in this chapter is small and tolerable for practical applications. Additionally, we can observe that CA-SVM reduces the number of iterations, which means it is intrinsically more efficient than other algorithms. For DC-SVM, DC-Filter, CP-SVM, and BKM the majority of the Init time is spent on K-means clustering. K-means itself does not significantly increase

Table 2.8: The Test Datasets

Dataset	Application Field	#samples	#features
adult (John C Platt, 1999)	Economy	32,561	123
epsilon (Sonnenburg et al., 2008)	Character Recognition	400,000	2,000
face (Tsang, Kwok, and Zurada, 2006)	Face Detection	489,410	361
gisette (Isabelle Guyon et al., 2004)	Computer Vision	6,000	5,000
ijcnn (Prokhorov, 2001)	Text Decoding	49,990	22
usps (Hull, 1994)	Transportation	266,079	675
webspam (Webb, Caverlee, and Pu, 2006)	Management	350,000	16,609,143

Table 2.9: adult dataset on Hopper (K-means converged in 8 loops)

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	84.3%	8,054	5.64s (0.006, 5.64)
Cascade	83.6%	1,323	1.05s (0.007, 1.04)
DC-SVM	83.7%	8,699	17.1s (0.042, 17.1)
DC-Filter	84.4%	3,317	2.23s (0.042, 2.18)
CP-SVM	83.0%	2,497	1.66s (0.041, 1.59)
BKM-SVM	83.3%	1,482	1.61s (0.057, 1.54)
FCFS-SVM	83.6%	1,621	1.21s (0.005, 1.19)
CA-SVM	83.1%	1,160	0.96s (4e-4, 0.95)

the computation or communication cost. However, we need to redistribute the data after K-means, which increases the communication cost.

Strong Scaling and Weak Scaling

Tables 2.15 and 2.16 show the results of strong scaling time and efficiency. We observe that the strong scaling efficiency of CA-SVM is increasing with the number of processors. The reason is that the number of iterations is decreasing since the number of samples (m/P) on each node is decreasing. The single iteration time is also reduced with fewer samples on each node. For the weak scaling results in Tables 2.17 and 2.18, we observe that all the efficiencies of these six algorithms are decreasing with the increasing number of processors. In theory, the work load of CA-SVM on each node is constant with the increasing number processors. However, in practice, the system overhead is higher with more processors. The weak scaling efficiency of CA-SVM only decreases 4.7% with a 16× increase in the number of processors.

Table 2.10: face dataset on Hopper (K-means converged in 29 loops)

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.0%	17,501	358s (2e-4, 358)
Cascade	98.0%	2,274	67.0s (0.10, 66.9)
DC-SVM	98.0%	20,331	445s (13.6, 431)
DC-Filter	98.0%	13,999	314s (13.6, 297)
CP-SVM	98.0%	13,993	311s (13.6, 295)
BKM-SVM	98.0%	2,209	88.9s (17.8, 71.0)
FCFS-SVM	98.0%	2,194	65.3s (0.43, 64.9)
CA-SVM	98.0%	2,268	66.4s (0.08, 66.4)

Table 2.11: gisette dataset on Hopper (K-means converged in 31 loops)

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	97.6%	1,959	8.1s (0.26, 7.86)
Cascade	88.3%	1,520	15.9s (0.20, 15.7)
DC-SVM	90.9%	4,689	130.7s (2.35, 127.9)
DC-Filter	85.7%	1,814	20.1s (2.39, 17.2)
CP-SVM	95.8%	521	8.30s (2.30, 5.4)
BKM-SVM	95.8%	452	4.75s (2.29, 2.46)
FCFS-SVM	96.5%	441	2.48s (0.07, 2.41)
CA-SVM	94.0%	487	2.9s (0.014, 2.87)

Efficiency of CA-SVM

Here, we use m for simplicity to refer to the problem size and P to refer to the number of nodes. To be more precise, let $t(m, P)$ be the per-iteration time, which is a function of m and P; and let $i(m, P)$ be the number of iterations, a function of m and P. For Dis-SMO, we observe $i(m, P) = \Theta(m)$, that is, there is no actual dependence on P. Then, the total time should really be

$$T(m, P) = i(m, P) \times t(m, P)$$

Thus, the efficiency becomes

$$E(m, P) = \frac{i(m, 1) \times t(m, 1)}{P \times i(m, P) \times t(m, P)}$$

Table 2.12: ijcnn dataset on Hopper (K-means converged in 7 loops)

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.7%	30,297	23.8s (0.008, 23.8)
Cascade	95.5%	37,789	13.5s (0.007, 13.5)
DC-SVM	98.3%	31,238	59.8s (0.04, 59.7)
DC-Filter	95.8%	17,339	8.4s (0.04, 8.3)
CP-SVM	98.7%	7,915	6.5s (0.04, 6.4)
BKM-SVM	98.3%	5,004	3.0s (0.08, 2.87)
FCFS-SVM	98.5%	7,450	3.6s (0.005, 3.55)
CA-SVM	98.0%	6,110	3.4s (3e-4, 3.4)

Table 2.13: usps dataset on Edison (K-means converged in 28 loops)

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	99.2%	47,214	65.9s (2e-4, 65.9)
Cascade	98.7%	132,503	969s (0.008, 969)
DC-SVM	98.7%	83,023	1889s (1.5, 1887)
DC-Filter	99.6%	67,880	242s (1.5, 240)
CP-SVM	98.9%	7,247	35.7s (1.5, 33.9)
BKM-SVM	98.9%	6,122	30.4s (2.02, 28.4)
FCFS-SVM	99.0%	6,513	30.1s (0.04, 29.7)
CA-SVM	98.9%	6,435	24.5s (0.0018, 24.5)

For Dis-SMO, $i(m,1) = i(m,P)$, which means

$$E(m,P) = \frac{t(m,1)}{P \times t(m,P)}$$

If the per-iteration time scales perfectly — meaning $t(m,P) = t(m,1)/P$ — the efficiency of SMO should be $E(m,P) = 1$ in theory. For CA-SVM, each node is actually an independent SVM. Thus we expect that $i(m,P) = \Theta(m/P)$ because each node only trains m/P samples. In other words, each node is a SMO problem with m/P samples. Therefore, $i(m,1)$ is close to $P \times i(m,P)$, which means

$$E(m,P) = \frac{t(m,1)}{t(m,P)}$$

On the other hand $t(m,1)$ is close to $P \times t(m,P)$ because each node only has m/P

Table 2.14: webspam dataset on Hopper (K-means converged in 38 loops)

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.9%	164,465	269.1s (0.05, 269.0)
Cascade	96.3%	655,808	2944s (0.003, 2944)
DC-SVM	97.6%	229,905	3093s (0.95, 3092)
DC-Filter	97.2%	108,980	345s (1.0, 345)
CP-SVM	98.7%	14,744	41.8s (1.0, 40.7)
BKM-SVM	98.5%	14,208	24.3s (1.12, 23.0)
FCFS-SVM	98.3%	12,369	21.2s (0.03, 21.0)
CA-SVM	96.9%	10,430	17.3s (0.003, 17.3)

Table 2.15: Strong Scaling Time for epsilon dataset on Hopper: 128k samples, 2k nnz per sample

Processors	96	192	384	768	1536
Dis-SMO	2067s	1135s	777s	326s	183s
Cascade	1207s	376s	154s	76.1s	165s
DC-SVM	11841s	8515s	4461s	3909s	3547s
DC-Filter	2473s	1517s	1100s	1519s	1879s
CP-SVM	2248s	1332s	877s	546s	202s
BKM-SVM	1031s	355s	137s	88.6s	48.4s
FCFS-SVM	1064s	303s	85.8s	25.4s	15.6s
CA-SVM	1095s	313s	86s	23s	6s

samples. In this way, we get

$$E(m, P) = \frac{P \times t(m, P)}{t(m, P)} = P$$

This means the efficiency of CA-SVM is close to P in theory. Usually, we expect efficiency to lie between 0 and 1. The way we set this up is perhaps not quite right – the sequential baseline should be the best sequential baseline, not the naive (plain SMO) one. If we execute CA-SVM sequentially by simulating P nodes with only 1 node, then the sequential time would be

$$P \times (i(m, 1)/P \times t(m, 1)/P) = i(m, 1) \times t(m, 1)/P$$

Table 2.16: Strong Scaling Efficiency for epsilon dataset on Hopper: 128k samples, 2k nnz per sample

Processors	96	192	384	768	1536
Dis-SMO	100%	91.1%	66.5%	79.2%	70.4%
Cascade	100%	160.5%	195.4%	198.4%	45.7%
DC-SVM	100%	69.5%	66.4%	37.9%	20.9%
DC-Filter	100%	81.5%	56.2%	20.3%	8.2%
CP-SVM	100%	84.4%	64.1%	51.4%	69.7%
BKM-SVM	100%	145.2%	188.1%	145.5%	133.1%
FCFS-SVM	100%	175.6%	310.0%	523.6%	426.3%
CA-SVM	100%	175.0%	319.5%	603.0%	1068.7%

Table 2.17: Weak Scaling Time for epsilon dataset on Hopper: 2k samples per node, 2k nnz per sample

Processors	96	192	384	768	1536
Dis-SMO	14.4s	27.9s	51.3s	94.8s	183s
Cascade	7.9s	8.5s	11.9s	52.9s	165s
DC-SVM	17.8s	67.9s	247s	1002s	3547s
DC-Filter	16.8s	51.2s	181s	593s	1879s
CP-SVM	13.8s	36.1s	86.8s	165s	202s
BKM-SVM	6.72s	9.14s	16.6s	31.2s	48.4s
FCFS-SVM	6.14s	6.71s	6.88s	10.2s	15.6s
CA-SVM	6.1s	6.2s	6.2s	6.4s	6.4s

So then $E(m, P)$ would approach 1 rather than P. Put another way, CA-SVM is better than SMO, even in the sequential case. That is, we can beat SMO by running CA-SVM to simulate P nodes using only 1 node.

The Approximation Accuracy

The Mathematical Derivation

The intuition behind the divide-and-conquer heuristic is this: Suppose we can partition (say by K-means) the m training samples into p disjoint clusters $D_1 \cup D_2 \cup \cdots \cup D_p$, where the samples in each D_i are close together, and far from other D_j. Then classifying a new sample

Table 2.18: Weak Scaling Efficiency for epsilon dataset on Hopper: 2k samples per node, 2k nnz per sample

Processors	96	192	384	768	1536
Dis-SMO	100%	51.7%	28.2%	15.2%	7.9%
Cascade	100%	93.2%	66.2%	14.9%	4.8%
DC-SVM	100%	26.3%	7.2%	1.8%	0.5%
DC-Filter	100%	32.8%	9.3%	2.8%	0.9%
CP-SVM	100%	38.2%	15.9%	8.3%	6.8%
BKM-SVM	100%	73.5%	40.5%	21.5%	13.9%
FCFS-SVM	100%	91.5%	89.2%	60.2%	39.4%
CA-SVM	100%	98.9%	97.8%	96.0%	95.3%

$\hat{X}$ may be done by (1) finding the cluster D_i to which $\hat{X}$ is closest, and (2) using the samples inside D_i to classify $\hat{X}$ (say by an SVM using only D_i as training data). Since we use nearby data to classify $\hat{X}$, we expect this to work well in many situations.

In this section we quantify this observation as follows: Let K be the m-by-m kernel matrix, with $K_{i,j} = K(X_i, X_j)$, permuted so that the first $|D_1|$ indices are in D_1, the next $|D_2|$ indices are in D_2, etc. Let K_1 be the leading $|D_1|$-by-$|D_1|$ diagonal submatrix of K, K_2 the next $|D_2|$-by-$|D_2|$ diagonal submatrix, etc. Let $\widetilde{K} = \text{diag}(K_1, K_2, ..., K_p)$ be the submatrix of K consisting just of these diagonal blocks. Then we may ask how well $\widetilde{K}$ "approximates" K. In the extreme case, when $\widetilde{K} = K$ and all samples in each cluster have the same classification, it is natural to assign $\hat{X}$ the same classification as its closest cluster. As we will see, depending both on the kernel function $K()$ and our metric for how we measure how well $\widetilde{K}$ approximates K, our algorithm for finding clusters D_i will naturally improve the approximation. One can also view choosing D_i as a graph partitioning problem, where $K_{i,j}$ is the weight of edge (i, j) (Shi and Malik, 2000)(Von Luxburg, 2007)(Si Si, Cho-Jui Hsieh, and I. Dhillon, 2014).

In the simple case of a linear kernel $K(X_i, X_j) = X_i^T \cdot X_j$, a natural metric to try to maximize (inspired by (Shi and Malik, 2000)) is

$$J_1 = \sum_{k=1}^{p} |D_k|^{-1} \sum_{i,j \in D_k} K_{i,j}$$

Letting $X = [X_1, ..., X_m]$, it is straightforward to show that

$$\|X\|_F^2 - J_1 = \sum_{k=1}^{p} \sum_{i \in D_k} \|X_i - \mu_k\|_2^2 \equiv J^{kmeans}$$

where $\mu_k = |D_k|^{-1} \sum_{i \in D_k} X_i$ is the mean of cluster D_k. It is also known that the goal of K-means is to choose clusters to minimize the objective function J^{kmeans}, i.e. to maximize J_1. Since the polynomial and sigmoid kernels are also increasing functions of $X_i^T \cdot X_j$, we also

expect K-means to choose a good block diagonal approximation for them.

Now we consider the Gaussian kernel, or more generally kernels for which $K(X_i, X_j) = f(\|X_i - X_j\|_2)$ for some function $f()$. (The argument below may also be generalized to shift-invariant kernels $K(X_i, X_j) = f(X_i - X_j)$.) Now we use the metric

$$J_2 = \sum_{k=1}^{p} |D_k|^{-1} \sum_{i,j \in D_k} K_{i,j}^2$$

to measure how well $\widetilde{K}$ approximates K, and again relate minimizing J^{kmeans} to maximizing J_2.

The mean value theorem tells us that $K(X_i, X_j) = f(\|X_i - X_j\|_2) = f(0) + f'(s)\|X_i - X_j\|_2$ for some $s \in [0, \|X_i - X_j\|_2]$. For the Gaussian Kernel $f(s) = e^{-\gamma s^2}$ so $f'(s) = -2\gamma s e^{-\gamma s^2}$, which lies in the range $0 > f'(s) \geq -\sqrt{2\gamma} e^{-1/2} \equiv R$. Thus $1 \geq K(X_i, X_j) \geq 1 + R\|X_i - X_j\|_2$, which in turn implies $1 \leq (K(X_i, X_j) - R\|X_i - X_j\|^2)^2 \leq 2K^2(X_i, X_j) + 2R^2\|X_i - X_j\|_2^2$ or $K^2(X_i, X_j) \geq \frac{1}{2} - R^2\|X_i - X_j\|_2^2$. Substituting this into the above expression for J_2 and simplifying we get

$$\begin{aligned} J_2 &\geq \frac{m}{2} - R^2 \sum_{k=1}^{p} |D_k|^{-1} \sum_{i,j \in D_k} \|X_i - X_j\|_2^2 \\ &= \frac{m}{2} - 2R^2 J^{kmeans} \end{aligned}$$

So again minimizing J^{kmeans} means maximizing (a lower bound for) J_2.

The Block Diagonal Matrix

For the experiment, we use 5,000 samples from the UCI covtype dataset (Blackard, D. Dean, and C. Anderson, 1998). The kernel matrix is 5,000-by-5,000 with 458,222 nonzeroes. In Fig. 2.9, the first part is the original kernel matrix, the second part is the kernel matrix after clustering. From these figures we can observe that the kernel matrix is block-diagonal after clustering. Let us use Fn to represent the Frobenius norm (F-norm) and $\widehat{Fn}$ means the F-norm of the original kernel matrix. The definition of Error (kernel approximation error) is given by

$$Error = \frac{|\widehat{Fn} - Fn|}{\widehat{Fn}} \tag{2.14}$$

The γ in Table 2.19 is defined in the Gaussian kernel of Table 2.1. From Table 2.19 we can observe that when γ is small, the approximation error of Random method is much larger than the approximation error of Clustering method. Based on F-norm, the approximation matrix by clustering method is almost the same with the original matrix. The approximation error of Random partition is decreasing when the γ is increasing. There, if we can use large γ parameter in the real-world applications, the approximation error of Random partition (i.e. CA-SVM) can be very low.

Table 2.19: The error of different kernel approximations. The definition of Error is in Equation (2.14).

γ	Original F-norm / Error	Random F-norm / Error	Clustering F-norm / Error
20	154.239899 / 0.0%	98.661888 / 36.0%	154.279709 / 0.0%
30	117.745422 / 0.0%	85.005592 / 27.8%	117.765900 / 0.0%
40	100.739906 / 0.0%	79.307716 / 21.3%	100.755119 / 0.0%
50	91.576241 / 0.0%	76.469208 / 16.5%	91.585464 / 0.0%
60	86.123299 / 0.0%	74.869514 / 13.1%	86.129494/ 0.0%
70	82.630066 / 0.0%	73.882416 / 10.6%	82.635559/ 0.0%

Tradeoffs

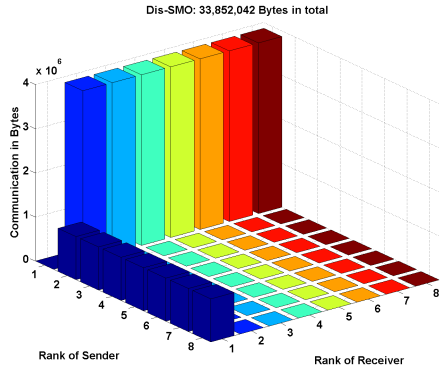
CA-SVM is the only algorithm presented that can achieve nearly zero communication. Thus, CA-SVM should be the fastest one in general. However, CA-SVM also suffers the most loss in accuracy, if surprisingly little. Basically, our methods are for applications that most need to be accelerated or scaled up, and do not require the highest accuracy. For applications that require both accuracy and speed, using FCFS or BKM is a better choice. So there is a trade-off between time (communication) and accuracy.

Accuracy of CA-SVM

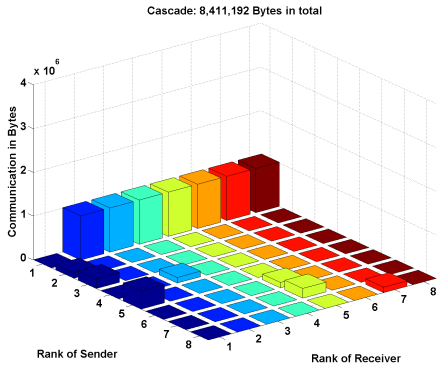
We conduct the following two experiments: (1) **Random-Assign**: assign the test sample not to the processor with the closest data center, but to a random processor. (2) **Sub-Sampling**: pick random subset of 1/p-th of all the data, and just use it to build an SVM for all the test samples. Let use the ijcnn dataset (Table 2.8) as an example. We use 8 nodes and divide the dataset into 8 parts for CA-SVM. After the experiment, the accuracy of **Random-Assign** is 85% (77987/91701), the accuracy of **Sub-Sampling** is 68% (62340/91701), and the accuracy of **CA-SVM** is 98% (89852/91701). **Sub-Sampling** has the lowest accuracy because it uses a much smaller dataset (6k training samples) and thus can only build an inferior model. The difference between **Random-Assign** and **CA-SVM** is that each model of **Random-Assign** roughly receives the same number of test samples because it used the random assignment method. However, a test sample of **CA-SVM** will be sent to its closest cluster rather than a random cluster. This makes different nodes of **CA-SVM** have different numbers of test samples. For example, the 0-th node of **Random-Assign** receives 11,518 test samples while the 0-th node of **CA-SVM** only receives 5,037 test samples.

2.6 Conclusion

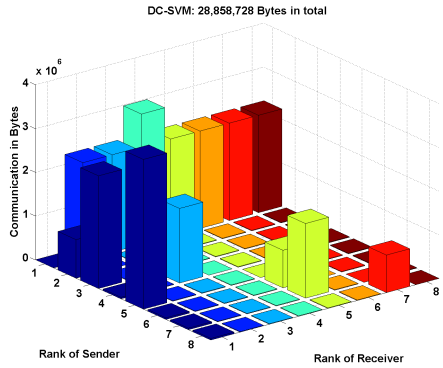
Existing distributed SVM approaches like Dis-SMO, Cascade, and DC-SVM suffer from intensive communication, computation inefficiency and bad scaling. In this chapter, we design and implement five efficient approaches (i.e. DC-Filter, CP-SVM, BKM, FCFS, and CA-SVM) through step-by-step optimizations. BKM, FCFS, and CA-SVM all reduce communication significantly compared to previous methods, with CA-SVM avoiding all communication. We manage to obtain a perfect load-balancing, and achieve 7× average speedup with only 1.3% average loss in accuracy for six real-world application datasets. Because of faster iteration and reduced number of iterations, CA-SVM can achieve 1068.7% strong scaling when we increase the number of processors from 96 to 1536. Thanks to the removal of communication overhead, CA-SVM attains a 95.3% weak scaling from 96 to 1536 processors. The results justify that the approaches proposed in this chapter can be used in large-scale applications.



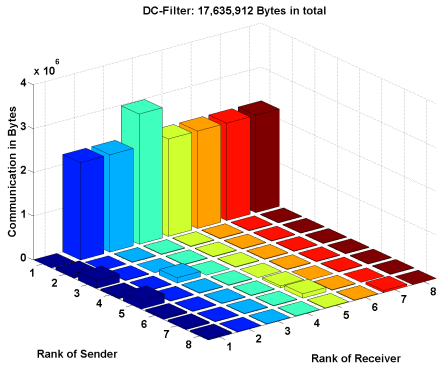
2.7.1 Dis-SMO: 34MB



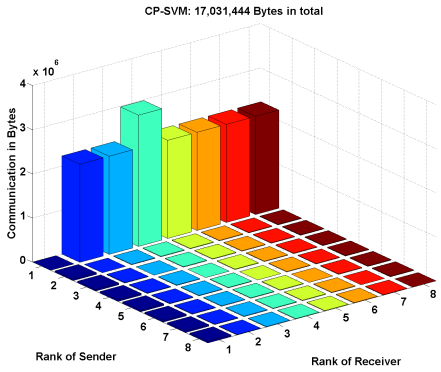
2.7.2 Cascade: 8MB



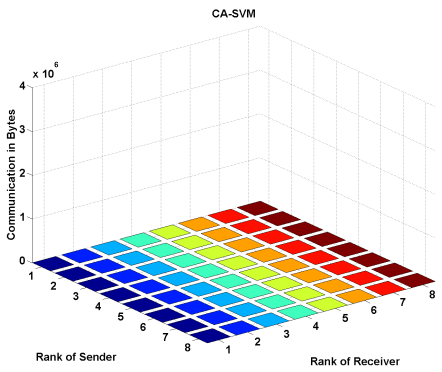
2.7.3 DC-SVM: 29MB



2.7.4 DC-Filter: 18MB



2.7.5 CP-SVM: 17MB



2.7.6 CA-SVM: 0MB

Figure 2.7: Communication Patterns of different approaches. The data is from running the 6 approaches on 8 nodes with the same 5MB real-world dataset (subset of ijcnn dataset). x-axis is the rank of sending processors, y-axis is the rank of receiving processors, and z-axis is the volume of communication in bytes. The vertical ranges (z-axis) of these 6 sub-figures are the same. The communication pattern of BKM-SVM is similar to that of CP-SVM. The communication pattern of FCFS-SVM is similar to that of cascade without point-to-point communication.

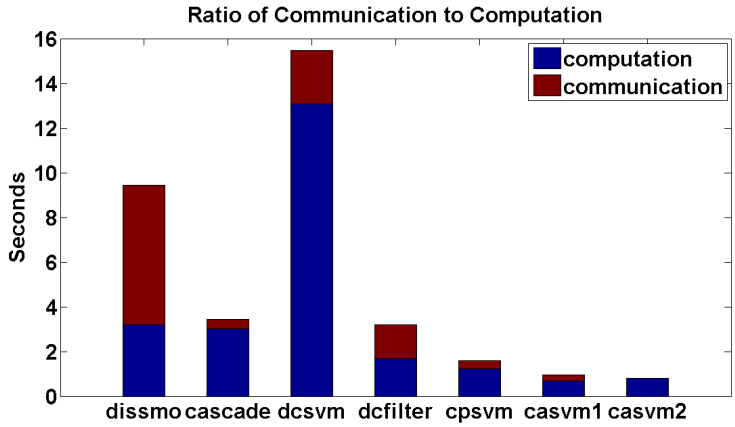
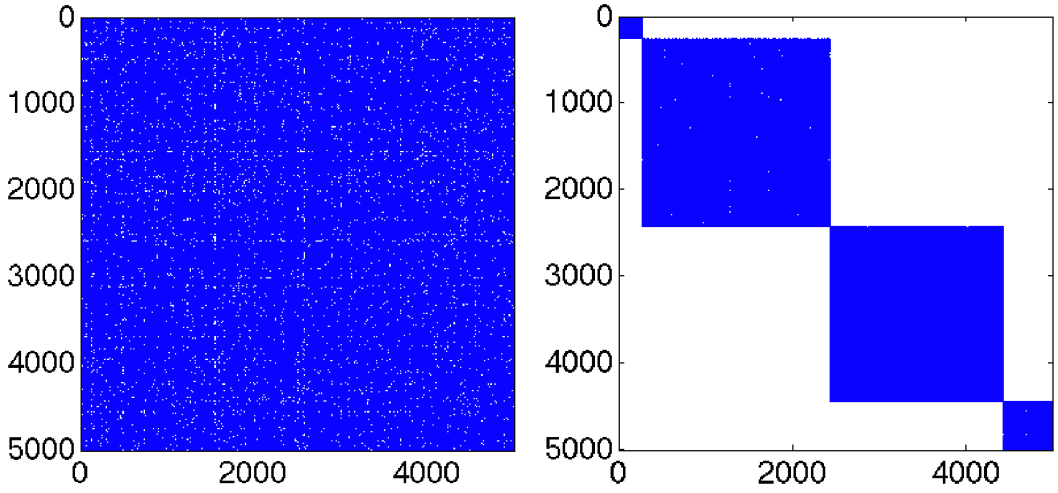


Figure 2.8: The ratio of computation to communication. The experiment is based on a subset of ijcnn dataset. To give a fair comparison, we implemented two versions of CA-SVM. **casvm1** means that we put the initial dataset on the same node, which needs communication to distribute the dataset to different nodes. **casvm2** means that we put the initial dataset on different nodes, which needs no communication.



2.9.1 Original Kernel Matrix

2.9.2 Kernel Matrix After Clustering

Figure 2.9: We use the 5,000 samples from the UCI covtype dataset (Blackard, D. Dean, and C. Anderson, 1998) for this experiment. The kernel matrix is 5,000-by-5,000 with 458,222 nonzeroes. The first figure is the original kernel matrix, the second figure is the kernel matrix after clustering. From these figures we can observe that the kernel matrix is block-diagonal after the clustering algorithm.

Chapter 3

Communication-Avoiding Kernel Ridge Regression

3.1 Introduction

Learning non-linear relationships between predictor variables and responses is a fundamental problem in machine learning (Barndorff-Nielsen and Shephard, 2004), (Bertin-Mahieux et al., 2011). One state-of-the-art method is Kernel Ridge Regression (KRR) (Y. Zhang, J. Duchi, and Wainwright, 2013), which we target in this chapter. It combines ridge regression, a method to address ill-posedness and overfitting in the standard regression setting via L2 regularization, with kernel techniques, which adds flexible support for capturing non-linearity.

Computationally, the input of KRR is an n-by-d data matrix with n training samples and d features, where typically $n \gg d$. At training time, KRR needs to solve a large linear system $(K + \lambda nI)\alpha = y$ where K is an n-by-n matrix, α and y are n-by-1 vectors, and λ is a scalar. Forming and solving this linear system is the major bottleneck of KRR. For example, even on a relatively small dataset—357 megabytes (MB) for a 520,000-by-90 (Bertin-Mahieux et al., 2011)—KRR needs to form a 2 terabyte dense kernel matrix. A standard distributed parallel dense linear solver on a p-processor system will require $\Theta(n^3/p)$ arithmetic operations per processor; we refer to this approach hereafter as Distributed KRR (DKRR). In machine learning, where weak scaling is of primary interest, DKRR fares poorly: keeping n/p fixed, the total storage grows as $\Theta(p)$ per processor and the total flops as $\Theta(p^2)$ per processor. In the perfect weak scaling situation, both the memory needed and the flops grow as $\Theta(1)$ per processor (i.e. memory and flops are constant). In one experiment, the weak scalability of DKRR dropping from 100% to 0.32% as p increased from 96 to 1536 processors.

Divide-and-Conquer KRR (DC-KRR) (Y. Zhang, J. Duchi, and Wainwright, 2013), addresses the scaling problems of DKRR. Its main idea is to randomly shuffle the rows of the n-by-d data matrix and then partition it to p different n/p-by-d matrices, one per machine, leading to an n/p-by-n/p kernel matrix on each machine; it builds local KRR models that are then reduced globally to obtain the final model. DC-KRR reduced the memory and computational requirement. However, it can't be used in practice because it sacrifices accuracy.

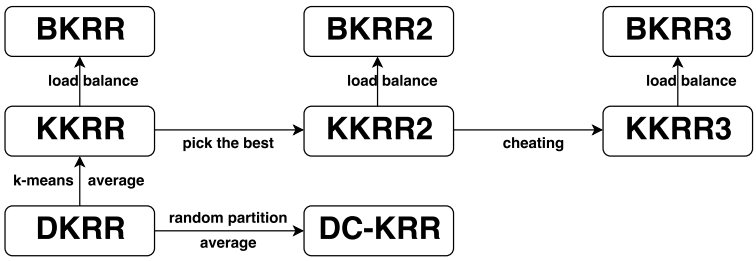


Figure 3.1: Optimization flow of our algorithm. DKRR is the baseline. DC-KRR is the existing method. All the others are the new approaches proposed in this chapter.

For example, on the Million Song Dataset (a recommendation system application) with 2k test samples, the mean squared error (MSE) decreased only from 88.9 to 81.0 as the number of training samples increased from 8k to 128k. We double the number of processors as we double the number of samples. This is a bad weak scaling problem in terms of accuracy. By contrast, the MSE of DKRR decreased from 91 to 0.002, which is substantially better.

The idea of DC-KRR is to partition the dataset into p **similar** parts and generate p **similar** models, and then **average** these p models to get the final solution. We seek other ways to partition the problem that scale as well as DC-KRR while improving its accuracy. Our idea is to partition the input dataset into p **different** parts and generate p **different** models, from which we then select the **best** model among them. Further addressing particular communication overheads, we obtain two new methods, which we call Balanced KRR (BKRR) and K-means KRR (KKRR). Figure 3.1 is the summary of our optimization flow. We proposed a series of approaches with details explained later (KKRR3 is an impractical algorithm used later to bound the attainable accuracy). Figure 3.2 shows the fundamental trade-off between accuracy and scaling for these approaches. Among them, we recommend BKRR2 (optimized version of BKRR) and KKRR2 (optimized version of KKRR) to use in practice. BKRR2 is optimized for scaling and has good accuracy. KKRR2 is optimized for accuracy and has good speed.

When we increase the number of samples from 8k to 128k, KKRR2 (optimized version of KKRR) reduces the MSE from 95 to 10^{-7}, which addresses the poor accuracy of DC-KRR. In addition, KKRR2 is faster than DC-KRR for a variety of datasets. Our KKRR2 method improves weak scaling efficiency over DKRR from 0.32% to 38% and achieves a 591× speedup over DKRR on the same data at the same accuracy and the hardware (1536 processors). For the applications requiring only approximate solutions, BKRR2 improves the weak scaling efficiency to 92% and achieves 3505× speedup with only a slight loss in accuracy.

This chapter is based on a joint work with James Demmel, Cho-Jui Hsieh, and Richard Vuduc. It was published as a conference paper entitled *Accurate, fast and scalable kernel ridge regression on parallel and distributed systems* (You, Demmel, Cho-Jui Hsieh, et al., 2018).

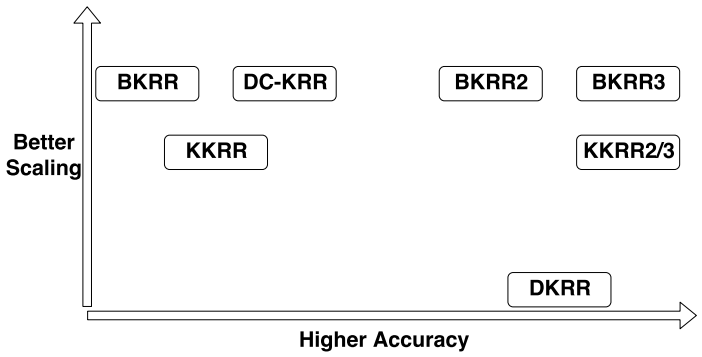


Figure 3.2: Trade-off between accuracy and speed for large-scale weak scaling study. BKRR3 and KKRR3 are the unrealistic approaches. BKRR2 is optimal for scaling and has good accuracy. KKRR2 is optimal for accuracy and has good speed.

3.2 Background

Linear Regression and Ridge Regression

In machine learning, linear regression is a widely-used method for modeling the relationship between a scalar dependent variable y (regressand) and multiple explanatory variables (independent variables) denoted by x, where x is a d-dimensional vector. The training data of a linear regression problem consists of n data points, where each data point (or training sample) is a pair (x_i, y_i) and $1 \leq i \leq n$. Linear regression has two phases: training and prediction. The training phase builds a model from an input set of training samples, while the prediction phase uses the model to predict the unknown regressands $\hat{y}$ of a new data point $\hat{x}$. The training phase is the main limiter to scaling, both with respect to increasing the training set size n and number of processors p. In contrast, prediction is embarrassingly parallel and cheap per data point.

Training Process. Given n training data points $\{(x_i, y_i)\}_{i=1}^{n}$ where each $x_i = (x_i^1, ..., x_i^j, ..., x_i^d)$ and y_i is a scalar, we want to build a model relating a given sample (x_i) to its measured regressand (y_i). For convenience, we define X as an n-by-d matrix with $X_{ij} = x_i^j$ and $y = (y_1, \cdots, y_n)$ be an n-dimensional vector. The goal of regression is to find a w such that $y_i \approx w^T \cdot x_i$. This can be formulated as a least squares problem in Equation (3.1). To solve some ill-posed problems or prevent overfitting, L2 regularization (Schmidt, 2005) is used, as formulated in Equation (3.2), which is called **Ridge Regression**. λ is a positive parameter that controls w: the larger is λ, the smaller is $\|w\|_2$.

$$argmin_w \|Xw - y\|_2^2 \tag{3.1}$$

$$argmin_w \{\|Xw - y\|_2^2 + \lambda\|w\|_2^2\} \tag{3.2}$$

Table 3.1: Standard Kernel Functions

Linear	$\Phi(x_i, x_j) = {x_i}^\top x_j$
Polynomial	$\Phi(x_i, x_j) = (a{x_i}^\top x_j + r)^d$
Gaussian	$\Phi(x_i, x_j) = \exp(-\|\|x_i - x_j\|\|^2/(2\sigma^2))$
Sigmoid	$\Phi(x_i, x_j) = \tanh(a{x_i}^\top x_j + r)$

Prediction Process. Given a new sample $\hat{x}$, we can use the w computed in the training process to predict the regressand of $\hat{x}$ by $\tilde{y} = w^T \cdot \hat{x}$. If we have k test samples $\{\hat{x}_i\}_{i=1}^k$ and their true regressands $\{\hat{y}_i\}_{i=1}^k$, the accuracy of the estimated regressand $\{\tilde{y}_i\}_{i=1}^n$ can be evaluated by MSE (Mean Squared Error) defined in Equation (3.3).

$$MSE = \frac{1}{k}\sum_{i=1}^{k}(\tilde{y}_i - \hat{y}_i)^2 \tag{3.3}$$

Kernel Method

For many real problems, the underlying model cannot be described by a linear function, so linear ridge regression suffers from poor prediction error. In those cases, a common approach is to map samples to a high dimensional space using a nonlinear mapping, and then learn the model in the high dimensional space. Kernel method (Hofmann, Schölkopf, and A. J. Smola, 2008) is a widely used approach to conduct this learning procedure implicitly by defining the kernel function—the similarity of samples in the high dimensional space. The commonly used kernel functions are shown in Table 3.1, and we use the most widely used Gaussian kernel in this chapter.

Kernel Ridge Regression (KRR)

Combining the Kernel method with Ridge Regression yields Kernel Ridge Regression, which is presented in Equations (3.4) and (3.5). The $\|\cdot\|_H$ in Equation (3.4) is a Hilbert space norm (Y. Zhang, J. Duchi, and Wainwright, 2013). Given the n-by-n kernel matrix K, this problem reduces to a linear system defined in Equation (3.6). K is the kernel matrix constructed from training data by $K_{i,j} = \Phi(x_i, x_j)$, y is the input n-by-1 regressand vector corresponding to X, and α is the n-by-1 unknown solution vector.

$$argmin\ \frac{1}{n}\sum_{i=1}^{n}\|f_i - y_i\|_2^2 + \lambda\|f\|_H^2 \tag{3.4}$$

$$f_i = \sum_{j=1}^{n}\alpha_j\Phi(x_j, x_i) \tag{3.5}$$

In the Training phase, the algorithm's goal is to get α by (approximately) solving the linear system in (3.6). In the Prediction phase, the algorithm uses α to predict the regressand of any unknown sample $\hat{x}$ using Equation (3.7). KRR is specified in Algorithm 6. The algorithm searches for the best σ and λ from parameter sets. Thus, in practice, Algorithm 6 is only a

single iteration of KRR because people do not know the best parameters before using the dataset. $|\Lambda| \times |\Sigma|$ is the number of iterations where Λ and Σ are the parameter sets of λ and σ (Gaussian Kernel) respectively. Thus, the computational cost of KRR method is $\Theta(|\Lambda||\Sigma|n^3)$. In a typical case, if $|\Lambda| = 50$ and $|\Sigma| = 50$, then the algorithm needs to finish thousands of iterations. People also use cross-validation technique to select the best parameters, which needs much more time.

$$(K + \lambda nI)\alpha = y \tag{3.6}$$

$$\tilde{y} = \sum_{i=1}^{n} \alpha_i \Phi(x_i, \hat{x}) \tag{3.7}$$

Algorithm 6 Kernel Ridge Regression (KRR)

Input:

n labeled data points (x_i, y_i) for training;
another k labeled data points $(\hat{x}_j, \hat{y}_j)$ for testing;
both x_i and $\hat{x}_j$ are d-dimensional vectors;
$i \in \{1, 2, ..., n\}$, $j \in \{1, 2, ..., k\}$;
tuned parameters λ and σ

Output:

Mean Squared Error (MSE) of prediction

Create a n-by-n kernel matrix K
for $i \in 1 : n$ **do**
  for $j \in 1 : n$ **do**
    $K[i][j] \leftarrow \exp(-||x_i - x_j||^2/(2\sigma^2))$
  end
end
Solve linear equation $(K + \lambda nI)\alpha = y$ for α
for $j \in 1 : k$ **do**
  $\tilde{y}_j \leftarrow \sum_{i=1}^{n} \alpha_i K(x_i, \hat{x}_j)$
end
$MSE \leftarrow \frac{1}{k}\sum_{j=1}^{k}(\tilde{y}_j - \hat{y}_j)^2$

K-means clustering

Here we review K-means clustering algorithm, which will be used in our algorithm. The objective of K-means clustering is to partition a dataset TD into $k \in Z^+$ subsets $(TD_1, TD_2, ..., TD_k)$, using a notion of proximity based on Euclidean distance (Forgy, 1965). The value of k is chosen by the user. Each subset has a center $(CT_1, CT_2, ..., CT_k)$, each of which is a d-dimensional vector. A sample x belongs to TD_i if CT_i is its closest center. K-means is shown in Algorithm 7.

Algorithm 7 Plain K-means Clustering

Input the training samples x_i, $i \in \{1, 2, ..., n\}$
Initialize data centers $CT_1, CT_2, ..., CT_k$ randomly
$\delta \leftarrow 0$
For every $i = 1, \cdots, n$
— $c^i \leftarrow argmin_j ||x_i - CT_j||$
— If c^i has been changed, $\delta \leftarrow \delta + 1$
End For
For every $j = 1, \cdots, k$
— $CT_j \leftarrow \frac{\sum_{i=1}^{n} 1\{c^i=j\} x_i}{\sum_{i=1}^{n} 1\{c^i=j\}}$
End For
If $\delta/n >$ threshold, then repeat

3.3 Existing Methods

Distributed KRR (DKRR)

The bottleneck of KRR is solving the n-by-n linear system (3.6), which is generated by a much smaller n-by-d input matrix with $n \gg d$. As stated before, this makes weak-scaling problematic, because memory-per-machine grows like $\Theta(p)$, and flops-per-processor grows like $\Theta(p^2)$. In the perfect weak scaling situation, both the memory needed and the flops grow as $\Theta(1)$ per processor (i.e. memory and flops are constant). Our experiments show the weak scaling efficiency of DKRR decreases from 100% to 0.3% as we increase the number of processors from 96 to 1536. Since the n-by-n matrix K cannot be created on a single node, we create it in a distributed way on a $\sqrt{p}$-by-$\sqrt{p}$ machine grid (Fig. 3.3). We first divide the sample matrix into $\sqrt{p}$ equal parts by rows. To generate $\frac{1}{p}$ of the kernel matrix, each machine will need two of these $\sqrt{p}$ parts of the sample matrix. For example, in Fig. 3.3, to generate the $K(1, 2)$ block, machine 6 needs the second and the third parts of the blocked sample matrix. Thus, we reduce the storage and computation for kernel creation from $\Theta(n^2)$ to $\Theta(n^2/p)$ per machine. Then we use distributed linear solver in ScaLAPACK (Choi et al., 1995) to solve for α.

Divide-and-Conquer KRR (DC-KRR)

DC-KRR (Y. Zhang, J. Duchi, and Wainwright, 2013) showed that using divide-and-conquer can reduce the memory and computational requirement. We first shuffle the original data matrix $M = [X, y]$ by rows. Then we distribute M to all the nodes evenly by rows (lines 1-5 of Algorithm 8). On each node, we construct a much smaller matrix ($\Theta(n^2/p^2)$) than the original kernel matrix ($\Theta(n^2)$) (lines 6-11 of Algorithm 8). After getting the local kernel matrix K, we use it to solve a linear equation $(K + \lambda nI)\alpha = y$ on each node where y is the input labels and α is the solution. After getting α, the training step is complete. Then we use the local α to make predictions for each unknown data point and do a global average

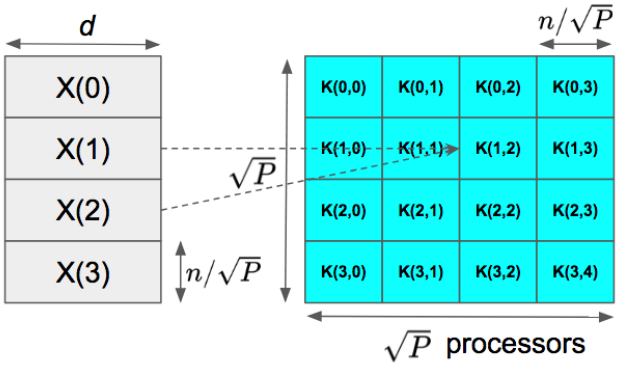


Figure 3.3: Implementation of Distributed KRR. We divide the input sample matrix into $\sqrt{p}$ parts by rows, and each machine gets two of these $\sqrt{p}$ parts. Then each machine generates $1/p$ part of the kernel matrix.

(reduction) to output the final predicted label (lines 13-15 of Algorithm 8). If we get a set of better hyper-parameters, we record them by overwriting the old versions (lines 16-19 of Algorithm 8).

3.4 Accurate, Fast, and Scalable KRR

Kmeans KRR (KKRR)

DC-KRR performs better than state-of-the-art methods (Y. Zhang, J. Duchi, and Wainwright, 2013). However, based on our observation, DC-KRR still needs to be improved. DC-KRR has a poor weak scaling in terms of accuracy, which is the bottleneck for distributed machine learning workloads. We observe that the poor accuracy of DC-KRR is mainly due to the random partitioning of training samples. Thus, our objective is to design a better partitioning method to achieve a higher accuracy and maintain a high scaling efficiency. Our algorithm is accurate, fast, and scalable on distributed systems.

The analysis in this section is based on the Gaussian kernel because it is the most widely used case (Y. Zhang, J. Duchi, and Wainwright, 2013). Other kernels can work in the same way with different distance metrics. For any two training samples, their kernel value ($exp\{-\|x_i - x_j\|^2/(2\sigma^2)\}$) is close to zero when their Euclidean distance ($\|x_i - x_j\|^2$) is large. Therefore, for a given sample $\hat{x}$, only the training points close to $\hat{x}$ in Euclidean distance can have an effect on the result (Equation (3.7)) of the prediction process. Based on this idea, we partition the training dataset into p subsets $(TD_1, TD_2, ..., TD_p)$. We use K-means to partition the dataset since K-means maximizes Euclidean distance between any two clusters. The samples with short Euclidean distance will be clustered into one group. After K-means, each subset will have its data center $(CT_1, CT_2, ..., CT_p)$. Then we launch p

Algorithm 8 Divide-and-Conquer KRR (DC-KRR)

Input:

n labeled data points (x_i, y_i) for training;
k labeled data points $(\hat{x}_j, \hat{y}_j)$ for testing;
x_i and $\hat{x}_j$ are d-dimensional vectors;
$i \in \{1, 2, ..., n\}$, $j \in \{1, 2, ..., k\}$;
$\widehat{MSE} \leftarrow \infty$ (Initial Mean Squared Error);

Output:

Mean Squared Error ($\widehat{MSE}$) of prediction;
best parameters $\hat{\lambda}, \hat{\sigma}$;

if $rank = 0$ **then**
  Store data points (x_i, y_i) $i \in \{1, ..., n\}$ as $[X, y]$
  $M = [X, y]$ is a n-by-$(d+1)$ matrix
  Shuffle M by rows
  Scatter M to all the nodes evenly by rows
end
Create a m-by-m kernel matrix K, $m = n/p$
for $i \in 1 : m$ **do**
  x_i, y_i = M[i][1:d+1]
end
for $\lambda \in \Lambda$ *and* $\sigma \in \Sigma$ **do**
  for $i \in 1 : m$ **do**
    for $j \in 1 : m$ **do**
      $K[i][j] = \Phi(x_i, x_j)$ based on Table 3.1
    end
  end
  Solve linear equation $(K + \lambda m I)\alpha = y$ for α
  for $j \in 1 : k$ **do**
    $\bar{y}_j = \sum_{i=1}^{m} \alpha_i K(x_i, \hat{x}_j)$
  end
  Global reduce: $\tilde{y} = \sum \bar{y}/p$
  if $rank = 0$ **then**
    $MSE = \frac{1}{k} \sum_{j=1}^{k} (\tilde{y}_j - \hat{y}_j)^2$
    if $MSE < \widehat{MSE}$ **then**
      $\widehat{MSE} \leftarrow MSE$, $\widehat{\lambda} \leftarrow \lambda$, $\widehat{\sigma} \leftarrow \sigma$
    end
  end
end

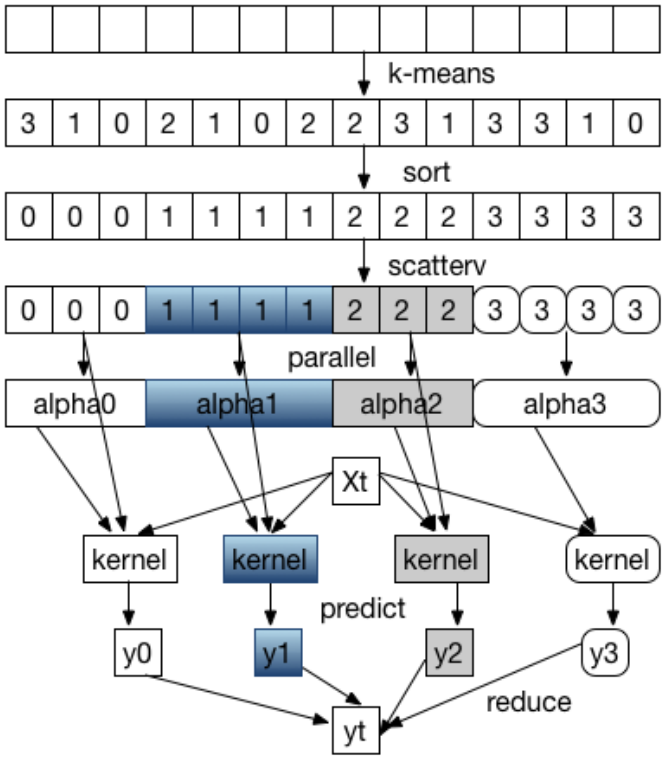
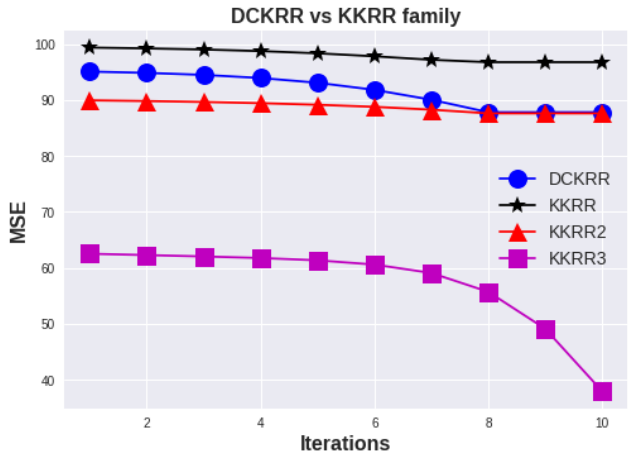


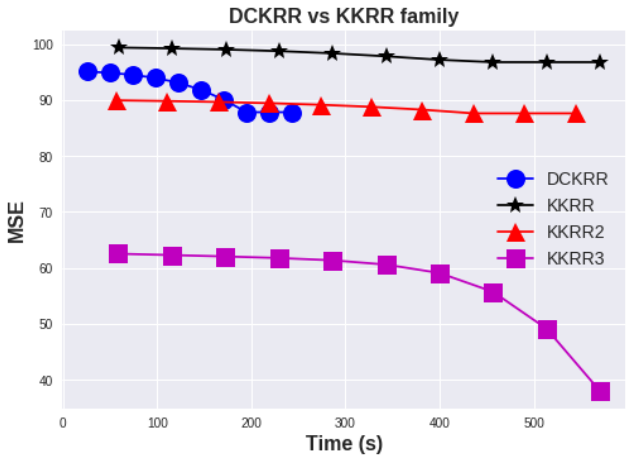
Figure 3.4: Implementation of Kmeans KRR (KKRR). Both k-means and sort can be parallelized. We use the standard MPI implementation for scatter operation.

independent KRRs ($KRR_1, KRR_2, ..., KRR_p$) to process these p subsets.

Given a test sample $\hat{x}$, instead of only using one model to predict $\hat{x}$, we make all the nodes have a copy of $\hat{x}$. This additional cost is trivial for two reasons: (1) the test dataset is much smaller than the training dataset, and the training dataset is much smaller than the kernel matrix, which is the major memory overhead. (2) This broadcast is finished at initial step and there is no need to do it at every iteration. Each node i makes a prediction $\bar{y}_i$ for $\hat{x}$ by using its local model. Then we get the final prediction $\tilde{y}$ by a global reduction (average) over all nodes ($\tilde{y} = \sum_{i=1}^{p} \bar{y}_i/p$). Figure 3.4 is the framework of KKRR. KKRR is highly parallel because all the subproblems are independent of each other. However, the performance of KKRR is not good as we expect. From Figure 3.5.1, we observe that DC-KRR achieves better accuracy than KKRR. In addition, KKRR is slower than DC-KRR. In the following sections, we will make the algorithm more accurate and scalable, which is better than DC-KRR.



3.5.1



3.5.2

Figure 3.5: Comparison between DC-KRR and KKRR family, using same parameter set on 96 NERSC Edison processors. The test data set MSD is described in Section 3.5. KKRR2 is an accurate but slow algorithm. KKRR3 is an optimal but unrealistic method for comparison purposes.

KKRR2

The low accuracy of KKRR is mainly due to its conquer step. For DC-KRR, because the divide step is a random and even partition, the sub-datasets and local models are similar to each other, and thus averaging works pretty well. For KKRR, the clustering method divides the original dataset into different sub-datasets which are far away from each other in Euclidean distance. The local models generated by sub-datasets are also totally different from each other. Thus, using their average will get worse accuracy because some models are

unrelated to the test sample $\hat{x}$. For example, if the data center of i-th partition (CT_i) is far away from $\hat{x}$ in Euclidean distance, the i-th model should not be used to make prediction for $\hat{x}$. Since we divide the original dataset based on Euclidean distance, the similar procedure should be used in the prediction phase. Thus, we design the following algorithm.

After the training process, each sub-KRR will generate its own model file ($MF_1, MF_2, ..., MF_P$). We can use these models independently for prediction. For a given unknown sample $\hat{x}$, if its closest data center (in Euclidean distance) is CT_i, we only use MF_i to make a prediction for $\hat{x}$. We call this version KKRR2. From Figures 3.5.1 and 3.5.2 we observe that KKRR2 is more accurate than DCKRR. However, KKRR2 is slower than DCKRR. For example, to get the same accuracy (MSE=88), KKRR2 is 2.2× (436s vs 195s) slower than DCKRR. Thus we need to focus on speed and scalability.

Balanced KRR (BKRR)

Based on the profiling results in Figure 3.6, we observe that the major source of KKRR's poor efficiency is load imbalance. The reason is that the partitioning by K-means is irregular and imbalanced. For example, processor 2 in Figure 3.6 has to handle n = 35,137 samples while processor 3 only needs to process n = 7,349 samples. Since the memory requirement grows like $\Theta(n^2)$ and the number of flops grows like $\Theta(n^3)$, processor 3 runs 51× faster than processor 1 (Figure 3.6). On the other hand, partitioning by K-means is data-dependent and the sizes of clusters cannot be controlled. This makes it unreliable to be used in practice, and thus we need to replace K-means with a load-balanced partitioning algorithm. To this end, we design a K-balance clustering algorithm and use it to build Balanced Kernel Ridge Regression (BKRR).

In our design, a machine corresponds to a clustering center. If we have p machines, then we partition the training dataset into p parts. As mentioned above, the objective of K-balance partitioning algorithm is to make the number of samples on each node close to n/p. If a data center has n/p samples, then we say it is balanced. The basic idea of this algorithm is to find the closest center for each sample, and if a given data center has been balanced, no additional sample will be added to this center. The detailed K-balance clustering method is in Algorithm 9. Line 1 of Algorithm 9 is an important step because K-balance needs to first run K-means algorithm to get the data centers. This makes K-balance have a similar clustering pattern as K-means. Lines 3-12 find the center for each sample. Lines 6-10 find the best under-load center for the i-th sample. Lines 15-19 recompute the data center by averaging all the samples in a certain center. Recomputing the centers by averaging is optional because it will not necessarily make the results better. From Figure 3.6 we observe that K-balance partitions the dataset in a balanced way. After K-balance clustering, all the nodes have the same number of samples, so in the training phase all the nodes roughly have the same training time and memory requirement.

After replacing K-means with K-balance, KKRR becomes BKRR, and KKRR2 becomes BKRR2. Algorithm 10 is a framework of BKRR2. As we mentioned in the sections of Abstract and Introduction, KKRR2 is the optimized version of KKRR and BKRR2 is the optimized version of BKRR. Lines 1-7 perform the partition. Lines 9-22 perform one iteration of

Algorithm 9 K-balance Clustering

Input:

CT[i] is the center of i-th cluster;
CS[i] is the size of i-th cluster;
SA[i] is the i-th sample;
n is the number of samples;
P is the number of clusters (#machines);

Output:

MB[i] is the closest center to i-th sample;
CT[i] is the center of i-th cluster;

Run K-means to get CT[1], CT[2], ..., CT[p]
$balanced = n/p$
for $i \in 1 : n$ **do**
  $mindis$ = inf
  $minind = 0$
  for $j \in 1 : p$ **do**
    $dist$ = EuclidDistance($SA[i]$, $CT[j]$)
    if $dist < mindis$ ***and*** $CS[j] < balanced$ **then**
      $mindis = dist$
      $minind = j$
    end
  end
  $CS[minind]$++
  $MB[i] = minind$
end
for $i \in 1 : p$ **do**
  $CT[i] = 0$
end
for $i \in 1 : n$ **do**
  $j = MB[i]$
  $CT[j]$ += $SA[i]$
end
for $i \in 1 : p$ **do**
  $CT[i] = CT[i]$ / $CS[i]$
end

Algorithm 10 Balanced KRR2 (BKRR2)

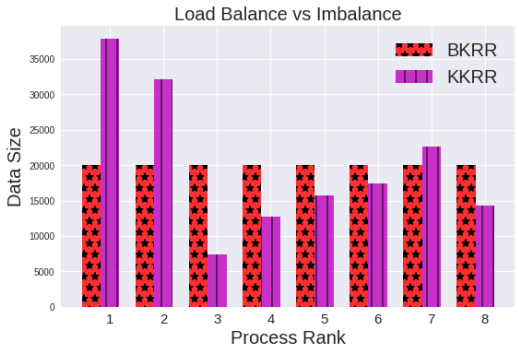
Input:

n labeled data points (x_i, y_i) for training;
k labeled data points $(\hat{x}_j, \hat{y}_j)$ for testing;
both x_i and $\hat{x}_j$ are d dimensional vectors;
$i \in \{1, 2, ..., n\}$, $j \in \{1, 2, ..., k\}$;
$\widehat{MSE} \leftarrow \infty$ (Initial Mean Squared Error);

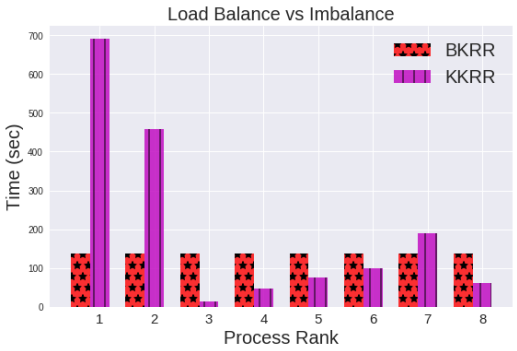
Output:

Mean Squared Error ($\widehat{MSE}$) of prediction
best parameters $\hat{\lambda}, \hat{\sigma}$

$t \leftarrow$ rank of a machine, $t \in \{1, ..., p\}$
Do K-balance clustering (Algorithm 9)
$M = [X, y]$ is a (n/p)-by-$(d+1)$ matrix
Store data points (x_i, y_i) $i \in \{1, ..., n/p\}$ as $[X, y]$
Create a m-by-m kernel matrix K, $m = n/p$
for $i \in 1 : m$ **do**
  x_i, y_i = M[i][1:d+1]
  Machine t: $(x_i^t, y_i^t) = (x_i, y_i)$, $t \in \{1, ..., p\}$
end
for $\lambda \in \Lambda$ *and* $\sigma \in \Sigma$ **do**
  for $i \in 1 : m$ **do**
    for $j \in 1 : m$ **do**
      $K[i][j] = \Phi(x_i, x_j)$ based on Table 3.1
    end
  end
  Solve linear equation $(K + \lambda m I)\alpha = y$ for α
  for $t \in 1 : p$ **do**
    $err_t \leftarrow 0$
  end
  for $j \in 1 : k$ **do**
    if $t = MyCluster(\hat{x}_j)$ **then**
      $\tilde{y}_j \leftarrow \sum_{i=1}^{m} \alpha_i^t K(x_i^t, \hat{x}_j)$
      $err_t \leftarrow err_t + ||\tilde{y}_j - \hat{y}_j||^2$
    end
  end
  Reduce: $MSE = (\sum_{t=1}^{p} err_t)/k$
  if $t = 0$ **then**
    if $MSE < \widehat{MSE}$ **then**
      $\widehat{MSE} \leftarrow MSE$, $\widehat{\lambda} \leftarrow \lambda$, $\widehat{\sigma} \leftarrow \sigma$
    end
  end
end



3.6.1 Load Balance for Data Size



3.6.2 Load Balance for Time

Figure 3.6: This experiment is conducted on NERSC Cori Supercomputer (WRIGHT et al., 2015). We use 8 nodes for load balance test. The test dataset is MSD dataset and we use 16000 samples. We observe that BKRR has roughly 2000 samples on each node, which leads to a perfect load balance. For KKRR, because different nodes have different number of samples, the fastest node is 51× faster than the slowest node, which leads to a huge resource waste.

the training. Lines 9-11 construct the kernel matrix. Line 12 solves the linear equation on each machine. Lines 13-18 perform the prediction. Only the best model does the prediction for a certain test sample. Lines 19-22 evaluate the accuracy of the system. There is no communication cost of BKRR2 during training except for the initial data partition and the final model selection. The additional cost of BKRR2 comes from two parts: (1) the K-balance algorithm and partition, and (2) the prediction.

The overhead of K-balance is tiny compared with the training part. K-balance first does K-means. The cost of K-means is $\Theta(In)$ where I is the number of K-means iterations, which is usually less 100. The cost of lines 3-12 in Algorithm 9 is $\Theta(pn)$ where p is the number of partitions and also the number of machines. $\Theta(In + pn)$ is tiny compared with the training part, which is $\Theta(|\Sigma||\Lambda|n^3/p^3)$. For example, if we use BKRR2 to process the 32k MSD training samples on 96 NERSC Edison processors, the single-iteration training time is 20 times larger than the K-balance and partition time. Since the computational overhead of K-balance is low compared to KRR training ($n = \Theta(p^4)$ in practice), we can just use one node to finish the K-balance algorithm. Although it is not necessary, we have an approximate parallel algorithm for K-balance. We conduct parallel K-means clustering and distribute the centers to all the nodes. Then we partition the samples to all the processors and set *balanced* as n/p^2. Since $n = \Theta(p^4)$ in practice, this parallel implementation roughly gets the same results with the single-node implementation. The overhead of this approximate parallel algorithm is $\Theta(In/p + n)$.

For the prediction part, instead of conducting k small communications, we make each machine first compute its own error (line 18 of Algorithm 10) and then only conduct one global communication (line 19 of Algorithm 10). The reason is detailed below. The runtime

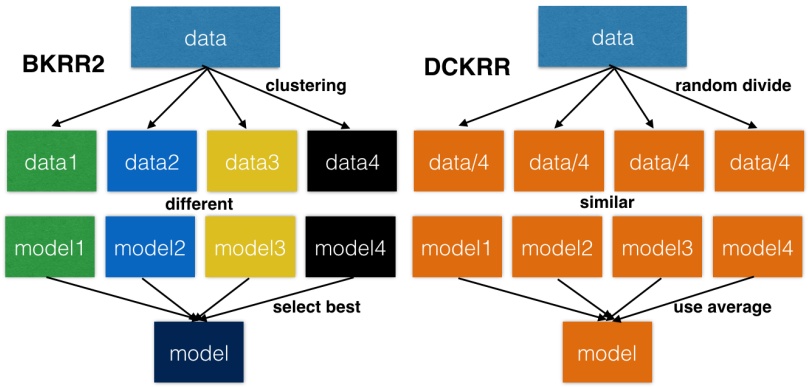


Figure 3.7: Difference between BKRR2 and DCKRR. BKRR2: partition the dataset into p different parts, generate p different models, and select the best model. DCKRR: partition the dataset into p similar parts, generate p similar models, and use the average of all the models.

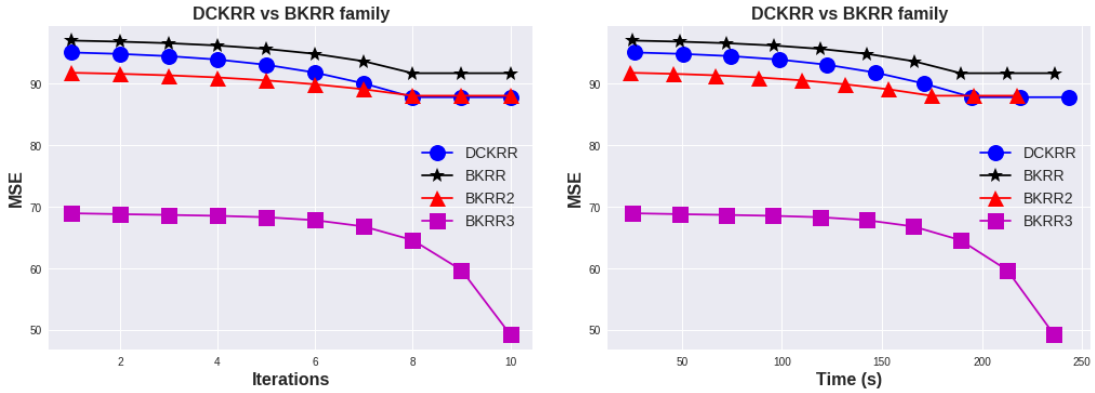
on a distributed system can be estimated by the sum of the time for computation and the time for communication: $f \times \gamma + n_m \times (\alpha + n_b \times \beta)$, where f is the number of floating point operations, γ is the time per floating point operation, α is latency, $1/\beta$ is the bandwidth, n_m is the number of messages, and n_b is the number of bytes per message. In practice, $\alpha \gg \beta \gg \gamma$ (e.g. $\alpha = 7.2 \times 10^{-6}$s, $\beta = 0.9 \times 10^{-9}$s for Intel NetEffect NE020, $\gamma = 2 \times 10^{-11}$s for Intel 5300-series Clovertown). Therefore, one big message is much cheaper than k small messages because $k \times (\alpha + n_b \times \beta) \gg (\alpha + k \times n_b \times \beta)$. This optimization reduces the latency overhead. Figures 3.8 and 3.9 show that BKRR2 achieves lower error rate than DC-KRR in a shorter time for a variety of datasets. Figure 3.7 shows the difference between BKRR2 and DC-KRR.

BKRR3 and KKRR3: Error Lower Bound and The Unrealistic Approach

The KKRR and BKRR families share the same idea of partitioning the data into p parts and making p different models, but they are different in that the KKRR family is optimized for accuracy while the BKRR family is optimized for performance. We want to know the gap between our method and the best theoretical method. Let us refer to the theoretical KKRR method as KKRR3 and the theoretical BKRR method as BKRR3.

BKRR3 is similar to BKRR2 in terms of communication and computation pattern. Like BKRR and BKRR2, after the training process, each sub-BKRR will generate its own model file $(MF_1, MF_2, ..., MF_P)$. We can use these model files independently for prediction. For a given test sample $\hat{x}$ ($\hat{y}$ is its true regressand), we make all the nodes have a copy of $\hat{x}$ (like KKRR). Each model will make a prediction for $\hat{x}$. We get $\tilde{y}_1$, $\tilde{y}_2$, ..., $\tilde{y}_p$ from p models respectively. We select MF_i for prediction where $i = argmin_j ||\tilde{y}_j - \hat{y}||^2$. This means we **inspect** the true regressand to make sure we select the best model for each test sample.

When changing K-balance to K-means, BKRR3 becomes KKRR3, which is much more accurate than DCKRR (Figures 3.5.1 and 3.5.2). The MSE of BKRR3 is the lower bound of the MSE of BKRR2 because BKRR3 can always use the best model for each test sample. In fact, BKRR3 is even much more accurate than the original method (DKRR) for all the testings in our experiments. The framework of BKRR3 is in Algorithm 11. Figures 3.8 and 3.9 show the results. BKRR3 is always the best approach in these comparisons. BKRR3 achieves much higher accuracy than DCKRR and BKRR2.



3.8.1 3.8.2

Figure 3.8: These figures do a point-do-point comparison between DC-KRR and BKRR family using test data set MSD (described in Section 3.5). They use the same parameter set and conduct the same number of iterations on 96 NERSC Edison processors. DCKRR is more accurate than BKRR. BKRR2 is faster than DCKRR for getting the same accuracy. BKRR3 is an optimal but unrealistic implementation for comparison purposes.

3.5 Implementation and Analysis

Real-World Dataset

To give a fair comparison with DC-KRR, we use the Million Song Dataset (MSD) (Bertin-Mahieux et al., 2011) as our major dataset in our experiments because MSD was used in the chapter of DC-KRR. It is a freely-available collection of audio features and metadata for a million contemporary popular music tracks. The dataset contains $n = 515{,}345$ samples. Each sample is a song (track) released between 1922 and 2011, and the song is represented as a vector of timbre information computed from the song. Each sample consists of a pair of (x_i, y_i) where x_i is a d-dimensional ($d = 90$) vector and $y_i \in [1922, 2011]$ is the year that the song was released. The Million Song Dataset Challenge aims at being the best possible offline evaluation of a music recommendation system. It is a large-scale, personalized music

Algorithm 11 BKRR3

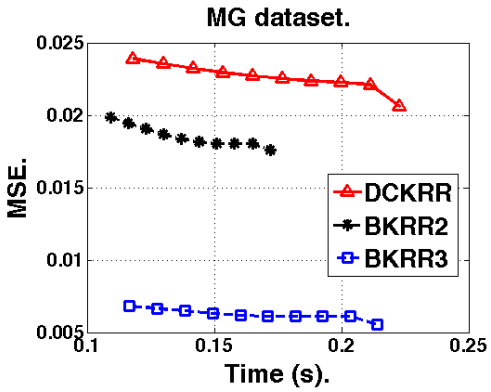
Input:

n labeled data points (x_i, y_i) for training;
another k labeled data points $(\hat{x}_j, \hat{y}_j)$ for testing;
both x_i and $\hat{x}_j$ are d dimensional vectors;
$i \in \{1, 2, ..., n\}$, $j \in \{1, 2, ..., k\}$;
t is the rank of a machine, $t \in \{1, ..., p\}$;

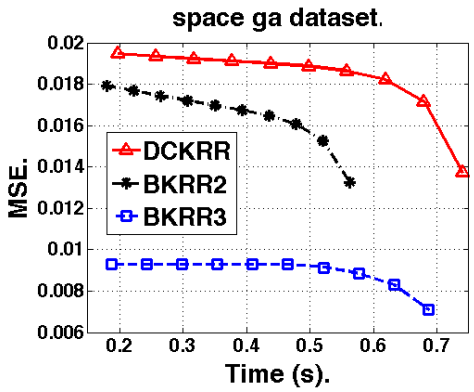
Output:

Mean Squared Error ($\widehat{MSE}$) of prediction
best parameters $\hat{\lambda}, \hat{\sigma}$

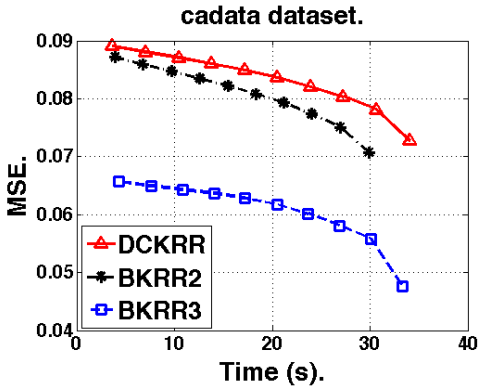
Do K-balance clustering
$M = [X, y]$ is a (n/p)-by-$(d+1)$ matrix
Store data points (x_i, y_i) $i \in \{1, ..., n/p\}$ as $[X, y]$
Create a m-by-m kernel matrix K, $m = n/p$
for $i \in 1 : m$ **do**
  x_i, y_i = M[i][1:d+1]
  Machine t: $(x_i^t, y_i^t) = (x_i, y_i)$, $t \in \{1, ..., p\}$
end
for $\lambda \in \Lambda$ *and* $\sigma \in \Sigma$ **do**
  for $i \in 1 : m$ **do**
    for $j \in 1 : m$ **do**
      $K[i][j] = \Phi(x_i, x_j)$ based on Table 3.1
    end
  end
  Solve linear equation $(K + \lambda m I)\alpha = y$ for α
  for $j \in 1 : k$ **do**
    Machine t: $\tilde{y}_j^t = \sum_{i=1}^{m} \alpha_i^t K(x_i^t, \hat{x}_j)$
    Global reduce: $id = argmin_t ||\tilde{y}_j^t - \hat{y}_j||^2$
    Send/Receive: $\tilde{y}_j = \tilde{y}_j^{id}$
  end
  if $rank = 0$ **then**
    $MSE = \frac{1}{k}\sum_{j=1}^{k}(\tilde{y}_j - \hat{y}_j)^2$
    if $MSE < \widehat{MSE}$ **then**
      $\widehat{MSE} \leftarrow MSE$, $\widehat{\lambda} \leftarrow \lambda$, $\widehat{\sigma} \leftarrow \sigma$
    end
  end
end



3.9.1 1024 training, 361 test samples



3.9.2 2560 training, 547 test samples



3.9.3 18432 training, 2208 test samples

Figure 3.9: These figures do a point-to-point comparison between BKRR2, BKRR3 and DC-KRR. They use the same parameter set and conduct the same number of iterations on 96 NERSC Edison processors. BKRR2 is faster than DCKRR for getting the same accuracy. BKRR3 is an optimal but unrealistic implementation for comparison purposes.

recommendation challenge, where the goal is to predict the songs that a user will listen to, given both the user's listening history and full information (including meta-data and content analysis) for all songs. To justify the efficiency of our approach, we use another three real-world datasets. The information of these four datasets are summarized in Table 3.2. All these datasets were downloaded from (C.-J. Lin, 2017).

Fair Comparison

Let us use p as the number of partitions or nodes, ρ as the number of processors. Each node has 24 processors. When we use ρ=1536 processors, we actually divide the dataset into p=64 parts. Each machine generates a local model for BKRR2. To give a fair comparison, we make

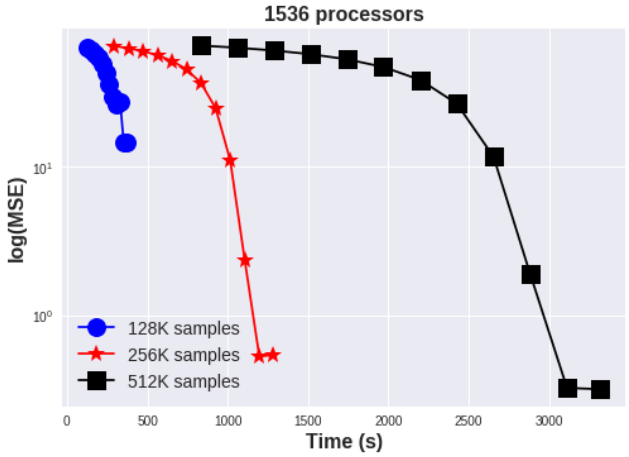


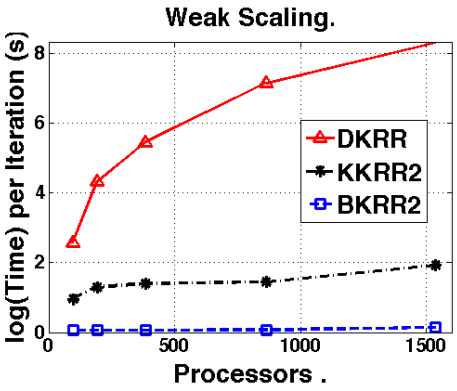
Figure 3.10: BKRR2 results based on MSD dataset. Using increasing number of samples on a fixed number of processors, we can get a better model but also observe the time increase in the speed of $\Theta(n^3)$.

Table 3.2: The test datasets

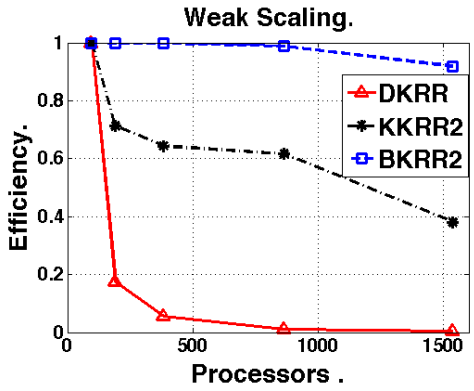
name	MSD	cadata	MG	space-ga
# **Train**	463,715	18,432	1,024	2,560
# **Test**	51,630	2,208	361	547
Dimension	90	8	6	6
Application	Music	Housing	Dynamics	Politics

sure all the comparisons were tuned based on the same parameter set. Different methods may select different best-parameters from the same parameter set to achieve its lowest MSE. From Figures 3.8, 3.9 and 3.11, we clearly observe BKRR2 is faster than DC-KRR and also achieves lower prediction error on all the datasets. In other words, BKRR2 and DC-KRR may use different parameters to achieve their lowest MSEs. The lowest MSE of BKRR2 is lower than the lowest MSE of DC-KRR. On the other hand, BKRR2 is slightly faster than both DC-KRR and BKRR3 (for both single iteration time and overall time). The reason is that each machine only needs to process $1/p$ of the test samples for prediction. For DC-KRR and BKRR3, each machine needs to process all the test samples for prediction. BKRR2 achieves 1.2× speedup over DC-KRR on average and has a much lower error rate for 128k-sample MSD dataset (14.7 vs 81.0).

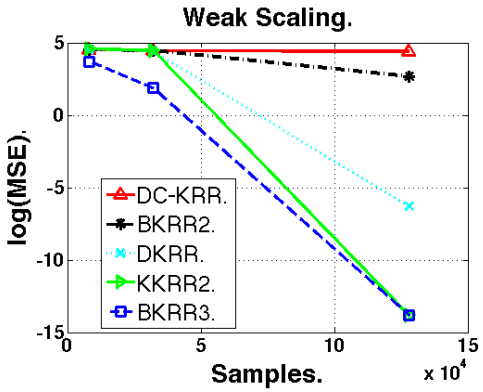
Because of DKRR's poor weak scaling, BKRR2 runs much faster than DKRR for 1536 processors and 128k training samples. The single iteration time of BKRR2, t_b, is 1.15 sec while the single iteration time of DKRR, t_d, is 4048 sec. Here, single iteration means picking a pair of parameters and solving the linear equation once (e.g. lines 9-22 of Algorithm 10 are



3.11.1 Time



3.11.2 Efficiency



3.11.3 Accuracy

Figure 3.11: Results based on MSD dataset. We use 96 processors (i.e. 4 nodes) and 8k samples as the baseline. The lowest MSE of DKRR is 0.001848 on 1536 processors. The lowest MSE of BKRR3 is 10^{-7}. The weak scaling efficiency of DKRR at 1536 processors is 0.32%. The weak scaling efficiency of KKRR2 and BKRR2 at 1536 processors is 38% and 92%, respectively. The MSE of DC-KRR for 2k test samples only decreases from 88.9 to 81.0, which is a bad weak scaling accuracy. The MSE of BKRR2 decreases from 93.1 to 14.7. The MSE of KKRR2 decreases from 95.0 to 10^{-7}. The data is in Tables 3.3 and 3.4. We conclude that the proposed methods outperform the existing methods.

one iteration). The algorithm can get a better pair of parameters after each iteration. All the algorithms in this chapter run the same number of iterations because they use the same parameter tuning space. However, it is unfair to say BKRR2 achieves 3505× speedup over DKRR because, for the same 2k test dataset, the lowest MSE of BKRR2 is 14.7 while that of DKRR is 0.002. This means the BKRR2 model with 128K samples model (bm_{128}) is worse than DKRR model (dm_{128}) for accuracy. To give a fair comparison, we increase the training

Table 3.3: Weak Scaling in time. We use 96 processors and 8000 MSD samples as the baseline. Constant time means perfect scaling. BKRR2 has very good scaling efficiency. DKRR's scaling efficiency is poor.

Method	Processors	96	192	384	768	1536
BKRR2	IterTime (s)	1.06	1.06	1.06	1.08	1.15
KKRR2	IterTime (s)	2.60	3.66	4.05	4.23	6.85
DKRR	IterTime (s)	13	75.1	234	1273	4048
BKRR2	Efficiency	1.0	1.0	1.0	0.99	0.92
KKRR2	Efficiency	1.0	0.71	0.64	0.62	0.38
DKRR	Efficiency	1.0	0.17	0.06	0.01	0.003

Table 3.4: Weak Scaling in Accuracy on MSD Dataset. Lower is better. DCKRR is a bad algorithm.

Samples	DCKRR	BKRR2	DKRR	KKRR2	BKRR3
8k	88.9	93.1	90.9	95.0	40.2
32k	85.5	87.7	85.0	87.5	6.6
128k	81.0	14.7	0.002	10^{-7}	10^{-7}

samples of BKRR2 to 256k so that its lowest MSE can generate a better model (bm_{256}). By using bm_{256}, the lowest MSE of BKRR2 is 0.53 (Figure 3.10). We run it on the same 1536 processors and observe t_b becomes 5.6 sec. In this way, we can say that BKRR2 achieves 723× speedup over DKRR for achieving roughly the same accuracy by using the same hardware and the same test dataset. It is worth noting that the biggest dataset that DKRR can handle on 1536 processors is 128k samples, otherwise it will collapse. Therefore, dm_{128} is the best model DKRR can get on 1536 processors. However, BKRR2 can use a bigger dataset to get a better model than bm_{256} on 1536 processors (e.g. 512k model in Figure 3.10).

The theoretical speedup of bm_{128} over dm_{128} is the ratio of $\Theta(n^3/p)$ and $\Theta((n/p)^3)$, which is 4096× for $p = 64$ and $\rho = 1536$. We achieve 3505× speedup. The theoretical speedup of bm_{256} over dm_{128} is the ratio of $\Theta(n^3/p)$ and $\Theta((2n/p)^3)$, which is 512× for $p = 64$ and $\rho = 1536$. We achieve 723× speedup. The difference between theoretical speedup and practical speedup comes from systems and low-level libraries (e.g. the implementation of LAPACK and ScaLAPACK). In this comparison, BKRR2's better scalability allows it to use more data than DKRR, which cannot run on an input of the same size. We also want to compare KKRR2 to DKRR for using the same amount of data. Let us refer to the model generated by KKRR2 using 128k samples as km_{128} and the single iteration time as t_k. t_k is 6.9 sec in our experiment. The MSE of km_{128} is 10^{-7}, which is even lower than the MSE of dm_{128} (0.002). Thus, KKRR2 achieves 591× speedup over DKRR for the same accuracy by using the same data and hardware (Table 3.3 and 3.4).

Weak-Scaling Issue

As we mentioned in a previous section, weak scaling means we fix the data size per node and increase the number of nodes. We use from 96 to 1536 processors (4 to 64 partitions) for scaling tests on the NERSC Edison supercomputer. The test dataset is the MSD dataset. We set the 96-processor case as the baseline and assume it has a 100% weak-scaling. We double the number of samples as we double the number of processors. Figure 3.11 and Table 3.3 show the time weak-scaling and accuracy weak-scaling of BKRR. In terms of time weak-scaling, BKRR2 achieves 92% efficiency on 1536 processors while DKRR only achieves 0.32%. We then compare the weak scaling in terms of test accuracy: the MSE of DC-KRR for 2k test samples only decreases from 88.9 to 81.0, while the MSE of BKRR2 decreases from 93.1 to 14.7. KKRR2 reduced the MSE from 95 to 10^{-7}. In conclusion, we observe that DKRR has a bad time scaling and DC-KRR has a poor accuracy weak scaling. Our proposed method, BKRR2, has good weak scaling behavior both in terms of time and accuracy. The weak scaling of BKRR3 can be shown in Table 3.5.

Table 3.5: Weak scaling results of BKRR3. We use 96 processors and 8k samples as the baseline. We double the number of samples as we double the number of processors.

processors	96	192	384	768	1536
Time (s)	1311	1313	1328	1345	1471
Efficiency	100%	99.8%	98.7%	97.5%	89.1%
MSE	40.2	16.5	6.62	1.89	10^{-7}

Method Selection

KKRR2 achieves better accuracy than the state-of-the-art method. If the users need the most accurate method, KKRR2 is the best choice. KKRR2 runs slower than DC-KRR but much faster than DKRR. BKRR2 achieves better accuracy and also runs faster than the DCKRR. BKRR2 is much faster than DKRR with the same accuracy. The reason why BKRR2 is more efficient than DKRR is not that it assumes more samples, but rather, because it has much better weak scaling, so it is easier to use more samples to get a better model. Thus, BKRR2 is the most practical method because it balances the speed and the accuracy. BKRR3 can be used to evaluate the performance of systems both in accuracy and speed. KKRR2 is optimized for accuracy. We recommend using either BKRR2 or KKRR2.

Implementation Details

We use CSR (Compressed Row Storage) format for processing the sparse matrix (the original n-by-d input matrix, not the dense kernel matrix) in our implementation. We use MPI for distributed processing. To give a fair comparison, we use ScaLAPACK (Choi et al., 1995) for solving the linear equation on distributed systems and LAPACK (E. Anderson et al.,

1999) on shared memory systems, both of which are Intel MKL versions. Our experiments are conducted on NERSC Cori and Edison supercomputer systems (NERSC, 2016). Our source code is available online[1].

The kernel matrix is symmetric positive definite (SPD), and so is $K + \lambda nI$. Thus we use Cholesky decomposition to solve $(K + \lambda nI)\alpha = y$ in DKRR, which is 2.2× faster than the Gaussian Elimination version.

To conduct a weak scaling study, we select a subset (e.g. 64k, 128k, 256k) from 463,715 samples as the training data to generate a model. Then we use the model to make a prediction. We use 2k test samples to show the MSE comparisons among different methods. If we change the number of test samples (e.g. change 2k to 20k), the testing accuracy will be roughly the same. The reason is that all the test samples are randomly shuffled so the 2k case will have roughly the same sample pattern as the 20k case.

3.6 GPU Acceleration

GPUs are popular tools for accelerating machine learning applications (You, Gitman, and Ginsburg, 2017) and many machine learning applications are built on top of GPU-based systems. Thus, we also built our system on GPU clusters. We use Piz Daint supercomputer located at CSCS (the Swiss National Supercomputing Centre). Each GPU node is equipped with one Intel Xeon E5-2690 v3 at 2.60GHz (12 cores, 64GB RAM) and one NVIDIA Tesla P100 16GB. There are a total of 5704 GPU nodes. Since 2018 Piz Daint is ranked 6th on the TOP500 list. Our results are shown in Fig. 3.12 and Fig. 3.13. We can observe our algorithms achieve similar speedups with the Knights Landing cluster. We can conclude that our algorithms perform well for both CPU/KNL based systems and GPU based systems.

3.7 Parallel Efficiency of BKRR2

In the following, we use n to refer to the problem size (the number of samples). Like the previous part, we use p to refer to the number of machines. To be more precise, let $t(n; p)$ be the per-iteration time, which is a function of n and p; and let $i(\Lambda; \Sigma)$ be the number of iterations, which is $|\Lambda| \times |\Sigma|$). For KRR, $i(\Lambda; \Sigma)$ typically is at least 1000, that is, there is no actual dependence on n or p. Then, the total time should ideally be

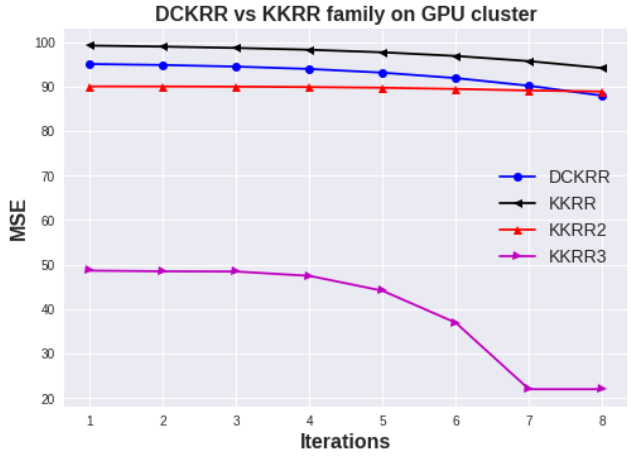
$$T(\Lambda; \Sigma; n; p) = i(\Lambda; \Sigma) \times t(n; p)$$

Thus, the parallel efficiency of a p-machine systems becomes

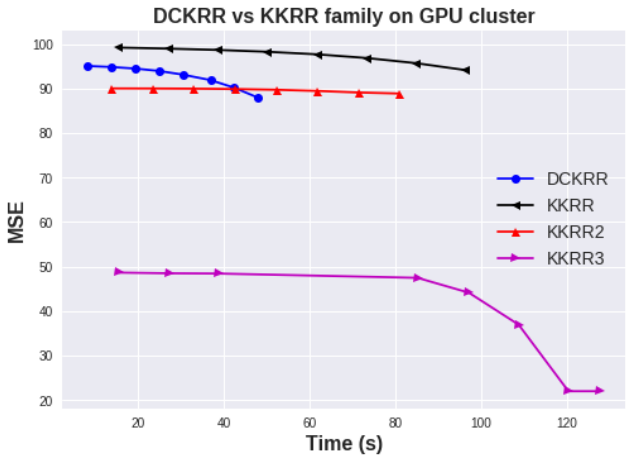
$$E(n; p) = \frac{i(\Lambda; \Sigma) \times t(n; 1)}{p \times i(\Lambda; \Sigma) \times t(n; p)} = \frac{t(n; 1)}{p \times t(n; p)}$$

If the per-iteration time scales perfectly — meaning $t(n; p) = t(n; 1)/p$ — the efficiency should be 1. The $t(n; 1)$ of KRR is $\Theta(n^3)$ because it needs to solve a dense linear equation.

[1] `https://people.eecs.berkeley.edu/~youyang/cakrr.zip`



3.12.1



3.12.2

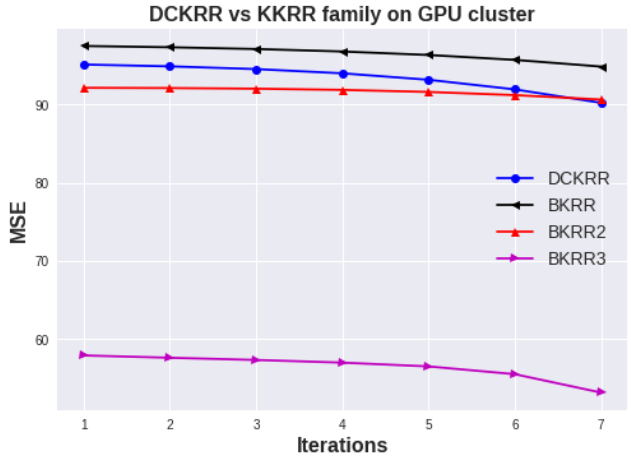
Figure 3.12: Comparison between DC-KRR and KKRR family, using same parameter set on eight nodes of Piz Daint supercomputer (each node has one Intel Xeon E5-2690 v3 CPU and one NVIDIA P100 GPUs). The test data set MSD is described in Section 3.5. KKRR2 is an accurate but slow algorithm. KKRR3 is an optimal but unrealistic method for comparison purposes.

For BKRR2, each node is actually an independent KRR. Thus we can get

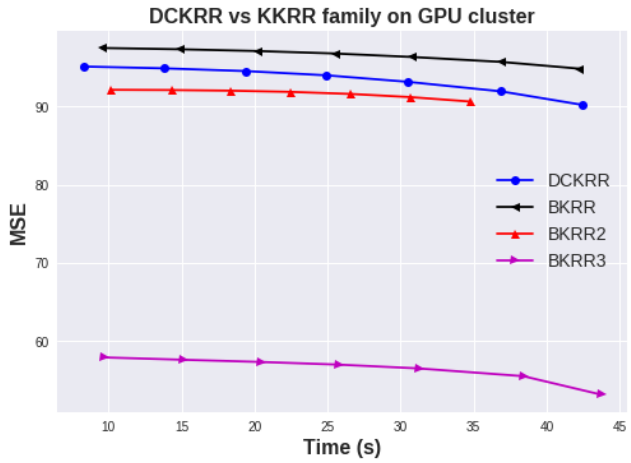
$$t(n;p) = t(n/p;1) = \Theta(n^3/p^3) = t(n;1)/p^3$$

because each node only trains n/p samples.

In other words, each node is a BKRR problem with $\Theta((n/p)^3)$ operations, and



3.13.1



3.13.2

Figure 3.13: These figures do a point-to-point comparison between DC-KRR and BKRR family using test data set MSD (described in Section 3.5). They use the same parameter set and conduct the same number of iterations on eight nodes of Piz Daint supercomputer (each node has one Intel Xeon E5-2690 v3 CPU and one NVIDIA P100 GPUs). DCKRR is more accurate than BKRR. BKRR2 is faster than DCKRR for getting the same accuracy. BKRR3 is an optimal but unrealistic implementation for comparison purposes.

$$E(n;p) = \frac{t(n;1)}{p \times t(n;p)} = p^2$$

This means the parallel efficiency of BKRR2 is p^2 in theory. Usually, we expect the parallel efficiency of a regular distributed system to lie between 0 and 1.

The reason behind the $\Theta(p^2)$ parallel efficiency is that the way we set this up is perhaps not quite right – the sequential baseline should be the best sequential baseline, not the naive (plain KRR) one. If we execute BKRR2 sequentially by simulating p partitions with only 1 machine, then the sequential time would be

$$\hat{t}(n;1) = p \times t(n/p;1) = p \times \Theta((n/p)^3) = t(n;1)/p^2$$

So then $E(n;p)$ would approach 1 rather than p^2 because $\hat{t}(n;1)$ is close to $t(n;p)$. Put another way, this means that BKRR2 is a perfect algorithm, even in the sequential case. That indicates we can beat KRR by running BKRR2 to simulate p partitions using only 1 machine. We reserve the experiments as our future work.

3.8 Related Work

There are papers on using low rank techniques to approximate the kernel matrix. Schölkopf, A. Smola, and Müller, 1998 conducted a nonlinear form of principal component analysis (PCA) in high-dimensional feature spaces. Fine and Scheinberg, 2002 used Incomplete Cholesky Factorization (ICF) to design an efficient Interior Point Method (IPM). Williams and Seeger, 2001 designed a Nyström sampling method by carrying out an eigendecomposition on a smaller dataset. Si Si, Cho-Jui Hsieh, and I. Dhillon, 2014 designed a memory efficient kernel approximation method by first dividing the kernel matrix and then doing low-rank approximation on the smaller matrices. These methods can reduce the running time from $\Theta(n^2d)$ to $\Theta(nd^2)$ or $\Theta(ndk)$ where k is the rank of the kernel approximation. However, all of these methods are proposed for serial analysis on single node systems. On the other hand, Kernel method is more accurate than existing approximate methods (Y. Zhang, J. Duchi, and Wainwright, 2013). Y. Zhang, J. Duchi, and Wainwright, 2013 showed DC-KRR can beat all the previous approximate methods. DC-KRR is considered as state-of-the-art approach. Thus, we focus on the comparison with Y. Zhang, J. Duchi, and Wainwright, 2013.

The second line of work is to use iterative methods such as gradient descent (Yao, Rosasco, and Caponnetto, 2007), block Jacobi method (Schreiber, 1986) and conjugate gradient method (Blanchard and Krämer, 2010) to reduce the running time. However, an iterative method can not make full use of computational powers efficiently because they only load a small amount of data to memory. If we scale the algorithm to 100+ cores, Kernel matrix method will be much faster. These methods provide a trade-off between time and accuracy, and we reserve them for future research. ASKIT (March, Xiao, and Biros, 2015) used n-body ideas to reduce the time and storage of kernel matrix evaluation, which is a direction of our future work. CA-SVM (You, Demmel, Kenneth Czechowski, et al., 2015) (You, Demmel, Kent Czechowski, L. Song, and Rich Vuduc, 2016) used the divide-and-conquer approach for machine learning applications. The differences include: (1) CA-SVM is for classification while this work is for regression. (2) CA-SVM uses an iterative method, whereas we use a direct method. (3) The iterative method did not store the huge Kernel matrix, thus CA-SVM does not need to deal with the huge Kernel matrix. The scalability of the iterative method is also limited.

Lu et al., 2013 designed a fast ridge regression algorithm, which reduced the running time from $\Theta(n^2d)$ to $\Theta(log(n)nd)$. However, first, this work made an unreasonable assumption: the number of features is much larger than the number of samples ($d \gg n$). This is true for some datasets like the webspam dataset (C.-J. Lin, 2017), which has 350,000 training samples and each sample has 16,609,143 features. However, on average only about 10 of these 16 million features are nonzero. This can be processed efficiently by the sparse format like CSR (Compressed Sparse Row). Besides, this method only works for the linear ridge regression algorithm rather than the kernel version, which will lose non-linear space information.

The most related work to this chapter is Divide-and-Conquer Kernel Ridge Regression (Y. Zhang, J. Duchi, and Wainwright, 2013), which is the serial version of DC-KRR. The basic idea is to divide the original data into p parts, and then from each part construct a local smaller kernel matrix and finish the training process individually. DC-KRR is generally a fast serial algorithm, which reduces the memory requirement from $\Theta(n^2)$ to $\Theta(n^2/p^2)$. We finish a parallel implementation to justify it is also an efficient parallel approach. The idea of DC-KRR is to randomly and evenly divide the dataset into p similar parts and get p models. These p models are **similar** to each other because of the dividing algorithm. DC-KRR use an **average** of these models, which has been shown better than the model generated by $1/p$ of the original dataset. The idea of BKRR2 or KKRR2 is to use a better partition algorithm to divide the datasets into p different parts. There p models are **different** from each other because of the clustering approach. Either BKRR2 or KKRR2 are actually finding the **best** model from these p models. By doing so, BKRR2 achieves a much better accuracy than DC-KRR and a higher speed by using the same hardware. KKRR2 even achieves higher accuracy than the original method. Fig. 3.7 summarizes the difference between BKRR2 and DC-KRR.

3.9 Conclusion

Due to a $\Theta(n^2)$ memory requirement and $\Theta(n^3)$ arithmetic operations, KRR is prohibitive for large-scale machine learning datasets when n is very large. The weak scaling of KRR is problematic because the total memory required will grow like $\Theta(p)$ per processor, and the total flops will grow like $\Theta(p^2)$ per processor. The reason why BKRR2 is more scalable than DKRR is that it removes all the communication in the training part and reduces the computation cost from $\Theta(n^3/p)$ to $\Theta((n/p)^3)$. The reason why BKRR2 is more accurate than DC-KRR is that it creates p different models and selects the best one. Compared to DKRR, BKRR2 improves the weak scaling efficiency from 0.32% to 92% and achieves 723× speedup for the same accuracy on the same 1536 processors. Compared to DC-KRR, BKRR2 achieves much higher accuracy with faster speed. DC-KRR can never get the best accuracy achieved by BKRR2. When we increase # samples from 8k to 128k and # processors from 96 to 1536, BKRR2 reduces the MSE from 93 to 14.7, which solves the poor accuracy weak-scaling problem of DC-KRR (MSE only decreases from 89 to 81). KKRR2 achieves 591× speedup over DKRR for the same accuracy by using the same data and hardware. Based on a variety of datasets used in this chapter, we observe that KKRR2 is the most accurate method. BKRR2

is the most practical algorithm that balances the speed and accuracy. In conclusion, BKRR2 and KKRR2 are accurate, fast, and scalable methods.

Chapter 4

Asynchronous parallel greedy coordinate descent

4.1 Introduction

Asynchronous parallel optimization has recently become a popular way to speedup machine learning algorithms using multiple processors. The key idea of asynchronous parallel optimization is to allow machines work independently without waiting for the synchronization points. It has many successful applications including linear SVM (Niu et al., 2011; C.-J. Hsieh, H. F. Yu, and I. S. Dhillon, 2015), deep neural networks (J. Dean et al., 2012b; M. Li et al., 2014), matrix completion (Niu et al., 2011; Yun et al., 2014), linear programming (Sridhar et al., 2013), and its theoretical behavior has been deeply studied in the past few years (Liu and Wright, 2014; Avron, Druinsky, and A. Gupta, 2014; J. C. Duchi, Chaturapruek, and Ré, 2015).

The most widely used asynchronous optimization algorithms are stochastic gradient method (SG) (Niu et al., 2011; J. Dean et al., 2012b; J. C. Duchi, Chaturapruek, and Ré, 2015) and coordinate descent (CD) (Liu and Wright, 2014; Avron, Druinsky, and A. Gupta, 2014; C.-J. Hsieh, H. F. Yu, and I. S. Dhillon, 2015), where the workers keep selecting a sample or a variable randomly and conduct the corresponding update asynchronously. Although these stochastic algorithms have been studied deeply, in some important machine learning problems a "greedy" approach can achieve much faster convergence speed. A very famous example is greedy coordinate descent: instead of randomly choosing a variable, at each iteration the algorithm selects the most important variable to update. If this selection step can be implemented efficiently, greedy coordinate descent can often make bigger progress compared with stochastic coordinate descent, leading to a faster convergence speed. For example, the decomposition method (a variant of greedy coordinate descent) is widely known as best solver for kernel SVM (John C. Platt, 1998; Joachims, 1998a), which is implemented in LIBSVM and SVMLight. Other successful applications can be found in (Cho-Jui Hsieh and Inderjit S. Dhillon, 2011; I. S. Dhillon, Ravikumar, and Tewari, 2011; Yen et al., 2013).

In this chapter, we study the asynchronous greedy coordinate descent algorithm framework.

The variables are partitioned into subsets, and each worker asynchronously conducts greedy coordinate descent in one of the blocks. To our knowledge, this is the first work to present a theoretical analysis or practical applications of this asynchronous parallel algorithm. In the first part of the chapter, we formally define the asynchronous greedy coordinate descent procedure, and prove a linear convergence rate under mild assumptions. In the second part of the chapter, we discuss how to apply this algorithm to solve the kernel SVM problem on multi-core machines. Our algorithm achieves linear speedup with the number of cores, and performs better than other multi-core SVM solvers.

The rest of the chapter is outlined as follows. The related work is discussed in Section 4.2. We propose the asynchronous greedy coordinate descent algorithm in Section 4.3 and derive the convergence rate in the same section. In Section 4.4 we show the details how to apply this algorithm for training kernel SVM, and the experimental comparisons are presented in Section 4.5.

This chapter is based on a joint work with Xiangru Lian, Ji Liu, Hsiang-Fu Yu, Inderjit S Dhillon, James Demmel, and Cho-Jui Hsieh. It was published as a conference paper entitled *Asynchronous parallel greedy coordinate descent* (You, Lian, et al., 2016).

4.2 Related Work

Coordinate Descent. Coordinate descent (CD) has been extensively studied in the optimization community (Bertsekas, 1999), and has become widely used in machine learning. At each iteration, only one variable is chosen and updated while all the other variables remain fixed. CD can be classified into stochastic coordinate descent (SCD), cyclic coordinate descent (CCD) and greedy coordinate descent (GCD) based on their variable selection scheme. In SCD, variables are chosen randomly based on some distribution, and this simple approach has been successfully applied in solving many machine learning problems (Shalev-Shwartz and T. Zhang, 2013; Cho-Jui Hsieh, K.-W. Chang, et al., 2008). The theoretical analysis of SCD has been discussed in (Yurii E. Nesterov, 2012; Richtárik and Takáč, 2014). Cyclic coordinate descent updates variables in a cyclic order, and has also been applied to several applications (Canutescu and Dunbrack, 2003; H.-F. Yu et al., 2013).

Greedy Coordinate Descent (GCD). The idea of GCD is to select a good, instead of random, coordinate that can yield better reduction of objective function value. This can often be measured by the magnitude of gradient, projected gradient (for constrained minimization) or proximal gradient (for composite minimization). Since the variable is carefully selected, at each iteration GCD can reduce objective function more than SCD or CCD, which leads to faster convergence in practice. Unfortunately, selecting a variable with larger gradient is often time consuming, so one needs to carefully organize the computation to avoid the overhead, and this is often problem dependent. The most famous application of GCD is the decomposition method (John C. Platt, 1998; Joachims, 1998a) used in kernel SVM. By exploiting the structure of quadratic programming, selecting the variable with largest gradient magnitude can be done without any overhead; as a result GCD becomes the dominant

technique in solving kernel SVM, and is implemented in LIBSVM (C.-C. Chang and C.-J. Lin, 2000) and SVMLight (Joachims, 1998a). There are also other applications of GCD, such as non-negative matrix factorization (Cho-Jui Hsieh and Inderjit S. Dhillon, 2011), large-scale linear SVM (Yen et al., 2013), and (I. S. Dhillon, Ravikumar, and Tewari, 2011) proposed an approximate way to select variables in GCD. Recently, Nutini et al., 2015 proved an improved convergence bound for greedy coordinate descent. We focus on parallelizing the GS-r rule in this chapter but our analysis can be potentially extended to the GS-q rule mentioned in that chapter.

To the best of our knowledge, the only literature discussing how to parallelize GCD was in C. Scherrer et al., 2012; Chad Scherrer et al., 2012. A thread-greedy/block-greedy coordinate descent is a *synchronized* parallel GCD for L_1-regularized empirical risk minimization. At an iteration, each thread randomly selects a block of coordinates from a pre-partitioned block partition and proposes the best coordinate from this block along with its increment (i.e., step size). Then all the threads are synchronized to perform the actual update to the variables. However, the method can potentially diverge; indeed, this is mentioned in Chad Scherrer et al., 2012 about the potential divergence when the number of threads is large. C. Scherrer et al., 2012 establishes sub-linear convergence for this algorithm.

Asynchronous Parallel Optimization Algorithms. In a synchronous algorithm each worker conducts local updates, and in the end of each round they have to stop and communicate to get the new parameters. This is not efficient when scaling to large problem due to the curse of last reducer (all the workers have to wait for the slowest one). In contrast, in asynchronous algorithms there is no synchronization point, so the throughput will be much higher than a synchronized system. As a result, many recent work focus on developing asynchronous parallel algorithms for machine learning as well as providing theoretical guarantee for those algorithms (Liu and Wright, 2014; Avron, Druinsky, and A. Gupta, 2014; J. C. Duchi, Chaturapruek, and Ré, 2015; Niu et al., 2011; Yun et al., 2014; C.-J. Hsieh, H. F. Yu, and I. S. Dhillon, 2015; J. Dean et al., 2012b; M. Li et al., 2014; Xing et al., 2015).

In distributed systems, asynchronous algorithms are often implemented using the concept of parameter servers (J. Dean et al., 2012b; M. Li et al., 2014; Xing et al., 2015). In such setting, each machine asynchronously communicates with the server to read or write the parameters. In our experiments, we focus on another multi-core shared memory setting, where multiple cores in a single machine conduct updates independently and asynchronously, and the communication is implicitly done by reading/writing to the parameters stored in the shared memory space. This has been first discussed in Niu et al., 2011 for the stochastic gradient method, and recently proposed for parallelizing stochastic coordinate descent (Liu, Wright, et al., 2014; C.-J. Hsieh, H. F. Yu, and I. S. Dhillon, 2015).

This is the first work proposing an asynchronous *greedy* coordinate decent framework. The closest work to ours is Liu, Wright, et al., 2014 for asynchronous stochastic coordinate descent (ASCD). In their algorithm, each worker asynchronously conducts the following updates: (1) randomly select a variable (2) compute the update and write to memory or server. In our AGCD algorithm, each worker will select the best variable to update in a block, which leads

to faster convergence speed. We also compare with ASCD algorithm in the experimental results for solving the kernel SVM problem.

4.3 Asynchronous Greedy Coordinate Descent

We consider the following constrained minimization problem:

$$\min_{x\in\Omega} f(x), \tag{4.1}$$

where f is convex and smooth, $\Omega \subset \mathbb{R}^N$ is the constraint set, $\Omega = \Omega_1 \times \Omega_2 \times \cdots \times \Omega_N$ and each $\Omega_i, i = 1, 2, \ldots, N$ is a closed subinterval of the real line.

Notation: We denote S to be the optimal solution set for equation 4.1 and $\mathcal{P}_S(x), \mathcal{P}_\Omega(x)$ to be the Euclidean projection of x onto S, Ω, respectively. We also denote f^* to be the optimal objective function value for equation 4.1.

We propose the following Asynchronous parallel Greedy Coordinate Descent (Asy-GCD) for solving equation 4.1. Assume N coordinates are divided into n non-overlapping sets $S_1 \cup \ldots \cup S_n$. Let k be the global counter of total number of updates. In Asy-GCD, each processor repeatedly runs the following GCD updates:

- Randomly select a set $S_k \in \{S_1, \ldots, S_n\}$ and pick the coordinate $i_k \in S_k$ where the projected gradient (defined in equation 4.2) has largest absolute value.
- Update the parameter by
$$x_{k+1} \leftarrow \mathcal{P}_\Omega(x_k - \gamma \nabla_{i_k} f(x_k)),$$
where γ is the step size.

Here the projected gradient defined by

$$\nabla^+_{i_k} f(\hat{x}_k) := x_k - \mathcal{P}_\Omega(x_k - \nabla_{i_k} f(\hat{x}_k)) \tag{4.2}$$

is a measurement of optimality for each variable, where $\hat{x}_k$ is current point stored in memory used to calculate the update. The processors will run concurrently without synchronization. In order to analyze Asy-GCD, we capture the system-wise global view in Algorithm 12.

Algorithm 12 Asynchronous Parallel Greedy Coordinate Descent (Asy-GCD)

Input: $x_0 \in \Omega, \gamma, K$
Output: x_{K+1}
Initialize $k \leftarrow 0$ **while** $k \leq K$ **do**
  Choose S_k from $\{S_1, \ldots, S_n\}$ with equal probability
  Pick $i_k = \arg\max_{i \in S_k} \|\nabla^+_i f(\hat{x})\|$
  $x_{k+1} \leftarrow \mathcal{P}_\Omega(x_k - \gamma \nabla_{i_k} f(\hat{x}_k))$
  $k \leftarrow k + 1$
end

The update in the k^{th} iteration is

$$x_{k+1} \leftarrow \mathcal{P}_\Omega(x_k - \gamma \nabla_{i_k} f(\hat{x}_k)),$$

where i_k is the selected coordinate in k^{th} iteration, $\hat{x}_k$ is the point used to calculate the gradient and $\nabla_{i_k} f(\hat{x}_k)$ is a zero vector where the i_kth coordinate is set to the corresponding coordinate of the gradient of f at $\hat{x}_k$. Note that $\hat{x}_k$ may not be equal to the current value of the optimization variable x_k due to asynchrony. Later in the theoretical analysis we will need to assume $\hat{x}_k$ is close to x_k using the bounded delay assumption.

In the following we prove the convergence behavior of Asy-GCD. We first make some commonly used assumptions:

Assumption 1

1. **(Bounded Delay)** There is a set $J(k) \subset \{k-1, \ldots, k-T\}$ for each iteration k such that
$$\hat{x}_k \;\coloneqq\; x_k - \sum_{j \in J(k)} (x_{j+1} - x_j), \tag{4.3}$$
where T is the upper bound of the staleness. In this "inconsistent read" model, we assume some of the latest T updates are not yet written back to memory. This is also used in some previous papers (Liu, Wright, et al., 2014; Avron, Druinsky, and A. Gupta, 2014), and is more general than the "consistent read" model that assumes $\hat{x}_k$ is equal to some previous iterate.

2. For simplicity, we assume each set $S_i, i \in \{1, \ldots, n\}$ has m coordinates.

3. **(Lipschitzian Gradient)** The gradient function of the objective $\nabla f(\cdot)$ is Lipschitzian. That is to say,
$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \quad \forall x, \forall y. \tag{4.4}$$
Under the Lipschitzian gradient assumption, we can define three more constants L_{res}, L_s and $L_{\max}$. Define L_{res} to be the restricted Lipschitz constant satisfying the following inequality:
$$\|\nabla f(x) - \nabla f(x + \alpha e_i)\| \leq L_{\text{res}}|\alpha|, \quad \forall i \in \{1, 2, ..., N\} \text{ and } t \in \mathbb{R} \text{ with } x, x + te_i \in \Omega \tag{4.5}$$
Let ∇_i be the operator calculating a zero vector where the i^{th} coordinate is set to the i^{th} coordinate of the gradient. Define $L_{(i)}$ for $i \in \{1, 2, \ldots, N\}$ as the minimum constant that satisfies:
$$\|\nabla_i f(x) - \nabla_i f(x + \alpha e_i)\| \leq L_{(i)}|\alpha|. \tag{4.6}$$
Define $L_{\max} \coloneqq \max_{i \in \{1,\ldots,N\}} L_{(i)}$. It can be seen that $L_{\max} \leq L_{\text{res}} \leq L$.

Let s be any positive integer bounded by N. Define L_s to be the minimal constant satisfying the following inequality: $\forall x \in \Omega, \forall S \subset \{1, 2, ..., N\}$ where $|S| \leq s$:
$$\left\|\nabla f(x) - \nabla f\left(x + \textstyle\sum_{i \in S} \alpha_i e_i\right)\right\| \leq L_s \left\|\textstyle\sum_{i \in S} \alpha_i e_i\right\|.$$

4. **(Global Error Bound)** We assume that our objective f has the following property: when $\gamma = \frac{1}{3L_{\max}}$, there exists a constant κ such that
$$\|x - \mathcal{P}_S(x)\| \;\leqslant\; \kappa\|\tilde{x} - x\|, \forall x \in \Omega. \tag{4.7}$$
Where $\tilde{x}$ is defined by $\operatorname{argmin}_{x' \in \Omega} \left(\langle \nabla f(x), x' - x\rangle + \frac{1}{2\gamma}\|x' - x\|^2\right)$. This is satisfied by strongly convex objectives and some weakly convex objectives. For example, it is proved

in P.-W. Wang and C.-J. Lin, 2014 that the kernel SVM problem equation 4.9 satisfies the global error bound even when the kernel is not strictly positive definite.

5. **(Independence)** All random variables in $\{S_k\}_{k=0,1,\cdots,K}$ in Algorithm 12 are independent to each other.

We then have the following convergence result:

Theorem 4.1 (convergence)

Choose $\gamma = 1/(3L_{\max})$ in Algorithm 12. Suppose $n \geq 6$ and that the upper bound for staleness T satisfies the following condition

$$T(T+1) \leqslant \frac{\sqrt{n}L_{\max}}{4eL_{\text{res}}}. \tag{4.8}$$

Under Assumption 1, we have the following convergence rate for Algorithm 12:

$$\mathbb{E}(f(x_k) - f^*) \leqslant \left(1 - \frac{2L_{\max}b}{L\kappa^2 n}\right)^k (f(x_0) - f^*).$$

where b is defined as

$$b = \left(\frac{L_T^2}{18\sqrt{n}L_{\max}L_{\text{res}}} + 2\right)^{-1}.$$

This theorem indicates a linear convergence rate under the global error bound and the condition $T^2 \leq O(\sqrt{n})$. Since T is usually proportional to the total number cores involved in the computation, this result suggests that one can have linear speedup as long as the total number of cores is smaller than $O(n^{1/4})$. Note that for $n = N$ Algorithm 12 reduces to the standard asynchronous coordinate descent algorithm (ASCD) and our result is essentially consistent with the one in (Liu, Wright, et al., 2014), although they use the optimally strong convexity assumption for $f(\cdot)$. The optimally strong convexity is a similar condition to the global error bound assumption (H. Zhang, 2015).

Here we briefly discuss the constants involved in the convergence rate. Using Gaussian kernel SVM on covtype as a concrete sample, $L_{\max} = 1$ for Gaussian kernel, L_{res} is the maximum norm of columns of kernel matrix (≈ 3.5), L is the 2-norm of Q[1] (21.43 for covtype), and conditional number $\kappa \approx 1190$. As number of samples increased, the conditional number κ will become a dominant term, and this also appears in the rate of serial greedy coordinate descent. In terms of speedup when increasing number of threads (T), although L_T may grow, it only appears in $b = (\frac{L_T^2}{18\sqrt{n}L_{max}L_{res}} + 2)^{-1}$, where the first term inside b is usually small since there is a $\sqrt{n}$ in the demominator. Therefore, $b \approx 2^{-1}$ in most cases, which means the convergence rate does not slow down too much when we increase T.

4.4 Application to Multi-core Kernel SVM

In this section, we demonstrate how to apply asynchronous parallel greedy coordinate descent to solve kernel SVM (Boser, I. Guyon, and V. Vapnik, 1992; Corina Cortes and Vladimir Vapnik, 1995). We follow the conventional notations for kernel SVM, where the variables

[1] Q is the kernel matrix, which is formally defined in Section 4.4.

for the dual form are $\alpha \in \mathbb{R}^n$ (instead of x in the previous section). Given training samples $\{a_i\}_{i=1}^{\ell}$ with corresponding labels $y_i \in \{+1, -1\}$, kernel SVM solves the following quadratic minimization problem:

$$\min_{\alpha \in \mathbb{R}^n} \left\{ \frac{1}{2} \alpha^T Q \alpha - e^T \alpha \right\} := f(\alpha) \quad \text{s.t.} \quad 0 \leq \alpha \leq C, \tag{4.9}$$

where Q is an ℓ by ℓ symmetric matrix with $Q_{ij} = y_i y_j K(a_i, a_j)$ and $K(a_i, a_j)$ is the kernel function. Gaussian kernel is a widely-used kernel function, where $K(a_i, a_j) = e^{-\gamma \|a_i - a_j\|^2}$.

Greedy coordinate descent is the most popular way to solve kernel SVM. In the following, we first introduce greedy coordinate descent for kernel SVM, and then discuss the detailed update rule and implementation issues when applying our proposed Asy-GCD algorithm on multi-core machines.

Kernel SVM and greedy coordinate descent

When we apply coordinate descent to solve the dual form of kernel SVM equation 4.9, the one variable update rule for any index i can be computed by:

$$\delta_i^* = P_{[0,\ C]}\big(\alpha_i - \nabla f_i(\alpha)/Q_{ii}\big) - \alpha_i \tag{4.10}$$

where $P_{[0,\ C]}$ is the projection to the interval $[0,\ C]$ and the gradient is $\nabla f_i(\alpha) = (Q\alpha)_i - 1$. Note that this update rule is slightly different from equation 4.2 by setting the step size to be $\gamma = 1/Q_{ii}$. For quadratic functions this step size leads to faster convergence because δ_i^* obtained by equation 4.10 is the closed form solution of

$$\delta^* = \arg\min_{\delta} f(\alpha + \delta e_i),$$

and e_i is the i-th indicator vector.

As in Algorithm 12, we choose the best coordinate based on the magnitude of projected gradient. In this case, by definition

$$\nabla_i^+ f(\alpha) = \alpha_i - P_{[0,\ C]}\big(\alpha_i - \nabla_i f(\alpha)\big). \tag{4.11}$$

The success of GCD in solving kernel SVM is mainly due to the maintenance of the gradient

$$g := \nabla_i f(\alpha) = (Q\alpha) - 1.$$

Consider the update rule equation 4.10: it requires $O(\ell)$ time to compute $(Q\alpha)_i$, which is the cost for stochastic coordinate descent or cyclic coordinate descent. However, in the following we show that GCD has the same time complexity per update by using the trick of maintaining g during the whole procedure. If g is available in memory, each element of the projected gradient equation 4.11 can be computed in $O(1)$ time, so selecting the variable with maximum magnitude of projected gradient only costs $O(\ell)$ time. The single variable update equation 4.10 can be computed in $O(1)$ time. After the update $\alpha_i \leftarrow \alpha_i + \delta$, the g has to be updated by $g \leftarrow g + \delta q_i$, where q_i is the i-th column of Q. This also costs $O(\ell)$ time. Therefore, each GCD update only costs $O(\ell)$ using this trick of maintaining g.

Therefore, for solving kernel SVM, GCD is faster than SCD and CCD since there is no additional cost for selecting the best variable to update. Note that in the above discussion we assume Q can be stored in memory. Unfortunately, this is not the case for large scale

problems because Q is an ℓ by ℓ dense matrix, where ℓ can be millions. We will discuss how to deal with this issue in Section 4.4.

With the trick of maintaining $g = Q\alpha - 1$, the GCD for solving equation 4.9 can be summarized in Algorithm 13.

Algorithm 13 Greedy Coordinate Descent (GCD) for Dual Kernel SVM

Initial $g = -\mathbf{1}$, $\alpha = 0$ **for** $k = 1, 2, \cdots$ **do**
- step 1: Pick $i = \arg\max_i |\nabla_i^+ f(\alpha)|$ using g (See eq equation 4.11)
- step 2: Compute δ_i^* by eq equation 4.10
- step 3: $g \leftarrow g + \delta^* q_i$
- step 4: $\alpha_i \leftarrow \alpha_i + \delta^*$

end

Asynchronous greedy coordinate descent

When we have n threads in a multi-core shared memory machine, and the dual variables (or corresponding training samples) are partitioned into the same number of blocks:

$$S_1 \cup S_2 \cup \cdots \cup S_n = \{1, 2, \cdots, \ell\} \quad \text{and} \quad S_i \cap S_j = \phi \text{ for all } i, j.$$

Now we apply Asy-GCD algorithm to solve equation 4.9. For better memory allocation of kernel cache (see Section 4.4), we bind each thread to a partition. The behavior of our algorithm still follows Asy-GCD because the sequence of updates are asynchronously random. The algorithm is summarized in Algorithm 14.

Algorithm 14 Asy-GCD for Dual Kernel SVM

Initial $g = -\mathbf{1}$, $\alpha = 0$

Each thread t repeatedly performs the following updates *in parallel*:
- step 1: Pick $i = \arg\max_{i \in S_t} |\nabla_i^+ f(\alpha)|$ using g (See eq equation 4.11)
- step 2: Compute δ_i^* by eq equation 4.10
- step 3: For $j = 1, 2, \cdots, \ell$
  - $g_j \leftarrow g_j + \delta^* Q_{j,i}$ **using atomic update**
- step 4: $\alpha_i \leftarrow \alpha_i + \delta^*$

Note that each thread will read the ℓ-dimensional vector g in step 2 and update g in step 3 in the shared memory. For the read, we do not use any atomic operations. For the writes, we maintain the correctness of g by atomic writes, otherwise some updates to g might be overwritten by others, and the algorithm cannot converge to the optimal solution. Theorem 4.1, suggests a linear convergence rate of our algorithm, and in the experimental results we will see the algorithm is much faster than the widely used Asynchronous Stochastic Coordinate Descent (Asy-SCD) algorithm (Liu, Wright, et al., 2014).

Implementation Issues

In addition to the main algorithm, there are some practical issues we need to handle in order to make Asy-GCD algorithm scales to large-scale kernel SVM problems. Here we discuss these implementation issues.

Kernel Caching. The main difficulty for scaling kernel SVM to large dataset is the memory requirement for storing the Q matrix, which takes $O(\ell^2)$ memory. In the GCD algorithm, step 2 (see eq equation 4.10) requires a diagonal element of Q, which can be pre-computed and stored in memory. However, the main difficulty is to conduct step 3, where a column of Q (denoted by q_i)is needed. If q_i is in the memory, the algorithm only takes $O(\ell)$ time; however, if q_i is not in the memory, re-computing it from scratch takes $O(dn)$ time. As a result, how to maintain most important columns of Q in memory is an important implementation issues in SVM software.

In LIBSVM, the user can specify the size of memory they want to use for storing columns of Q. The columns of Q are stored in a linked-list in the memory, and when memory space is not enough the Least Recent Used column will be kicked out (LRU technique).

In our implementation, instead of sharing the same LRU for all the cores, we create an individual LRU for each core, and make the memory space used by a core in a contiguous memory space. As a result, remote memory access will happen less in the NUMA system when there are more than 1 CPU in the same computer. Using this technique, our algorithm is able to scale up in a multi-socket machine (see Figure 4.5).

Variable Partitioning. The theory of Asy-GCD allows any non-overlapping partition of the dual variables. However, we observe a better partition that minimizes the between-cluster connections can often lead to faster convergence. This idea has been used in a divide-and-conquer SVM algorithm (C. J. Hsieh, S. Si, and I. S. Dhillon, 2014), and we use the same idea to obtain the partition. More specifically, we partition the data by running kmeans algorithm on a subset of 20000 training samples to obtain cluster centers $\{c_r\}_{r=1}^n$, and then assign each i to the nearest center: $\pi(i) = \arg\min_r \|c_r - x_i\|$. This steps can be easily parallelized, and costs less than 3 seconds in all the datasets used in the experiments. Note that we include this kmeans time in all our experimental comparisons.

4.5 Experimental Results

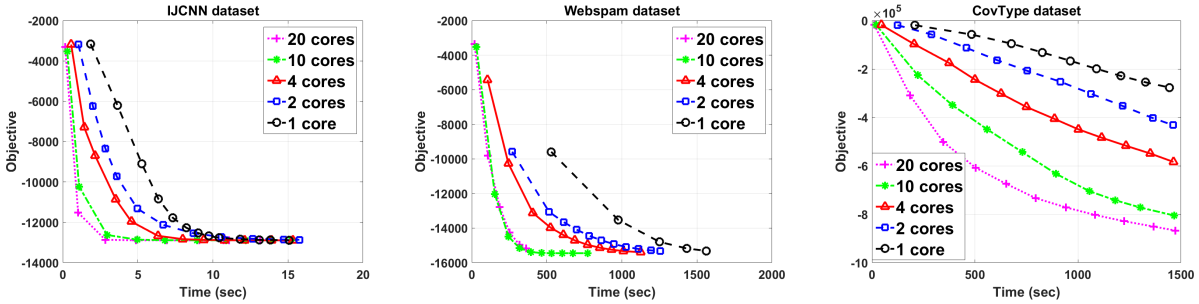
We conduct experiments to show that the proposed method Asy-GCD achieves good speedup in parallelizing kernel SVM in multi-core systems. We consider three datasets: IJCNN, COVTYPE and WebSpam (see Table 4.1 for detailed information). We follow the parameter settings in C. J. Hsieh, S. Si, and I. S. Dhillon, 2014, where C and γ are selected by cross validation. All the experiments are run on the same system with 20 CPUs and 256GB memory, where the CPU has two sockets, each with 10 cores. We locate 64GB for kernel caching for

Table 4.1: Data statistics. ℓ is number of training samples, d is dimensionality, ℓ_t is number of testing samples.

	ℓ	ℓ_t	d	C	γ
IJCNN	49,990	91,701	22	32	2
COVTYPE	464,810	116,202	54	32	32
WebSpam	280,000	70,000	254	8	32

all the algorithms. In our algorithm, the 64GB will distribute to each core; for example, for Asy-GCD with 20 cores, each core will have 3.2GB kernel cache.

Figure 4.1: Comparison of Asy-GCD with 1–20 threads on IJCNN, COVTYPE and WebSpam datasets.



(a) IJCNN time vs obj (b) WebSpam time vs obj (c) COVTYPE time vs obj

We include the following algorithms/implementations into our comparison:

1. Asy-GCD: Our proposed method implemented by C++ with OpenMP. Note that the preprocessing time for computing the partition is included in all the timing results.

2. PSCD: We implement the asynchronous stochastic coordinate descent (Liu, Wright, et al., 2014) approach for solving kernel SVM. Instead of forming the whole kernel matrix in the beginning (which cannot scale to all the dataset we are using), we use the same kernel caching technique as Asy-GCD to scale up PSCD.

3. LIBSVM (OMP): In LIBSVM, there is an option to speedup the algorithm in multi-core environment using OpenMP (see `http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html#f432`). This approach uses multiple cores when computing a column of kernel matrix (q_i used in step 3 of Algorithm 13).

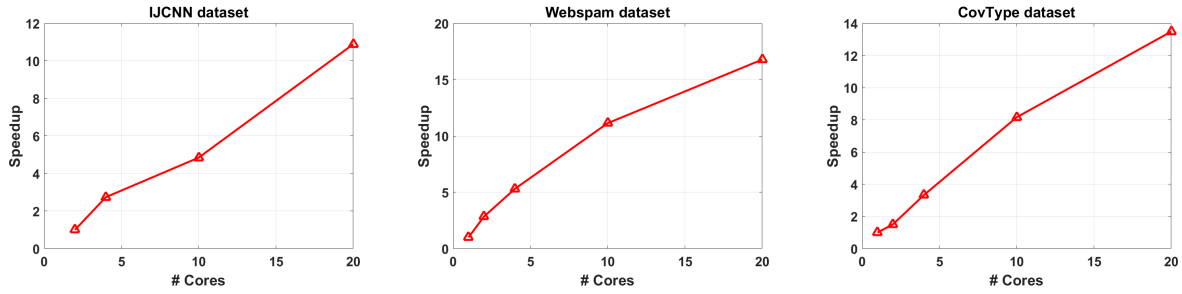
All the implementations are modified from LIBSVM (e.g., they share the similar LRU cache class), so the comparison is very fair. We conduct the following two sets of experiments. Note that another recent proposed DC-SVM solver (C. J. Hsieh, S. Si, and I. S. Dhillon, 2014) is currently not parallelizable; however, since it is a meta algorithm and requires training a series of SVM problems, our algorithm can be naturally served as a building block of DC-SVM.

Scaling with number of cores

In the first set of experiments, we test the speedup of our algorithm with varying number of cores. The results are presented in Figure 4.5 and Figure 4.5. We have the following observations:

- **Time vs obj (for 1, 2, 4, 10, 20 cores).** From Fig. 4.5 (a)-(c), we observe that when we use more CPU cores, the objective decreases faster.
- **Cores vs speedup.** From Fig. 4.5, we can observe that we got good strong scaling when we increase the number of threads. Note that our computer has two sockets, each with 10 cores, and our algorithm can often achieve 13-15 times speedup. This suggests our algorithm can scale to multiple sockets in a Non-Uniform Memory Access (NUMA) system. Previous asynchronous parallel algorithms such as HogWild (Niu et al., 2011) or PASSCoDe (C.-J. Hsieh, H. F. Yu, and I. S. Dhillon, 2015) often struggle when scaling to multiple sockets.

Figure 4.2: The scalability of Asy-GCD with up to 20 threads.



(a) IJCNN cores vs speedup (b) WebSpam cores vs speedup (c) COVTYPE cores vs speedup

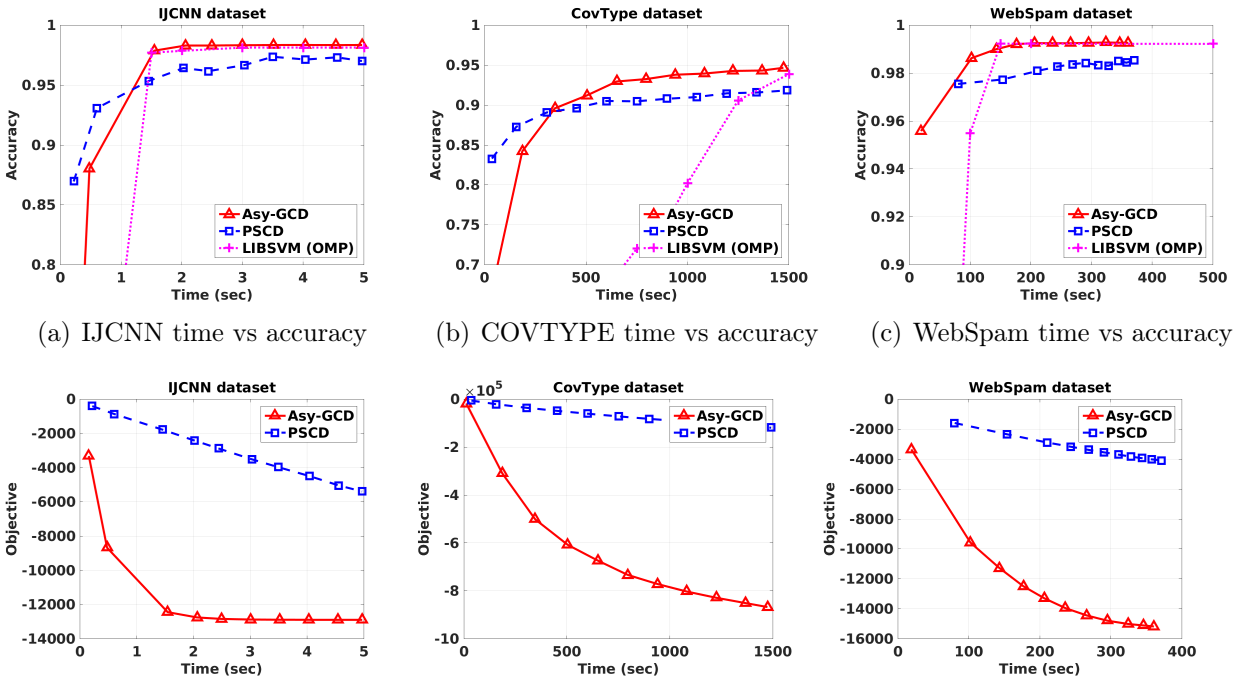
Comparison with other methods

Now we compare the efficiency of our proposed algorithm with other multi-core parallel kernel SVM solvers on real datasets in Figure 4.5. All the algorithms in this comparison are using 20 cores and 64GB memory space for kernel caching. Note that LIBSVM is solving the kernel SVM problem with the bias term, so the objective function value is not showing in the figures.

We have the following observations:

- Our algorithm achieves much faster convergence in terms of objective function value compared with PSCD. This is not surprising because using the trick of maintaining g (see details in Section 4.4) greedy approach can select the best variable to update, while stochastic approach just chooses variables randomly. In terms of accuracy, PSCD is sometimes good in the beginning, but converges very slowly to the best accuracy. For

Figure 4.3: Comparison among multi-core kernel SVM solvers. All the solvers use 20 cores and the same amount of memory.



(a) IJCNN time vs accuracy (b) COVTYPE time vs accuracy (c) WebSpam time vs accuracy

(d) IJCNN time vs objective (e) COVTYPE time vs objective (f) WebSpam time vs objective

example, in COVTYPE data the accuracy of PSCD remains 93% after 4000 seconds, while our algorithm can achieve 95% accuracy after 1500 seconds.

- LIBSVM (OMP) is slower than our method. The main reason is that they only use multiple cores when computing kernel values, so the computational power is wasted when the column of kernel (q_i) needed is available in memory.

4.6 Conclusions

In this chapter, we propose an Asynchronous parallel Greedy Coordinate Descent (Asy-GCD) algorithm, and prove a linear convergence rate under mild conditions. We show our algorithm is useful for parallelizing the greedy coordinate descent method for solving kernel SVM, and the resulting algorithm is much faster than existing multi-core SVM solvers. Since our algorithm is fully asynchronous—each core does not need to idle and wait for the other cores—our implementation enjoys good speedup and outperforms asynchronous stochastic coordinate descent and multi-core LIBSVM. Specifically, our method can achieve a speedup of 2.7× over PSCD and improve the accuracy by 2% at the same time. We conclude that Asy-GCD is fast and accurate in real-world applications.

Chapter 5

Efficient Large-Batch Optimization for DNNs

5.1 Introduction

For deep learning applications, larger datasets and bigger models lead to significant improvements in accuracy (Amodei et al., 2015), but at the cost of longer training time. For example, it takes a Nvidia M40 GPU 14 days to finish just one 90-epoch ResNet-50 training execution on the ImageNet-1k dataset. This long experiment turnaround motivates the research of training time reduction of Deep Neural Networks (DNN). The 90-epoch ResNet-50 training requires 10^{18} single precision operations. On the other hand, according to the Nov 2018 Top 500 results, the world's current fastest supercomputer can finish 3×10^{17} single precision operations per second. So, if the training can make full use of the computing capability of the supercomputer, it should be able to finish in several seconds.

So far, the best results on scaling ImageNet-1k training have used the synchronous stochastic gradient descent method (synchronous SGD) to enable parallelism. The synchronous SGD algorithm has many inherent advantages, but at the root of these advantages is determinism (modulo floating point round-off). Determinism[1] ensures that all valid parallel implementations of the algorithm match the behavior of the sequential version. This property is invaluable during DNN design and during the debugging of optimization algorithms. In parallel DNN training with synchronous SGD, larger batch size is important to keep up machine efficiency, as it assigns each processor sufficient amount of work in each iteration. For example, engaging 512 processors with a batch size of 1k would mean that each processor only gets a local batch of 2 images. In contrast, a larger batch size of 32k assigns each processor 64 images in each iteration. The latter case thus makes more efficient use of the machines, as the computation to communication ratio is higher.

Over the last two years, we have seen the focus on increasing the batch size and number

[1]Here, we just want to make a comparison between asynchronous SGD and synchronous SGD. Given the same random-number generator, synchronous SGD is able to reproduce the same numerical results in different runs. However, asynchronous SGD does not provide this determinism.

of processors used in image classification training, with a resulting reduction in training time. FireCaffe (Iandola et al., 2016) demonstrated scaling the training of GoogleNet to 128 Nvidia K20 GPUs with a batch size of 1k and a total training time of 10.5 hours for 72 epochs. Although using a larger batch size naively can lead to significant loss in test accuracy, with the warm-up technique coupled with the linear scaling rule, Goyal et al., 2017 were able to scale the training of ResNet-50 to 256 Nvidia P100 GPUs with a batch size of 8k and a total training time of 65 minutes. Using a more sophisticated approach to adapting the learning rate in the Layer-wise Adaptive Rate Scaling (LARS) algorithm (You, Gitman, and Ginsburg, 2017), You, Gitman, and Ginsburg were able to scale the batch size dramatically to 32k in a **simulation study**. On eight Nvidia P100 GPUs (simulating the distributed system[2]), they (You, Gitman, and Ginsburg) reported a 3.4% reduction in accuracy due to the absence of data augmentation.

Given the large batch sizes that the LARS algorithm enables, it is natural to ask: how much further can we scale out the training of DNN on the ImageNet-1k dataset? This is the investigation that led to this chapter. At a high level, we find out that the 32k batch size can efficiently scale DNN training on ImageNet-1k dataset up to thousands of processors. In particular, we are able to finish the **100-epoch training with AlexNet in 11 minutes with 58.6% top-1 test accuracy** (defined in §5.2) on 2,048 Intel Xeon Platinum 8160 processors. With 2,048 Intel Xeon Phi 7250 Processors, we are able to reduce the turnaround time of the **90-epoch ResNet-50 training to 20 minutes without losing accuracy**, inside which the top-1 test accuracy (defined in §5.2) converges to 74.9% at 64th epoch (14 minutes from starting time).

In summary, we make the following contributions:

- We show the scaling capability of LARS up to thousands of CPUs with no loss of accuracy. Meanwhile we demonstrate that DNN can be successfully trained by CPU-based systems instead of using GPUs. We achieved the best scaling results on Intel hardware.

- We examine the generality of the LARS algorithm on both AlexNet and ResNet-50, while many other works are ResNet-50 specific. AlexNet's communication overhead is much higher than ResNet-50, which makes it hard to scale up on many machines (Section 5.4). Moreover, we found that it is hard to keep the accuracy of AlexNet when we increase the batch size (Section 5.3). Our approaches are general and can be used in all the DNN models.

- Empirically, we demonstrate that LARS is more robust than the recent work (Goyal et al., 2017) at a batch size of 32K on large-scale computers. When we use the same baseline, our results are better than Goyal et al. for all the batch sizes (Fig. 5.20).

- Our work has been open sourced and released in the Intel distribution of Caffe, Facebook's PyTorch, and Google's TensorFlow.

[2]The idea is to simply iterate over multiple batches and accumulate the resulting gradients before committing a weight update.

This chapter is based on a joint work with Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. It was published as a journal paper entitled *Fast deep neural network training on distributed systems and cloud TPUs* (You, Z. Zhang, Cho-Jui Hsieh, et al., 2018).

5.2 Background and Related Work

In this section, we discuss the details of data-parallel stochastic gradient descent (SGD) method, the model parallel approach, and similar work of parallelizing DNN training. There are two major directions for parallelizing DNN: data parallelism and model parallelism. All the later parallel methods are the variants of them.

Data-Parallelism SGD

In the data parallelism method, the dataset is partitioned into P parts stored on each machine, and each machine will have a local copy of the neural network and the weights (w^j). In synchronized data parallelism, the communication happens at two places: the sum of local gradients and the broadcast of the global weights. For the first part, each worker computes the local gradient ∇w^j independently, and sends the update to the master node. The master then updates $\tilde{w} \leftarrow \tilde{w} - \eta/P \sum_{j=1}^{P} \nabla w^j$ after it gets all the gradients from workers. Here, η is the algorithm's learning rate. For the second part, the master broadcasts $\tilde{w}$ to all workers.

There is another implementation for the communication part. The **1 reduce + 1 broadcast** pattern can be replaced by **1 all-reduce** operation. In this situation, each worker computes the local gradient ∇w^j independently, and then the system conducts an all-reduce operation to send the sum of the gradients ($\sum_{j=1}^{P} \nabla w^j$) to all the machines. After that, each machine will do the weight updating ($w^j \leftarrow w^j - \eta/P \sum_{j=1}^{P} \nabla w^j$) locally. In this chapter, we use the all-reduce method rather than the reduce-broadcast method. This synchronized approach is a widely-used method on large-scale systems (Iandola et al., 2016).

Scaling synchronous SGD to more processors has two challenges. The first is giving each processor enough useful work to do; this has already been discussed in §5.1. The second challenge is the inherent problem that after processing each local batch all processors must synchronize their gradient updates via a barrier before proceeding. This problem can be partially ameliorated by overlapping computation and communication (Das et al., 2016) (Goyal et al., 2017), but the inherent synchronization barrier remains. A more radical approach to breaking this synchronization barrier is to pursue a purely asynchronous method. A variety of asynchronous approaches have been proposed (Recht et al., 2011; S. Zhang, Choromanska, and LeCun, 2015; Jin et al., 2016; Mitliagkas et al., 2016). The communication and updating rules differ in the asynchronous approach and the synchronous approach. The simplest version of the asynchronous approach is a master-worker scheme. At each step, the master only communicates with one worker. The master gets the gradients ∇w^j from the j-th worker, updates the global weights, and sends the global weight back to the j-th worker. The order of workers is based on first-come-first-serve strategy. The master machine is also called as *parameter server*. The idea of a parameter server was used in real-world commercial

applications by the Downpour SGD approach (J. Dean et al., 2012a), which has successfully scaled to 16,000 cores. However, Downpour's performance on 1,600 cores for a globally connected network is not significantly better than a single GPU (Seide et al., 2014).

Other Work

Goyal et al. reported finishing the 90-epoch ResNet-50 training on ImageNet-1k dataset with 256 Nvidia P100 GPUs within one hour (Goyal et al., 2017). Their work uses a batch size of 8k. Though we are able to achieve faster training speed than the reported case, our baseline's accuracy is slightly lower than Facebook's version (76.2% vs 75.3%) due to our usage of weaker data augmentation. However, our approach has a higher accuracy with batch sizes that are larger than 16k, as shown in Figure 5.1. Recently, we implemented their baseline on our platform, which helps us to achieve 76.3% top-1 accuracy. Now our implementation is able to outperform the state-of-the-art approaches for all the batch sizes, which is shown in Fig. 5.20.

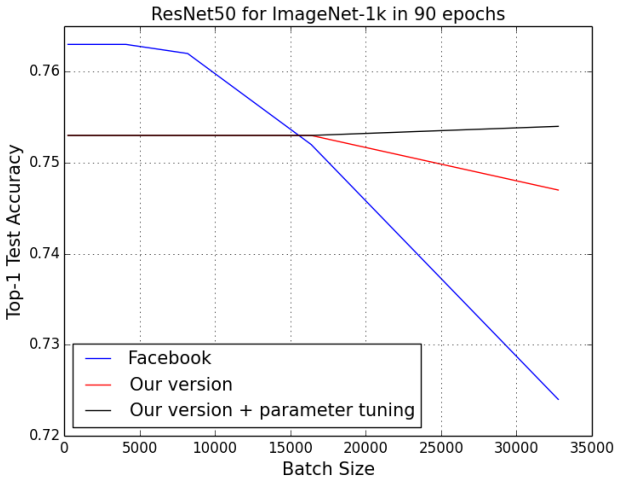


Figure 5.1: Top-1 Test Accuracy Comparison on Various Batch Sizes between Our Approach and Facebook's solution

Codreanu, Podareanu, and Saletore, 2017 reported achieving 73.78% accuracy on ResNet-50 (with data augmentation) in less than 40 minutes on 512 Intel Xeon Phi 7250 Processors. There are two things worth noting: Firstly, their batch size is 8k. Secondly, Codreanu et al. only ran for 37 epochs[3]. The complete 90-epoch training would take 80 minutes with 75.25% accuracy.

Akiba, Suzuki, and Fukuda, 2017 reported finishing the 90-epoch ResNet-50 training within 15 minutes on 1,024 Nvidia P100 GPUs. However, the baseline accuracy is missing

[3]https://blog.surf.nl/en/imagenet-1k-training-on-intel-xeon-phi-in-less-than-40-minutes/

in the report, so it is difficult to tell how much their 74.9% accuracy using the 32k batch size diverges from the baseline. Secondly, both Akiba et al. and Goyal et al. are ResNet-50 specific, while we also show the generality of our approach with AlexNet. It is worth noting that our online preprint (You, Z. Zhang, Demmel, et al., 2017) is two months earlier than Akiba et al.

Top-1 accuracy and Top-5 accuracy

In this section, we explain the difference between the top-1 accuracy and the top-5 accuracy. Top-1 accuracy means the conventional accuracy: the model's prediction (the one with highest probability) must be exactly the expected answer. Top-5 accuracy means that any of your model's five highest probability predictions match the expected answer. For example, let us apply machine learning to object recognition using a neural network. A picture of an airplane is shown, and these are the outputs of our neural network:

- Car with 68% probability
- Train with 11% probability
- Bus with 10% probability
- Airplane with 9% probability
- Tank with 8% probability
- Gun with 2% probability
- Building with 1% probability

If we use top-1 accuracy, we count this output as false, because it predicted a car. If we use top-5 accuracy, we count this output as true, because airplane is among the top-5 guesses. In this example, the dataset has seven classes. The ImageNet-1k dataset has 1,000 classes. All the accuracy in this chapter means top-1 test accuracy. Here, **test accuracy** is the prediction accuracy of the model on the validation dataset, which the model is not trained on.

Recurrent Neural Networks

Our conference publication (You, Z. Zhang, Cho-Jui Hsieh, et al., 2018) is focused on CNN (Convolutional Neural Network) applications. Another popular deep learning research direction is based on Recurrent Neural Networks or RNN (Elman, 1990). RNN has been widely used in language modeling, machine translation, sentiment analysis, speech recognition, and speech synthesis (Goodfellow et al., 2016). A RNN layer is a matrix that can be used t times at each iteration where t is the input sequence length at runtime. Because of the vanishing and exploding gradient problems, it is hard for RNN to learn long-range dependencies. To solve this problem, the LSTM (long-short term memory) technique was proposed by Hochreiter and Schmidhuber, 1997.

We picked Google's Neural Machine Translation (GNMT) or Sequence-to-Sequence (Y. Wu et al., 2016) as our evaluating application. Our GNMT model includes the encoder part and the decoder part. The encoder has four LSTM layers with the hidden size of 1024. The first layer uses bidirectional LSTM. The other three layers use undirectional LSTM. The residual connections start from the 3rd layer. The decoder includes four unidirectional LSTM layers with the hidden size of 1024 and a fully-connected classifier. The residual connections also start from the 3rd layer. The dataset is WMT'16 English-German translation challenge[4]. We use the BLEU score (higher is better) on newstest2014 dataset as the accuracy metric (BLEU score is calculated by sacrebleu package[5]). Our baseline model achieves a BLEU score of 21.80 (Luong, Brevdo, and Zhao, 2017).

Tensor Processing Unit

In this section, we introduce Google's Tensor Processing Unit (TPU), which was used in our experiments. This section is based on the public information from the internet. We do not claim any contribution of this section. TPUv1 chip is focused on DNN inference, which was released in 2017 (Jouppi et al., 2017). This chapter is focused on DNN training, which uses TPUv2. TPUv2 was open to limited users via Google Cloud in February of 2018. This chapter does not include any data on TPUv1.

TPU Chip. Figure 5.2 shows the architecture of a TPU chip. Each TPU chip includes two TPU cores. Inside each TPU core, an 128-by-128 MXU (Matrix Unit) is connected to the scalar/vector units. Scalar/vector units are only for 32-bit floating point operations. MXU supports 32-bit precision for accumulation and 16-bit precision for multiplication. TPU does not support double-precision floating point operations because 32-bit operations are enough for the accuracy of machine learning applications. The performance of each TPU chip is 45 TFlops for 32-bit and 16-bit mixed-precision computation.

TPU. A TPU is made up of four TPU chips. Each TPU provides 180 Teraflops performance (32-bit and 16-bit mixed precision), 64 GB memory, and 2400 GB/s total peak bandwidth. Currently, TPU is open to the public on Google cloud. Figure 5.3 is the structure of a TPU.

TPU Pod. TPU Pod is the first supercomputer for deep learning applications in the cloud (one can also run other applications on TPU Pod if the precision is enough). A TPU Pod can provide 11.5 Petaflops (32-bit and 16-bit mixed precision) performance and 4 terabytes of high-bandwidth memory. TPU Pod is open to the public in near future. Figure 5.4 shows the structure of a TPU Pod.

XLA. Like GPU's cuBLAS and CPU's MKL, TPU provides Accelerated Linear Algebra (XLA) for users to build their own high-level implementations without rewriting the math kernel. XLA supports compiler, runtime, and TPU-specific optimization. Together with the basic math operations, XLA also supports the basic DL operations like convolution.

AutoGraph. If the users choose TensorFlow as the programming framework, Google cloud provides a tool called AutoGraph. Graphs are the basic form of interacting with Tensorflow

[4]http://www.statmt.org/wmt16/translation-task.html

[5]https://pypi.org/project/sacrebleu/

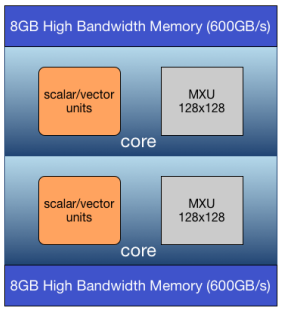


Figure 5.2: The architecture of Tensor Processing Unit (TPU) chip. MXU means Matrix Unit. The performance of a TPU chip is 45 Tflops (32-bit and 16-bit mixed precision). We made this figure based on Jeff Dean's talk at NIPS'17.

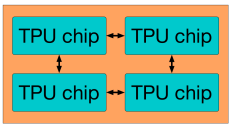


Figure 5.3: Each Tensor Processing Unit (TPU) includes four TPU chips. The performance of a TPU is 180 Tflops. This figure was made by ourselves based on Jeff Dean's talk at NIPS'17.

and have plenty of advantages. Graphs simplify deployment to all sorts of environments, as they can be a platform-independent view of the application's computation. In a dataflow graph, dependencies are explicit which makes it relatively easy to parallelize and distribute the computation. Graphs also allow for whole-program optimizations like kernel fusion which replaces a subgraph with an optimized version by combining nodes. However, the programming model for building graphs in Python is complex and unintuitive. It is like constructing the Abstract Syntax Tree for the program by hand. When it comes to control flow, programmers have to implement it using an unusual functional style. On the other hand, TensorFlow's Eager execution is an imperative programming environment that evaluates operations immediately without building graphs. The users can actually combine the Eager mode and the Graph mode within a single project. The users can call from Eager into graph mode for extra performance. For example, one can implement an especially tricky algorithm in Eager and call it from graph mode. Graph code invoked from Eager is subject to the same clunky interfaces, and Eager code run from the graph is a black box to any graph-based optimizations. It is also not something that can currently be done on a TPU. So these two modes can be combined, but the integration is not perfect. And they can be used at different points of the project, but translating from one to another can be tricky. This is where AutoGraph comes in. AutoGraph allows the users to write Eager-style code and generate graphs anyway. AutoGraph can take a function written using standard Python syntax (that can run in Eager mode) and generate the

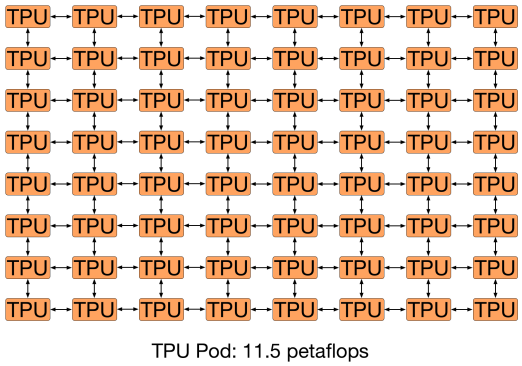


Figure 5.4: Each TPU Pod is made up of 64 second generation TPUs. The performance of a TPU Pod is 11.5 Petaflops (32-bit and 16-bit mixed precision). This figure was made by ourselves based on Jeff Dean's talk at NIPS'17.

code that creates a graph performing the same computation. AutoGraph will automatically convert nice Python constructs like if statements and loops into tf.conds and tf.while_loops. The users just annotate the functions they want to be converted with Autograph. In summary, AutoGraph helps users more easily express complex models in TensorFlow without sacrificing the benefits of graph mode such as performance and portability. The users can think of AutoGraph as adding another stage of computation. Without AutoGraph graph-style Python code is translated into graphs which can be executed by TensorFlow. With AutoGraph users write Eager-style code, autograph generates the graph style code, and TensorFlow executes the resulting graph.

Profiling. The TPU users are able to do profiling by TensorBoard, where they can easily check CPU usage, TPU usage, and the efficiency of Tensorflow/XLA XLA ops. Another useful tool is CTPU (The Cloud TPU Provisioning Utility), which allows the users to configure cloud TPUs and monitor them at runtime.

Using TPUs. Fig. 5.5 illustrates how a regular user can get access to a cloud TPU. Cloud TPUs are network-attached. The user only needs to create a compute engine virtual machine and apply a cloud TPU. The virtual machine connects to the cloud TPU through grpc. The regular users do not need to install any driver and can just use the machine images provided by Google cloud. However, the users still need to design the algorithm and write the code for their own applications. TPUs only provide the basic linear algebra and machine learning functions. In this way, using a cloud TPU is similar to using a GPU or FPGA.

5.3 Large Batch DNN Training

In this section, we discuss the benefits and challenges of large batch training, and the rationale of our model selection along with a learning rate profile study. Throughout the discussion, we focus on the data-parallel synchronous SGD approach, as it is proven to be stable for

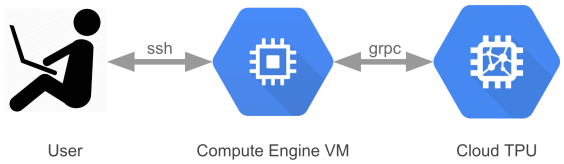


Figure 5.5: This figure illustrates how a regular user can use a cloud TPU. Cloud TPUs are network-attached. Google cloud does not require the user to install any driver. The user can just use the machine images provided by Google cloud.

DNN training at scale (Goyal et al., 2017). In contrast, the asynchronous methods using a parameter server are not guaranteed to be stable in a distributed environment (Jianmin Chen et al., 2016).

Benefits of Large Batch Training

The **prominent advantage of large batch training is that it can reduce the overall training time**. The idea is straightforward—by using a **large batch size** for SGD, the work for each iteration can be distributed to multiple processors. Consider the following ideal case. ResNet-50 requires 7.72 billion single-precision operations to process one 225x225 image. If we run 90 epochs on the ImageNet-1k dataset of 1.28×10^6 images, the total number of operations is $90 * 1.28 \times 10^6 * 7.72 \times 10^9 \approx 10^{18}$. Currently, the world's current fastest supercomputer can finish 3×10^{17} single precision operations per second (according to the Nov 2018 Top 500 results). If there is an algorithm allowing us to make full use of the supercomputer, we can finish the training in several seconds.

To do so, we need to make the algorithm use more processors and load more data at each iteration, which corresponds to increasing the batch size in synchronous SGD. Ideally, if we fix total number of data accesses and grow the batch size linearly with number of processors, the number of SGD iterations will decrease linearly and the time cost of each iteration remains constant, so the total time will also reduce linearly with number of processors. A detailed analytical study on the ResNet-50 training is shown in Table 5.1.

In the strong scaling situation, a large batch does not change the number of floating point operations (computation volume), as the number of epochs is fixed. However, a large batch can reduce the overall communication volume. The reason is that a larger batch size results in fewer iterations and so less overall communication volume, as the single iteration communication volume remains relatively constant given the fact that it is only related to the model size and the networking system. In this way, a large batch size reduces the overall DNN training time in a scalable manner.

A **second benefit of large batch training is that it can keep up the high machine utilization**, which is especially important in a distributed environment.

Let us use one Nvidia M40 GPU to illustrate this benefit on a single machine. Figure 5.6 shows the M40 GPU performance measurements (in images/sec) for AlexNet with varying

batch size from 16 to 512. Increasing the batch size from 16 to 32, the performance almost doubles. And from 128 to 512, the curve flattens, which means the M40 GPU approaches its peak performance at the batch size of 512. A large batch size, such as 8,192, can keep a 16 M40 GPU cluster at its peak performance during the training execution.

Table 5.1: An Analytical Scaling Performance Study with ResNet-50 as the Example. t_1 is the computation time and t_2 is communication time. We fix the number of epochs as 100. Larger batch size needs less iterations. We set batch size as 512 per machine. Then we increase the number of machines. Since $t_1 \gg t_2$ for using ImageNet-1k dataset to train ResNet-50 on GPUs (Goyal et al., 2017), the single iteration time is dominant by the computation. Thus the total time will be reduced approximately linearly.

Batch Size	Num of Epochs	Num of Iterations	Num of GPUs	Single Iteration Time	Total Time
512	100	250k	1	t_1	250kt_1
1024	100	125k	2	$t_1 + \log(2)t_2$	125k$(t_1 + \log(2)t_2)$
2048	100	62500	4	$t_1 + \log(4)t_2$	62500$(t_1 + \log(4)t_2)$
4096	100	31250	8	$t_1 + \log(8)t_2$	31250$(t_1 + \log(8)t_2)$
8192	100	15625	16	$t_1 + \log(16)t_2$	15625$(t_1 + \log(16)t_2)$
...	...	...	...		
1.28M	100	100	2500	$t_1 + \log(2500)t_2$	100$(t_1 + \log(2500)t_2)$

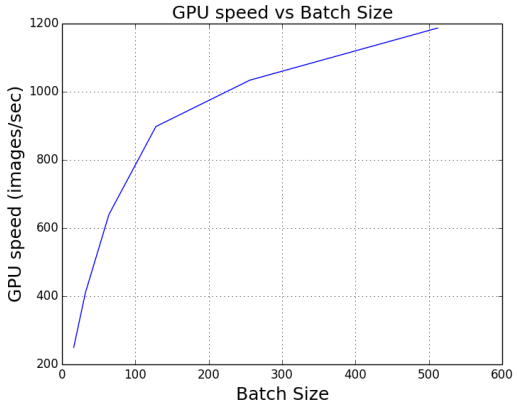


Figure 5.6: AlexNet Training Performance on Various Batch Sizes on a Nvidia M40 GPU. Peak performance is reached with the batch size of 512, while a 1,024 batch size runs out of memory.

Model Selection

To scale out DNN training to many machines, a major overhead is the communication among different machines (S. Zhang, Choromanska, and LeCun, 2015). Here we define the notion of **scaling ratio** as ratio between computation and communication. For DNN models, the computation is proportional to the number of floating point operations required for processing an image. Since we focus on synchronous SGD approach, the communication is proportional to model size (or the number of parameters). Different DNN models have different scaling ratios. To generalize our study, we pick four representative models: AlexNet, GoogleNet, ResNet-50, and VGG. From Table 5.2, we see that ResNet-50's scaling ratio is 12.5× larger than that of AlexNet. This means scaling ResNet-50 is easier than scaling AlexNet. Our experiments in Figures 5.15, 5.16, 5.17, and 5.18 confirmed this conclusion.

For VGG-19, the model size is 575MB (144 million parameters). The model requires around 120 billion operations to process one image. Thus, the scaling ratio is around 855. For GoogleNet, the model size is 54MB (13.5 million parameters). The model requires around 9.7 billion operations to process one image. Thus, the scaling ratio is around 736. In summary, the scaling ratio comparison is: VGG-19 > GoogleNet > ResNet-50 > AlexNet. Although VGG-19 has the best scaling ratio, it is slow to train because it needs many more operations than other models. For the image classification accuracy: ResNet-50 > GoogleNet > VGG-19 > AlexNet.

Table 5.2: Scaling Ratio for state-of-the-art models. The rows of this table are sorted by the 4th column (Scaling Ratio).

Model Name	Comm: Parameters	Comp: Flops per Image	Comp/Comm Scaling Ratio	ImageNet Accuracy
AlexNet	61 million	1.5 billion	24.6	58.0%
ResNet50	25 million	7.7 billion	308	76.3%
GoogleNet	13.5 million	9.7 billion	736	72.5%
VGG-19	144 million	120 billion	855	71.3%

Challenges of Large Batch Training

Large batch size comes with the benefits of potential shorter training time and high machine utilization. However, naively using synchronous SGD with a large batch size usually suffers test accuracy degradation compared to smaller batch sizes with a fixed number of epochs. Unfortunately, there is no algorithm allowing us to effectively use unlimitedly large batch sizes (Keskar et al., 2016). Table 5.3 shows the target test accuracy by standard benchmarks. For example, when we set the batch size of AlexNet larger than 1,024 or the batch size of ResNet-50 larger than 8,192, the test accuracy will be significantly degraded, as shown in Table 5.4 and Figure 5.8, respectively.

Table 5.3: Standard Benchmarks for ImageNet-1k training.

Model	Epochs	Test Top-1 Accuracy
AlexNet	100	58% (Iandola et al., 2016)
ResNet-50	90	75.3% (K. He et al., 2016)
ResNet-50-v2	90	76.3% (Goyal et al., 2017)

Table 5.4: AlexNet Test Accuracy with varying Batch Size. Current approaches (linear scaling + warmup) do not work for AlexNet with a batch size larger than 1k. We tune the warmup epochs from 0 to 10 and pick the one with highest accuracy. According to linear scaling, the optimal learning rate (LR) of batch size 4k should be 0.16. We use the poly learning rate policy (Ge et al., 2018), and the poly power is 2. The momentum is 0.9 and the weight decay is 0.0005.

Batch Size	Base LR	warmup	epochs	test accuracy
512	0.02	N/A	100	0.583
1k	0.02	no	100	0.582
4k	0.01	yes	100	0.509
4k	0.02	yes	100	0.527
4k	0.03	yes	100	0.520
4k	0.04	yes	100	0.530
4k	0.05	yes	100	0.531
4k	0.06	yes	100	0.516
4k	0.07	yes	100	**0.001**
...	...	...	...	...
4k	0.16	yes	100	**0.001**

For large batch training, it is essential to keep up the test accuracy with smaller batches under the constraint of the same number of epochs. Here we fix the number of epochs because: statistically, one epoch means the algorithm touches the entire dataset once; and computationally, fixing the number of epochs means fixing the number of floating point operations. State-of-the-art techniques for large batch training to remedy the test accuracy degradation issue include:

(1) **Linear Scaling** (Krizhevsky, 2014): With an increase of the batch size from B to kB, we should also increase the learning rate from η to $k\eta$.

(2) **Warmup Scheme** (Goyal et al., 2017): With a large learning rate (η), we should start from a small η and increase it to the large η in the first few epochs.

The intuition of linear scaling is related to the number of iterations (Krizhevsky, 2014;

Goyal et al., 2017). Let us use B, η, and I to denote the batch size, the learning rate, and the number of iterations. If we increase the the batch size from B to kB, then the number of iterations is reduced from I to I/k. This indicates that the frequency of weight updating is reduced by k times. Thus, we make the updating of each iteration $k\times$ more efficient by enlarging the learning rate by k times. The purpose of a warmup scheme is to avoid the situation in which the algorithm diverges at the beginning because we have to use a very large learning rate based on linear scaling. With these techniques, researchers can use the relatively large batch in a certain range (Table 5.5). However, we observe that these state-of-the-art approaches can only scale batch size to 1k for AlexNet and 8k for ResNet-50. With the batch size of 4k for AlexNet, we can only achieve a 53.1% test accuracy in 100 epochs (Table 5.4). Our target is to preserve the 58% test accuracy even when using large batch sizes, such as 32k.

Table 5.5: State-of-the-art Large Batch Training and Test Accuracy. Batch1 means baseline batch size. Batch2 means large batch size. Acc1 means baseline accuracy. Acc2 means large-batch accuracy.

Team	Model	Batch1	Batch2	Acc1	Acc2
Google (Krizhevsky, 2014)	AlexNet	128	1024	57.7%	56.7%
Amazon (Mu Li, 2017)	ResNet101	256	5120	77.8%	77.8%
Facebook (Goyal et al., 2017)	ResNet50	256	8192	76.30%	76.26%

Scaling up Batch Size

To improve the accuracy for large batch training, we developed a new rule of learning rate (LR) schedule (You, Gitman, and Ginsburg, 2017). As discussed in §5.2, we use $w = w - \eta\nabla w$ to update the weights. Each layer has its own weight w and gradient ∇w. Standard SGD algorithm uses the same LR (η) for all the layers. However, from our experiments, we observe that different layers may need different LRs. The reason is that the ratio between $||w||_2$ and $||\nabla w||_2$ varies significantly for different layers. From example, we observe that $||w||_2/||\nabla w||_2$ is only 20 for conv1.1 layer (Table 5.6). However, $||w||_2/||\nabla w||_2$ is 3,690 for fc6.1 layer. To speed up the convergence for fc6.1 layer, the users need to use a large LR. However, this large LR may lead to divergence on the conv1.1 layer. We believe this is an important reason for the optimization difficulty in large batch training.

Goyal et al. (Goyal et al., 2017) proposed the warmup scheme to solve this problem. The warmup scheme works well for ResNet-50 training with a batch size $\leq$ 8k. However, only using this recipe does not work for AlexNet with batch size $>$ 10k and ResNet-50 with batch size $>$ 8k.

Together with researchers at Nvidia, we proposed the Layer-wise Adaptive Rate Scaling (LARS) algorithm (You, Gitman, and Ginsburg, 2017) to improve large batch training's test accuracy. The base LR rule is defined in Equation (5.1). l is the scaling factor, which we set

Table 5.6: The ratios between $||w||_2$ and $||\nabla w||_2$ for different layers of AlexNet with batch size = 4k at the 1st epoch. We observe that they are very different from each other. fc is the fully connected layer and conv is the convolutional layer. x.0 is a layer's weight. x.1 is a layer's bias.

Layers	$\|\|w\|\|_2$	$\|\|\nabla w\|\|_2$	$\|\|w\|\|_2/\|\|\nabla w\|\|_2$
fc8.0	20.24	0.078445	258
fc8.1	0.316	0.006147	51
fc7.0	20.48	0.110949	184
fc7.1	6.400	0.004939	1296
fc6.0	30.72	0.097996	314
fc6.1	6.400	0.001734	**3690**
conv5.0	6.644	0.034447	193
conv5.1	0.160	0.000961	166
conv4.0	8.149	0.039939	204
conv4.1	0.196	0.000486	403
conv3.0	9.404	0.049182	191
conv3.1	0.196	0.000511	384
conv2.0	5.545	0.057997	96
conv2.1	0.160	0.000649	247
conv1.0	1.866	0.071503	26
conv1.1	0.098	0.004909	**20**

as 0.001 for AlexNet and ResNet training. γ is a tuning parameter for users. Usually γ can be chosen by linear scaling.

$$\eta = l \times \gamma \times \frac{||w||_2}{||\nabla w||_2} \tag{5.1}$$

In this formulation, different layers can have different LRs. In practice, we add momentum (denoted as μ) and weight decay (denoted as β) to SGD, and use the following sequence for LARS:

(1) get the local LR for each learnable parameter by $\alpha = l \times ||w||_2/(||\nabla w||_2 + \beta||\nabla w||_2)$;
(2) get the LR for each layer by $\eta = \gamma \times \alpha$;
(3) update the gradients by $\nabla w = \nabla w + \beta w$;
(4) update acceleration term a by $a = \mu a + \eta \nabla w$;
(4) update the weights by $w = w - a$.

Using this approach together with the warmup technique, SGD with a large batch can achieve identical test accuracy with the batch size of 32k as the baseline for AlexNet (Table 5.7). Technically, we change the local response normalization (LRN) to batch normalization (BN). We add BN after each convolutional layer. As shown in Figure 5.8, we can see that the LARS

algorithm can keep up the test accuracy for ResNet-50 using 32k batch with the baseline of ∼73% without data augmentation. In comparison, the current approaches of combining linear scaling and warmup has lower accuracy on ResNet-50 for batch size of 16k and 32k (68% and 56%, respectively).

Table 5.7: Test Accuracy of AlexNet with Batch Size of 32k using KNL Nodes on Stampede2. We use poly learning rate policy, and the poly power is 2. The momentum is 0.9 and the weight decay is 0.0005. For a batch size of 32K, we changed local response norm in AlexNet to batch norm. Specifically, we use the refined AlexNet model by Boris Ginsburg.

Batch Size	LR rule	warmup	Epochs	test accuracy
512	regular	N/A	100	0.583
4096	LARS	13 epochs	100	0.584
8192	LARS	8 epochs	100	0.583
32768	LARS	5 epochs	100	0.585

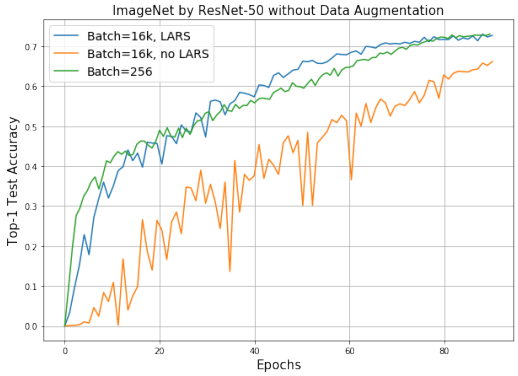


Figure 5.7: **Batch Size=16k**. Test accuracy comparison between Large-Batch Training, Large Batch Training with LARS, and the Baseline. The base learning rate of Batch 256 is 0.2 with poly policy (power=2). For the version without LARS, we use the state-of-the-art approach (Goyal et al., 2017): 5-epoch warmup and linear scaling for LR. For the version with LARS, we also use 5-epoch warmup. Clearly, the existing method does not work for Batch Size larger than 8K. LARS algorithm can help the large batch to achieve the same accuracy with baseline in the same number of epochs.

5.4 Performance Evaluation

In this section, we evaluate the 100-epoch AlexNet and 90-epoch ResNet-50 training on a number of platforms. We will briefly introduce the hardware and software settings, and

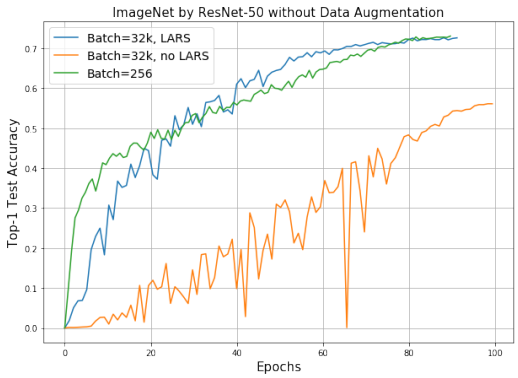


Figure 5.8: **Batch Size=32k**. Test accuracy comparison between Large Batch Training, Large Batch Training with LARS, and the Baseline. The base learning rate of Batch 256 is 0.2 with poly policy (power=2). For the version without LARS, we use the state-of-the-art approach (Goyal et al., 2017): 5-epoch warmup and linear scaling for LR. For the version with LARS, we also use 5-epoch warmup. Clearly, the existing method does not work for Batch Size larger than 8K. LARS algorithm can help the large batch to achieve the same accuracy with baseline in the same number of epochs.

present the comparison baseline and code change. In addition to performance results, we will also discuss the communication overhead analysis.

System Configuration

Throughout this section, we run experiments on four types of hardware. The Intel Xeon Phi 7250 Processors and the Xeon Platinum 8160 processors are part of the Stampede2 supercomputer hosted at Texas Advanced Computing Center[6]. The eight Nvidia P100 GPU server is locally hosted. We also use the cloud TPUs.

We perform the large batch scaling efficiency study on the eight Nvidia P100 GPU server. Each P100 GPU has the performance of 10.6 teraflops and has 16 GB memory. The Intel Xeon Phi 7250 Processor (referred to as KNL) is the latest version of Intel's general-purpose accelerator. It is a self-hosted platform running CentOS 7 on our testbed. Each processor has 68 physical cores, and four hardware threads per core. All cores are running at 1.4 GHz clock rate. On Stampede2, 3,696 out of the total 4,200 KNL nodes are configured in the following way: On each node, there is one processor with 96 GB DDR4 RAM, 16 GB MCDRAM, and a 200 GB local Solid State Drive, of which 144 GB is available. The memory is configured as cache-quadrant mode, where MCDRAM is used as an L3 cache. The Intel Xeon Platinum 8160 processors (referred to as SKX) are part of Intel Xeon Scalable Processors collection. Each SKX node in Stampede2 has two such processors with 48 physical cores in total. Each

[6]portal.tacc.utexas.edu/user-guides/stampede2

core is documented to have a 2.1 GHz clock rate, however, the clock rate varies from 1.4 GHz to 3.7 GHz depending on the instruction set and the number of active cores. Our measured runtime clock rate for AlexNet and ResNet-50 are 2.1 GHz and 2.0 GHz, respectively. Each SKX node is equipped with 192 GB RAM and a 200 GB Solid State Drive, of which 144 GB is available. There are 1,600 SKX nodes (3,200 processors) on Stampede2.

On Nvidia GPUs, we used the Nvidia distribution of Caffe[7]. And on Intel processors, we used two variants of the official Caffe (Y. Jia et al., 2014): 1) our customized parallel implementation that uses MPI (Gropp et al., 1996) for the communication across nodes and 2) the Intel distribution of Caffe v1.0.3[8], which supports multi-node training by Intel Machine Learning Scaling Library (MLSL) v2017.1.016[9].

Data and Baseline

Throughout the performance evaluation, we use the ImageNet-1k (Deng et al., 2009) dataset. The dataset has 1.28 million images for training and 50,000 images for testing. There are two top-1 test accuracy baselines for ResNet-50 in 90 epochs: the case without data augmentation is 73% while the case with data augmentation is 75.3%. The top-1 test accuracy baseline for AlexNet in 100 epochs is about 58%. We also got the same ResNet-50 baseline used by Goyal et al., 2017. The learning rate is 0.1 for a batch size of 256. The weight decay is 1e-4. The momentum is 0.9. The batch norm decay is 0.9. The batch norm epsilon is 1e-5. The label smoothing parameter used in the softmax-cross-entropy is 0.0. We got 76.3% top-1 accuracy from this baseline. The clipping threshold for the trust-ratio of LARS solver is [0.0, 50.0]. The clipping threshold for the gradient of LARS solver is [-10.0, 10.0][10].

Data Shuffling in Distributed Training

For the extremely large dataset, our approach is to partition the full data set on all nodes and configure the data layer to draw a different subset on each node. Since each node has its own unique randomizing seed in this situation, it will effectively draw a unique image subset. If the dataset is not large (e.g. ImageNet), we just copy the full dataset on all nodes. Like Goyal et al., we use a single random shuffling of the training data (per epoch) that is divided amongst all the nodes.

Scaling Efficiency of Large Batches

As discussed in §5.3, using large batch size can reduce the communication volume with fewer iterations, thus yielding higher scaling efficiency than small batch size. Here, we present an analytical study and the empirical performance evaluation to validate this hypothesis.

[7]https://github.com/NVIDIA/caffe
[8]https://github.com/intel/caffe
[9]https://github.com/intel/MLSL
[10]https://www.cs.berkeley.edu/~youyang/lars_optimizer.py

Communication often is the major bottleneck for efficient scaling for applications across many processors (Table 5.8). On a distributed system, communication means moving the data over the network (e.g. master machine broadcast its data to all the worker machines). In DNN training, communication across nodes is in the form of a all-reduce (sum of local gradients). These communication patterns have a higher scaling overhead than the matrix computations (i.e. the matrix computations on each machine can be finished independently). In particular, the all-reduce on N nodes have the scaling factor of $O(\log N)$ or $O(N)$ depending on the network topology (Thakur, Rabenseifner, and Gropp, 2005; Rabenseifner, 2004; Rabenseifner and Träff, 2004; van de Geijn, 1994). The scaling factor of broadcast is $O(\log N)$. In contrast, the matrix computation in DNN training can be distributed almost evenly to N nodes with the scaling factor of $O(1/N)$.

For finishing the same number of epochs, the communication overhead is lower in the large batch version than in the small batch version, as the large batch version sends fewer messages (latency overhead) and moves less data (bandwidth overhead). For synchronous SGD, the algorithm needs to conduct an all-reduce operation (sum of gradients on all machines) in each iteration. The number of messages sent is linear with the number of iterations. And for each iteration, the communication volume is constant regardless of the batch size, as the gradients have identical size as the model weights ($|W|$).

Let us use the following notations for the analytical evaluation:

E the number of epochs

n the total number of images in the training dataset

B the batch size

Then the number of iterations is $E \times n/B$. Holding E and n constant, with the large batch size, the program finishes with fewer iterations. By fixing E, the number of epochs, it is fixing the total number of floating point operations. Meanwhile, the number of iterations is consistent with the communication frequency of the training process. Let us denote $|W|$ as the neural network model size. Then we can get the communication volume is $|W| \times E \times n/B$.

Thus, the large batch version transfers less data than the small batch version to finish the same number of floating point operations. As mentioned before, the number of floating point operations remain constant when the number of epochs is fixed. The larger batch size increases the computation-communication ratio because it reduces the communication frequency. As a result, the larger batch size makes the algorithm more scalable on distributed systems.

For the empirical performance evaluation, we use the ImageNet-1k training with AlexNet-BN on eight P100 GPUs in this experiment. Here, AlexNet-BN means changing the local response normalization in AlexNet to batch normalization. The baseline's batch size is 512 and is referred to as the small batch size. The large batch size is 4k. In this example, we focus on the the communication across GPUs. Firstly, the experimental results confirm that the large batch size achieves the same test accuracy as the small batch size in 100 epochs

Table 5.8: Communication unit is much slower than computation unit because time-per-flop (γ) $\ll$ 1/ bandwidth (β) $\ll$ latency (α). For example, $\gamma = 0.9 \times 10^{-13}$s for NVIDIA P100 GPUs.

Network	α (latency)	β (1/bandwidth)
Mellanox 56Gb/s FDR IB	0.7×10^{-6}s	0.2×10^{-9}s
Intel 40Gb/s QDR IB	1.2×10^{-6}s	0.3×10^{-9}s
Intel 10GbE NetEffect NE020	7.2×10^{-6}s	0.9×10^{-9}s

(Figure 5.9). Thus, large batch size achieves the same test accuracy as the small batch size in fixed number of floating point operations (Figure 5.10).

We observed a 3x reduction in training time with the large batch size compared to that of the small batch size, as shown in Figure 5.11. The number of iterations in large-batch training is much less than small-batch training (Figure 5.12). The number of messages is equal to the number of iterations. Our experimental results also confirmed that large batch will reduce the accumulated latency overhead (Figure 5.13) and bandwidth overhead (Figure 5.14).

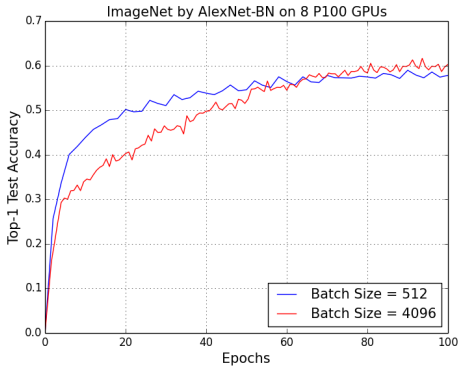


Figure 5.9: Test Accuracy Comparison between the Small Batch Size and the Large Batch Size. The 512 small batch size is the baseline.

ImageNet training with AlexNet

In this experiment, we use the AlexNet training case to evaluate our approach's effectiveness in scaling DNN training at large scale. Previously, Nvidia reported that using one DGX-1 station they were able to finish 90-epoch ImageNet-1k training with AlexNet in two hours[11]. However, they used half-precision or FP16, whose cost is half of the standard single-precision operation. We run the AlexNet training with standard single-precision. It takes 6 hours 9 minutes with the batch size of 512 on one NVIDIA DGX-1 station. With our approach, using

[11] www.nextplatform.com/2016/04/06/dgx-1-nvidias-deep-learning-system-newbies

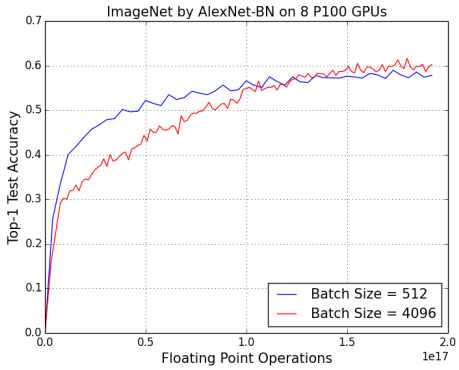


Figure 5.10: Increasing the batch size does not increase the number of floating point operations. Large batch can achieve the same accuracy in the fixed number of floating point operations.

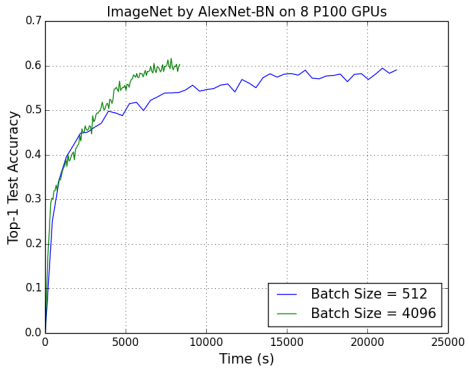


Figure 5.11: Time-to-solution Comparison between the Small Batch Size and the large Batch Size. To achieve the 58% accuracy, the large batch size of 4k only needs about two hours while the smaller batch size of 512 needs about six hours.

the large batch size of 4k achieves similar test accuracy as the small batch size case (Line 2 in Table 5.7), and it finishes in two hours and ten minutes on the same station. Thus, using large batch size can significantly speedup DNN training on GPU cluster.

Then we scale the same AlexNet training case with a batch size of 32k, and run it on multiple scales of the KNL nodes and the SKX nodes on the Stampede2 supercomputer. Figure 5.15 and 5.16 show the strong scaling performance on each type of nodes with the ideal scaling curve relative to the performance of 128 nodes in each case.

Despite the 11-minute training time on 1,024 SKX nodes, the AlexNet training does not scale well beyond 512 nodes in both cases. The inefficient scaling performance is due to its low scaling ratio, as defined in §5.3. On the other hand, on 512 KNL nodes, the AlexNet

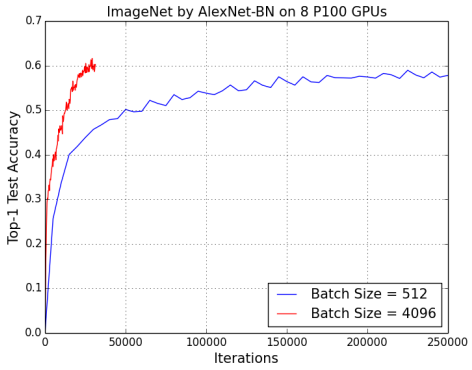


Figure 5.12: When we fix the number of epochs and increase the batch size, we need much fewer iterations.

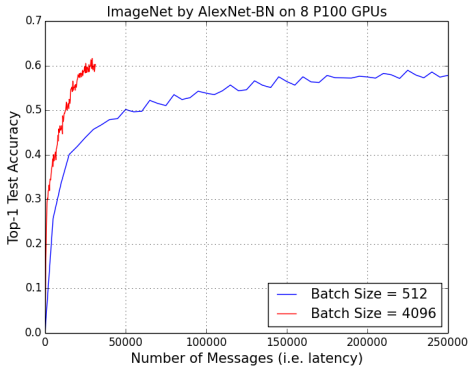


Figure 5.13: When we fix the number of epochs and increase the batch size, we need much fewer iterations. The number of iterations is linear with the number of messages the algorithm sent.

training finished in 24 minutes. The minute-level training performance is remarkable given the current practice, a comparison against known performance is presented in Table 5.9.

ImageNet training with ResNet-50

In this experiment, we use the ResNet-50 training case to evaluate the effectiveness of our approach in scalable DNN training. We use the ResNet-50 training on the ImageNet-1k dataset for 90 epochs with the batch size of 32k as the test case, run it at multiple scales on the KNL and SKX nodes on Stampede2, then compare the test accuracy and time-to-solution to the published results.

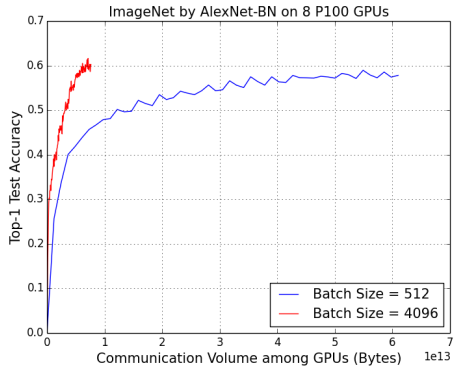


Figure 5.14: The number of iterations is linear with the number of messages the algorithm sent. Let us denote $|W|$ as the neural network model size. Then we can get the communication volume is $|W| \times E \times n/B$. Thus, the larger batch version needs to move much less data than the smaller batch when they finish the number of floating point operations.

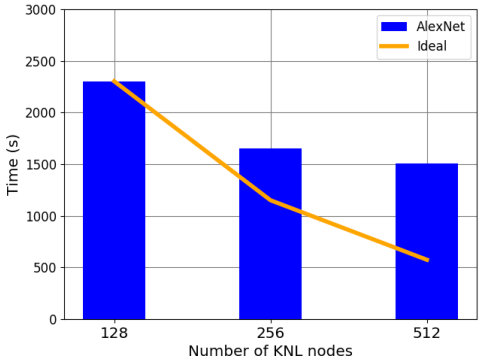


Figure 5.15: Strong Scaling Performance of AlexNet with 32k Batch Size on KNL Nodes

Figure 5.17 and 5.18 show the strong scaling performance of the ResNet-50 case at various scales. Compared to AlexNet, ResNet-50 scales more efficiently to 1,024 KNL nodes and 1,600 SKX nodes. This is because ResNet-50 has a relatively higher scaling ratio. In particular, the ResNet-50 case finishes in 32 minutes on 1,600 SKX nodes (3,200 Intel Xeon Platinum 8160 processors) and 20 minutes on 2,048 KNL nodes (2,048 Intel Xeon Phi 7250 processors).

A comprehensive result comparison against existing results is presented in Table 5.10. Codreanu *et al.* reported their experience on using Intel KNL clusters to speed up ImageNet-1k training by a blogpost[12]. They reported a 73.78% accuracy (with data augmentation) in less than 40 minutes on 512 KNL nodes with the batch size of 8k. However, this case only ran for

[12]https://blog.surf.nl/en/imagenet-1k-training-on-intel-xeon-phi-in-less-than-40-minutes/

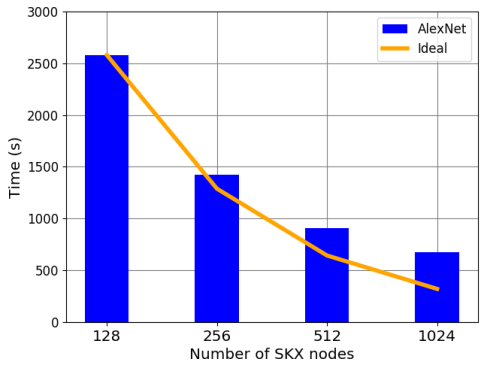


Figure 5.16: Strong Scaling Performance of AlexNet with 32k Batch Size on SKX Nodes

Table 5.9: Time-to-solution Comparison against Other Published Performance

Batch Size	epochs	Accuracy	hardware	time
256	100	58.7%	CPU + K20 GPU	144h
512	100	58.8%	DGX-1 station	6h 10m
4096	100	58.4%	DGX-1 station	2h 19m
32768	100	58.5%	512 KNLs	24m
32768	100	58.6%	1024 CPUs	11m

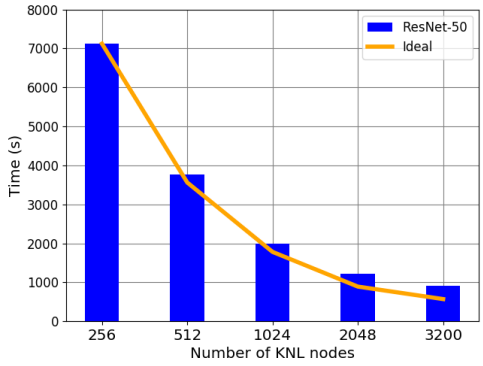


Figure 5.17: Strong Scaling Performance of ResNet-50 with 32k Batch Size on KNL Nodes

37 epochs. The complete 90-epoch training would take 80 minutes with a 75.25% accuracy.

Based on the original ResNet-50 model (K. He et al., 2016), we added data augmentation to our baseline. Our baseline is 75.3% top-1 test accuracy in 90 epochs. The model we used is available upon request. The test accuracy comparison is shown in Table 5.11. Although our baseline's accuracy is lower than Goyal et al., we achieve a correspondingly higher accuracy with batch sizes that are greater than 10k. When we use the same baseline, our results are

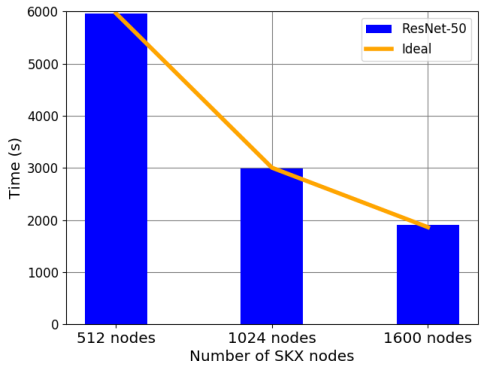


Figure 5.18: Strong Scaling Performance of ResNet-50 with 32k Batch Size on SKX Nodes

Table 5.10: ResNet-50 Result. The DA means Data Augmentation. The result of 76.4% accuracy is achieved by ResNet-50-v2.

Batch	DA	epochs	accuracy	hardware	time
256	NO	90	73.0%	DGX-1 station	21h
256	YES	90	75.3%	16 KNLs	45h
8k	NO	90	72.7%	DGX-1 station	21h
8k	NO	90	72.7%	256 P100 GPUs	1h
8k	YES	90	75.3%	256 P100 GPUs	1h
16k	YES	90	75.3%	1024 SKX nodes	52m
16k	YES	90	75.3%	1600 SKX nodes	31m
32k	NO	90	72.6%	512 KNL nodes	1h
32k	YES	90	75.4%	512 KNL nodes	1h
32k	YES	90	75.4%	1024 SKX nodes	48m
32k	YES	90	74.2%	1600 SKX nodes	32m
32k	YES	90	75.4%	2048 KNL nodes	20m
32k	YES	90	76.4%	2048 KNL nodes	20m

better than Goyal et al. for all the batch sizes (Fig. 5.20).

Superlinear Speedup

For AlexNet training on eight NVIDIA P100 GPUs, we can achieve three times speedup. In this situation, the baseline and our approach used the same hardware. For ResNet-50 on distributed systems, the speedup is extremely large. Because the baseline's workload is limited to 256 samples at each iteration, it can only fully utilize 16 KNL nodes (Intel Xeon Phi 7250) or 1088 cores. Now our approach is able to process 32,768 samples each iteration, we can use 2048 KNL nodes or 139,264 cores. The perfect speedup is 139264/1088 = 128×. But we achieved a superlinear speedup of 45h/20m = 135×. The reason is that our communication overhead is much lower than the baseline (fewer messages and less volume of data moved

Table 5.11: Comparison by 90-epoch ResNet50 Accuracy. DA means Data Augmentation

Batch Size	256	8K	16K	32K	64K	DA
MSRA	75.3%	75.3%	—	—	—	weak
IBM	—	75.0%	—	—	—	—
SURFsara	—	75.3%	—	—	—	—
Facebook	76.3%	76.2%	75.2%	72.4%	66.0%	heavy
Our	73.0%	72.7%	72.7%	72.6%	70.0%	no
Our	75.3%	75.3%	75.3%	75.4%	73.2%	weak
Our	76.3%	76.6%	76.7%	76.4%	75.1%	heavy

over network), which is shown in Fig. 5.13 and Fig. 5.14. On the other hand, the single-node efficiency of our approach is higher (Fig. 5.6).

Implementation of LARS on Cloud TPUs

Most of the emerging computer architectures support low-precision floating-point formats such as float16 format and bfloat16 format. The illustration of bfloat16 and float16 are shown in Fig. 5.19. As introduced in §5.2, we are interested in evaluating Google's TPU architecture. TPU's designers picked bfloat16 to support the low-precision format. In our experiments, we confirm that using bfloat16 rather than float16 is a good choice because most of the deep learning applications do not need a high precision to get the target application accuracy. To reduce the storage overhead and speed up the floating point operations, we implement our large-batch algorithm in bfloat16. All the weights, gradients, data, and activations in the deep neural networks training are represented in bfloat16 format. However, to avoid accuracy loss or even divergence, we store the hyper-parameters (e.g. learning rate, weight decay) into 32-bit single precision format. It is worth noting we only have fewer than ten hyper-parameters. Thus, the computation and storage costs of hyper-parameters are trivial. The computation and storage costs are dominated by the large matrices like weights, gradients, data, and activations. We tried a series of applications like AlexNet, GoogleNet, ResNet, and LSTM (using our large-batch algorithm), we confirm that we can achieve the results with the 32-bit single-precision CPU implementation. We write our code based on TensorFlow framework. We did some optimizations to reduce the synchronous communication overhead and improve the single-node performance.

The main deep neural networks are based on CNN structures and RNN structures. For CNN applications, we picked ImageNet training with ResNet-50. For RNN applications, we picked GNMT. We finished the Imagenet training with ResNet-50 on a cloud TPU (v2-8) in 7 hours and 50 minutes. Since one image requires 7.7 billion operations and the dataset has 1.28 million images, 90-epoch ImageNet training with ResNet-50 needs 8.87×10^{17} ($90 \times 1.28 \times 10^6 \times 7.7 \times 10^9$) operations. Thus, we achieved 31.5 TFlops on a cloud TPU for ResNet-50 training. LARS is an efficient algorithm that helps to improve the floating-point performance.

For GNMT, we are able to scale the batch size to 4K on a cloud TPU and get a BLEU

score of 22.2. The application is out of memory on a cloud TPU when we increase the batch size more than 4K. So we stop the batch size scaling at 4K. The speed is 3637 samples per second. The sequence length of the data is 150, which means the eight LSTM layers are unrolled to 1200 layers. Each GNMT layer is a 2048-by-4096 matrix and the input data is a 1-by-2048 vector. Thus, one input data sample requires 1200 matrix-vector multiplications. We achieved 33.3 TFlops on a cloud TPU for GNMT training. The baseline uses a batch size of 256, which only achieves the speed of 997.8 samples per second. Thus, we achieved around 4 times speedup on the same hardware.

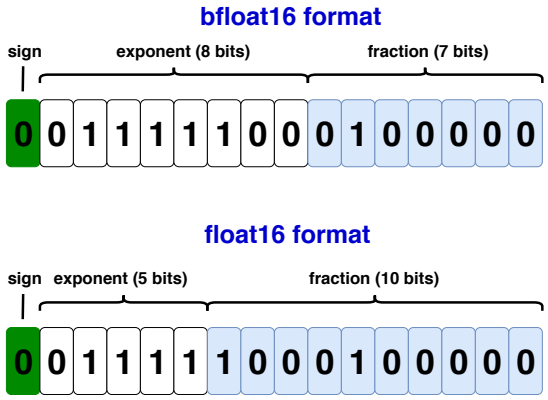


Figure 5.19: The difference between bfloat16 and float16. Compared to float16 format, bfloat16 format has a wider range but a lower precision. bfloat16 format was supported in Google's TPU.

Compared to State-of-the-art Implementation

To further improve our accuracy, we did some additional optimizations in our implementation. We implemented the strong data augmentation that was used in the system developed by Goyal et al., 2017. In this way, our baseline achieved the same accuracy with Goyal et al., 2017. By using the gradient clipping technique (Pascanu, Mikolov, and Bengio, 2013), we are able to achieve a higher accuracy for the baseline. Specifically, we use a mix of constant gradient clip and norm gradient clipping. We used label smoothing technique to improve the accuracy from 75.9 to 76.4 for batch size = 32K. Compared to the state-of-the-art implementation by Goyal et al., 2017, our approach achieves a higher accuracy. Our approach has more consistent accuracy across batch sizes (Fig. 5.20).

5.5 Conclusion

In conclusion, we explore the large batch size approach to enable scalable DNN training on large scale computers. We examine the benefits and the challenges of this approach, and

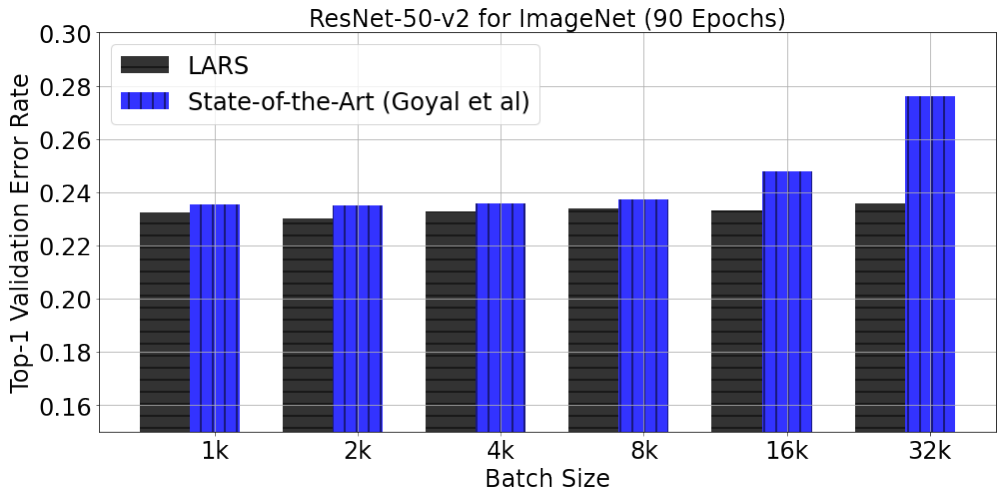


Figure 5.20: Compared to Goyal et al., 2017, our approach achieves a higher accuracy. Our approach has more consistent accuracy across batch sizes. Here, both Goyal et al. and we use the same ResNet-50 baseline (i.e. ResNet-50-v2).

incorporate the LARS algorithm as the solution. We evaluate the implementation with the ImageNet-1k dataset and two neural network models of AlexNet and ResNet-50 at scale for the efficiency, test accuracy, training speed, and solution generality. Our solution is able to keep up with the baseline test accuracy for both test cases within the same number of epochs. We are able to reduce the ImageNet-1k training time from hours to minutes: With 1,024 SKX nodes, the AlexNet case finished in 11 minutes. While with 2,048 KNL nodes, the ResNet-50 case finished in 20 minutes. Our solution is general to be effective for both the AlexNet and ResNet-50 cases. We showcase large scale computers' capability in accelerating DNN training with massive computing resource with standard ImageNet-1k based benchmark. For GNMT, we achieved around 4 times speedup on the same hardware. We believe this is a pilot use case to motivate future DNN based research on large scale computers across domains in both industry and academia.

Chapter 6

Fast BERT Pre-Training

6.1 Introduction

With the advent of large scale datasets, training large deep neural networks, even using computationally efficient optimization methods like Stochastic gradient descent (*SGD*), has become particularly challenging. For instance, training state-of-the-art deep learning models like BERT and ResNet-50 takes 3 days on 16 TPUv3 chips and 29 hours on 8 Tesla P100 gpus respectively (Devlin et al., 2018; K. He et al., 2016). Thus, there is a growing interest to develop optimization solutions to tackle this critical issue. The goal of this chapter is to investigate and develop optimization techniques to accelerate training large deep neural networks, mostly focusing on approaches based on variants of SGD.

Methods based on SGD iteratively update the parameters of the model by moving them in a scaled (negative) direction of the gradient calculated on a minibatch. However, SGD's scalability is limited by its inherent sequential nature. Owing to this limitation, traditional approaches to improve SGD training time in the context of deep learning largely resort to distributed asynchronous setup (J. Dean et al., 2012a; Recht et al., 2011). However, the implicit staleness introduced due to the asynchrony limits the parallelization of the approach, often leading to degraded performance. The feasibility of computing gradient on *large minibatches* in parallel due to recent hardware advances has seen the resurgence of simply using synchronous SGD with large minibatches as an alternative to asynchronous SGD. However, naïvely increasing the batch size typically results in degradation of generalization performance and reduces computational benefits (Goyal et al., 2017).

Synchronous SGD on large minibatches benefits from reduced variance of the stochastic gradients used in SGD. This allows one to use much larger learning rates in SGD, typically of the order square root of the minibatch size. Surprisingly, recent works have demonstrated that up to certain minibatch sizes, linear scaling of the learning rate with minibatch size can be used to further speed up the training Goyal et al., 2017. These works also elucidate two interesting aspects to enable the use of linear scaling in large batch synchronous SGD: (i) linear scaling of learning rate is harmful during the initial phase; thus, a hand-tuned warmup strategy of slowly increasing the learning rate needs to be used initially, and (ii) linear scaling

of learning rate can be detrimental beyond a certain batch size. Using these tricks, Goyal et al., 2017 was able to drastically reduce the training time of ResNet-50 model from 29 hours to 1 hour using a batch size of 8192. While these works demonstrate the feasibility of this strategy for reducing the wall time for training large deep neural networks, they also highlight the need for an adaptive learning rate mechanism for large batch learning.

Variants of SGD using layerwise adaptive learning rates have been recently proposed to address this problem. The most successful in this line of research is the LARS algorithm (You, Gitman, and Ginsburg, 2017), which was initially proposed for training ResNet and discussed in Chap 5. Using LARS, ResNet-50 can be trained on ImageNet in just a few minutes! However, it has been observed that its performance gains are *not* consistent across tasks. For instance, LARS performs poorly for attention models like BERT. Furthermore, theoretical understanding of the adaptation employed in LARS is largely missing. To this end, we study and develop new approaches specially catered to the large batch setting of our interest.

Contributions. More specifically, we make the following main contributions in this chapter.

- Inspired by LARS, we investigate a general adaptation strategy specially catered to large batch learning and provide intuition for the strategy.
- Based on the adaptation strategy, we develop a new optimization algorithm (LAMB) for achieving adaptivity of learning rate in SGD. Furthermore, we provide convergence analysis for both LARS and LAMB to achieve a stationary point in nonconvex settings. We highlight the benefits of using these methods for large batch settings.
- We propose the Linear Epoch Gradual Warmup (LEGW) scheme, which can potentially avoid all the hyper-parameter tuning effort. In our experiments, we observe that LEGW performs well with LAMB in ResNet and BERT training.
- We demonstrate the strong empirical performance of LAMB across several challenging tasks. Using LAMB we scale the batch size in training BERT to more than 32k without degrading the performance, thereby cutting the time down from 3 days to 76 minutes. Ours is the first work to reduce BERT training wall time to less than a couple of hours.
- We also demonstrate the efficiency of LAMB for training state-of-the-art image classification models like ResNet. To the best of our knowledge, ours is first adaptive solver that can achieve state-of-the-art accuracy for ResNet-50 as adaptive solvers like Adam fail to obtain the accuracy of SGD with momentum for these tasks.

This chapter is based on a joint work with Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. It was published as a conference paper entitled *Large Batch Optimization for Deep Learning: Training BERT in 76 minutes* (You, J. Li, et al., 2019).

Related Work

The literature on optimization for machine learning is vast and hence, we restrict our attention to the most relevant works here. Earlier works on large batch optimization for machine learning mostly focused on convex models, benefiting by a factor of square root of batch size using appropriately large learning rate. Similar results can be shown for nonconvex settings wherein using larger minibatches improves the convergence to stationary points; albeit at the cost of extra computation. However, several important concerns were raised with respect to generalization and computational performance in large batch nonconvex settings. It was observed that training with extremely large batch was difficult (Keskar et al., 2016; Hoffer, Hubara, and Soudry, 2017). Thus, several prior works carefully hand-tune training hyper-parameters, like learning rate and momentum, to avoid degradation of generalization performance (Goyal et al., 2017; Mu Li, 2017; You, Z. Zhang, Cho-Jui Hsieh, et al., 2018; Shallue et al., 2018).

(Krizhevsky, 2014) empirically found that simply scaling the learning rate linearly with respect to batch size works better up to certain batch sizes. To avoid optimization instability due to linear scaling of learning rate, Goyal et al. (2017) proposed a highly hand-tuned learning rate which involves a warm-up strategy that gradually increases the LR to a larger value and then switching to the regular LR policy (e.g. exponential or polynomial decay). Using LR warm-up and linear scaling, Goyal et al. (2017) managed to train ResNet-50 with batch size 8192 without loss in generalization performance. However, empirical study (Shallue et al., 2018) shows that learning rate scaling heuristics with the batch size do not hold across all problems or across all batch sizes.

More recently, to reduce hand-tuning of hyper-parameters, adaptive learning rates for large batch training garnered significant interests (Reddi, Kale, and Sanjiv Kumar, 2018; Zaheer et al., 2018; J. Zhang et al., 2019). Several recent works successfully scaled the batch size to large values using adaptive learning rates without degrading the performance, thereby, finishing ResNet-50 training on ImageNet in a few minutes (You, Z. Zhang, Cho-Jui Hsieh, et al., 2018; Iandola et al., 2016; Codreanu, Podareanu, and Saletore, 2017; Akiba, Suzuki, and Fukuda, 2017; X. Jia et al., 2018; Smith, Kindermans, and Le, 2017; Martens and Grosse, 2015; Devarakonda, Naumov, and Garland, 2017; Mikami et al., 2018; Osawa et al., 2018; You, Hseu, et al., 2019; Yamazaki et al., 2019). To the best of our knowledge, the fastest training result for ResNet-50 on ImageNet is due to Ying et al., 2018, who achieve 76+% top-1 accuracy. By using the LARS optimizer and scaling the batch size to 32K on a TPUv3 Pod, Ying et al. (2018) was able to train ResNet-50 on ImageNet in 2.2 minutes. However, it was empirically observed that none of these performance gains hold in other tasks such as BERT training (see Section 6.4).

6.2 Preliminaries

Notation. For any vector $x_t \in \mathbb{R}^d$, either $x_{t,j}$ or $[x_t]_j$ are used to denote its j^{th} coordinate where $j \in [d]$. Let $\mathbb{I}$ be the $d \times d$ identity matrix, and let $\mathbb{I} = [\mathbb{I}_1, \mathbb{I}_2, ..., \mathbb{I}_h]$ be its decomposition

into column submatrices $\mathbb{I}_i = d \times d_h$. For $x \in \mathbb{R}^d$, let $x^{(i)}$ be the block of variables corresponding to the columns of I_i i.e., $x^{(i)} = \mathbb{I}_i^\top x \in \mathbb{R}^{d_i}$ for $i = \{1, 2, \cdots, h\}$. For any function $f : \mathbb{R}^d \to \mathbb{R}$, we use $\nabla_i f(x)$ to denote the gradient with respect to $x^{(i)}$. For any vectors $u, v \in \mathbb{R}^d$, we use u^2 and u/v to denote elementwise square and division operators respectively. We use $\|.\|$ and $\|.\|_1$ to denote l_2-norm and l_1-norm of a vector respectively.

We start our discussion by formally stating the problem setup. In this chapter, we study nonconvex stochastic optimization problems of the form

$$\min_{x \in \mathbb{R}^d} f(x) := \mathbb{E}_{s \sim \mathbb{P}}[\ell(x, s)] + \frac{\lambda}{2}\|x\|^2, \tag{6.1}$$

where ℓ is a smooth (possibly nonconvex) function and $\mathbb{P}$ is a probability distribution on the domain $\mathcal{S} \subset \mathbb{R}^k$. Here, x corresponds to model parameters, ℓ is the loss function and $\mathbb{P}$ is an unknown data distribution.

We assume function $\ell(x)$ is L_i-*smooth* with respect to i^{th} block, i.e., there exists a constant L_i such that

$$\|\nabla_i \ell(x, s) - \nabla_i \ell(x + I_i \delta, s)\| \leq L_i \|\delta\|, \quad \forall\ x \in \mathbb{R}^d, \delta \in \mathbb{R}^{d_i} \text{ and } s \in \mathcal{S}, \tag{6.2}$$

for all $i \in [h]$. We use $L = (L_1, \cdots, L_h)^\top$ to denote the h-dimensional vector of Lipschitz constants. We use L_∞ and L_{avg} to denote $\max_i L_i$ and $\sum_i \frac{L_i}{h}$ respectively. We assume the following bound on the variance in stochastic gradients: $\mathbb{E}\|\nabla_i \ell(x, s) - \nabla_i f(x)\|^2 \leq \sigma_i^2$ for all $x \in \mathbb{R}^d$ and $i \in [h]$. Furthermore, we also assume $\mathbb{E}\|[\nabla \ell(x, s)]_i - [\nabla f(x)]_i\|^2 \leq \tilde{\sigma}_i^2$ for all $x \in \mathbb{R}^d$ and $i \in [d]$. We use $\sigma = (\sigma_1, \cdots, \sigma_h)^\top$ and $\tilde{\sigma} = (\tilde{\sigma}_1, \cdots, \tilde{\sigma}_d)^\top$ to denote the vectors of standard deviations of stochastic gradient per layer and per dimension respectively. Finally, we assume that the gradients are bounded i.e., $|[\nabla l(x, s)]_j| \leq G$ for all $j \in [d]$, $x \in \mathbb{R}^d$ and $s \in \mathcal{S}$. Note that such assumptions are typical in the analysis of stochastic first-order methods (cf. (Ghadimi and G. Lan, 2013a; Ghadimi, G. Lan, and Hongchao Zhang, 2014; Reddi, Hefny, et al., 2016; Reddi, Kale, and Sanjiv Kumar, 2018)).

Stochastic gradient descent (SGD) is one of the simplest first-order algorithms for solving problem in Equation 6.1. The update at the t^{th} iteration of SGD is of the following form:

$$x_{t+1} = x_t - \eta_t \frac{1}{|\mathcal{S}_t|} \sum_{s_t \in \mathcal{S}_t} \nabla \ell(x_t, s_t) + \lambda x_t, \tag{SGD}$$

where S_t is set of b random samples drawn from the distribution $\mathbb{P}$. For very large batch settings, the following is a well-known result for SGD.

Theorem 6.1 (Ghadimi and G. Lan, 2013b) With large batch $b = T$ and using appropriate learning rate, we have the following for the iterates of SGD:

$$\mathbb{E}\left[\|\nabla f(x_a)\|^2\right] \leq O\left(\frac{(f(x_1) - f(x^*))L_\infty}{T} + \frac{\|\sigma\|^2}{T}\right).$$

where x^* is an optimal solution to the problem in equation 6.1 and x_a is an iterate uniformly randomly chosen from $\{x_1, \cdots, x_T\}$.

However, tuning the learning rate η_t in SGD, especially in large batch settings, is difficult in practice. Furthermore, the dependence on L_∞ (the maximum of smoothness across dimension) can lead to significantly slow convergence. In the next section, we discuss algorithms to circumvent this issue.

6.3 Algorithms

In this section, we first discuss a general strategy to adapt the learning rate in large batch settings. Using this strategy, we discuss two specific algorithms in the later part of the section. Since our primary focus is on deep learning, our discussion is centered around training a h-layer neural network.

General Strategy. Suppose we use an iterative *base* algorithm $\mathcal{A}$ (e.g. SGD or Adam) in the small batch setting with the following layerwise update rule:

$$x_{t+1} = x_t + \eta_t u_t,$$

where u_t is the update made by $\mathcal{A}$ at time step t. We propose the following two changes to the update for large batch settings:

1. The update is normalized to unit l_2-norm. This is ensured by modifying the update to the form $u_t/\|u_t\|$. Throughout this chapter, such a normalization is done layerwise i.e., the update for each layer is ensured to be unit l_2-norm.

2. The learning rate is scaled by $\phi(\|x_t\|)$ for some function $\phi : \mathbb{R}^+ \to \mathbb{R}^+$. Similar to the normalization, such a scaling is done layerwise.

Suppose the base algorithm $\mathcal{A}$ is SGD, then the modification results in the following update rule:

$$x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|g_t^{(i)}\|} g_t^{(i)}, \tag{6.3}$$

for all layers $i \in [h]$ and where $x_t^{(i)}$ and $g_t^{(i)}$ are the parameters and the gradients of the i^{th} layer at time step t. The normalization modification is similar to one typically used in normalized gradient descent except that it is done layerwise. Note that the modification leads to a biased gradient update; however, in large-batch settings, it can be shown that this bias is small. It is intuitive that such a normalization provides robustness to exploding gradients (where the gradient can be arbitrarily large) and plateaus (where the gradient can be arbitrarily small). Normalization of this form essentially ignores the size of the gradient and is particularly useful in large batch settings where the direction of the gradient is largely preserved.

The scaling term involving ϕ ensures that the norm of the update is of the same order as that of the parameter. We found that this typically ensures faster convergence in deep neural networks. In practice, we observed that a simple function of $\phi(z) = \min\{\max\{z, \gamma_l\}, \gamma_u\}$ works well. It is instructive to consider the case where $\phi(z) = z$. In this scenario, the overall change in the learning rate is $\frac{\|x_t^{(i)}\|}{\|g_t^{(i)}\|}$, which can also be interpreted as an estimate on the inverse of Lipschitz constant of the gradient (see equation 6.2). We now discuss different instantiations of the strategy discussed above. In particular, we focus on two algorithms: LARS (6.3) and the proposed method, LAMB (6.3).

Algorithm 15 LARS

Input: $x_1 \in \mathbb{R}^d$, learning rate $\{\eta_t\}_{t=1}^T$, parameter $0 < \beta_1 < 1$, scaling function ϕ, $\epsilon > 0$

Set $m_0 = 0$
for $t = 1$ **to** T **do**
  Draw b samples S_t from $\mathbb{P}$
  Compute $g_t = \frac{1}{|\mathcal{S}_t|}\sum_{s_t \in \mathcal{S}_t} \nabla \ell(x_t, s_t)$
  $m_t = \beta_1 m_{t-1} + (1-\beta_1)(g_t + \lambda x_t)$
  $x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|m_t^{(i)}\|} m_t^{(i)}$ for all $i \in [h]$
end for

Algorithm 16 LAMB

Input: $x_1 \in \mathbb{R}^d$, learning rate $\{\eta_t\}_{t=1}^T$, parameters $0 < \beta_1, \beta_2 < 1$, scaling function ϕ, $\epsilon > 0$

Set $m_0 = 0$, $v_0 = 0$
for $t = 1$ **to** T **do**
  Draw b samples S_t from $\mathbb{P}$.
  Compute $g_t = \frac{1}{|\mathcal{S}_t|}\sum_{s_t \in \mathcal{S}_t} \nabla \ell(x_t, s_t)$.
  $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $m_t = m_t / (1-\beta_1^t)$
  $v_t = v_t / (1-\beta_2^t)$
  Compute ratio $r_t = \frac{m_t}{\sqrt{v_t}+\epsilon}$
  $x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|r_t^{(i)} + \lambda x_t^{(i)}\|}(r_t^{(i)} + \lambda x_t^{(i)})$
end for

LARS Algorithm

The first instantiation of the general strategy is LARS algorithm (You, Gitman, and Ginsburg, 2017), which is obtained by using momentum optimizer as the base algorithm $\mathcal{A}$ in the framework. LARS was earlier proposed for large batch learning for ResNet on ImageNet. In general, it is observed that the using (heavy-ball) momentum, one can reduce the variance in the stochastic gradients at the cost of little bias. The pseudocode for LARS is provide in Algorithm 15.

We now provide convergence analysis for LARS in general nonconvex setting stated in this chapter. For the sake of simplicity, we analyze the case where $\beta_1 = 0$ and $\lambda = 0$ in Algorithm 15. However, our analysis should extend to the general case as well. We will defer all discussions about the convergence rate to the end of the section.

Theorem 6.2 Let $\eta_t = \eta = \sqrt{\frac{2(f(x_1)-f(x^*))}{\alpha_u^2 \|L\|_1 T}}$ for all $t \in [T]$, $b = T$, $\alpha_l \le \phi(v) \le \alpha_u$ for all $v > 0$ where $\alpha_l, \alpha_u > 0$. Then for x_t generated using LARS (Algorithm 15), we have the following bound

$$\left(\mathbb{E}\left[\frac{1}{\sqrt{h}}\sum_{i=1}^{h}\|\nabla_i f(x_a)\|\right]\right)^2 \le O\left(\frac{(f(x_1)-f(x^*))L_{avg}}{T} + \frac{\|\sigma\|_1^2}{Th}\right),$$

where x^* is an optimal solution to the problem in equation 6.1 and x_a is an iterate uniformly randomly chosen from $\{x_1, \cdots, x_T\}$. The proof is in Section 6.6.

LAMB Algorithm

The second instantiation of the general strategy is obtained by using Adam as the base algorithm $\mathcal{A}$. Adam optimizer is popular in deep learning community and has shown to have

good performance for training state-of-the-art language models like BERT. Unlike LARS, the adaptivity of LAMB is two-fold: (i) per dimension normalization with respect to the square root of the second moment used in Adam and (ii) layerwise normalization obtained due to layerwise adaptivity. The pseudocode for LAMB is provided in Algorithm 16. When $\beta_1 = 0$ and $\beta_2 = 0$, the algorithm reduces to be Sign SGD where the learning rate is scaled by square root of the layer dimension (Bernstein et al., 2018).

The following result provides convergence rate for LAMB in general nonconvex settings. Similar to the previous case, we focus on the setting where $\beta_1 = 0$ and $\lambda = 0$. As before, our analysis extends to the general case; however, the calculations become messy.

Theorem 6.3 Let $\eta_t = \eta = \sqrt{\frac{2(f(x_1)-f(x^*))}{\alpha_u^2 \|L\|_1 T}}$ for all $t \in [T]$, $b = T$, $d_i = d/h$ for all $i \in [h]$, and $\alpha_l \leq \phi(v) \leq \alpha_u$ for all $v > 0$ where $\alpha_l, \alpha_u > 0$. Then for x_t generated using LAMB (Algorithm 16), we have the following bounds:

1. When $\beta_2 = 0$, we have
$$\left(\mathbb{E}\left[\frac{1}{\sqrt{d}}\|\nabla f(x_a)\|_1\right]\right)^2 \leq O\left(\frac{(f(x_1) - f(x^*))L_{avg}}{T} + \frac{\|\tilde{\sigma}\|_1^2}{Th}\right),$$
2. When $\beta_2 > 0$, we have
$$\mathbb{E}[\|\nabla f(x_a)\|^2] \leq O\left(\sqrt{\frac{G^2 d}{h(1-\beta_2)}} \times \left[\sqrt{\frac{2(f(x_1) - f(x^*))\|L\|_1}{T}} + \frac{\|\tilde{\sigma}\|_1}{\sqrt{T}}\right]\right),$$

where x^* is an optimal solution to the problem in equation 6.1 and x_a is an iterate uniformly randomly chosen from $\{x_1, \cdots, x_T\}$. The proof is in Section 6.6.

Discussion on convergence rates. We first start our discussion with the comparison of convergence rate of LARS with that of SGD (Theorem 6.1). The convergence rates of LARS and SGD differ in two ways: (1) the convergence criterion is $(\mathbb{E}[\sum_{i=1}^h \|\nabla_i f\|])^2$ as opposed to $\mathbb{E}[\|\nabla f\|^2]$ in SGD and (2) the dependence on L and σ in the convergence rate. Briefly, the convergence rate of LARS is better than SGD when the gradient is denser than curvature and stochasticity. This convergence rate comparison is similar in spirit to the one obtained in (Bernstein et al., 2018). Assuming that the convergence criterion in Theorem 6.1 and Theorem 6.2 is of similar order (which happens when gradients are fairly dense), convergence rate of LARS and LAMB depend on L_{avg} instead of L_∞ and are thus, significantly better than that of SGD. A more quantitative comparison is provided in Section 6.6. The comparison of LAMB (with $\beta_2 = 0$) with SGD is along similar lines. We obtain slightly worse rates for the case where $\beta_2 > 0$; although, we believe that its behavior should be better than the case $\beta_2 = 0$. We leave this investigation to future work.

Linear Epoch Gradual Warmup (LEGW)

The warmup technique has been successfully applied in the CNN applications (Goyal et al., 2017; You, Gitman, and Ginsburg, 2017). However, warmup has become an additional parameter that requires developers to tune, which further increases the effort of DNN system

Table 6.1: Large-batch training is a sharp minimum problem. It is easy to miss the global minimum. Tuning the hyper-parameters requires a lot of effort. In this example (ImageNet/ResNet-50 training by LARS solver), we only slightly changed the LR, the accuracy dropped below the target 76.3% accuracy.

Batch Size	Init LR	Warmup	Epochs	Top-1 Test Accuracy
2048	9.94	0.6875 epochs	90	76.97%
2048	10.0	0.6875 epochs	90	75.59%

implementation. To make things worse, large-batch training usually converges to a sharp local minimum, so a tiny change in the hyper-parameters may have a significant influence on the test accuracy (Table 6.1). We propose the Linear-Epoch Gradual Warmup (LEGW or Leg-Warmup) scheme. When we increase the batch size by k times, we also increase the warmup epochs by k times. The intuition is that larger batch size usually needs a large LR. However, a larger LR may make the training algorithm more easily diverge because the gradient changes dramatically in the beginning of neural network training. We use a longer warmup to avoid the divergence of larger LR. Linear epoch warmup means fixing the warmup iterations as we increase the batch size, which helps us to stabilize the chaotic early-learning state. It worth noting the users do not need to change anything.

Explanation of LEGW

In general, it is hard to prove why a specific learning rate schedule works. However, some experimental findings on the change of local Lipschitz constant during iterations partially explain why LEGW works better than previous methods.

Consider the update along the gradient direction $g = \nabla f(x)$. Assume the update is $x \leftarrow x - \eta g$, the question is: how to choose learning rate η? One classical idea is to form a second order approximation around current solution x.

$$f(x+\Delta) \approx \tilde{f}(x+\Delta) := f(x) + \Delta^T \nabla f(x) + \frac{1}{2}\Delta^T \nabla^2 f(x) \Delta, \tag{6.4}$$

and then find Δ to minimize the approximation function. If we assume Δ is in the form of $-\eta g$ and the Hessian is positive definite along the direction of g ($g^T \nabla^2 f(x) g > 0$), then the optimal η^* is

$$\arg\min_{\eta} \tilde{f}(x - \eta g) = \frac{1}{g^T \nabla^2 f(x) g / \|g\|^2} := \frac{1}{L(x,g)}.$$

Therefore, ideally the learning rate should be inversely proportional to $L(x,g)$. Moreover, it is known (Goldstein, 1977) that the update $-\eta g$ will decrease the objective function in a small compact region S if $\eta < \min_{x' \in S} \frac{1}{L(x',g)}$. The optimal learning rate is also called the local Lipchitz constant along the gradient direction, and $L(x,g)$ can be viewed as its approximation. In Figure 6.1, we plot the values of $L(x,g)$ for all the iterations in MNIST training with LSTM. It is hard to compute $L(x,g)$ exactly since $\nabla^2 f(x)$ involves all the training samples. So we approximate it using a small batch and compute the Hessian-vector product by finite difference. For the same reason it is hard to apply a second order method exactly, but the plots

in the figures show an interesting phenomenon that explains why linear warmup works. We observe that the value of $L(x, g)$ usually has a peak in the early iterations, implying a smaller step size should be used in the beginning (which implies warmup is needed). Furthermore, the peak tends to shift toward right (almost linearly) as batch size grows. This intuitively explains our linear warm-up strategy: when batch size increases, the warm up should be longer to cover the "peak region".

6.4 Experiments

We now present empirical results comparing LAMB with existing optimizers on two important large batch training tasks: BERT and ResNet-50 training. We also compare LAMB with existing optimizers for small batch size ($< 1K$) and small dataset (e.g. CIFAR, MNIST) (see Section 6.6).

Experimental Setup. To demonstrate its robustness, we use very minimal hyper-parameter tuning for the LAMB optimizer. Thus, it is possible to achieve better results by further tuning the hyper-parameters. The parameters β_1 and β_2 in Algorithm 16 are set to 0.9 and 0.999 respectively in all our experiments; we only tune the learning rate. We use a polynomially decaying learning rate of $\eta_t = \eta_0 \times (1 - t/T)$ in Algorithm 16), which is the same as in BERT baseline. This setting also works for all other applications in this chapter. Furthermore, for BERT and ResNet-50 training, we did not tune the hyper-parameters of LAMB while increasing the batch size. We use the square root of LR scaling rule to automatically adjust learning rate and linear-epoch gradual warmup (LEGW) scheduling. We use TPUv3 in all the experiments. A TPUv3 Pod has 1024 chips and can provide more than 100 petaflops performance for mixed precision computing. To make sure we are comparing with solid baselines, we use grid search to tune the hyper-parameters for Adam, AdaGrad, AdamW (Adam with weight decay), and LARS. We also tune weight decay for AdamW. All the hyper-parameter tuning settings are reported in Section 6.6.

BERT Training

We first discuss empirical results for speeding up BERT training. For this experiment, we use the same dataset as Devlin et al., 2018, which is a concatenation of Wikipedia and BooksCorpus with 2.5B and 800M words respectively. We specifically focus on the SQuAD task[1] in this chapter. The F1 score on SQuAD-v1 is used as the accuracy metric in our experiments. All our comparisons are with respect to the baseline BERT model by Devlin et al., 2018. To train BERT, Devlin et al. (2018) first train the model for 900k iterations using a sequence length of 128 and then switch to a sequence length of 512 for the last 100k iterations. This results in a training time of around 3 days on 16 TPUv3 chips. The baseline BERT model[2] achieves a F1 score of 90.395. To ensure a fair comparison, we follow the same SQuAD fine-tune procedure of Devlin et al., 2018 without modifying any configuration

[1] https://rajpurkar.github.io/SQuAD-explorer/

[2] Pre-trained BERT model can be downloaded from https://github.com/google-research/bert

(including number of epochs and hyper-parameters). As noted earlier, we could get even better results by changing the fine-tune configuration. For instance, by just slightly changing the learning rate in the fine-tune stage, we can obtain a higher F1 score of 91.688 for the batch size of 16K using LAMB. We report a F1 score of 91.345 in Table 6.2, which is the score obtained for the untuned version. Below we describe two different training choices for training BERT and discuss the corresponding speedups.

Table 6.2: We use the F1 score on SQuAD-v1 as the accuracy metric. The baseline F1 score is the score obtained by the pre-trained model (BERT-Large) provided on BERT's public repository (as of February 1st, 2019). We use TPUv3s in our experiments. We use the same setting as the baseline: the first 9/10 of the total epochs used a sequence length of 128 and the last 1/10 of the total epochs used a sequence length of 512. All the experiments run the same number of epochs. Dev set means the test data. It is worth noting that we can achieve better results by manually tuning the hyper-parameters. The data in this table is collected from the untuned version.

Solver	batch size	steps	F1 score on dev set	TPUs	Time
Baseline	512	1000k	90.395	16	81.4h
LAMB	512	1000k	91.752	16	82.8h
LAMB	1k	500k	91.761	32	43.2h
LAMB	2k	250k	91.946	64	21.4h
LAMB	4k	125k	91.137	128	693.6m
LAMB	8k	62500	91.263	256	390.5m
LAMB	16k	31250	91.345	512	200.0m
LAMB	32k	15625	91.475	1024	101.2m
LAMB	64k/32k	8599	90.584	1024	76.19m

For the first choice, we maintain the same training procedure as the baseline except for changing the training optimizer to LAMB. We run with the same number of epochs as the baseline but with batch size scaled from 512 to 32K. The choice of 32K batch size (with sequence length 512) is mainly due to memory limits of TPU Pod. Our results are shown in Table 6.2. By using the LAMB optimizer, we are able to achieve a F1 score of 91.460 in 15625 iterations for a batch size of 32768 (14063 iterations for sequence length 128 and 1562 iterations for sequence length 512). With 32K batch size, we reduce BERT training time from 3 days to around 100 minutes. We achieved 49.1 times speedup by 64 times computational resources (76.7% efficiency). We consider the speedup is great because we use the synchronous data-parallelism. There is a communication overhead coming from transferring of the gradients over the interconnect. For ResNet-50, researchers are able to achieve 90% scaling efficiency because ResNet-50 has much fewer parameters (# parameters

is equal to #gradients) than BERT (25 million versus 300 million).

To obtain further improvements, we use the **Mixed-Batch Training** procedure with LAMB. Recall that BERT training involves two stages: the first 9/10 of the total epochs use a sequence length of 128, while the last 1/10 of the total epochs use a sequence length of 512. For the second stage training, which involves a longer sequence length, due to memory limits, a maximum batch size of only 32768 can be used on a TPUv3 Pod. However, we can potentially use a larger batch size for the first stage because of a shorter sequence length. In particular, the batch size can be increased to 131072 for the first stage. However, we did not observe any speedup by increasing the batch size from 65536 to 131072 for the first stage, thus, we restrict the batch size to 65536 for this stage. By using this strategy, we are able to make full utilization of the hardware resources throughout the training procedure. Increasing the batch size is able to warm-up and stabilize the optimization process (Smith, Kindermans, and Le, 2017), but decreasing the batch size brings chaos to the optimization process and can cause divergence. In our experiments, we found a technique that is useful to stabilize the second stage optimization. Because we switched to a different optimization problem, it is necessary to re-warm-up the optimization. Instead of decaying the learning rate at the second stage, we ramp up the learning rate from zero again in the second stage (re-warm-up). As with the first stage, we decay the learning rate after the re-warm-up phase. With this method, we only need 8599 iterations and finish BERT training in 76 minutes (100.2% efficiency).

Comparison with AdamW and LARS. To ensure that our approach is compared to a solid baseline for the BERT training, we tried three different strategies for tuning AdamW (Loshchilov and Hutter, 2017): (1) AdamW with default hyper-parameters (Devlin et al., 2018) (2) AdamW with the same hyper-parameters as LAMB, and (3) AdamW with tuned hyper-parameters. AdamW stops scaling at the batch size of 16K because it is not able to achieve the target F1 score (88.1 vs 90.4). The tuning information of AdamW is shown in Section 6.6. For 64K/32K mixed-batch training, even after extensive tuning of the hyper-parameters, we fail to get any reasonable result with AdamW optimizer. We conclude that AdamW does not work well in large-batch BERT training or is at least hard to tune. We also observe that LAMB performs better than LARS for all batch sizes (Table 6.3).

Table 6.3: LAMB achieves a higher performance (F1 score) than LARS for all the batch sizes. The baseline achieves a F1 score of 90.390. Thus, LARS stops scaling at the batch size of 16K.

Batch Size	512	1K	2K	4K	8K	16K	32K
LARS	90.717	90.369	90.748	90.537	90.548	89.589	diverge
LAMB	91.752	91.761	91.946	91.137	91.263	91.345	91.475

ImageNet Training with ResNet-50.

ImageNet training with ResNet-50 is an industry standard metric that is being used in MLPerf[3]. The baseline can get 76.3% top-1 accuracy in 90 epochs (Goyal et al., 2017). All the successful implementations are based on momentum SGD (K. He et al., 2016; Goyal et al., 2017) or LARS optimizer (Ying et al., 2018; X. Jia et al., 2018; Mikami et al., 2018; You, Z. Zhang, Cho-Jui Hsieh, et al., 2018; Yamazaki et al., 2019). Before our study, we did not find any paper reporting a state-of-the-art accuracy achieved by Adam (Kingma and Ba, 2014), AdaGrad, or AdamW optimizer. In our experiments, even with comprehensive hyper-parameter tuning, AdaGrad/Adam/AdamW (with batch size 16K) only achieves 55.38%/66.04%/67.27% top-1 accuracy. After adding learning rate scheme of Goyal et al., 2017, the top-1 accuracy of AdaGrad/Adam/AdamW was improved to 72.0%/73.48%/73.07%. However, they are still much lower than 76.3%. The details of the tuning information are in Section 6.6. Table 6.4 shows that LAMB can achieve the target accuracy. Beyond a batch size of 8K, LAMB's accuracy is higher than the momentum. LAMB's accuracy is also slightly better than LARS. At a batch size of 32K, LAMB achieves 76.4% top-1 accuracy while LARS achieves 76.3%. At a batch size of 2K, LAMB is able to achieve 77.11% top-1 accuracy while LARS achieves 76.6%.

Table 6.4: Top-1 validation accuracy of ImageNet/ResNet-50 training at the batch size of 16K (90 epochs). The performance of momentum was reported by (Goyal et al., 2017). + means adding the learning rate scheme of Goyal et al., 2017 to the optimizer: (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 0.763 (Goyal et al., 2017). All the adaptive solvers were comprehensively tuned. The tuning information is in Section 6.6.

optimizer	adagrad/adagrad+	adam/adam+	adamw/adamw+	momentum	lamb
Accuracy	0.5538/0.7201	0.6604/0.7348	0.6727/0.7307	0.7520	0.7666

Hyper-parameters for scaling the batch size

For BERT and ImageNet training, we did not tune the hyper-parameters of LAMB optimizer when increasing the batch size. We use the square root LR scaling rule and LEGW scheduling to automatically adjust learning rate. The details can be found in Tables 6.5 and 6.6

6.5 Conclusion

Large batch techniques are critical to speeding up deep neural network training. In this chapter, we propose the LAMB optimizer, which supports adaptive elementwise updating and

[3]https://mlperf.org/

Table 6.5: Untuned LAMB for BERT training across different batch sizes (fixed #epochs). We use square root LR scaling and LEGW. For example, batch size 32K needs to finish 15625 iterations. It uses $0.2\times15625 = 3125$ iterations for learning rate warmup. BERT's baseline achieved a F1 score of 90.395. We can achieve an even higher F1 score if we manually tune the hyper-parameters.

Batch Size	512	1K	2K	4K	8K	16K	32K
Learning Rate	$\frac{5}{2^{3.0}\times10^3}$	$\frac{5}{2^{2.5}\times10^3}$	$\frac{5}{2^{2.0}\times10^3}$	$\frac{5}{2^{1.5}\times10^3}$	$\frac{5}{2^{1.0}\times10^3}$	$\frac{5}{2^{0.5}\times10^3}$	$\frac{5}{2^{0.0}\times10^3}$
Warmup Ratio	$\frac{1}{320}$	$\frac{1}{160}$	$\frac{1}{80}$	$\frac{1}{40}$	$\frac{1}{20}$	$\frac{1}{10}$	$\frac{1}{5}$
F1 score	91.752	91.761	91.946	91.137	91.263	91.345	91.475
Exact Match	85.090	85.260	85.355	84.172	84.901	84.816	84.939

Table 6.6: Untuned LAMB for ImageNet training with ResNet-50 for different batch sizes (90 epochs). We use square root LR scaling and LEGW. The baseline Goyal et al., 2017 gets 76.3% top-1 accuracy in 90 epochs. Stanford DAWN Bench (Coleman et al., 2017) baseline achieves 93% top-5 accuracy. LAMB achieves both of them. LAMB can achieve an even higher accuracy if we manually tune the hyper-parameters.

Batch Size	512	1K	2K	4K	8K	16K	32K
Learning Rate	$\frac{4}{2^{3.0}\times100}$	$\frac{4}{2^{2.5}\times100}$	$\frac{4}{2^{2.0}\times100}$	$\frac{4}{2^{1.5}\times100}$	$\frac{4}{2^{1.0}\times100}$	$\frac{4}{2^{0.5}\times100}$	$\frac{4}{2^{0.0}\times100}$
Warmup Epochs	0.3125	0.625	1.25	2.5	5	10	20
Top-5 Accuracy	0.9335	0.9349	0.9353	0.9332	0.9331	0.9322	0.9308
Top-1 Accuracy	0.7696	0.7706	0.7711	0.7692	0.7689	0.7666	0.7642

layerwise learning rates. Furthermore, LAMB is a general purpose optimizer that works for both small and large batches. We also provided theoretical analysis for the LAMB optimizer, highlighting the cases where it performs better than standard SGD. LAMB achieves a better performance than existing optimizers for a wide range of applications. By using LAMB, we are able to scale the batch size of BERT pre-training to 64K without losing accuracy, thereby, reducing the BERT training time from 3 days to around 76 minutes. LAMB is also the first large batch adaptive solver that can achieve state-of-the-art accuracy on ImageNet training with ResNet-50. We also propose the LEGW scheme, which can potentially auto-tune some hyper-parameters in large-batch training. LEGW performs well with LAMB in our experiments.

6.6 Supplementary material

Proof of Theorem 6.2

We analyze the convergence of LARS for general minibatch size here. Recall that the update of LARS is the following

$$x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \phi(\|x_t^{(i)}\|) \frac{g_t^{(i)}}{\|g_t^{(i)}\|},$$

for all $i \in [h]$. Since the function f is L-smooth, we have the following:

$$\begin{aligned}
f(x_{t+1}) &\leq f(x_t) + \langle \nabla_i f(x_t), x_{t+1}^{(i)} - x_t^{(i)} \rangle + \sum_{i=1}^{h} \frac{L_i}{2} \|x_{t+1}^{(i)} - x_t^{(i)}\|^2 \\
&= f(x_t) - \eta_t \sum_{i=1}^{h} \sum_{j=1}^{d_i} \phi(\|x_t^{(i)}\|) \times \left([\nabla_i f(x_t)]_j \times \frac{g_{t,j}^{(i)}}{\|g_t^{(i)}\|} \right) + \sum_{i=1}^{h} \frac{L_i \eta_t^2 \phi^2(\|x_t^{(i)}\|)}{2} \\
&\leq f(x_t) - \eta_t \sum_{i=1}^{h} \sum_{j=1}^{d_i} \phi(\|x_t^{(i)}\|) \times \left([\nabla_i f(x_t)]_j \times \left(\frac{g_{t,j}^{(i)}}{\|g_t^{(i)}\|} - \frac{[\nabla_i f(x_t)]_j}{\|\nabla_i f(x_t)\|} + \frac{[\nabla_i f(x_t)]_j}{\|\nabla_i f(x_t)\|} \right) \right) + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1 \\
&= f(x_t) - \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|) \times \|\nabla_i f(x_t)\| \\
&\qquad - \eta_t \sum_{i=1}^{h} \sum_{j=1}^{d_i} \phi(\|x_t^{(i)}\|) \times \left([\nabla_i f(x_t)]_j \times \left(\frac{g_{t,j}^{(i)}}{\|g_t^{(i)}\|} - \frac{[\nabla_i f(x_t)]_j}{\|\nabla_i f(x_t)\|} \right) \right) + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1
\end{aligned} \tag{6.5}$$

The first inequality follows from the lipschitz continuous nature of the gradient. Let $\Delta_t^{(i)} = g_t^{(i)} - \nabla_i f(x_t)$. Then the above inequality can be rewritten in the following manner:

$$
\begin{aligned}
f(x_{t+1}) &\leq f(x_t) - \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|)\|\nabla_i f(x_t)\| \\
&\quad - \eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i} \phi(\|x_t^{(i)}\|) \times \left([\nabla_i f(x_t)]_j \times \left(\frac{(\Delta_{t,j}^{(i)} + [\nabla_i f(x_t)]_j)}{\|\Delta_t^{(i)} + \nabla_i f(x_t)\|} - \frac{[\nabla_i f(x_t)]_j}{\|\nabla_i f(x_t)\|} \right) \right) + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1 \\
&= f(x_t) - \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|)\|\nabla_i f(x_t)\| \\
&\quad - \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|) \times \left(\frac{\langle \Delta_t^{(i)} + \nabla_i f(x_t), \nabla_i f(x_t) \rangle}{\|\Delta_t^{(i)} + \nabla_i f(x_t)\|} - \|\nabla_i f(x_t)\| \right) + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1 \\
&= f(x_t) - \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|)\|\nabla_i f(x_t)\| \\
&\quad + \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|) \times \left(\frac{\|\nabla_i f(x_t)\| \|\Delta_t^{(i)} + \nabla_i f(x_t)\| - \langle \Delta_t^{(i)} + \nabla_i f(x_t), \nabla_i f(x_t) \rangle}{\|\Delta_t^{(i)} + \nabla_i f(x_t)\|} \right) + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1 \\
&= f(x_t) - \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|)\|\nabla_i f(x_t)\| + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1 \\
&\quad + \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|) \times \left(\frac{\|\nabla_i f(x_t)\| \|\Delta_t^{(i)} + \nabla_i f(x_t)\| - \|\Delta_t^{(i)} + \nabla_i f(x_t)\|^2 + \langle \Delta_t^{(i)}, \Delta_t^{(i)} + \nabla_i f(x_t) \rangle}{\|\Delta_t^{(i)} + \nabla_i f(x_t)\|} \right).
\end{aligned} \tag{6.6}
$$

Using Cauchy-Schwarz inequality in the above inequality, we have:

$$
\begin{aligned}
f(x_{t+1}) &\leq f(x_t) - \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|)\|\nabla_i f(x_t)\| \\
&\quad + \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|) \times \left(\|\nabla_i f(x_t)\| - \|\Delta_t^{(i)} + \nabla_i f(x_t)\| + \|\Delta_t^{(i)}\| \right) + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1 \\
&\leq f(x_t) - \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|)\|\nabla_i f(x_t)\| + 2\eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|) \times \|\Delta_t^{(i)}\| + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1
\end{aligned}
$$

Taking expectation, we obtain the following:

$$
\begin{aligned}
\mathbb{E}[f(x_{t+1})] &\leq f(x_t) - \eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|)\|\nabla_i f(x_t)\| + 2\eta_t \sum_{i=1}^{h} \phi(\|x_t^{(i)}\|) \times \mathbb{E}[\|\Delta_t^{(i)}\|] + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1 \\
&\leq f(x_t) - \eta_t \alpha_l \sum_{i=1}^{h} \|\nabla_i f(x_t)\| + 2\eta_t \alpha_u \frac{\|\sigma\|_1}{\sqrt{b}} + \frac{\eta_t^2 \alpha_u^2}{2} \|L\|_1.
\end{aligned} \tag{6.7}
$$

Summing the above inequality for $t = 1$ to T and using telescoping sum, we have the following

inequality:

$$\mathbb{E}[f(x_{T+1})] \leq f(x_1) - \eta\alpha_l \sum_{t=1}^{T}\sum_{i=1}^{h} \mathbb{E}[\|\nabla_i f(x_t)\|] + 2\eta T \frac{\alpha_u \|\sigma\|_1}{\sqrt{b}} + \frac{\eta^2 \alpha_u^2 T}{2}\|L\|_1.$$

Rearranging the terms of the above inequality, and dividing by $\eta T \alpha_l$, we have:

$$\begin{aligned}\frac{1}{T}\sum_{t=1}^{T}\sum_{i=1}^{h} \mathbb{E}[\|\nabla_i f(x_t)\|] &\leq \frac{f(x_1) - \mathbb{E}[f(x_{T+1})]}{T\eta\alpha_l} + \frac{2\alpha_u\|\sigma\|_1}{\sqrt{b}\alpha_l} + \frac{\eta\alpha_u^2}{2\alpha_l}\|L\|_1 \\ &\leq \frac{f(x_1) - f(x^*)}{T\eta\alpha_l} + \frac{2\alpha_u\|\sigma\|_1}{\alpha_l\sqrt{b}} + \frac{\eta\alpha_u^2}{2\alpha_l}\|L\|_1.\end{aligned}$$

Proof of Theorem 6.3

We analyze the convergence of LAMB for general minibatch size here. Recall that the update of LAMB is the following

$$x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \phi(\|x_t^{(i)}\|)\frac{r_t^{(i)}}{\|r_t^{(i)}\|},$$

for all $i \in [h]$. For simplicity of notation, we reason the

Since the function f is L-smooth, we have the following:

$$\begin{aligned} f(x_{t+1}) &\leq f(x_t) + \langle \nabla_i f(x_t), x_{t+1}^{(i)} - x_t^{(i)}\rangle + \sum_{i=1}^{h}\frac{L_i}{2}\|x_{t+1}^{(i)} - x_t^{(i)}\|^2 \\ &= f(x_t) \underbrace{-\eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i} \phi(\|x_t^{(i)}\|) \times \left([\nabla_i f(x_t)]_j \times \frac{r_{t,j}^{(i)}}{\|r_t^{(i)}\|}\right)}_{T_1} + \sum_{i=1}^{h}\frac{L_i\alpha_u^2\eta_t^2}{2} \end{aligned} \tag{6.8}$$

The above inequality simply follows from the lipschitz continuous nature of the gradient. We bound term T_1 in the following manner:

$$\begin{aligned} T_1 &\leq -\eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i} \phi(\|x_t^{(i)}\|) \times \left([\nabla_i f(x_t)]_j \times \frac{r_{t,j}^{(i)}}{\|r_t^{(i)}\|}\right) \\ &\leq -\eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i} \sqrt{\frac{1-\beta_2}{G^2 d_i}}\left(\phi(\|x_t^{(i)}\|) \times [\nabla_i f(x_t)]_j \times g_{t,j}^{(i)}\right) \\ &\quad - \eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i}\left(\phi(\|x_t^{(i)}\|) \times [\nabla_i f(x_t)]_j \times \frac{r_{t,j}^{(i)}}{\|r_t^{(i)}\|}\right)\mathbb{1}(sign(\nabla_i f(x_t)]_j) \neq sign(r_{t,j}^{(i)})) \end{aligned} \tag{6.9}$$

This follows from the fact that $\|r_t^{(i)}\| \leq \sqrt{\frac{d_i}{1-\beta_2}}$ and $\sqrt{v_t} \leq G$. If $\beta_2 = 0$, then T_1 can be

bounded as follows:

$$T_1 \leq -\eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i}\sqrt{\frac{1}{d_i}}\left(\phi(\|x_t^{(i)}\|) \times |[\nabla_i f(x_t)]_j|\right)$$
$$- \eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i}\left(\phi(\|x_t^{(i)}\|) \times [\nabla_i f(x_t)]_j \times \frac{r_{t,j}^{(i)}}{\|r_t^{(i)}\|}\right)\mathbb{1}(sign(\nabla_i f(x_t)]_j) \neq sign(r_{t,j}^{(i)}))$$

The rest of the proof for $\beta_2 = 0$ is similar to argument for the case $\beta_2 > 0$, which is shown below. Taking expectation, we have the following:

$$\mathbb{E}[T_1] \leq -\eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i}\sqrt{\frac{1-\beta_2}{G^2 d_i}}\mathbb{E}\left[\phi(\|x_t^{(i)}\|) \times \left([\nabla_i f(x_t)]_j \times g_{t,j}^{(i)}\right)\right]$$
$$- \eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i}\mathbb{E}\left[\phi(\|x_t^{(i)}\|) \times \left([\nabla_i f(x_t)]_j \times \frac{r_{t,j}^{(i)}}{\|r_t^{(i)}\|}\right)\mathbb{1}(sign(\nabla_i f(x_t)]_j) \neq sign(g_{t,j}^{(i)}))\right]$$
$$\leq -\eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i}\sqrt{\frac{1-\beta_2}{G^2 d_i}}\mathbb{E}\left[\left(\phi(\|x_t^{(i)}\|) \times [\nabla_i f(x_t)]_j \times g_{t,j}^{(i)}\right)\right]$$
$$+ \eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i}\mathbb{E}\left[\alpha_u |[\nabla_i f(x_t)]_j| \mathbb{1}(sign(\nabla_i f(x_t)]_j) \neq sign(g_{t,j}^{(i)}))\right]$$
$$\leq -\eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i}\sqrt{\frac{1-\beta_2}{G^2 d_i}}\mathbb{E}\left[\phi(\|x_t^{(i)}\|) \times \left([\nabla_i f(x_t)]_j \times g_{t,j}^{(i)}\right)\right]$$
$$+ \eta_t \sum_{i=1}^{h}\sum_{j=1}^{d_i}\alpha_u |[\nabla_i f(x_t)]_j| \mathbb{P}(sign(\nabla_i f(x_t)]_j) \neq sign(g_{t,j}^{(i)}))$$

Using the bound on the probability that the signs differ, we get:

$$\mathbb{E}[T_1] \leq -\eta_t \alpha_l \sqrt{\frac{h(1-\beta_2)}{G^2 d}}\|\nabla f(x_t)\|^2 + \eta_t \alpha_u \sum_{i=1}^{h}\sum_{j=1}^{d_i}\frac{\sigma_{i,j}}{\sqrt{b}}.$$

Substituting the above bound on T_1 in equation 6.8, we have the following bound:

$$\mathbb{E}[f(x_{t+1})] \leq f(x_t) - \eta_t \alpha_l \sqrt{\frac{h(1-\beta_2)}{G^2 d}}\|\nabla f(x_t)\|^2 + \eta_t \alpha_u \frac{\|\tilde{\sigma}\|_1}{\sqrt{b}} + \frac{\eta_t^2 \alpha_u^2 \|L\|_1}{2} \tag{6.10}$$

Summing the above inequality for $t = 1$ to T and using telescoping sum, we have the following inequality:

$$\mathbb{E}[f(x_{T+1})] \leq f(x_1) - \eta_t \alpha_l \sqrt{\frac{h(1-\beta_2)}{G^2 d}}\sum_{t=1}^{T}\mathbb{E}[\|\nabla f(x_t)\|^2] + \eta T \alpha_u \frac{\|\tilde{\sigma}\|_1}{\sqrt{b}} + \frac{\eta^2 \alpha_u^2 T}{2}\|L\|_1.$$

Rearranging the terms of the above inequality, and dividing by $\eta T \alpha_l$, we have:

$$\sqrt{\frac{h(1-\beta_2)}{G^2 d}} \frac{1}{T} \sum_{t=1}^{T} \mathbb{E}[\|\nabla f(x_t)\|^2] \leq \frac{f(x_1) - \mathbb{E}[f(x_{T+1})]}{T\eta\alpha_l} + \frac{\alpha_u \|\tilde{\sigma}\|_1}{\alpha_l \sqrt{b}} + \frac{\eta}{2}\|L\|_1$$

$$\leq \frac{f(x_1) - f(x^*)}{T\eta\alpha_l} + \frac{\alpha_u \|\tilde{\sigma}\|_1}{\alpha_l \sqrt{b}} + \frac{\eta \alpha_u^2}{2\alpha_l}\|L\|_1.$$

Comparison of Convergence Rates of LARS and SGD

Inspired by the comparison used by (Bernstein et al., 2018) for comparing SIGN SGD with SGD, we define the following quantities:

$$\left(\sum_{i=1}^{h} \|\nabla_i f(x_t)\|\right)^2 = \frac{\psi(\nabla f(x_t)) d \|\nabla f(x_t)\|^2}{h} \geq \frac{\psi_g d \|\nabla f(x_t)\|^2}{h}$$

$$\|L\|_1^2 \leq \frac{\psi_L d^2 \|L\|_\infty^2}{h^2}$$

$$\|\sigma\|_1^2 = \frac{\psi_\sigma d \|\sigma\|^2}{h}.$$

Then LARS convergence rate can be written in the following manner:

$$\left(\mathbb{E}[\|\nabla f(x_a)\|]\right)^2 \leq O\left(\frac{(f(x_1) - f(x^*))L_\infty}{T} \frac{\psi_L}{\psi_g^2} + \frac{\|\sigma\|^2}{T} \frac{\psi_\sigma^2}{\psi_g^2}\right).$$

If $\psi_L \ll \psi_g^2$ and $\psi_\sigma \ll \psi_g^2$ then LARS (i.e., gradient is more denser than curvature or stochasticity), we gain over SGD. Otherwise, SGD's upper bound on convergence rate is better.

N-LAMB: Nesterov Momentum for LAMB

Sutskever et al., 2013 report that Nesterov's accelerated gradient (NAG) proposed by Yurii E Nesterov, 1983 is conceptually and empirically better than the regular momentum method for convex, non-stochastic objectives. Dozat, 2016 incorporated Nesterov's momentum into Adam optimizer and proposed the Nadam optimizer. Specifically, only the first moment of Adam was modified and the second moment of Adam was unchanged. The results on several applications (Word2Vec, Image Recognition, and LSTM Language Model) showed that Nadam optimizer improves the speed of convergence and the quality of the learned models. We also tried using Nesterov's momentum to replace the regular momentum of LAMB optimizer's first moment. In this way, we got a new algorithm named as N-LAMB (Nesterov LAMB). The complete algorithm is in Algorithm 17. We can also Nesterov's momentum to replace the regular momentum of LAMB optimizer's second moment. We refer to this algorithm as NN-LAMB (Nesterov's momentum for both the first moment and the second moment). The details of NN-LAMB were shown in Algorithm 18.

Dozat, 2016 suggested the best performance of Nadam was achieved by $\beta_1 = 0.975$, $\beta_2 = 0.999$, and $\epsilon =$ 1e-8. We used the same settings for N-LAMB and NN-LAMB. We scaled the batch size to 32K for ImageNet training with ResNet-50. Our experimental results show that

N-LAMB and NN-LAMB can achieve a comparable accuracy compared to LAMB optimizer. Their performances are much better than momentum solver (Figure 6.2).

LAMB with learning rate correction

There are two operations at each iteration in original Adam optimizer (let us call it adam-correction):

$$m_t = m_t/(1-\beta_1^t)$$
$$v_t = v_t/(1-\beta_2^t)$$

It has an impact on the learning rate by $\eta_t := \eta_t * \sqrt{(1-\beta_2^t)/(1-\beta_1^t)}$. According to our experimental results, adam-correction essentially has the same effect as learning rate warmup (see Figure 6.3). The warmup function often was implemented in the modern deep learning system. Thus, we can remove adam-correction from the LAMB optimizer. We did not observe any drop in the test or validation accuracy for BERT and ImageNet training.

LAMB with different norms

We need to compute the matrix/tensor norm for each layer when we do the parameter updating in the LAMB optimizer. We tried different norms in LAMB optimizer. However, we did not observe a significant difference in the validation accuracy of ImageNet training with ResNet-50. In our experiments, the difference in validation accuracy is less than 0.1 percent (Figure 6.4). We use L2 norm as the default.

Regular Batch Sizes for Small Datasets: MNIST and CIFAR-10.

According to DAWNBench, DavidNet (a custom 9-layer Residual ConvNet) is the fastest model for CIFAR-10 dataset (as of April 1st, 2019)[4]. The baseline uses the momentum SGD optimizer. Table 6.7 and Figure 6.5 show the test accuracy of CIFAR-10 training with DavidNet. The PyTorch implementation (momentum SGD optimizer) on GPUs was reported on Standford DAWNBench's website, which achieves 94.06% in 24 epochs. The Tensorflow implementation (momentum SGD optimizer) on TPU achieves a 93.72% accuracy in 24 epochs[5]. We use the implementation of TensorFlow on TPUs. LAMB optimizer is able to achieve 94.08% test accuracy in 24 epochs, which is better than other adaptive optimizers and momentum SGD. Even on the smaller tasks like MNIST training with LeNet, LAMB is able to achieve a better accuracy than existing solvers (Table 6.8).

Implementation Details and Additional Results

There are several hyper-parameters in LAMB optimizer. Although users do not need to tune them, we explain them to help users to have a better understanding. β_1 is used for decaying

[4]https://dawn.cs.stanford.edu/benchmark/CIFAR10/train.html

[5]https://github.com/fenwickslab/dl_tutorials/blob/master/tutorial3_cifar10_davidnet_fix.ipynb

Table 6.7: CIFAR-10 training with DavidNet (batch size = 512). All of them run 24 epochs and finish the training under one minute on one cloud TPU. We make sure all the solvers are carefully tuned. The learning rate tuning space of Adam, AdamW, Adagrad and LAMB is {0.0001, 0.0002, 0.0004, 0.0006, 0.0008, 0.001, 0.002, 0.004, 0.006, 0.008, 0.01, 0.02, 0.04, 0.06, 0.08, 0.1, 0.2, 0.4, 0.6, 0.8, 1, 2, 4, 6, 8, 10, 15, 20, 25, 30, 35, 40, 45, 50}. The momentum optimizer was tuned by the baseline implementer. The weight decay term of AdamW was tuned by {0.0001, 0.001, 0.01, 0.1, 1.0}.

Optimizer	AdaGrad	Adam	AdamW	momentum	LAMB
Test Accuracy	0.9074	0.9225	0.9271	0.9372	0.9408

Table 6.8: Test Accuracy by MNIST training with LeNet (30 epochs for Batch Size = 1024). The tuning space of learning rate for all the optimizers is {0.0001, 0.001, 0.01, 0.1}. We use the same learning rate warmup and decay schedule for all of them.

Optimizer	Momentum	Addgrad	Adam	AdamW	LAMB
Average accuracy over 5 runs	0.9933	0.9928	0.9936	0.9941	0.9945

the running average of the gradient. β_2 is used for decaying the running average of the square of gradient. The default setting for other parameters: weight decay rate λ=0.01, β_1=0.9, β_2=0.999, ϵ=1e-6. We did not tune β_1 and β_2. However, our experiments show that tuning them may get a higher accuracy.

Based on our experience, learning rate is the most important hyper-parameter that affects the learning efficiency and final accuracy. Bengio, 2012 suggests that it is often the single most important hyper-parameter and that it always should be tuned. Thus, to make sure we have a solid baseline, we carefully tune the learning rate of Adam, AdamW, AdaGrad, and momentum SGD

In our experiments, we found that the validation loss is not reliable for large-batch training. A lower validation loss does not necessarily lead to a higher validation accuracy (Figure 6.6). Thus, we use the test/val accuracy or F1 score on dev set to evaluate the optimizers.

BERT

Table 6.9 shows some of the tuning information from BERT training with AdamW optimizer. AdamW stops scaling at the batch size of 16K. The target F1 score is 90.5. LAMB achieves a F1 score of 91.345. The table shows the tuning information of AdamW. In Table 6.9, we report the best F1 score we observed from our experiments.

The loss curves of BERT training by LAMB for different batch sizes are shown in Figure 6.7. We observe that the loss curves are almost identical to each other, which means our optimizer scales well with the batch size.

The training loss curve of BERT mixed-batch pre-training with LAMB is shown in Figure

Table 6.9: AdamW stops scaling at the batch size of 16K. The target F1 score is 90.5. LAMB achieves a F1 score of 91.345. The table shows the tuning information of AdamW. In this table, we report the best F1 score we observed from our experiments.

Solver	batch size	warmup steps	LR	last step infomation	F1 score on dev set
AdamW	16K	0.05×31250	0.0001	loss=8.04471, step=28126	diverged
AdamW	16K	0.05×31250	0.0002	loss=7.89673, step=28126	diverged
AdamW	16K	0.05×31250	0.0003	loss=8.35102, step=28126	diverged
AdamW	16K	0.10×31250	0.0001	loss=2.01419, step=31250	86.034
AdamW	16K	0.10×31250	0.0002	loss=1.04689, step=31250	88.540
AdamW	16K	0.10×31250	0.0003	loss=8.05845, step=20000	diverged
AdamW	16K	0.20×31250	0.0001	loss=1.53706, step=31250	85.231
AdamW	16K	0.20×31250	0.0002	loss=1.15500, step=31250	88.110
AdamW	16K	0.20×31250	0.0003	loss=1.48798, step=31250	85.653

6.8. This figure shows that LAMB can make the training converge smoothly at the batch size of 64K.

Figure 6.9 shows that we can achieve 76.8% scaling efficiency by scaling the batch size (49.1 times speedup by 64 times computational resources) and 101.8% scaling efficiency with mixed-batch (65.2 times speedup by 64 times computational resources)

ImageNet

Figures 6.10 - 6.15 show the LAMB trust ratio at different iterations for ImageNet training with ResNet-50. From these figures we can see that these ratios are very different from each other for different layers. LAMB uses the trust ratio to help the slow learners to train faster.

Baseline tuning details for ImageNet training with ResNet-50

If you are not interested in the baseline tuning details, please skip this section.

Goyal et al. (2017) suggested a proper learning rate warmup and decay scheme may help improve the ImageNet classification accuracy. We included these techniques in Adam/AdamW/AdaGrad tuning. Specifically, we use the learning rate recipe of Goyal et al., 2017: (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 76.3% (Goyal et al., 2017). There techniques help to improve the accuracy of Adam/AdamW/AdaGrad to around 73%. However, even with these techniques, Adam/AdamW/AdaGrad stil can not achieve the target validation accuracy.

To make sure our baseline is solid, we carefully tuned the hyper-parameters. Table 6.10 shows the tuning information of standard Adagrad. Table 6.11 shows the tuning information of adding the learning rate scheme of Goyal et al., 2017 to standard Adagrad. Table 6.12 shows the tuning information of standard Adam. Table 6.13 shows the tuning information of adding the learning rate scheme of Goyal et al., 2017 to standard Adam. It is tricky to tune the AdamW optimizer since both the L2 regularization and weight decay have the effect on the performance. Thus we have four tuning sets.

The first tuning set is based on AdamW with default L2 regularization. We tune the learning rate and weight decay. The tuning information is in Figures 6.14, 6.15, 6.16, and 6.17.

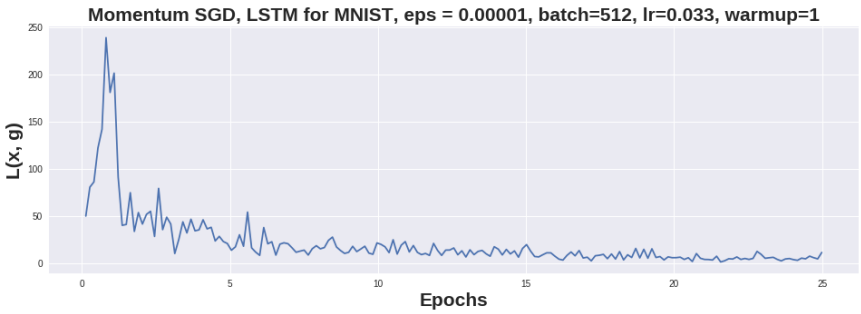
The second tuning set is based on AdamW with disabled L2 regularization. We tune the learning rate and weight decay. The tuning information is in Figures 6.18, 6.19, 6.20, and 6.21.

Then we add the learning rate scheme of Goyal et al., 2017 to AdamW and refer to it as AdamW+.

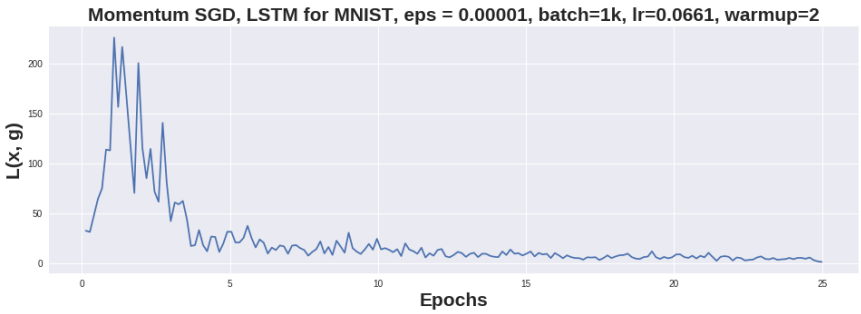
The third tuning set is based on AdamW+ with default L2 regularization. We tune the learning rate and weight decay. The tuning information is Figure 6.22 and 6.23.

The fourth tuning set is based on AdamW+ with disabled L2 regularization. We tune the learning rate and weight decay. The tuning information is in Figures 6.24, 6.25, 6.26.

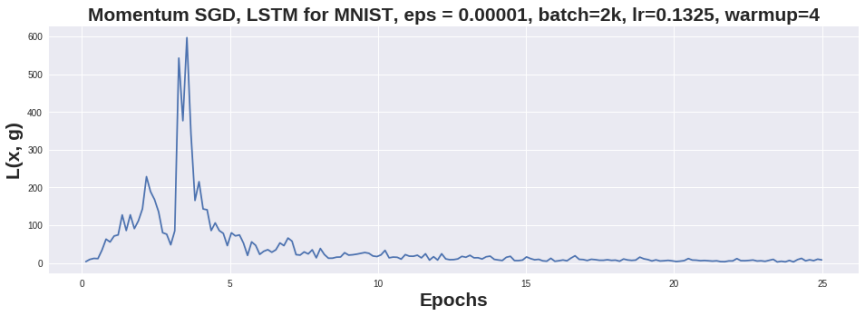
Based on our comprehensive tuning results, we conclude the existing adaptive solvers do not perform well on ImageNet training or at least it is hard to tune them.



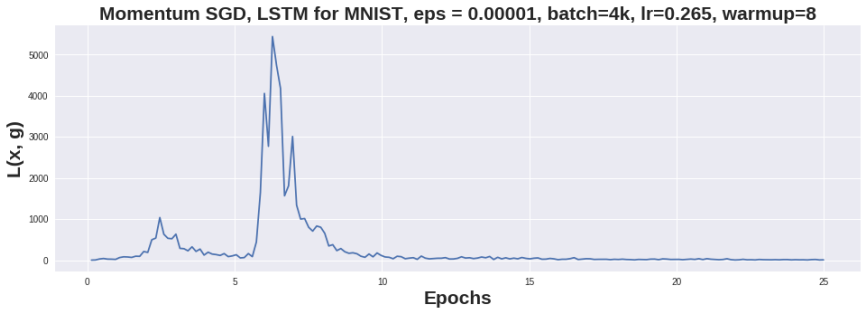
6.1.1 SGD with batch size 512.



6.1.2 SGD with batch size 1K.



6.1.3 SGD with batch size 2K.



6.1.4 SGD with batch size 4K.

Figure 6.1: The approximation of Lipchitz constant for different batch sizes.

Algorithm 17 N-LAMB

Input: $x_1 \in \mathbb{R}^d$, learning rate $\{\eta_t\}_{t=1}^T$, parameters $0 < \beta_1, \beta_2 < 1$, scaling function ϕ, $\epsilon > 0$, parameters $0 < \{\beta_1^t\}_{t=1}^T < 1$
Set $m_0 = 0$, $v_0 = 0$
for $t = 1$ **to** T **do**
  Draw b samples S_t from $\mathbb{P}$.
  Compute $g_t = \frac{1}{|\mathcal{S}_t|} \sum_{s_t \in \mathcal{S}_t} \nabla \ell(x_t, s_t)$.
  $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $\hat{m} = \frac{\beta_1^{t+1} m_t}{1 - \Pi_{i=1}^{t+1} \beta_1^i} + \frac{(1-\beta_1^t) g_t}{1 - \Pi_{i=1}^{t} \beta_1^i}$
  $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\hat{v} = \frac{\beta_2 v_t}{1 - \beta_2^t}$
  Compute ratio $r_t = \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$
  $x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|r_t^{(i)} + \lambda x_t^{(i)}\|} (r_t^{(i)} + \lambda x_t)$
end for

Algorithm 18 NN-LAMB

Input: $x_1 \in \mathbb{R}^d$, learning rate $\{\eta_t\}_{t=1}^T$, parameters $0 < \beta_1, \beta_2 < 1$, scaling function ϕ, $\epsilon > 0$, parameters $0 < \{\beta_1^t\}_{t=1}^T < 1$
Set $m_0 = 0$, $v_0 = 0$
for $t = 1$ **to** T **do**
  Draw b samples S_t from $\mathbb{P}$.
  Compute $g_t = \frac{1}{|\mathcal{S}_t|} \sum_{s_t \in \mathcal{S}_t} \nabla \ell(x_t, s_t)$.
  $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $\hat{m} = \frac{\beta_1^{t+1} m_t}{1 - \Pi_{i=1}^{t+1} \beta_1^i} + \frac{(1-\beta_1^t) g_t}{1 - \Pi_{i=1}^{t} \beta_1^i}$
  $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\hat{v} = \frac{\beta_2^{t+1} v_t}{1 - \Pi_{i=1}^{t+1} \beta_2^i} + \frac{(1-\beta_2^t) g_t^2}{1 - \Pi_{i=1}^{t} \beta_2^i}$
  Compute ratio $r_t = \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$
  $x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|r_t^{(i)} + \lambda x_t^{(i)}\|} (r_t^{(i)} + \lambda x_t)$
end for

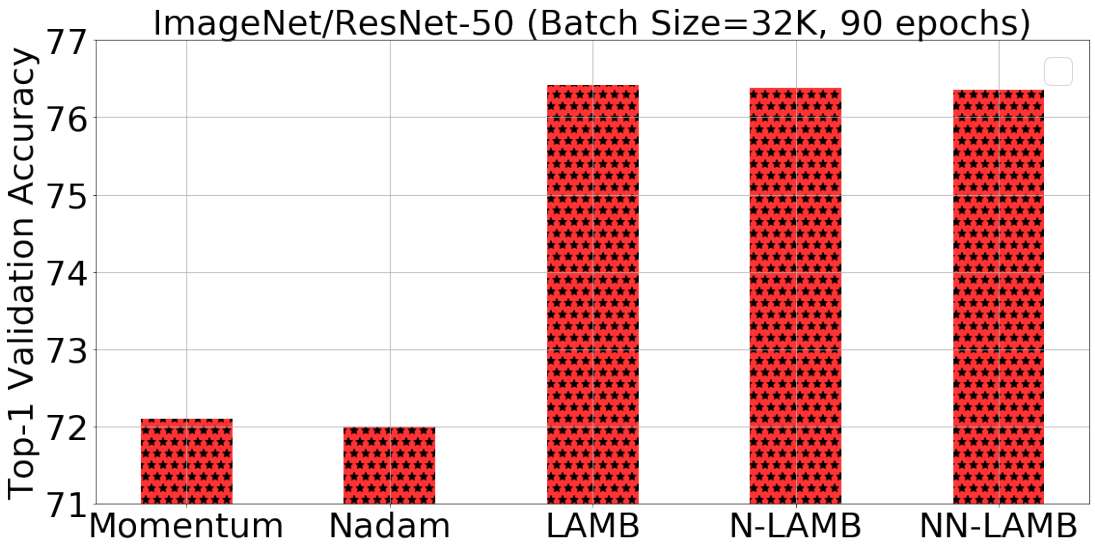


Figure 6.2: This figure shows N-LAMB and NN-LAMB can achieve a comparable accuracy compared to LAMB optimizer. Their performances are much better than momentum solver. The result of momentum optimizer was reported by Goyal et al., 2017. For Nadam, we use the learning rate recipe of (Goyal et al., 2017): (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 0.763 (Goyal et al., 2017). We also tuned the learning rate of Nadam in {1e-4, 2e-4, ..., 9e-4, 1e-3, 2e-3, ..., 9e-3, 1e-2}.

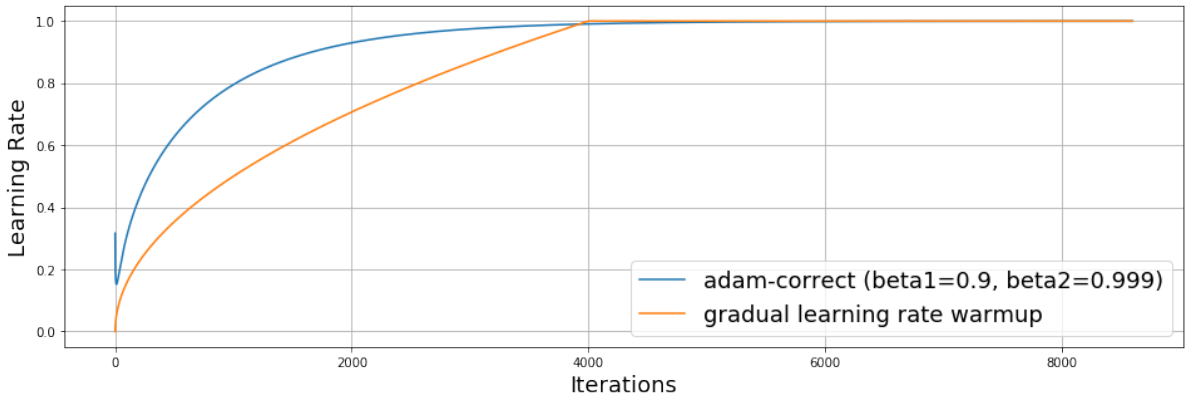


Figure 6.3: The figure shows that adam-correction has the same effect as learning rate warmup. We removed adam-correction from the LAMB optimizer. We did not observe any drop in the test or validation accuracy for BERT and ImageNet training.

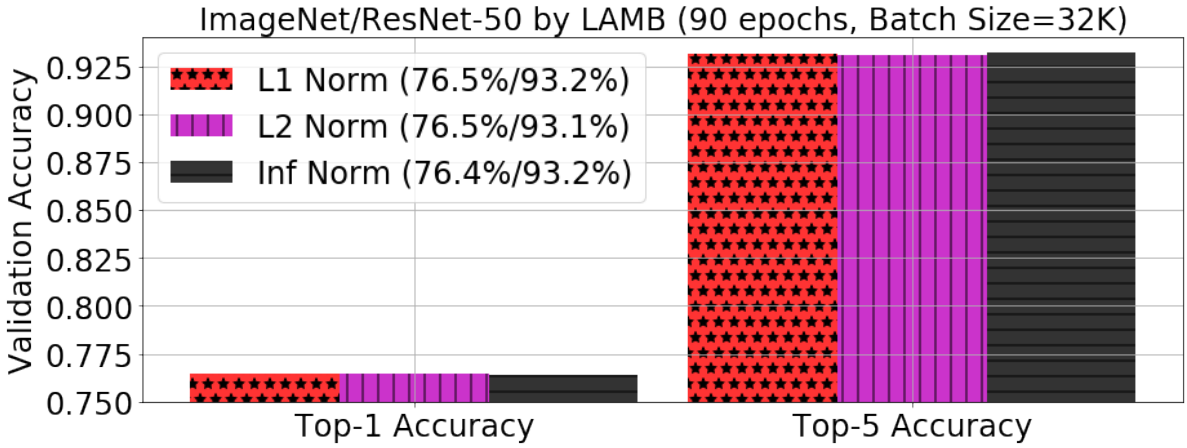


Figure 6.4: We tried different norms in LAMB optimizer. However, we did not observe a significant difference in the validation accuracy of ImageNet training with ResNet-50. We use L2 norm as the default.

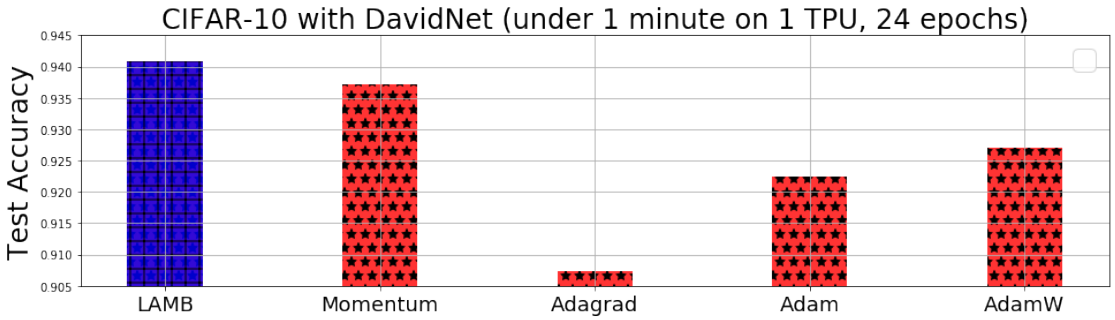


Figure 6.5: LAMB is better than the existing solvers (batch size = 512). We make sure all the solvers are carefully tuned. The learning rate tuning space of Adam, AdamW, Adagrad and LAMB is {0.0001, 0.0002, 0.0004, 0.0006, 0.0008, 0.001, 0.002, 0.004, 0.006, 0.008, 0.01, 0.02, 0.04, 0.06, 0.08, 0.1, 0.2, 0.4, 0.6, 0.8, 1, 2, 4, 6, 8, 10, 15, 20, 25, 30, 35, 40, 45, 50}. The momentum optimizer was tuned by the baseline implementer. The weight decay term of AdamW was tuned by {0.0001, 0.001, 0.01, 0.1, 1.0}.

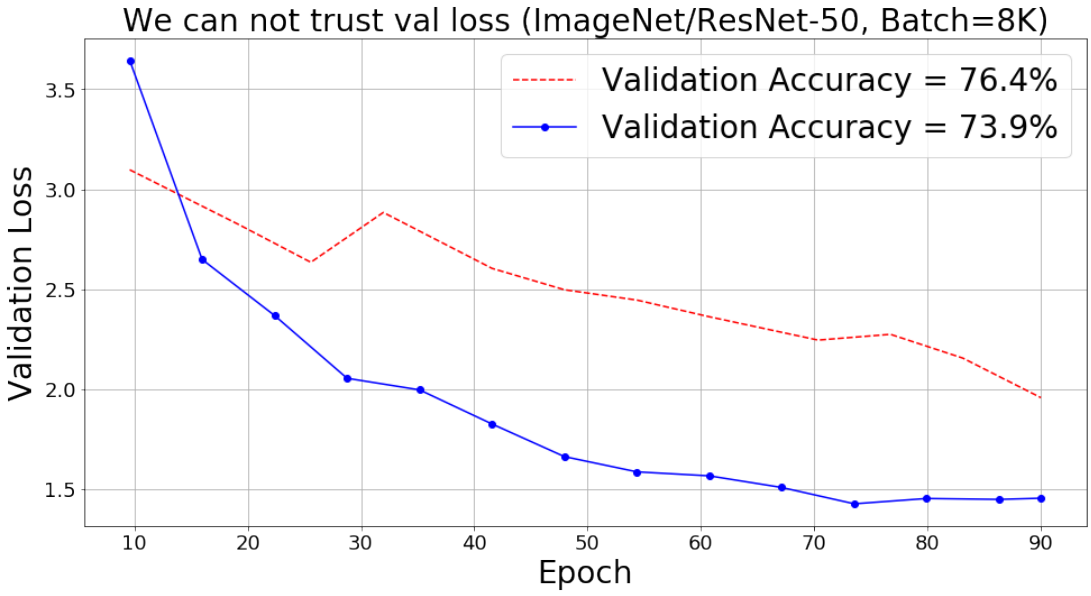


Figure 6.6: Our experiments show that even the validation loss is not reliable in the large-scale training. A lower validation loss may lead to a worse accuracy. Thus, we use the test/val accuracy or F1 score on dev set to evaluate the optimizers.

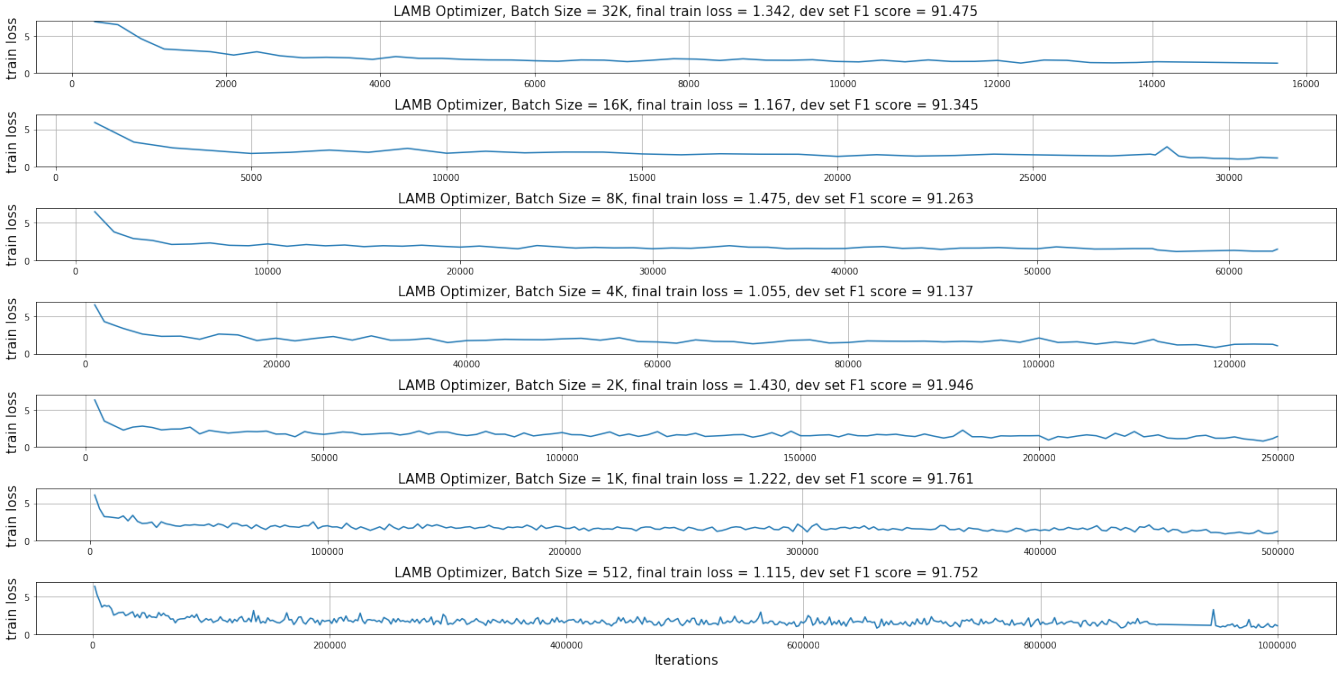


Figure 6.7: This figure shows the training loss curve of LAMB optimizer. We just want to use this figure to show that LAMB can make the training converge smoothly. Even if we scale the batch size to the extremely large cases, the loss curves are almost identical to each other.

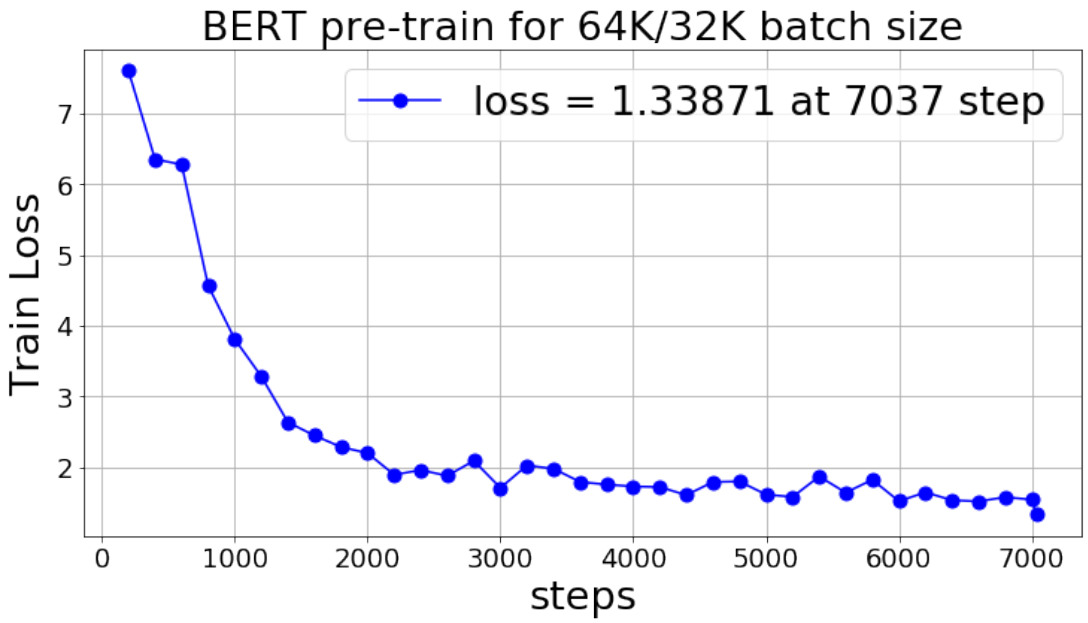


Figure 6.8: This figure shows the training loss curve of LAMB optimizer. This figure shows that LAMB can make the training converge smoothly at the extremely large batch size (e.g. 64K).

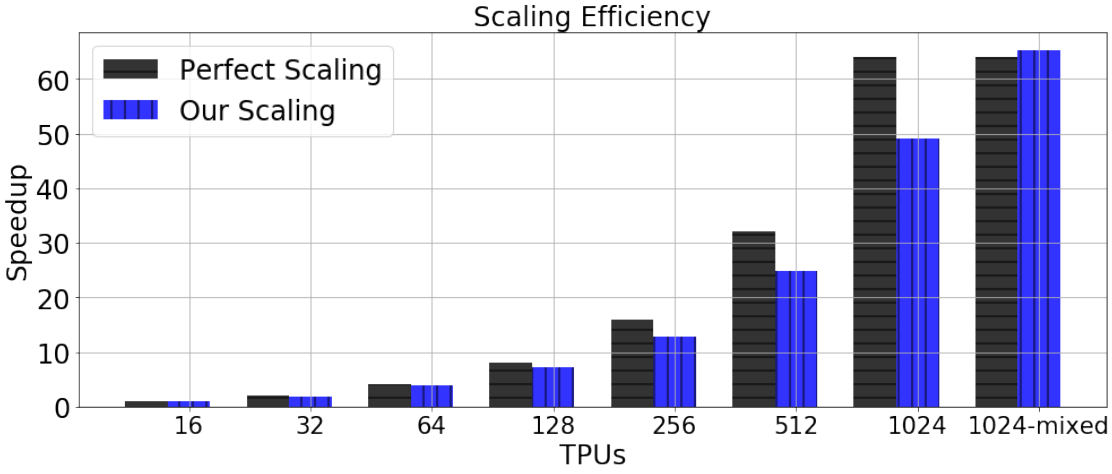


Figure 6.9: We achieve 76.8% scaling efficiency (49 times speedup by 64 times computational resources) and 101.8% scaling efficiency with a mixed, scaled batch size (65.2 times speedup by 64 times computational resources). 1024-mixed means the mixed-batch training on 1024 TPUs.

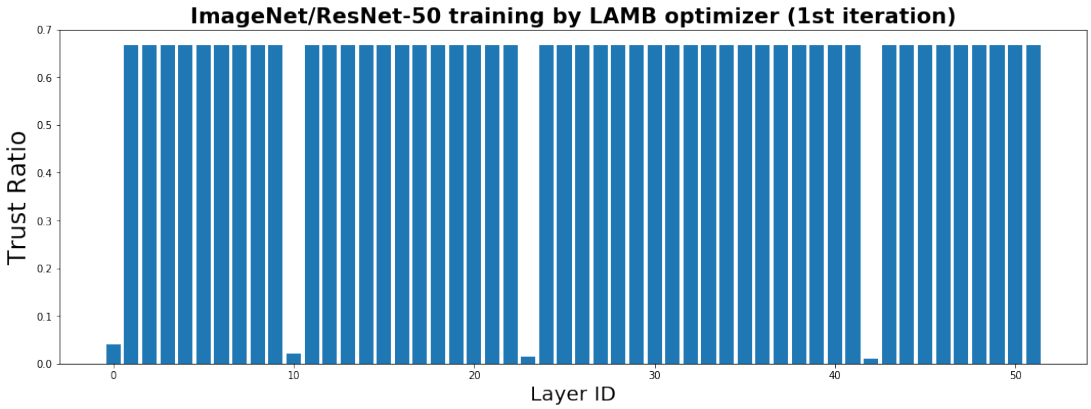


Figure 6.10: The LAMB trust ratio.

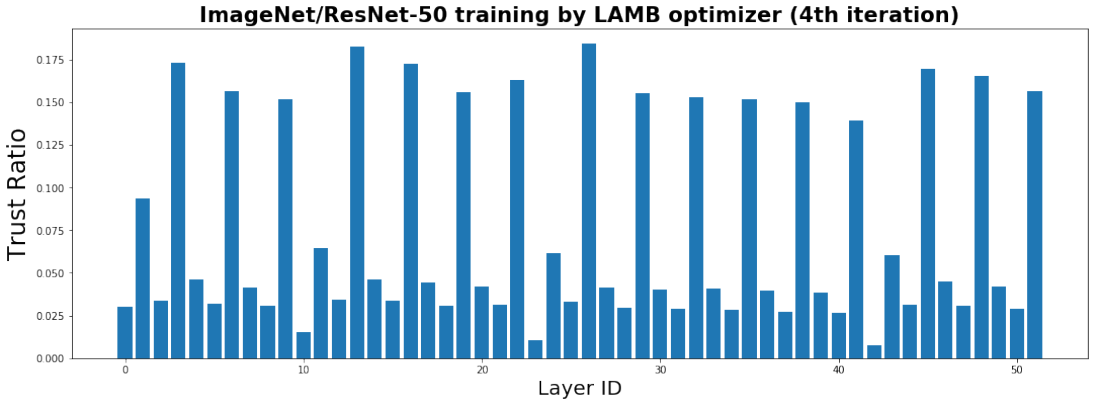


Figure 6.11: The LAMB trust ratio.

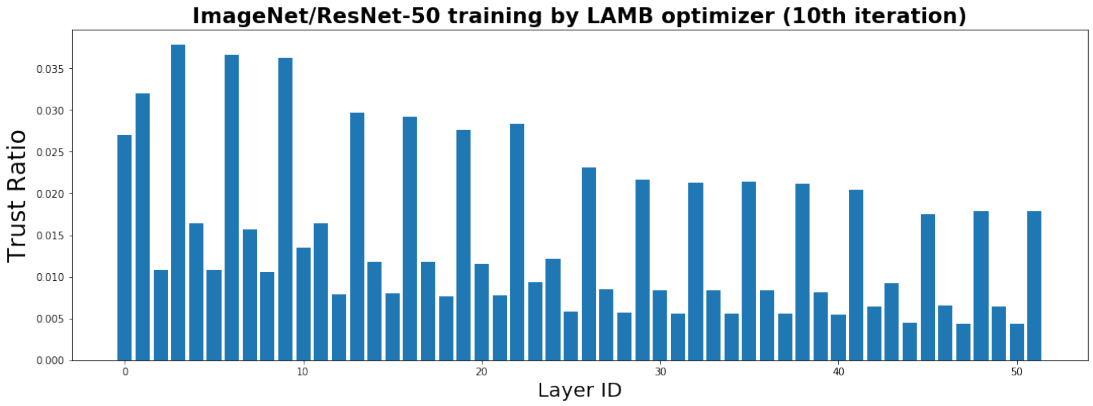


Figure 6.12: The LAMB trust ratio.

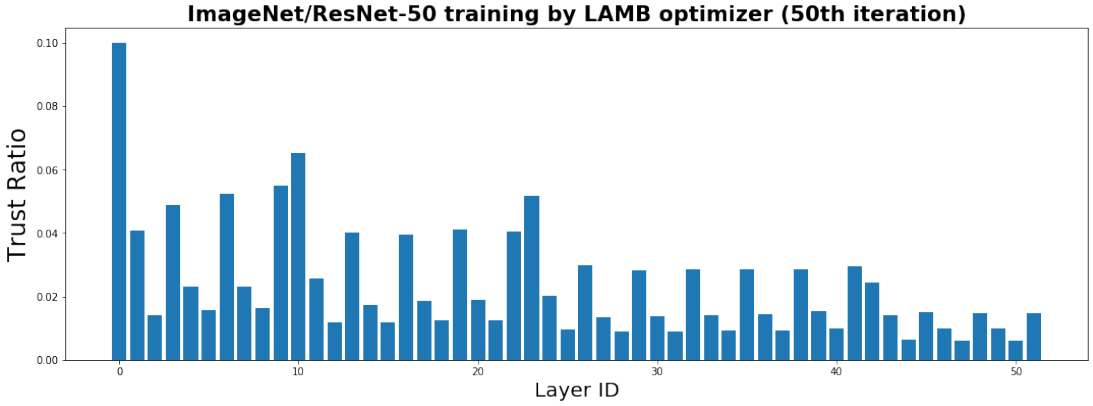


Figure 6.13: The LAMB trust ratio.

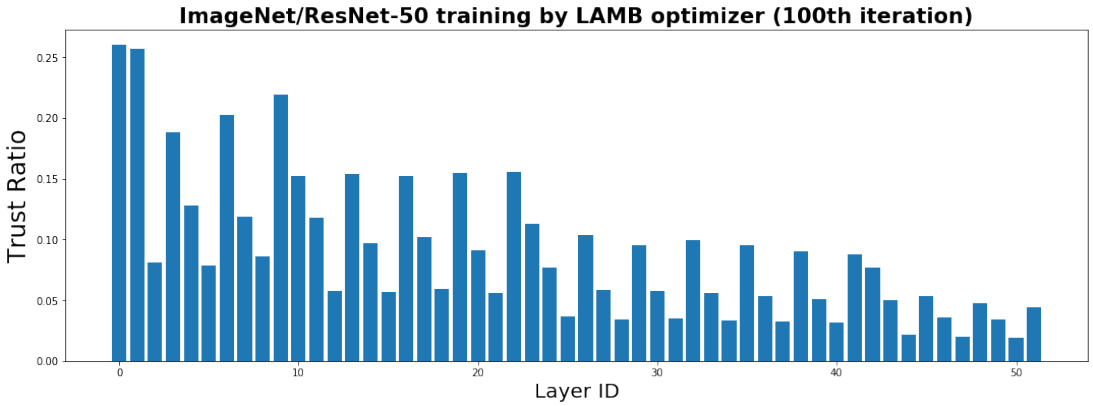


Figure 6.14: The LAMB trust ratio.

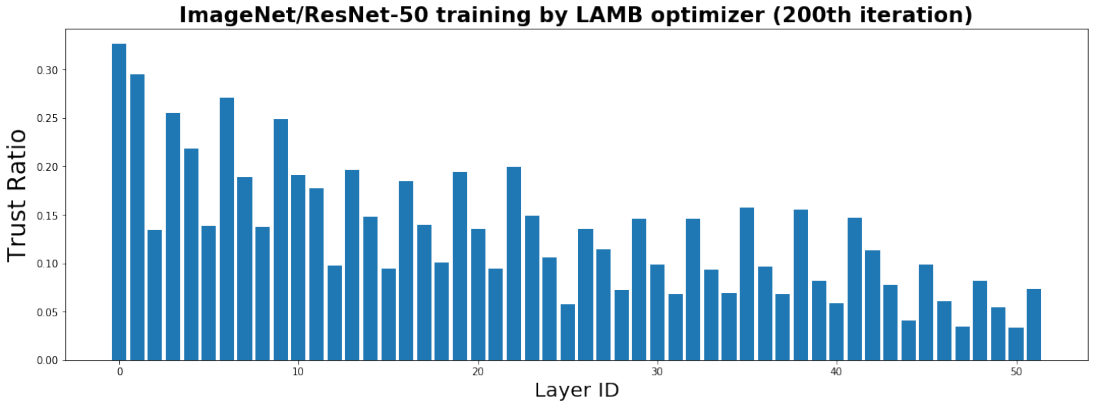


Figure 6.15: The LAMB trust ratio.

Table 6.10: The accuracy information of tuning default AdaGrad optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations).

Learning Rate	Top-1 Validation Accuracy
0.0001	0.0026855469
0.001	0.015563965
0.002	0.022684732
0.004	0.030924479
0.008	0.04486084
0.010	0.054158527
0.020	0.0758667
0.040	0.1262614
0.080	0.24037679
0.100	0.27357993
0.200	0.458313
0.400	**0.553833**
0.800	0.54103595
1.000	0.5489095
2.000	0.47680664
4.000	0.5295207
6.000	0.36950684
8.000	0.31081137
10.00	0.30670166
12.00	0.3091024
14.00	0.3227946
16.00	0.0063680015
18.00	0.11287435
20.00	0.21602376
30.00	0.08315023
40.00	0.0132039385
50.00	0.0009969076

Table 6.11: The accuracy information of tuning AdaGrad optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). We use the learning rate recipe of (Goyal et al., 2017): (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 0.763 (Goyal et al., 2017).

Learning Rate	Top-1 Validation Accuracy
0.0001	0.0011189779
0.001	0.00793457
0.002	0.012573242
0.004	0.019022623
0.008	0.027079264
0.010	0.029012045
0.020	0.0421346
0.040	0.06618246
0.080	0.10970052
0.100	0.13429768
0.200	0.26550293
0.400	0.41918945
0.800	0.5519816
1.000	0.58614093
2.000	0.67252606
4.000	0.70306396
6.000	0.709493
8.000	0.7137858
10.00	0.71797687
12.00	0.7187703
14.00	**0.72007245**
16.00	0.7194214
18.00	0.7149251
20.00	0.71293133
30.00	0.70458984
40.00	0.69085693
50.00	0.67976886

Table 6.12: The accuracy information of tuning default Adam optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). The target accuracy is around 0.763 (Goyal et al., 2017).

Learning Rate	Top-1 Validation Accuracy
0.0001	0.5521
0.0002	0.6089
0.0004	0.6432
0.0006	0.6465
0.0008	0.6479
0.001	**0.6604**
0.002	0.6408
0.004	0.5687
0.006	0.5165
0.008	0.4812
0.010	0.3673

Table 6.13: The accuracy information of tuning Adam optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). We use the learning rate recipe of (Goyal et al., 2017): (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 0.763 (Goyal et al., 2017).

Learning Rate	Top-1 Validation Accuracy
0.0001	0.410319
0.0002	0.55263263
0.0004	0.6455485
0.0006	0.6774495
0.0008	0.6996867
0.001	0.71010333
0.002	**0.73476154**
0.004	0.73286945
0.006	0.72648114
0.008	0.72214764
0.010	0.71466064
0.012	0.7081502
0.014	0.6993001
0.016	0.69108075
0.020	0.67997235
0.040	0.58658856
0.060	0.51090497
0.080	0.45174155
0.100	0.40297446

Table 6.14: The accuracy information of tuning default AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.00001	default (0.01)	0.53312176
0.0002	0.00001	default (0.01)	0.5542806
0.0004	0.00001	default (0.01)	0.48769125
0.0006	0.00001	default (0.01)	0.46317545
0.0008	0.00001	default (0.01)	0.40903726
0.001	0.00001	default (0.01)	0.42401123
0.002	0.00001	default (0.01)	0.33870444
0.004	0.00001	default (0.01)	0.12339274
0.006	0.00001	default (0.01)	0.122924805
0.008	0.00001	default (0.01)	0.08099365
0.010	0.00001	default (0.01)	0.016764322
0.012	0.00001	default (0.01)	0.032714844
0.014	0.00001	default (0.01)	0.018147787
0.016	0.00001	default (0.01)	0.0066731772
0.018	0.00001	default (0.01)	0.010294597
0.020	0.00001	default (0.01)	0.008260091
0.025	0.00001	default (0.01)	0.008870442
0.030	0.00001	default (0.01)	0.0064493814
0.040	0.00001	default (0.01)	0.0018107096
0.050	0.00001	default (0.01)	0.003540039

Table 6.15: The accuracy information of tuning default AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.0001	default (0.01)	0.55489093
0.0002	0.0001	default (0.01)	**0.56514484**
0.0004	0.0001	default (0.01)	0.4986979
0.0006	0.0001	default (0.01)	0.47595215
0.0008	0.0001	default (0.01)	0.44685873
0.001	0.0001	default (0.01)	0.41029868
0.002	0.0001	default (0.01)	0.2808024
0.004	0.0001	default (0.01)	0.08111572
0.006	0.0001	default (0.01)	0.068115234
0.008	0.0001	default (0.01)	0.057922363
0.010	0.0001	default (0.01)	0.05222575
0.012	0.0001	default (0.01)	0.017313639
0.014	0.0001	default (0.01)	0.029785156
0.016	0.0001	default (0.01)	0.016540527
0.018	0.0001	default (0.01)	0.00575765
0.020	0.0001	default (0.01)	0.0102335615
0.025	0.0001	default (0.01)	0.0060831704
0.030	0.0001	default (0.01)	0.0036417644
0.040	0.0001	default (0.01)	0.0010782877
0.050	0.0001	default (0.01)	0.0037638347

Table 6.16: The accuracy information of tuning default AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.001	default (0.01)	0.21142578
0.0002	0.001	default (0.01)	0.4289144
0.0004	0.001	default (0.01)	0.13537598
0.0006	0.001	default (0.01)	0.33803305
0.0008	0.001	default (0.01)	0.32611084
0.001	0.001	default (0.01)	0.22194417
0.002	0.001	default (0.01)	0.1833903
0.004	0.001	default (0.01)	0.08256022
0.006	0.001	default (0.01)	0.020507812
0.008	0.001	default (0.01)	0.018269857
0.010	0.001	default (0.01)	0.007507324
0.012	0.001	default (0.01)	0.020080566
0.014	0.001	default (0.01)	0.010762532
0.016	0.001	default (0.01)	0.0021362305
0.018	0.001	default (0.01)	0.007954915
0.020	0.001	default (0.01)	0.005859375
0.025	0.001	default (0.01)	0.009724935
0.030	0.001	default (0.01)	0.0019124349
0.040	0.001	default (0.01)	0.00390625
0.050	0.001	default (0.01)	0.0009969076

Table 6.17: The accuracy information of tuning default AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.01	default (0.01)	0.0009765625
0.0002	0.01	default (0.01)	0.0009969076
0.0004	0.01	default (0.01)	0.0010172526
0.0006	0.01	default (0.01)	0.0009358724
0.0008	0.01	default (0.01)	0.0022379558
0.001	0.01	default (0.01)	0.001566569
0.002	0.01	default (0.01)	0.009480794
0.004	0.01	default (0.01)	0.0033569336
0.006	0.01	default (0.01)	0.0029907227
0.008	0.01	default (0.01)	0.0018513998
0.010	0.01	default (0.01)	0.009134929
0.012	0.01	default (0.01)	0.0022176106
0.014	0.01	default (0.01)	0.0040690103
0.016	0.01	default (0.01)	0.0017293295
0.018	0.01	default (0.01)	0.00061035156
0.020	0.01	default (0.01)	0.0022379558
0.025	0.01	default (0.01)	0.0017089844
0.030	0.01	default (0.01)	0.0014241537
0.040	0.01	default (0.01)	0.0020345051
0.050	0.01	default (0.01)	0.0012817383

Table 6.18: The accuracy information of tuning default AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.00001	disable	0.48917642
0.0002	0.00001	disable	0.58152264
0.0004	0.00001	disable	0.63460284
0.0006	0.00001	disable	0.64849854
0.0008	0.00001	disable	0.6598918
0.001	0.00001	disable	0.6662801
0.002	0.00001	disable	**0.67266846**
0.004	0.00001	disable	0.6692708
0.006	0.00001	disable	0.6573079
0.008	0.00001	disable	0.6639404
0.010	0.00001	disable	0.65230304
0.012	0.00001	disable	0.6505534
0.014	0.00001	disable	0.64990234
0.016	0.00001	disable	0.65323895
0.018	0.00001	disable	0.67026776
0.020	0.00001	disable	0.66086835
0.025	0.00001	disable	0.65425617
0.030	0.00001	disable	0.6476237
0.040	0.00001	disable	0.55478925
0.050	0.00001	disable	0.61869305

Table 6.19: The accuracy information of tuning default AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.0001	disable	0.5033366
0.0002	0.0001	disable	0.5949707
0.0004	0.0001	disable	0.62561035
0.0006	0.0001	disable	0.6545207
0.0008	0.0001	disable	0.66326904
0.001	0.0001	disable	0.6677043
0.002	0.0001	disable	**0.67244464**
0.004	0.0001	disable	0.6702881
0.006	0.0001	disable	0.66033936
0.008	0.0001	disable	0.66426593
0.010	0.0001	disable	0.66151935
0.012	0.0001	disable	0.6545817
0.014	0.0001	disable	0.65509033
0.016	0.0001	disable	0.6529338
0.018	0.0001	disable	0.65651447
0.020	0.0001	disable	0.65334064
0.025	0.0001	disable	0.655009
0.030	0.0001	disable	0.64552814
0.040	0.0001	disable	0.6425374
0.050	0.0001	disable	0.5988159

Table 6.20: The accuracy information of tuning default AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.001	disable	0.4611206
0.0002	0.001	disable	0.0076293945
0.0004	0.001	disable	0.29233804
0.0006	0.001	disable	0.57295734
0.0008	0.001	disable	0.5574748
0.001	0.001	disable	0.5988566
0.002	0.001	disable	0.586263
0.004	0.001	disable	**0.62076825**
0.006	0.001	disable	0.61503094
0.008	0.001	disable	0.4697876
0.010	0.001	disable	0.619751
0.012	0.001	disable	0.54243976
0.014	0.001	disable	0.5429077
0.016	0.001	disable	0.55281574
0.018	0.001	disable	0.5819295
0.020	0.001	disable	0.5938924
0.025	0.001	disable	0.541097
0.030	0.001	disable	0.45890298
0.040	0.001	disable	0.56193036
0.050	0.001	disable	0.5279134

Table 6.21: The accuracy information of tuning default AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.01	disable	0.0009969076
0.0002	0.01	disable	0.0008951823
0.0004	0.01	disable	0.00095621747
0.0006	0.01	disable	0.0012817383
0.0008	0.01	disable	0.016886393
0.001	0.01	disable	0.038146973
0.002	0.01	disable	0.0015258789
0.004	0.01	disable	0.0014241537
0.006	0.01	disable	0.081441246
0.008	0.01	disable	0.028116861
0.010	0.01	disable	0.011820476
0.012	0.01	disable	0.08138021
0.014	0.01	disable	0.010111491
0.016	0.01	disable	0.0041910806
0.018	0.01	disable	0.0038248699
0.020	0.01	disable	0.002746582
0.025	0.01	disable	0.011555989
0.030	0.01	disable	0.0065104165
0.040	0.01	disable	0.016438803
0.050	0.01	disable	0.007710775

Table 6.22: The accuracy information of tuning AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). We use the learning rate recipe of (Goyal et al., 2017): (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.01	default (0.01)	0.0009969076
0.0002	0.01	default (0.01)	0.0009969076
0.0004	0.01	default (0.01)	0.0009969076
0.0006	0.01	default (0.01)	0.0009358724
0.0008	0.01	default (0.01)	0.0009969076
0.001	0.01	default (0.01)	0.0009765625
0.002	0.01	default (0.01)	0.0010172526
0.004	0.01	default (0.01)	0.0010172526
0.006	0.01	default (0.01)	0.0010172526
0.008	0.01	default (0.01)	0.0010172526
0.0001	0.001	default (0.01)	0.0010172526
0.0002	0.001	default (0.01)	0.0010172526
0.0004	0.001	default (0.01)	0.0010172526
0.0006	0.001	default (0.01)	0.0009969076
0.0008	0.001	default (0.01)	0.0010172526
0.001	0.001	default (0.01)	0.0010172526
0.002	0.001	default (0.01)	0.0010172526
0.004	0.001	default (0.01)	0.0038452148
0.006	0.001	default (0.01)	0.011881511
0.008	0.001	default (0.01)	0.0061442056

Table 6.23: The accuracy information of tuning AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). We use the learning rate recipe of (Goyal et al., 2017): (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.0001	default (0.01)	0.3665975
0.0002	0.0001	default (0.01)	0.5315755
0.0004	0.0001	default (0.01)	0.6369222
0.0006	0.0001	default (0.01)	0.6760457
0.0008	0.0001	default (0.01)	0.69557697
0.001	0.0001	default (0.01)	0.7076009
0.002	0.0001	default (0.01)	**0.73065186**
0.004	0.0001	default (0.01)	0.72806805
0.006	0.0001	default (0.01)	0.72161865
0.008	0.0001	default (0.01)	0.71816
0.0001	0.00001	default (0.01)	0.49804688
0.0002	0.00001	default (0.01)	0.6287028
0.0004	0.00001	default (0.01)	0.6773885
0.0006	0.00001	default (0.01)	0.67348224
0.0008	0.00001	default (0.01)	0.6622111
0.001	0.00001	default (0.01)	0.6468709
0.002	0.00001	default (0.01)	0.5846761
0.004	0.00001	default (0.01)	0.4868978
0.006	0.00001	default (0.01)	0.34969077
0.008	0.00001	default (0.01)	0.31193033

Table 6.24: The accuracy information of tuning AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). We use the learning rate recipe of (Goyal et al., 2017): (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.01	disable	0.0010172526
0.0002	0.01	disable	0.0009765625
0.0004	0.01	disable	0.0010172526
0.0006	0.01	disable	0.0009969076
0.0008	0.01	disable	0.0010172526
0.001	0.01	disable	0.0009765625
0.002	0.01	disable	0.0009969076
0.004	0.01	disable	0.0009969076
0.006	0.01	disable	0.0009765625
0.008	0.01	disable	0.0010172526
0.0001	0.001	disable	0.0009765625
0.0002	0.001	disable	0.0010172526
0.0004	0.001	disable	0.0010172526
0.0006	0.001	disable	0.0010172526
0.0008	0.001	disable	0.0010172526
0.001	0.001	disable	0.0009969076
0.002	0.001	disable	0.0010579427
0.004	0.001	disable	0.0016886393
0.006	0.001	disable	0.019714355
0.008	0.001	disable	0.1329956

Table 6.25: The accuracy information of tuning AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). We use the learning rate recipe of (Goyal et al., 2017): (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.0001	disable	0.28515625
0.0002	0.0001	disable	0.44055176
0.0004	0.0001	disable	0.56815594
0.0006	0.0001	disable	0.6234741
0.0008	0.0001	disable	0.6530762
0.001	0.0001	disable	0.6695964
0.002	0.0001	disable	0.70048016
0.004	0.0001	disable	0.71698
0.006	0.0001	disable	0.72021484
0.008	0.0001	disable	**0.7223918**
0.010	0.0001	disable	0.72017413
0.012	0.0001	disable	0.72058105
0.014	0.0001	disable	0.7188924
0.016	0.0001	disable	0.71695966
0.018	0.0001	disable	0.7154134
0.020	0.0001	disable	0.71358234
0.025	0.0001	disable	0.7145386
0.030	0.0001	disable	0.7114258
0.040	0.0001	disable	0.7066447
0.050	0.0001	disable	0.70284015

Table 6.26: The accuracy information of tuning AdamW optimizer for ImageNet training with ResNet-50 (batch size = 16384, 90 epochs, 7038 iterations). We use the learning rate recipe of (Goyal et al., 2017): (1) 5-epoch warmup to stablize the initial stage; and (2) multiply the learning rate by 0.1 at 30th, 60th, and 80th epoch. The target accuracy is around 0.763 (Goyal et al., 2017).

learning rate	weight decay	L2 regularization	Top-1 Validation Accuracy
0.0001	0.00001	disable	0.31247965
0.0002	0.00001	disable	0.4534912
0.0004	0.00001	disable	0.57765704
0.0006	0.00001	disable	0.6277669
0.0008	0.00001	disable	0.65321857
0.001	0.00001	disable	0.6682129
0.002	0.00001	disable	0.69938153
0.004	0.00001	disable	0.7095947
0.006	0.00001	disable	0.710612
0.008	0.00001	disable	0.70857745
0.010	0.00001	disable	0.7094116
0.012	0.00001	disable	0.70717365
0.014	0.00001	disable	**0.7109375**
0.016	0.00001	disable	0.7058309
0.018	0.00001	disable	0.7052409
0.020	0.00001	disable	0.7064412
0.025	0.00001	disable	0.7035319
0.030	0.00001	disable	0.6994629
0.040	0.00001	disable	0.6972656
0.050	0.00001	disable	0.6971232

Chapter 7

Conclusion and Future Work

7.1 Summary

We study the problem of distributed machine learning algorithms in this thesis. Each chapter reviews most of the important related work in the last few years. As mentioned in previous chapters, communication is a typical overhead in distributed machine learning. This overhead is likely growing in the future. Thus, one key issue is to improve the ratio between computation (the number of floating point operations) and communication (the number of words moved and the number of messages) for distributed machine learning algorithms. In this thesis, we propose a series of communication-avoiding machine learning algorithms like CA-SVM, CA-KRR, Asyn-GCD, LARS, and LAMB to close the gap between machine learning and distributed systems. We present strong experimental results on some real-world applications. Our methods are also supported by theoretical guarantees. Therefore, we conclude our approaches are accurate, fast, and scalable. We expect our approaches to be used in future hyper-scale applications.

7.2 Future Directions

My vision for the future is that there will be tremendous opportunities for supercomputing techniques in the growing scales of AI applications. My future research will explore the confluence of AI and supercomputing along the following dimensions.

Federated Machine Learning (ML)

As our society is becoming increasingly aware of user privacy and data confidentiality, data cannot be easily shared across multiple parties. Data sharing relates to corporate security and confidentiality concerns. AI applications must comply with privacy protection laws like the General Data Protection Regulation (GDPR). Traditional ML requires training data to be aggregated into a single machine or a data centre, which is in conflict with privacy protection. Federated ML, a decentralized ML approach, enables users to collaboratively

learn a model while keeping sensitive data private. In contrast to traditional distributed optimization, its characteristics are: (1) **Massively Distributed**: Data points are stored across a large number of compute nodes (e.g. mobile devices). The average number of samples per node is much smaller than the number of nodes. (2) **Highly Non-IID Data**: Data on each node may be drawn from a different distribution. No node has a representative sample of the overall distribution. (3) **Load Unbalanced**: Some nodes have a few examples but some have orders of magnitude more. (4) **Limited Communication**: Only a few rounds of unreliable communication are possible. The communication lower bound and convergence rate of Federated ML are unclear now. My central goal is to design algorithms that minimize the communication needed to train a good model. Traditional machine learning with synchronous optimization is usually reproducible and deterministic. The appearance of non-determinism in a modern system often signals a serious bug, and so having reproducibility as a goal greatly facilitates debugging. However, Federated ML typically does not provide reproducibility and is therefore more difficult to debug, which is a big concern for the real-world deployment. Moreover, compressing the data over the interconnect also has an influence on the convergence rate. My central goal is to design accurate, communication-efficient, and reproducible Federated ML algorithms. It is worth noting that LARS (You, Gitman, and Ginsburg, 2017) is powering some of Google's Federated Learning projects.

Hyper-Scale Machine Learning

The current optimization approaches still fail to fully utilize the increasing performance of supercomputers. The increasing parallelism requires future optimizers to continue to scale up. For example, a TPU Pod needs at least a batch size of 256K for an ImageNet-level dataset to reach its peak performance. On the other hand, researchers are collecting larger datasets by the auto-labeling technique. As a result, we will have enormous parallelism if we can put all the available data samples in the memory of a future supercomputer. One possibly interesting method is a variant of Gradient Descent (GD), which scales linearly with the dataset. The number of iterations of GD can be O(1). Moreover, the communication often brings a significant overhead to modern distributed systems. The number of network communication messages of GD can also be O(1). However, the concern is that the SGD-variant optimizer achieves a much better testing accuracy than the GD-variant optimizer. The GD-variant optimizer suffers a poor generalization performance on modern machine learning datasets. I would like to investigate various optimization techniques and invent new auto-learning techniques to improve the generalization performance of the GD-variant optimizer.

AutoML for Algorithm and System Co-design

The growing complexity of the cloud-based supercomputer is a nontrivial burden for the ML researchers and data scientists. In the ideal situation, our AI-HPC system needs to automatically select the arithmetic format, pick the hardware, deploy the resources, tune the hyper-parameters, and even write the code for any given application. Our AI-HPC system

should reach the peak performance in the cheapest way. The users should only need to input the high-level requirements. There is a huge design space in each part, which makes AutoML highly necessary. For example, let us use the arithmetic format as an example. Beyond traditional floating point and blocked floating point, there are formats people are exploring where the total number of bits is fixed, but the number allocated to exponent and mantissa can vary (which is encoded in each number). A recent example is Unums or universal numbers (Gustafson, 2015). The original proposal, Type I unum (Gustafson, 2015), had variable widths (making them very expensive to access in arrays). More recently a new format (Gustafson, 2017) has bee proposed called posits (Type III unum), with fixed width. We need to design a smart algorithm to explore the best format for a given deep learning application in a short time.

Energy-Efficient Deep Learning (DL)

Energy consumption is a big concern for DL, especially the search and tuning in AutoML. The estimated CO_2 emissions per person of one airline trip between New York and San Francisco is 1,984 lbs. However, the estimated CO_2 emissions of tuning a state-of-the-art NLP model is 626,155 lbs. This motivates designing better training schemes to reduce the energy consumption. First, we need auto-tuning algorithms to reduce the cost. We have done some early research in the DATE framework (You, Hseu, et al., 2019). Furthermore, we need to identify the high-energy operations and the low-energy operations. We can reorganize the algorithm to avoid the high-energy operations. I plan to start from communication-avoiding algorithms (Demmel, 2013).

7.3 Impact

The techniques in this thesis have been used in some state-of-the-art systems. For example, LAMB became an official optimizer of NVIDIA[1]. According to NVIDIA, LAMB optimizer is 14× faster than ADAM for language modeling applications (Figure 7.1). The LAMB optimizer is being used in real-world applications by state-of-the-art models such as ALBERT (Z. Lan et al., 2019), ELECTRA (Clark et al., 2020), and Chinese BERT (Cui et al., 2019). LAMB helped Google achieve state-of-the-art results on GLUE (A. Wang et al., 2018), RACE (Lai et al., 2017), and SQuAD (Rajpurkar et al., 2016) benchmarks. The training recipe of our LAMB paper (You, J. Li, et al., 2019) was used by NVIDIA to scale deep learning to 1,472 V100 GPUs[2]. LAMB helped fast.ai scaled Transformer-XL to 128 GPUs[3]. LAMB is also being used by Microsoft's DeepSpeed system[4].

[1] https://people.eecs.berkeley.edu/~youyang/publications/nvlamb_screenshot_March_6th_2020.png

[2] https://devblogs.nvidia.com/training-bert-with-gpus

[3] https://medium.com/south-park-commons/scaling-transformer-xl-to-128-gpus-85849508ec35

[4] https://people.eecs.berkeley.edu/~youyang/record/deepspeed.pdf

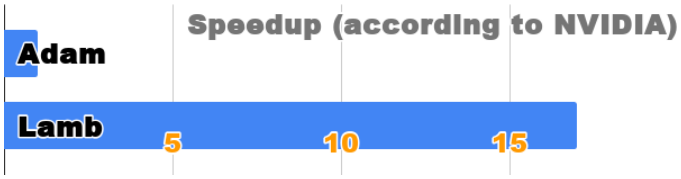


Figure 7.1: According to NVIDIA, LAMB optimizer is 14× faster than ADAM for language modeling applications.

As mentioned in the Introduction section, all the ImageNet training speed world records were created by the LARS optimizer since 2017 (Table 1.1). Recently, state-of-the-art results of self-supervised learning and semi-supervised learning were achieved by BYOL (Grill et al., 2020) and SimCLR (T. Chen et al., 2020), respectively. Both BYOL and SimCLR used our LARS optimizer. The LARS optimizer became a baseline optimizer in MLPerf (Mattson et al., 2019), which is the industry metric for fast deep learning.

7.4 Technical Acknowledgements

This thesis is based on the following publications:

- Chapter 2 is based on a joint work with James Demmel, Kent Czechowski, Le Song, and Rich Vuduc. It was published as a journal paper entitled *Design and implementation of a communication-optimal classifier for distributed kernel support vector machines* (You, Demmel, Kent Czechowski, L. Song, and Rich Vuduc, 2016).
- Chapter 3 is based on a joint work with James Demmel, Cho-Jui Hsieh, and Richard Vuduc. It was published as a conference paper entitled *Accurate, fast and scalable kernel ridge regression on parallel and distributed systems* (You, Demmel, Cho-Jui Hsieh, et al., 2018).
- Chapter 4 is based on a joint work with Xiangru Lian, Ji Liu, Hsiang-Fu Yu, Inderjit S Dhillon, James Demmel, and Cho-Jui Hsieh. It was published as a conference paper entitled *Asynchronous parallel greedy coordinate descent* (You, Lian, et al., 2016).
- Chapter 5 is based on a joint work with Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. It was published as a journal paper entitled *Fast deep neural network training on distributed systems and cloud TPUs* (You, Z. Zhang, Cho-Jui Hsieh, et al., 2018).
- Chapter 6 is based on a joint work with Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. It was published as a conference paper entitled *Large Batch Optimization for Deep Learning: Training BERT in 76 minutes* (You, J. Li, et al., 2019).

All the TPU results are based on the public cloud TPU system, we did not present any data related to Google's internal TPU system. The Layer-wise Adaptive Rate Scaling (LARS) algorithm (simulated on a single node) was developed by Y. You, B. Ginsburg, and I. Gitman when Y. You was an intern at NVIDIA (You, Gitman, and Ginsburg, 2017). All the algorithms, implementations and experiments on distributed systems are done by You, Z. Zhang, Cho-Jui Hsieh, et al., 2018. We got the TPU access from Google's TFRC program (https://www.tensorflow.org/tfrc). We thank Alex Passos for providing a detailed introduction to AutoGraph in a talk. We also thank the discussions with Frank Chen, Sameer Kumar, David Patterson, Shawn Wang and Cliff Young. The work presented in this work was supported by the National Science Foundation, through the Stampede 2 (OAC-1540931) award. By working with Prof. James Demmel, Yang You was supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC0010200; by DARPA Award Number HR0011-12-2-0016, ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle and Samsung. Other industrial sponsors include Mathworks and Cray. In addition to ASPIRE sponsors, Yang You is supported by an auxiliary Deep Learning ISRA from Intel. We thank CSCS for granting us access to Piz Daint resources.

Bibliography

Akiba, Takuya, Shuji Suzuki, and Keisuke Fukuda (2017). "Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes". In: *arXiv preprint arXiv:1711.04325*.

Amodei, Dario et al. (2015). "Deep speech 2: End-to-end speech recognition in english and mandarin". In: *arXiv preprint arXiv:1512.02595*.

Anderson, Edward et al. (1999). *LAPACK Users' guide*. Vol. 9. Siam.

Avron, H., A. Druinsky, and A. Gupta (2014). "Revisiting asynchronous linear solvers: Provable convergence rate through randomization". In: *IEEE International Parallel and Distributed Processing Symposium*.

Barndorff-Nielsen, Ole E and Neil Shephard (2004). "Econometric analysis of realized covariation: High frequency based covariance, regression, and correlation in financial economics". In: *Econometrica* 72.3, pp. 885–925.

Bengio, Yoshua (2012). "Practical recommendations for gradient-based training of deep architectures". In: *Neural networks: Tricks of the trade*. Springer, pp. 437–478.

Bernstein, Jeremy et al. (2018). "signSGD: compressed optimisation for non-convex problems". In: *CoRR* abs/1802.04434.

Bertin-Mahieux, Thierry et al. (2011). "The million song dataset". In: *ISMIR 2011: Proceedings of the 12th International Society for Music Information Retrieval Conference, October 24-28, 2011, Miami, Florida*. University of Miami, pp. 591–596.

Bertsekas, Dimitri P. (1999). *Nonlinear Programming*. Second. Belmont, MA 02178-9998: Athena Scientific.

Blackard, Jock, Denis Dean, and Charles Anderson (1998). *Covertype Data Set*. URL: `https://archive.ics.uci.edu/ml/datasets/Covertype`.

Blanchard, Gilles and Nicole Krämer (2010). "Optimal learning rates for kernel conjugate gradient regression". In: *Advances in Neural Information Processing Systems*, pp. 226–234.

Boser, B. E., I. Guyon, and V. Vapnik (1992). "A training algorithm for optimal margin classifiers". In: *COLT*.

Canutescu, A. and R Dunbrack (2003). "Cyclic coordinate descent: A robotics algorithm for protein loop closure". In: *Protein Science*.

Cao, Li Juan and SS Keerthi (2006). "Parallel sequential minimal optimization for the training of support vector machines". In: *IEEE Transactions on Neural Networks* 17.4, pp. 1039–1049.

Catanzaro, Bryan, Narayanan Sundaram, and Kurt Keutzer (2008). "Fast support vector machine training and classification on graphics processors". In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 104–111.

Chang, Chih-Chung and Chih-Jen Lin (2000). *LIBSVM: Introduction and Benchmarks*. Tech. rep. Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.

— (2011). "LIBSVM: a library for support vector machines". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 2.3, p. 27.

Chang, Edward Y (2011). "PSVM: Parallelizing support vector machines on distributed computers". In: *Foundations of Large-Scale Multimedia Information Management and Retrieval*. Springer, pp. 213–230.

Chen, Jianmin et al. (2016). "Revisiting distributed synchronous SGD". In: *arXiv preprint arXiv:1604.00981*.

Chen, Ting et al. (2020). "A simple framework for contrastive learning of visual representations". In: *arXiv preprint arXiv:2002.05709*.

Choi, Jaeyoung et al. (1995). "ScaLAPACK: A portable linear algebra library for distributed memory computersâDesign issues and performance". In: *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*. Springer, pp. 95–106.

Clark, Kevin et al. (2020). "Electra: Pre-training text encoders as discriminators rather than generators". In: *arXiv preprint arXiv:2003.10555*.

Codreanu, Valeriu, Damian Podareanu, and Vikram Saletore (2017). "Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train". In: *arXiv preprint arXiv:1711.04291*.

Coleman, Cody et al. (2017). "Dawnbench: An end-to-end deep learning benchmark and competition". In: *Training* 100.101, p. 102.

Cortes, Corina and Vladimir Vapnik (1995). "Support-vector network". In: *Machine Learning* 20, pp. 273–297.

Cortes, Corinna and Vladimir Vapnik (1995). "Support-vector networks". In: *Machine learning* 20.3, pp. 273–297.

Cui, Yiming et al. (2019). "Pre-training with whole word masking for chinese bert". In: *arXiv preprint arXiv:1906.08101*.

Das, Dipankar et al. (2016). "Distributed deep learning using synchronous stochastic gradient descent". In: *arXiv preprint arXiv:1602.06709*.

Dean, Jeffrey et al. (2012a). "Large scale distributed deep networks". In: *Advances in neural information processing systems*, pp. 1223–1231.

— (2012b). "Large scale distributed deep networks". In: *NIPS*.

Demmel, James (2013). "Communication-Avoiding Algorithms for Linear Algebra and Beyond". In: *Keynote Speech of 27th International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, pp. 585–585.

Demmel, James et al. (2013). "Perfect strong scaling using no additional energy". In: *2013 International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE.

Deng, Jia et al. (2009). "Imagenet: A large-scale hierarchical image database". In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, pp. 248–255.

Devarakonda, Aditya, Maxim Naumov, and Michael Garland (2017). "AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks". In: *arXiv preprint arXiv:1712.02029*.

Devlin, Jacob et al. (2018). "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805*.

Dhillon, I. S., P. Ravikumar, and A. Tewari (2011). "Nearest Neighbor based Greedy Coordinate Descent". In: *NIPS*.

Dongarra, J. (2014). *Top500 June 2014 list.* URL: `http://www.top500.org/lists/2014/06/`.

Dozat, Timothy (2016). "Incorporating nesterov momentum into adam". In:

Duchi, John C, Sorathan Chaturapruek, and Christopher Ré (2015). "Asynchronous stochastic convex optimization". In: *arXiv preprint arXiv:1508.00882*.

Eldan, Ronen and Ohad Shamir (2016). "The power of depth for feedforward neural networks". In: *Conference on learning theory*, pp. 907–940.

Elman, Jeffrey L (1990). "Finding structure in time". In: *Cognitive science* 14.2, pp. 179–211.

Fan, Rong-En, Pai-Hsuen Chen, and Chih-Jen Lin (2005). "Working set selection using second order information for training support vector machines". In: *The Journal of Machine Learning Research* 6, pp. 1889–1918.

Fine, Shai and Katya Scheinberg (2002). "Efficient SVM training using low-rank kernel representations". In: *The Journal of Machine Learning Research* 2, pp. 243–264.

Forgy, Edward W (1965). "Cluster analysis of multivariate data: efficiency versus interpretability of classifications". In: *Biometrics* 21, pp. 768–769.

Ge, Rong et al. (2018). "Rethinking learning rate schedules for stochastic optimization". In:

Ghadimi, Saeed and Guanghui Lan (2013a). "Stochastic First- and Zeroth-Order Methods for Nonconvex Stochastic Programming". In: *SIAM Journal on Optimization* 23.4, pp. 2341–2368. DOI: `10.1137/120880811`.

— (2013b). "Stochastic first-and zeroth-order methods for nonconvex stochastic programming". In: *SIAM Journal on Optimization* 23.4, pp. 2341–2368.

Ghadimi, Saeed, Guanghui Lan, and Hongchao Zhang (2014). "Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization". In: *Mathematical Programming* 155.1-2, pp. 267–305.

Goldstein, AA (1977). "Optimization of Lipschitz continuous functions". In: *Mathematical Programming* 13.1, pp. 14–22.

Goodfellow, Ian et al. (2016). *Deep learning*. Vol. 1. MIT press Cambridge.

Goyal, Priya et al. (2017). "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *arXiv preprint arXiv:1706.02677*.

Graf, Hans Peter et al. (2004). "Parallel support vector machines: The Cascade SVM". In: *Advances in neural information processing systems* 17, pp. 521–528.

Grama, Ananth (2003). *Introduction to parallel computing*. Pearson Education.

Grama, Ananth Y, Anshul Gupta, and Vipin Kumar (1993). "Isoefficiency: Measuring the scalability of parallel algorithms and architectures". In: *IEEE Parallel & Distributed Technology: Systems & Applications* 1.3, pp. 12–21.

Grill, Jean-Bastien et al. (2020). "Bootstrap Your Own Latent: A New Approach to Self-Supervised Learning". In: *arXiv preprint arXiv:2006.07733*.

Gropp, William et al. (1996). "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel computing* 22.6, pp. 789–828.

Gustafson, John L (2015). *The End of Error: Unum Computing*. CRC Press.

— (2017). "Posit arithmetic". In: *Mathematica Notebook describing the posit number system* 30.

Guyon, Isabelle et al. (2004). "Result analysis of the NIPS 2003 feature selection challenge". In: *Advances in Neural Information Processing Systems*, pp. 545–552.

He, Kaiming et al. (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Hoffer, Elad, Itay Hubara, and Daniel Soudry (2017). "Train longer, generalize better: closing the generalization gap in large batch training of neural networks". In: *arXiv preprint arXiv:1705.08741*.

Hofmann, Thomas, Bernhard Schölkopf, and Alexander J Smola (2008). "Kernel methods in machine learning". In: *The annals of statistics*, pp. 1171–1220.

Hsieh, C. J., S. Si, and I. S. Dhillon (2014). "A Divide-and-Conquer Solver for Kernel Support Vector Machines". In: *ICML*.

Hsieh, C.-J., H. F. Yu, and I. S. Dhillon (2015). "PASSCoDe: Parallel ASynchronous Stochastic dual Coordinate Descent". In: *International Conference on Machine Learning(ICML),*

Hsieh, Cho-Jui, Kai-Wei Chang, et al. (2008). "A dual coordinate descent method for large-scale linear SVM". In: *ICML*.

Hsieh, Cho-Jui and Inderjit S. Dhillon (2011). "Fast Coordinate Descent Methods with Variable Selection for Non-negative Matrix Factorization". In: *KDD*.

Hsieh, Cho-Jui, Si Si, and Inderjit S Dhillon (2013). "A Divide-and-Conquer Solver for Kernel Support Vector Machines". In: *arXiv preprint arXiv:1311.0914*.

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

Hull, Jonathan J. (1994). "A database for handwritten text recognition research". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 16.5, pp. 550–554.

Iandola, Forrest N et al. (2016). "FireCaffe: near-linear acceleration of deep neural network training on compute clusters". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2592–2600.

Jia, Xianyan et al. (2018). "Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes". In: *arXiv preprint arXiv:1807.11205*.

Jia, Yangqing et al. (2014). "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, pp. 675–678.

Jin, Peter H et al. (2016). "How to scale distributed deep learning?" In: *arXiv preprint arXiv:1611.04581*.

Joachims, Thorsten (1998a). "Making large-scale SVM learning practical". In: *Advances in Kernel Methods - Support Vector Learning*. MIT Press. Chap. 11.

Joachims, Thorsten (1998b). *Text categorization with support vector machines: Learning with many relevant features*. Springer.

— (1999). "Making large scale SVM learning practical". In: *Advances in Kernel Methods â Support Vector Learning*. MIT Press, pp. 169–184.

Jouppi, Norman P et al. (2017). "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, pp. 1–12.

Keskar, Nitish Shirish et al. (2016). "On large-batch training for deep learning: Generalization gap and sharp minima". In: *arXiv preprint arXiv:1609.04836*.

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

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

Kumar, Sameer et al. (2019). "Scale MLPerf-0.6 models on Google TPU-v3 Pods". In: *arXiv preprint arXiv:1909.09756*.

Lai, Guokun et al. (2017). "Race: Large-scale reading comprehension dataset from examinations". In: *arXiv preprint arXiv:1704.04683*.

Lan, Zhenzhong et al. (2019). "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations". In: *arXiv preprint arXiv:1909.11942*.

Leslie, Christina, Eleazar Eskin, and William Stafford Noble (2002). "The spectrum kernel: A string kernel for SVM protein classification". In: *Proceedings of the Pacific symposium on biocomputing*. Vol. 7. Hawaii, USA., pp. 566–575.

Li, M. et al. (2014). "Scaling distributed machine learning with the parameter server". In: *OSDI*.

Li, Mu (2017). "Scaling Distributed Machine Learning with System and Algorithm Co-design". PhD thesis. Intel.

Liao, Wei-Keng (2013). *Parallel K-Means*. URL: `http://users.eecs.northwestern.edu/~wkliao/Kmeans/`.

Lin, Chih-Jen (2017). *LIBSVM Machine Learning Regression Repository*. URL: `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

Lin, Tsung-Kai and Shao-Yi Chien (2010). "Support vector machines on gpu with sparse matrix format". In: *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*. IEEE, pp. 313–318.

Liu, J. and S. J. Wright (2014). "Asynchronous Stochastic Coordinate Descent: Parallelism and Convergence Properties". In: URL: `http://arxiv.org/abs/1403.3862`.

Liu, J., S. J. Wright, et al. (2014). "An asynchronous parallel stochastic coordinate descent algorithm". In: *ICML*.

Loshchilov, Ilya and Frank Hutter (2017). "Fixing weight decay regularization in adam". In: *arXiv preprint arXiv:1711.05101*.

Lu, Yichao et al. (2013). "Faster ridge regression via the subsampled randomized hadamard transform". In: *Advances in Neural Information Processing Systems*, pp. 369–377.

Luong, Minh-Thang, Eugene Brevdo, and Rui Zhao (2017). *Neural machine translation (seq2seq) tutorial*.

March, William B, Bo Xiao, and George Biros (2015). "ASKIT: Approximate skeletonization kernel-independent treecode in high dimensions". In: *SIAM Journal on Scientific Computing* 37.2, A1089–A1110.

Martens, James and Roger Grosse (2015). "Optimizing neural networks with kronecker-factored approximate curvature". In: *International conference on machine learning*, pp. 2408–2417.

Mattson, Peter et al. (2019). "MLPerf training benchmark". In: *arXiv preprint arXiv:1910.01500*.

Mikami, Hiroaki et al. (2018). "ImageNet/ResNet-50 Training in 224 Seconds". In: *arXiv preprint arXiv:1811.05233*.

Mitliagkas, Ioannis et al. (2016). "Asynchrony begets momentum, with an application to deep learning". In: *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*. IEEE, pp. 997–1004.

NERSC (2016). *NERSC Computational Systems*. URL: `https://www.nersc.gov/users/computational-systems/`.

Nesterov, Yurii E (1983). "A method for solving the convex programming problem with convergence rate O (1/kˆ 2)". In: *Dokl. akad. nauk Sssr*. Vol. 269, pp. 543–547.

— (2012). "Efficiency of coordinate descent methods on huge-scale optimization problems". In: *SIAM Journal on Optimization* 22.2, pp. 341–362.

Niu, F. et al. (2011). "HOGWILD!: a lock-free approach to parallelizing stochastic gradient descent". In: *NIPS*, pp. 693–701.

Nutini, J. et al. (2015). "Coordinate Descent Converges Faster with the Gauss-Southwell Rule Than Random Selection". In: *ICML*.

Osawa, Kazuki et al. (2018). "Second-order Optimization Method for Large Mini-batch: Training ResNet-50 on ImageNet in 35 Epochs". In: *arXiv preprint arXiv:1811.12019*.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013). "On the difficulty of training recurrent neural networks". In: *International Conference on Machine Learning*, pp. 1310–1318.

Platt, John C (1999). "Fast Training of Support Vector Machines using Sequential Minimal Optimization". In: *Advances in Kernel Methods â Support Vector Learning*. MIT Press, pp. 185–208.

— (1998). "Fast training of support vector machines using sequential minimal optimization". In: *Advances in Kernel Methods - Support Vector Learning*. Ed. by Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola. Cambridge, MA: MIT Press. Chap. 12.

Prokhorov, Danil (2001). "IJCNN 2001 neural network competition". In: *Slide presentation in IJCNN* 1.

Rabenseifner, Rolf (2004). "Optimization of collective reduction operations". In: *International Conference on Computational Science*. Springer, pp. 1–9.

Rabenseifner, Rolf and Jesper Larsson Träff (2004). "More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems". In: *PVM/MPI*. Springer, pp. 36–46.

Rajpurkar, Pranav et al. (2016). "Squad: 100,000+ questions for machine comprehension of text". In: *arXiv preprint arXiv:1606.05250*.

Recht, Benjamin et al. (2011). "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in Neural Information Processing Systems*, pp. 693–701.

Reddi, Sashank J., Ahmed Hefny, et al. (2016). "Stochastic Variance Reduction for Nonconvex Optimization". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pp. 314–323.

Reddi, Sashank J., Satyen Kale, and Sanjiv Kumar (2018). "On the Convergence of Adam & Beyond". In: *Proceedings of the 6th International Conference on Learning Representations.*

Richtárik, Peter and Martin Takáč (2014). "Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function". In: *Mathematical Programming* 144, pp. 1–38.

Scherrer, C. et al. (2012). "Feature Clustering for Accelerating Parallel Coordinate Descent". In: *NIPS*.

Scherrer, Chad et al. (2012). "Scaling Up Coordinate Descent Algorithms for Large l_1 Regularization Problems". In: *ICML*.

Schmidt, Mark (2005). "Least squares optimization with l1-norm regularization". In:

Schölkopf, Bernhard, Alexander Smola, and Klaus-Robert Müller (1998). "Nonlinear component analysis as a kernel eigenvalue problem". In: *Neural computation* 10.5, pp. 1299–1319.

Schreiber, R. (1986). "Solving eigenvalue and singular value problems on an undersized systolic array". In: *SIAM J. Sci. Stat. Comput.* 7. first block Jacobi reference?, pp. 441–451.

Seide, Frank et al. (2014). "On parallelizability of stochastic gradient descent for speech DNNs". In: *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, pp. 235–239.

Shalev-Shwartz, Shai and Tong Zhang (2013). "Stochastic Dual Coordinate Ascent Methods for Regularized Loss Minimization". In: *Journal of Machine Learning Research* 14, pp. 567–599.

Shallue, Christopher J et al. (2018). "Measuring the effects of data parallelism on neural network training". In: *arXiv preprint arXiv:1811.03600*.

Shazeer, Noam et al. (2018). "Mesh-tensorflow: Deep learning for supercomputers". In: *Advances in Neural Information Processing Systems*, pp. 10435–10444.

Shi, Jianbo and Jitendra Malik (2000). "Normalized cuts and image segmentation". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 22.8, pp. 888–905.

Si, Si, Cho-Jui Hsieh, and Inderjit Dhillon (2014). "Memory efficient kernel approximation". In: *Proceedings of The 31st International Conference on Machine Learning*, pp. 701–709.

Smith, Samuel L, Pieter-Jan Kindermans, and Quoc V Le (2017). "Don't Decay the Learning Rate, Increase the Batch Size". In: *arXiv preprint arXiv:1711.00489*.

Sonnenburg, Soeren et al. (2008). "PASCAL large scale learning challenge". In: *25th International Conference on Machine Learning (ICML2008) Workshop. http://largescale. first. fraunhofer. de. J. Mach. Learn. Res.* Vol. 10, pp. 1937–1953.

Sridhar, Srikrishna et al. (2013). "An approximate, efficient LP solver for LP rounding". In: *NIPS*.

Sutskever, Ilya et al. (2013). "On the importance of initialization and momentum in deep learning". In: *International conference on machine learning*, pp. 1139–1147.

Tay, Francis EH and Lijuan Cao (2001). "Application of support vector machines in financial time series forecasting". In: *Omega* 29.4, pp. 309–317.

Thakur, Rajeev, Rolf Rabenseifner, and William Gropp (2005). "Optimization of collective communication operations in MPICH". In: *The International Journal of High Performance Computing Applications* 19.1, pp. 49–66.

Tsang, Ivor W, JT-Y Kwok, and Jacek M Zurada (2006). "Generalized core vector machines". In: *Neural Networks, IEEE Transactions on* 17.5, pp. 1126–1140.

van de Geijn, Robert A (1994). "On global combine operations". In: *Journal of Parallel and Distributed Computing* 22.2, pp. 324–328.

Von Luxburg, Ulrike (2007). "A tutorial on spectral clustering". In: *Statistics and computing* 17.4, pp. 395–416.

Wang, Alex et al. (2018). "Glue: A multi-task benchmark and analysis platform for natural language understanding". In: *arXiv preprint arXiv:1804.07461*.

Wang, Po-Wei and Chih-Jen Lin (2014). "Iteration Complexity of Feasible Descent Methods for Convex Optimization". In: *Journal of Machine Learning Research* 15, pp. 1523–1548.

Webb, Steve, James Caverlee, and Calton Pu (2006). "Introducing the Webb Spam Corpus: Using Email Spam to Identify Web Spam Automatically." In: *CEAS*.

Williams, Christopher and Matthias Seeger (2001). "Using the Nystrom method to speed up kernel machines". In: *Proceedings of the 14th Annual Conference on Neural Information Processing Systems*. EPFL-CONF-161322, pp. 682–688.

WRIGHT, Nicholas J et al. (2015). "Cori: A Pre-Exascale Supercomputer for Big Data and HPC Applications". In: *Big Data and High Performance Computing* 26, p. 82.

Wu, Gang et al. (2006). "Incremental approximate matrix factorization for speeding up support vector machines". In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 760–766.

Wu, Yonghui et al. (2016). "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144*.

Xing, E. P. et al. (2015). "Petuum: A New Platform for Distributed Machine Learning on Big Data". In: *KDD*.

Yamazaki, Masafumi et al. (2019). "Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds". In: *arXiv preprint arXiv:1903.12650*.

Yao, Yuan, Lorenzo Rosasco, and Andrea Caponnetto (2007). "On early stopping in gradient descent learning". In: *Constructive Approximation* 26.2, pp. 289–315.

Yen, I. et al. (2013). "Indexed Block Coordinate Descent for Large-Scale Linear Classification with Limited Memory". In: *KDD*.

Ying, Chris et al. (2018). "Image Classification at Supercomputer Scale". In: *arXiv preprint arXiv:1811.06992*.

You, Yang, Aydın Buluç, and James Demmel (2017). "Scaling deep learning on GPU and Knights Landing clusters". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, p. 9.

You, Yang, James Demmel, Kenneth Czechowski, et al. (2015). "Ca-svm: Communication-avoiding support vector machines on distributed systems". In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, pp. 847–859.

You, Yang, James Demmel, Kent Czechowski, Le Song, and Rich Vuduc (2016). "Design and implementation of a communication-optimal classifier for distributed kernel support vector machines". In: *IEEE Transactions on Parallel and Distributed Systems* 28.4, pp. 974–988.

You, Yang, James Demmel, Kent Czechowski, Le Song, and Richard Vuduc (2015). *Appendix of CASVM*. URL: `https://sites.google.com/site/yangyouresearch/files/appendix.pdf`.

You, Yang, James Demmel, Cho-Jui Hsieh, et al. (2018). "Accurate, Fast and Scalable Kernel Ridge Regression on Parallel and Distributed Systems". In: *arXiv preprint arXiv:1805.00569*.

You, Yang, Haohuan Fu, et al. (2015). "Scaling support vector machines on modern HPC platforms". In: *Journal of Parallel and Distributed Computing* 76, pp. 16–31.

You, Yang, Igor Gitman, and Boris Ginsburg (2017). "Scaling sgd batch size to 32k for imagenet training". In: *arXiv preprint arXiv:1708.03888*.

You, Yang, Jonathan Hseu, et al. (2019). "Large-Batch Training for LSTM and Beyond". In: *arXiv preprint arXiv:1901.08256*.

You, Yang, Jing Li, et al. (2019). "Large batch optimization for deep learning: Training bert in 76 minutes". In: *International Conference on Learning Representations*.

— (2020). "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes". In: *International Conference on Learning Representations (ICLR)*.

You, Yang, Xiangru Lian, et al. (2016). "Asynchronous parallel greedy coordinate descent". In: *Advances in Neural Information Processing Systems*, pp. 4682–4690.

You, Yang, Shuaiwen Leon Song, et al. (2014). "MIC-SVM: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, pp. 809–818.

You, Yang, Shuaiwen Song, et al. (2014). "MIC-SVM: Designing A Highly Efficient Support Vector Machine For Advanced Modern Multi-Core and Many-Core Architectures". In: *2014 International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE.

You, Yang, Zhao Zhang, James Demmel, et al. (2017). "ImageNet Training in 24 Minutes". In: *arXiv preprint arXiv:1709.05011*.

You, Yang, Zhao Zhang, Cho-Jui Hsieh, et al. (2018). "Imagenet training in minutes". In: *Proceedings of the 47th International Conference on Parallel Processing*. ACM, p. 1.

— (2019). "Fast Deep Neural Network Training on Distributed Systems and Cloud TPUs". In: *IEEE Transactions on Parallel and Distributed Systems*.

Yu, H.-F. et al. (2013). "Parallel Matrix Factorization for Recommender Systems". In: *KAIS*.

Yun, Hyokun et al. (2014). "NOMAD: Non-locking, stochastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion". In: *VLDB*.

Zaheer, Manzil et al. (2018). "Adaptive Methods for Nonconvex Optimization". In: *Advances in Neural Information Processing Systems*, pp. 9815–9825.

Zanni, Luca, Thomas Serafini, and Gaetano Zanghirati (2006). "Parallel software for training large scale support vector machines on multiprocessor systems". In: *The Journal of Machine Learning Research* 7, pp. 1467–1492.

Zhang, H. (2015). "The restricted strong convexity revisited: Analysis of equivalence to error bound and quadratic growth". In: *ArXiv e-prints*.

Zhang, Jingzhao et al. (2019). "Why ADAM Beats SGD for Attention Models". In: *CoRR* abs/1912.03194.

Zhang, Sixin, Anna E Choromanska, and Yann LeCun (2015). "Deep learning with elastic averaging SGD". In: *Advances in Neural Information Processing Systems*, pp. 685–693.

Zhang, Yuchen, John Duchi, and Martin Wainwright (2013). "Divide and conquer kernel ridge regression". In: *Conference on Learning Theory*, pp. 592–617.