RICE UNIVERSITY

Distributed Machine Learning Scale Out with Algorithms and Systems

By

Binhang Yuan

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

Christopher Jermaine
Christopher Jermaine (Dec 4, 2020 16:50 CST)

Christopher Jermaine

Chair
Professor,
Computer Science, Rice University

Ce Zhang
Ce Zhang (Nov 29, 2020 23:04 GMT+1)

Ce Zhang

Assistant Professor,
Computer Science, ETH Zurich

Αναστασιος Κυριλλιδης
Αναστασιος Κυριλλιδης (Nov 29, 2020 06:41 GMT+2)

Anastasios Kyrillidis

Assistant Professor,
Computer Science, Rice University

Yingyan Lin
Yingyan Lin (Nov 29, 2020 23:08 CST)

Yingyan Lin

Assistant Professor,
Electrical and Computer Engineering,
RiceUniversity

HOUSTON, TEXAS

November 2020

ABSTRACT

Distributed Machine Learning Scale Out with Algorithms and Systems

by

Binhang Yuan

Machine learning (ML) is ubiquitous, and has powered the recent success of artificial intelligence. However, the state of affairs with respect to distributed ML is far from ideal. TensorFlow and PyTorch simply crash when an operation's inputs and outputs cannot fit on a GPU for model parallelism, or when a model cannot fit on a single machine for data parallelism. A TensorFlow code that works reasonably well on a single machine with eight GPUs procured from a cloud provider often runs slower on two machines totaling sixteen GPUs.

In this thesis, I propose solutions at both algorithm and system levels in order to scale out distributed ML. At the algorithm level, I propose a new method to distributed neural network learning, called independent subnet training (IST). In IST, per iteration, a neural network is decomposed into a set of subnetworks of the same depth as the original network, each of which is trained locally, before the various subnets are exchanged and the process is repeated. IST training has many advantages including reduction of communication volume and frequency, implicit extension to model parallelism, and memory limit decrease in each compute site. At the system level, I believe that proper computational and implementation abstractions will allow for the construction of self-configuring, declarative ML systems, especially when the goal is to execute tensor operations for ML in a distributed environment, or partitioned

across multiple AI accelerators (ASICs). To this end, I first introduce a tensor relational algebra (TRA), which is expressive to encode any tensor operation that can be written in the Einstein notation, and then consider how TRA expressions can be re-written into an implementation algebra (IA) that enables effective implementation in a distributed environment, as well as how expressions in the IA can be optimized. The empirical study shows that the optimized implementation provided by IA can reach or even out-perform carefully engineered HPC or ML systems for large scale tensor manipulations and ML workflows in distributed clusters.

Acknowledgement

In the process of completing this thesis, I have received great support and inspiration from both colleagues in Rice University and my external collaborators. I would like to express my sincere gratitude to each one of them for making this happen.

First and foremost, I would like to express my deepest gratitude to my PhD advisor Dr. Chris Jermaine. Without his guidance and supervision, I would not even land on the research field that I decided to work on for my professional career. I am also very grateful that he teaches me many fundamental methodologies in research without reservation, which would be of great benefit to my career. Additionally, I am deeply appreciative of Dr. Anastasios Kyrillidis's great help and advice for my research projects. His patience and consideration have been driving me towards the accomplishment of the independent subnet training project. I would also thank Dr. Yingyan Lin and Dr. Ce Zhang for serving on my committee and providing their helpful feedback on this thesis.

Second, it is my great pleasure to thank all of my colleagues in the PlinyCompute group including Dimitrije Jankov, Dr. Jia Zou, Yuxin Tang, Daniel Bourgeois, Dr. Shangyu Luo, Dr. Zekai J. Gao, and Dr. Zhuhua Cai, for sharing their wisdom with me and contributing to the research projects relevant to this thesis. I also appreciate the help from Dr. Chen Wang, Dr. Fei Jiang, Dr. Chen Luo, Wenhui Xing and Dr. Song Ge for research projects I participate during this period but not included in this thesis.

Last but not least, my most sincere appreciation goes to my dear families and friends for being mentally supportive during this period. Without their consideration and affection, I would not accomplish this thesis.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Motivation

Machine learning (ML), especially deep learning, has become an important mechanism to discover knowledge from large data collections. This knowledge takes the form of ML models that can make automatic predictions or provide actionable hypotheses for diverse data science applications [1]. Systems such as TensorFlow [2] and PyTorch [3] have revolutionized the practice of deep learning for vision, speech, language understanding, and many other fields [4], since the engineering effort to implement gradient decent for learning ML models has been minimized by these systems. Difficult gradient computations that would have been impossible to get right "by hand" are generated quickly (without programmer involvement) via auto-differentiation [5]. Operations in the resulting compute graph are automatically mapped to high-performance CPU and GPU kernels, with little programmer involvement.

Nevertheless, the state of affairs with respect to distributed ML is still far from ideal. First and foremost, these systems typically do not scale as well as one would hope. A classic measure of the scalability for a distributed program is the ratio of the execution time on a single compute unit to the execution time in a N-site cluster. Consider an example where a data scientist has accumulated a large amount of data from his/her enterprise, and he/she prototypes a state-of-the-art deep learning model using PyTorch. A back-of-the-envelope estimation suggests that it would take two

months to train the model using all of the data by an AWS EC2 GPU instance. If PyTorch could scale out distributed ML perfectly, the data scientist would simply rent N machines to cut the training time to $1/N$ of the estimation under the same budget[*]. Unfortunately, this is not the reality for existing systems. As I show, for very large models, renting more machines can actually result in a *higher* compute time.

Another problem with existing systems such as TensorFlow and PyTorch is that they struggle to train very large models. This can cause problems for practitioners in many scenarios. For example, a computational chemist who wishes to train a graph neural network to automate the classification of new compounds in drug discovery cannot easily get TensorFlow or PyTorch to work if the model to process the complex molecular fingerprints is too large to fit in a single GPU RAM. An enterprise IT team intending to train a transformer model for a special purpose question answering system from scratch by their private data would find it difficult to put hundreds of terabytes of documents as well as the multi-GB models needed to process them into tensors that fit into GPU RAM.

1.1.1 Background

Before I discuss potential ways to address these problems, I will begin by reviewing a few important concepts.

This thesis focuses on stochastic gradient descent (SGD)-based algorithms as the optimization strategy for ML. SGD minimize a loss function by repeatedly "moving" the model's parameters in the direction of the negative gradient of the function. The procedure is "stochastic" since the gradient is calculated from a randomly sampled

[*]One usually pays for compute capacity by the hour for a cloud computing service, like AWS EC2.

subset (called a mini-batch) of the training data[†]. The loss function is typically a proxy for the actual error to be minimized, e.g., the mean squared difference (or "error") calculated between the model outputs and desired outputs in the case of a regression problem, or the negative log likelihood of the ground truth class predicted by the model in the case of classification.

In order to distribute/parallelize SGD, there are two general approaches: parallelizing the data or the model. In the so-called "data parallel approach", the dataset is partitioned across all workers in the cluster, and all workers perform SGD iterations to update the entire model, using their own subset of the data, with or without synchronization of the model. The entire model is sent to all workers (either from a centralized location, or from a distributed storage), and the update of the model requires an aggregation of the gradients from all the workers. In the so-called "model parallel approach", different components of the model are distributed to workers, so that no worker has access to all of the data. The different components of the model are each processed locally, so intermediate results of the SGD computation must be communicated through the network [6].

The challenges w.r.t distributed ML fall into two categories: i) at the algorithmic level, SGD is not easily parallelized/distributed due to its inherently sequential nature; parallelization requires frequent synchronization which is expensive and limited by the network bandwidth. Developing algorithms that immunize this communication cost is crucial. And ii) at the systems level, there is little differentiation between the logical description of the computation to be run, and the actual computation that is run by

[†]In the literature, stochastic gradient descent sometimes refers to sampling a single sample from the training data. Here, I use stochastic gradient descent as a synonym for mini-batch stochastic gradient descent.

the system. Concepts such as automatic choice of algorithm, automated distribution of data and compute, re-writes of the computation itself to add to the efficiency, and automatic allocation of computational resources in a parallel/distributed runtime are almost absent from modern ML systems.

1.1.2 My Contributions

My thesis attempts to remedy these deficiencies in modern ML algorithms and systems, and make it easy for relatively naive end-users to use distributed ML. Distributed ML should be efficient and easy, and it should scale out. This thesis discusses solutions at both algorithmic and systematic levels.

The proposed algorithmic solution attempts to reduce the synchronization overhead of distributed SGD by decomposing a neural network into a set of subnetworks of the same depth, each of which is trained locally.

To help with the design and implementation of ML systems that allow for automatic optimization of distributed computations as well as excellent scale-out, I introduce a novel relational algebra, which is expressive enough to encode various machine learning operations. This thesis discusses how this algebra can be optimized for the construction of self-configuring, declarative ML implementations, especially in a distributed environment.

1.2 Algorithmic Solution — Independent Subnet Training

1.2.1 Problem Description

Distributed training of neural networks (NN) over a compute cluster has become an essential task in modern computing systems [7, 8, 9, 10, 11]. As discussed, distributed

training algorithms may be roughly categorized into model parallelism and data parallelism. In the former [8, 11], different compute nodes are responsible for different parts of a NN. In the latter [12, 13, 14], each compute node updates a complete copy of the NN's parameters on different training data. In both cases, the obvious way to speed up learning is to add more nodes. With more hardware, the model is split across more CPUs/GPUs in the model parallel setting, or gradients are computed using fewer data objects per compute node in the data parallel setting.

Due to its ease-of-implementation, data parallel training is most commonly used, and it is better supported by common deep learning software, such as TensorFlow and PyTorch. However, there are limitations preventing data parallelism from easily scaling out. Adding nodes means that each node can perform forward and backward propagation more quickly on its own local data, but it leaves the synchronization step no faster. In fact, if synchronization time dominates, adding more machines could actually make training even slower as the number of bytes transferred to broadcast an updated model grows linearly with cluster size. This is particularly problematic in public clouds, such as Amazon EC2[‡], that tend to couple relatively slow interconnects with high-performance GPUs, meaning that transfer costs dominate. One can increase batch size to decrease the relative cost of the synchronization step, but this can introduce statistical inefficiency. While there is some debate about the utility of large-batch methods in practice [15, 16], very large batch sizes often do not speed up convergence unless special configuration and tuning is performed, and large batches can also hurt generalizability [17, 18, 19, 20, 21, 22, 23].

[‡]89% of cloud-based deep learning projects are executed on EC2, according to Amazon's marketing materials.

1.2.2 Proposed Solution

To this end, a method called **independent subnet training** (IST) is proposed to facilitate combined model and data parallel distributed training at the algorithmic level. IST utilizes ideas from dropout [24] and approximate matrix multiplication [25] — IST decomposes the NN layers into a set of *subnets* for the same task, by partitioning the neurons across different sites. Each of those subnets is trained for one or more local SGD iterations, before synchronization.

Since subnets share no parameters in the distributed setting, synchronization requires no aggregation on these parameters, in contrast to the data parallel method — it is just an exchange of parameters. Moreover, because subnets are sampled without replacement, the interdependence among them is minimized, which allows their local SGD updates for a larger number of iterations without significant "model drifting", before synchronizing. This reduces communication frequency. Communication costs per synchronization step are also reduced because in an n-machine cluster, each machine gets between $\frac{1}{n^2}$ and $\frac{1}{n}$ of the weights — contrast this to data parallel training, when each machine must receive all of the weights.

IST has advantages over model parallel approaches. Since subnets are trained independently during local updates, no synchronization between subnetworks is required. Yet, IST inherits the advantages of model parallel methods. Since each machine gets just a small fraction of the overall model, IST allows the training of very large models that cannot fit into the RAM of a node or a device. This can be an advantage when training large models using GPUs, which tend to have limited memory.

1.2.3 Highlight of Empirical Study

An extensive empirical study is conducted to evaluate IST's ability to provide speedups and scalability on benchmarks concerned with speech recognition, image classification (CIFAR10 and full ImageNet), and a large-scale Amazon product recommendation task.

The experiments show that IST not only shows great advantages in processing the training data (in fact, IST shows super-linear scalability), but also results in up to a 10.6× speedup for end-to-end time-to-convergence, compared to a state-of-the-art data parallel realization, using bandwidth-optimal ring all-reduce [26], as well as a 4.3× speedup compared to the "vanilla" local SGD method [27].

Because IST allows for efficient implicit model parallel training, IST can solve an "extreme" Amazon product recommendation task with improved generalization, by supporting very high embedding dimensionality for large models, which is not easily supported by data parallel based training. The precisions @1, @3, and @5 for the task, are improved by 5.23%, 4.90% and 4.94% respectively, compared to the data parallel approach.

1.3 ML Systems Design — Tensor Relation Algebra

1.3.1 Problem Description

Systems such as TensorFlow and PyTorch successfully make ML computation declarative (at least within a single machine) by automatically constructing computation graph via auto-differentiation [5]. Operations in the resulting compute graph are then automatically mapped to hardware, with little programmer involvement.

However, the state-of-the-art of distributed ML systems is far from perfect. Ten-

sorFlow and PyTorch simply crash when an operation's inputs and outputs cannot fit on a GPU for model parallelism, or when a model cannot fit on a single machine for data parallelism. A TensorFlow code that works reasonably well on a single machine with eight GPUs procured from a cloud provider often runs slower on two machines totaling sixteen GPUs.

The fundamental problem is lack of abstraction in modern distributed ML systems. A user-requested operation such as a matrix multiply on TensorFlow or PyTorch is not a logical operation that the system figures out how to execute. Rather, it is a physical operation that has to be run as a kernel operation somewhere, on some hardware. These systems do not treat a compute graph as an abstract computation that is to be optimized and mapped to hardware, especially for a distributed runtime.

Modern ML systems were developed without asking: what are the foundational abstractions necessary for building such systems? Consider the development history of relational database management systems (RDBMSs) in the decades of the 1970's and 1980's. RDBMSs were designed by asking and answering a series of foundational questions:

1. What is the foundational computational abstraction upon which database systems should be built? That is, what is the "language" of database systems?

2. What is the implementation abstraction necessary to realize that computational abstraction?

3. And finally, how should that implementation abstraction be constructed in a real-life system?

In the context of database systems, the answer to the first question was first-order logic (FOL) [28]. The answer to the second question was relational algebra (RA),

which is computationally as powerful as FOL, but easily implementable (consisting of a small set of simple operations), and easily optimizable via a set of algebraic rewrite rules [29]. In answering the final question, the database research community designed a huge number of implementations for joins, selections, aggregations, etc.

1.3.2 Proposed Solution

In this thesis, I ask the fundamental question: what is an appropriate implementation abstraction for ML system design?

One may ask: why not simply use linear algebra as the implementation abstraction? As an example, a system may ship a large number of physical matrix multiply implementations, including multiple parallel, distributed, and local matrix multiply implementations that use (or do not use) hardware acceleration. For example, one could include a high-performance distributed ScaLAPACK implementation of the 2.5D matrix multiply algorithm [30], and build an optimization framework that allows the system to carefully design a physical plan for a given compute graph.

The problem with that approach is that the set of linear algebra operations is far too heavy and complex to serve as an implementation abstraction. This approach would require that a modern ML system includes dozens or even hundreds of operations. It is not practical to maintain a dozen implementations of each operation, nor is it practical to code up a dozen new implementations for each new operation that must be supported. Since each linear algebra operation is a separate code, an optimization applied to one physical implementation does not apply to any of the others.

In response, the proposed systematic approach introduces a simple and concise **tensor relational algebra** (TRA) over so-called tensor relations. At the highest level, a tensor relation is simply a binary relation between keys and multi-dimensional

arrays.

There are some key reasons that this makes sense as the implementation abstraction in ML system design. Crucially, just like the relational algebra, it is a set-based abstraction whose operations are easily parallelized in the same way that relational algebra is parallelized. Because it is designed to facilitate "chunking" of tensors into smaller pieces that can be operated over using efficient CPU or GPU kernels, it is by design easy to implement efficiently across multiple machines or ASICs. Second, like the relational algebra, it is simple and concise, so it should be possible to optimize ad infinitum. Finally, it is powerful. It is easy to prove (as discussion in Section 3.1.3) that the tensor relational algebra is at least as powerful as the Einstein notation, a standard tensor calculus. Hence, it can be used to implement anything that can be written in the Einstein notation (including matrix multiplications, convolutions, and so on).

The proposed TRA is complied to an **implementation algebra** (IA) which is easy to implement in a distributed system. The system considers how computations in the TRA can be transformed into computations in the IA, and leverages a set of transformations or equivalences that allow re-writes of computations in the IA.

1.3.3 Highlight of Empirical Study

A prototype of the TRA is implemented to evaluate if TRA can enable efficient, distributed implementations of ML computations. The empirical study aims at answering two key questions: i) given a dataset and a ML workflow/tensor computation, can the proposed system be used to convert a TRA expression into an expression in the IA which is optimized for, and runs well on, that particular dataset? ii) How does the execution time of such an optimized computation compare to what one might

expect for a competing, high-performance engine? To answer these questions, three benchmark tasks are designed: (i) distributed matrix multiplication, (ii) distributed nearest neighbor search in a Riemannian metric space, and (iii) distributed SGD in a two-layer, feed-forward neural network (FFNN).

The TRA system enables very fast ML computations: for different matrix multiplication settings, one of the IA implementations is nearly as fast, or significantly faster (1.5× speedup) than Intel's high-performance ScaLAPACK library; for the nearest neighbor computation, the optimized IA implementation is only 6% slower than a local implementation executed on a single machine with the same computation power. The most striking results are from the FFNN benchmark. The IA implementation reaches up to 12× speedup in a GPU cluster and 9.1× speedup in a CPU cluster, compared to the most advanced PyTorch data parallel approach.

According to the experimental results, it is reasonable to believe that the proposed optimization techniques are quite effective for various ML computations, since the TRA-based system can reach or even significantly outperform existing HPC and ML systems.

Chapter 2

Independent Subnet Training

This chapter introduces the proposed algorithmic level approach called *independent subnet training* (IST), which facilitates combined model and data parallel distributed training. When sharing the model among the cluster, IST only transmits the partitioned subnet to each node instead of the whole model. This reduces the overall communication overhead per iteration. By sampling without replacement, the model is split into non-overlapping networks, which reduces the interdependence between compute nodes, so that aggressive local updates can be applied, compared to vanilla local SGD. In order to justify the design of IST, I first include a theoretical discussion w.r.t the convergence guarantee of IST; then a group of experiments including speech recognition, image classifications (CIFAR10 and full ImageNet), and a large-scale Amazon product recommendation task are performed to evaluate IST.

This chapter is organized as below: Section 2.1 formally describes IST in depth; Section 2.2 provides the convergence guarantee of IST; and Section 2.3 enumerates the empirical study to verify the effectiveness of IST.

2.1 Training via Independent Subnetworks

2.1.1 Preliminaries

NN training. The goal is to optimize a loss function $\ell(\cdot, \cdot)$ over a set of labeled examples; the loss $\ell(w, \cdot)$ encodes the NN architecture, with parameters w. Given

samples $X := \{x_i, y_i\}_{i=1}^n$, deep learning aims in finding $w^\star$ that minimizes the *empirical loss*:

$$w^\star \in \arg\min_w \frac{1}{n} \sum_{i=1}^{n} \ell\left(w, \{x_i, y_i\}\right). \tag{2.1}$$

Formula (2.1) is completed using various approaches [31, 32, 33, 34, 35], but almost all NN training is accomplished via some variation on SGD: one computes (stochastic) gradient directions $\nabla \ell_{i_t}(w_i) := \nabla \ell(w_i, \{x_{i_t}, y_{i_t}\})$ that, on expectation, decrease the loss, and then set $w_{t+1} \leftarrow w_t - \eta \nabla \ell_{i_t}(w_t)$. Here, $\eta > 0$ is the learning rate, and i_t represents a mini-batch of examples, selected from X.

Why classical distributed approaches can be ineffective? Computing $\nabla \ell(w_t, X)$ over the whole X is wasteful [36]. Instead, mini-batch SGD computes $w_{t+1} \leftarrow w_t - \eta \nabla \ell(w_t, X_{i_t})$ for a small subsample X_{i_t} of X. In a centralized system, one often uses no more than a few hundred data items in X_{i_t}, and few would advocate using more than a few tens of thousands of X_{i_t} [17, 18, 20].

For distributed computation, this is problematic for two reasons: first, it makes it difficult to speed up the computation by adding more computing hardware. Since the batch size $|X_{i_t}|$ is small, splitting the task to more than a few compute nodes is no beneficial, which motivates different training approaches for NNs [37, 38, 39, 40, 41, 42]. Second, gathering the updates in a distributed setting introduces a non-negligible time overhead in large clusters, which is often the main bottleneck towards efficient large-scale computing. This imbalance between communication and computation capacity may lead to significant *increases* in training time when a larger cluster is used.

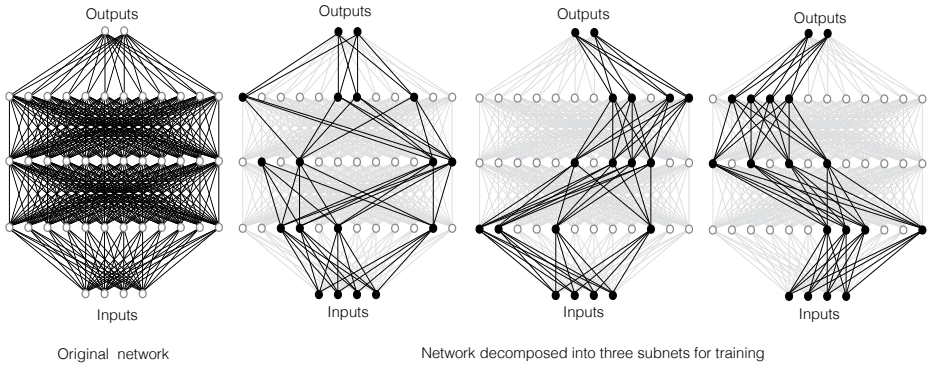

Figure 2.1 : Decomposing a NN with three hidden layers into three subnets.

2.1.2 Distributing Independent Subnets

Assume n sites in a distributed system. For simplicity, it is assumed that all layers of the NN utilize the same activation function. Let f^l denote that vector of activations at layer l. f^t denotes the set of activations at the final or "top" layer of the network, and f^0 denotes the feature vector that is input into the network. Assume that the number of neurons at layer l is N_l. IST is a randomized, distributed training regime that utilizes a set of *membership indicators*:

$$\{m^l_{s,i}\}_{s\in\{1...,n\},i\in\{1...,N_l\}}$$

Here, s ranges over the n sites, and i ranges over the neurons in layer l. Each $m^l_{s,i} \in \{0,1\}$, is randomly selected, where the marginal probability is $\mathbb{P}\left[m^l_{s,i}=1\right]=\frac{1}{n}$. Further, for each layer l and activation i, IST constrains $\sum_s m^l_{s,i}$ to be 1 and the covariance of $m^l_{s,i}$ and $m^{l-1}_{s,i'}$ must be zero, so that $\mathbb{E}\left[m^l_{s,i'}m^{l-1}_{s,i}\right]=\frac{1}{n^2}$.

Then, the definition of the recurrence at the heart of IST can be formalized as:

$$\hat{f}^l = f\left(n^2\sum_s m^l_s \odot \left(W^l\left(m^{l-1}_s \odot \hat{f}^{l-1}\right)\right)\right). \tag{2.2}$$

Here, W^l is the weight matrix connecting layer $l-1$ and layer l, and $\odot$ denotes

the Hadamard product of the two vectors. This recurrence is useful for two key reasons. First, it is easy to argue that if $\hat{f}^{l-1}$ is an unbiased estimator for f^{l-1}, then $n^2 \sum_s m_s^l \odot \left(W^l \left(m_s^{l-1} \odot \hat{f}^{l-1}\right)\right)$, is an unbiased estimator for $W^l f^{l-1}$. To show this, note that the j-th entry in the vector $n^2 \sum_s m_s^l \odot \left(W^l \left(m_s^{l-1} \dot{\wedge} f^{l-1}\right)\right)$ is computed as $n^2 \sum_s \sum_i \sum_{i'} W_{j,i}^l m_{s,i'}^l m_{s,i}^{l-1} \hat{f}_i^{l-1}$, and hence its expectation is:

$$\begin{aligned}
&\mathbb{E}\left[n^2 \sum_s \sum_i \sum_{i'} W_{j,i}^l m_{s,i'}^l m_{s,i}^{l-1} \hat{f}_i^{l-1}\right] \\
=&n^2 \sum_s \sum_i \sum_{i'} W_{j,i}^l \mathbb{E}\left[m_{s,i'}^l m_{s,i}^{l-1} \hat{f}_i^{l-1}\right] \\
=&n^2 \sum_s \sum_i \frac{1}{n^2} W_{j,i}^l \mathbb{E}\left[\hat{f}_i^{l-1}\right] \\
=&\sum_i W_{j,i}^l f_i^{l-1}
\end{aligned}$$

which is precisely the j-th entry in $W^l f^{l-1}$.

This unbiasedness suggests that this recurrence can be computed in place of the standard recurrence implemented by a NN, $f^l = f\left(W^l f^{l-1}\right)$. A feature vector can be pushed through the resulting "approximate" NN, and the final vector $\hat{f}^t$ can be used as an approximation for f^t.

The second reason the recurrence is useful is that it is much easier to distribute the computation of $\hat{f}^t$ — and its backward propagation — than that of f^t. When randomly generating $\{m_{s,i}^l\}_{s\in\{1\ldots,n\},i\in\{1\ldots,N_l\}}$, IST requires that $\sum_s m_{s,i}^l$ be 1. Two important aspects follow directly from this requirement. First, in the summation of Formula (2.2), only one "site" can contribute to the j-th entry in the vector $\hat{f}^l$; this is due to the Hadamard product with m_s^l, which implies that all other sites' contributions will be zeroed out. Second, only the entries in $\hat{f}^{l-1}$ that were *themselves* associated with the same site value for s can contribute to the j-th entry, again due to the Hadamard product with m_s^{l-1}.

This implies that IST can co-locate at site s the computation of all entries in $\hat{f}^{l-1}$ where $m_{s,i}^{l-1}$ is 1, and all entries in $\hat{f}^{l}$ where $m_{s,i}^{l}$ is 1. *No cross-site communication is required to compute the activations in layer l from the activations in layer $l-1$.* Further, since only the entries in W_j for which $m_{s,i'}^{l} m_{s,i}^{l-1} = 1$ are used at site i — and on expectation, only $\frac{1}{n^2}$ of the weights in W^l will be used — this implies that during an iteration of distributed backward propagation, each site needs only (and communicates gradients for) a fraction $\frac{1}{n^2}$ of the weights in each weight matrix.

The distributed implementation of the recurrence across three sites for a feed forward NN with three hidden layers is depicted in Figure 2.1. The convolutional NN case is described later. The neurons in each layer are partitioned randomly across the sites, except for the input layer, which is fully utilized at all sites and the output layer, which computes all of the activations at the top layer.

2.1.3 Distributed Training Algorithm

Algorithm 2 IST local SGD.

Require: subnet s: $\mathcal{W}_s^1, \mathcal{W}_s^2, \mathcal{W}_s^3, ...\rangle$, loss $\ell^{(i)}(\cdot)$, # of local iters. J, learning rate η, local batch size B

1: Let $\mathcal{W}^{(0)} = \langle \mathcal{W}_s^1, \mathcal{W}_s^2, \mathcal{W}_s^3, ...\rangle$

2: **for** $t = 1, \ldots, J$ **do**

3: Let $\mathcal{B}$ be a set of B samples from local data.

4: $\mathcal{W}^{(t)} = \mathcal{W}^{(t-1)} - \eta \cdot \nabla \ell_{\mathcal{B}}(\mathcal{W}^{(t-1)})$.

5: **end for**

6: Let $\langle \mathcal{W}_s^1, \mathcal{W}_s^2, \mathcal{W}_s^3, ...\rangle = \mathcal{W}^{(J)}$

7: Send $\langle \mathcal{W}_s^1, \mathcal{W}_s^2, \mathcal{W}_s^3, ...\rangle$ to coordinator

Algorithm 1 Independent subnet training.

Initialize weight matrices $W^1, W^2, ..., W^t$
while loss keeps improving **do**
  Sample $\{m^l_{s,i}\}_{t\in\{1...t-1\},s\in\{1...,n\},i\in\{1...,N_l\}}$
  /* Execute local SGDs */
  for each site s **do**
    /* Send weights to local SGD */
    for each layer l **do**
      Compute $\mathcal{W}^l_s = \{W^l_{i,j}\}_{\text{s.t. } m^l_{s,i}=1, m^{l-1}_{s,j}=1}$
      Send $\mathcal{W}^l_s$ to site s
    end for
    Run subnet local SGD at site s
  end for
  /* Retrieve results of local SGD */
  for each site s, layer l **do**
    Retrieve updated $\mathcal{W}^l_s$ from site s
    for each (i, j) s.t. $m^l_{s,i} = 1, m^{l-1}_{s,j} = 1$ **do**
      Update W^l by replacing $W^l_{i,j}$ with corresponding value from $\mathcal{W}^l_s$
    end for
  end for
end while

This suggests an algorithm for distributed learning, given in Algorithm 1 and Algorithm 2. Algorithm 1 repeatedly samples a set of membership indicators, and then partitions the model weights across the set of compute nodes. Since the weights

are fully partitioned, the independent subsets can be trained separately on local data for a number of iterations (Algorithm 2), before the indicators are re-sampled, and the weights are re-shuffled across the nodes. Note that periodic re-sampling of the indicators (followed by reshuffling) is necessary due to the possible accumulation of random effects. While the recurrence of Equation.(2.2) provides for an unbiased estimate for the input to a neuron, after backward propagation, the expected input to a neuron will change. Since each subset is being trained using samples from the same data distribution, this shift may be inconsistent across sites. Resampling guards against this.

2.1.4 Correcting Distributional Shift

There is, however, a significant problem with the above formulation. Specifically, by Equation.(2.2), I argued that $n^2 \sum_s m_s^l \odot \left(W^l \left(m_s^{l-1} \odot \hat{f}^{l-1}\right)\right)$ is an unbiased estimator for $W^l f^{l-1}$. Thus, it holds that $\hat{f}^l = f\left(n^2 \sum_s m_s^l \odot \left(W^l \left(m_s^{l-1} \odot \hat{f}^{l-1}\right)\right)\right)$ is a reasonable estimator for $f^l = f\left(W^l f^{l-1}\right)$. In doing so, I am guilty of applying a form of the classical statistical fallacy that for random variable x, if $\mathbb{E}[x] = b$, then $\mathbb{E}[f(x)] \approx f(b)$.

This fallacy is dangerous when the activation f is non-linear. Because the membership indicators force subsampling the inputs to each neuron (and a scale factor of n^2 is then applied to the resulting quantity to unbias it), IST ends up increasing the standard deviation of the input to each neuron by a factor of n during training, compared to the standard deviation that will be observed during inference, without the use of membership indicators. This increased variance means that IST is more likely to observe extreme inputs to each neuron during training than during actual deployment. The network learns to expect such extreme values and avoid saturation

during deployment, and adapts accordingly. However, the learned network fails when it is deployed.

To match the training and deployment distributions, IST could apply an analytic approach. Instead, IST simply removes the n^2 correction. I.e., during training, for a given neuron, IST compute the mean μ and standard deviation σ of the inputs to the neuron, and use a modified activation function $f'(x) = f\left((x-\mu)/\sigma\right)$. Before inference, IST can compute μ and σ for each neuron over a small subset of the training data using the full network, and use those values during deployment.

Note that this is similar to batch normalization [43] — IST can learn a scale and shift as well. Though the motivation for its use is somewhat different. Classically, batch normalization keeps the input in the non-saturated range of the activation function during training. This tends to speed convergence and improve generalization. Yet, IST *will simply not work* without some sort of normalization, due to the distributional shift that will be encountered when deploying the whole network.

2.1.5 Why Is This Fast?

By subsampling, IST reduces both network traffic and compute workload. In addition, IST allows for periods of local updates with no communication, again reducing network traffic.

For a feed-forward NN, at each round of "classical" data parallel training, the entire set of parameters must be broadcast to each site. Measuring the inflow to each site, the total network traffic per gradient step is (in floating point numbers transferred): $\sum_{i=1}^{t} nN_{i-1}N_i$. In contrast, during IST, each site receives the current parameters only one time every J gradient steps. Subsampling reduces this cost further; the matrices attached to the input and output layers are partitioned across nodes (not broadcast),

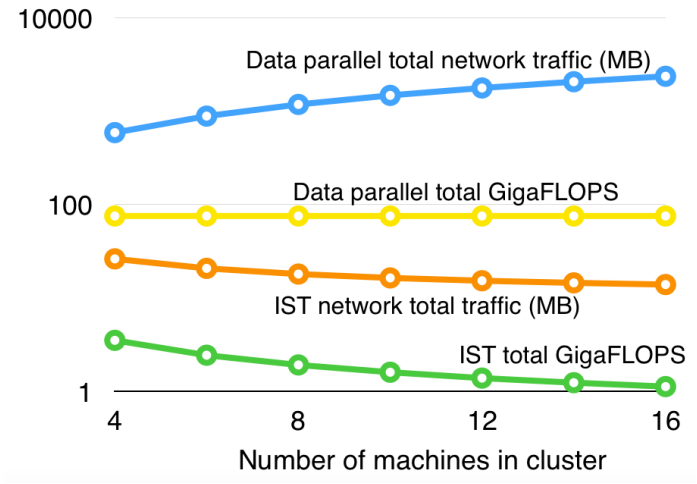

Figure 2.2 : Comparing the cost of IST with data parallel learning.

and only a $\frac{1}{n}$ fraction of the weights in each of the other matrices are sent to any node. The total network traffic per gradient step is: $\frac{N_0N_1+N_{t-1}N_t}{J} + \sum_{i=1}^{l} \frac{N_{i-1}N_i}{n \times J}$.

Computational resource utilization is reduced similarly. Considering the FLOPs required by matrix multiplications during forward and backward steps, during "classical" data parallel training, the number of FLOPS required per gradient step is: $4\sum_{i=1}^{l} BN_{i-1}N_i$. In contrast, the number of FLOPS per IST gradient step is: $4BN_0N_1 + 4BN_{t-1}N_t + 4B\sum_{i=1}^{l} \frac{N_{i-1}N_i}{n}$. Note that this also explains the reduction of RAM usage for IST, which enables training of larger models.

In Figure 2.2, the average cost of each gradient step as a function of the number of machines is plotted, assuming a feed forward NN with three hidden layers of 4,000 neurons, an input feature vector of 1,000 features, a batch size of 512 data objects, and 200 output labels, assuming J, the number of subnet local SGD steps, is 10. There is a radical decrease in both network traffic and FLOPS using IST. In particular, using IST both of these quantities *decrease* with the addition of more machines in the cluster.

Note that this plot does not tell the whole story, as IST may have lower (or higher) statistical efficiency. The fact that IST partitions the network and runs local updates may decrease efficiency, whereas the fact that each "batch" processed during IST actually consists of n independent samples of size B (compared to a single global sample in classical data parallel training) may tends to increase efficiency. This will be examined experimentally.

2.1.6 IST for other Architectures

As described, IST applies to fully-connected layers. Extending the method to other common neural constructs, such as convolutional layers, is straightforward. There are two ways that the IST decomposition can be used with a convolutional neural network (CNN).

The first and most straightforward way to do is to apply IST only to the fully-connected layer(s) that make part of nearly every modern CNN architecture. The fully-connected layers are decomposed as described in this section, but, during training, the rest of the network is broadcast to every site. Even this simple decomposition has significant benefits as the fully-connected layer(s) tend to be the most expensive to move between sites during training. Consider the full ImageNet [44] for a deep model as ResNet50: the convolutional layers have 17,614,016 parameters (67.2MB, 28.2%), whereas the fully-connected layer at the top has 44,730,368 parameters (170.6MB, 71.8%) amenable to IST. In the experimental section of this chapter, I show that this simple extension results in significant speedup of training VGG over the full ImageNet dataset.

A second and almost-as-straightforward way of applying IST to a CNN architecture is to partition the convolutional filters, by assigning non-overlapping subsets of filters

to each of the machines in the compute cluster. This natural extension is explicitly adopted to basic residual blocks in ResNet, where the intermediate activations are partitioned by filter channels. In the experimental section of this chapter, I show empirically that this natural extension can effectively leads to speedup in training ResNet over the CFAR10 dataset.

2.2 Convergence Guarantee of IST

Consider the problem of minimizing an average of loss functions:

$$x^\star \in \arg\min_{x\in\mathbb{R}^p} \left\{ f(x) := \frac{1}{n}\sum_{i=1}^{n} f_i(x) \right\}.$$

Here, $f_i(\cdot)$ denotes the contribution to the loss of the i-th data point. The components $f_i(\cdot)$ define the nature of the full function f: if f_i's are quadratics, one obtains the convex linear regression problem; if f_i's model the forward pass of a neural network, one obtains neural network inference.

In this note, the proof will consider f_i functions that do not follow exactly the architecture of a specific neural network, but satisfy general loss assumptions that could potentially be satisfied by neural network models.

Assumption 1 (L_i-smoothness) Given component f_i of f function, there exists constant $L_i > 0$ such that for every $x, y \in \mathbb{R}^p$ I have that:

$$\|\nabla f_i(x) - \nabla f_i(y)\|_2 \leq L_i \cdot \|x - y\|_2$$

or, equivalently,

$$f_i(y) \leq f_i(x) + \langle \nabla f_i(x), y - x\rangle + \tfrac{L_i}{2}\|x - y\|_2^2.$$

Further, define $L_{\max} := \max_i L_i$.

Another assumption that the proof uses in part of results is Q-Lipschitz assumption:

Assumption 2 (Q-Lipschitz continuity) Given f function, there exists constant $Q > 0$ such that for every $x, y \in \mathbb{R}^p$ I have that:

$$|f(x) - f(y)| \leq Q \cdot \|x - y\|_2$$

or, equivalently,

$$\|\nabla f(x)\|_2 \leq Q, \quad \forall x \in \mathbb{R}^p.$$

Two other assumptions for f functions that do not imply convexity but help as proof convergence rate techniques are:

Assumption 3 (Error Bound) Let $x^\star$ denote the global optimum of f. Then, under the Error Bound assumption, there exists constant $\mu > 0$ such that for every $x \in \mathbb{R}^p$ I have that:

$$\|\nabla f(x)\|_2 \geq \mu \|x^\star - x\|_2$$

Per [45], Error Bound $\equiv$ Polyak-Łojasiewicz inequality.

Regarding stochasticity in gradient descent, the proof will use the following general assumptions on the boundedness of stochastic gradient variance.

Assumption 4 (Stochastic gradient variance) I assume that there are constants $M, M_f > 0$, such that

$$\mathbb{E}_{i_t}\left[\|\nabla f_{i_t}(x)\|_2^2\right] \leq M + M_f \|\nabla f(x)\|_2^2,$$

where f_{i_t} is a randomly selected component from the sum $\frac{1}{n}\sum_{i=1}^{n} f_i(x)$.

For the rest of the text, note the distinction between the general indexing term i and the randomly selected index per SGD round, i_t. The differences are clear from the context.

2.2.1 Compressed Iterates

Following the problem formulation in [46] on compressed iterates and working on IST in a sequentially centralized fashion, IST performs the following motions:

- Given current model x_t at iteration t, IST generates a mask $\mathcal{M} : \mathbb{R}^p \to \mathbb{R}^p$ such that:

$$(\mathcal{M}(x_t))_i = \begin{cases} \frac{x_{t,i}}{\xi}, & \text{with probability } \xi, \\ 0, & \text{with probability } 1-\xi. \end{cases}$$

  This mask deviates from the proposed model in the sense that the input and output neurons in the neural network are always selected.

- Given mask $\mathcal{M}(\cdot)$, IST generates the subnetwork as in:

$$y_t \equiv \mathcal{M}(x_t) \in \mathbb{R}^p,$$

  where y_t has zeros at the positions where neurons are deactivated for this subnetwork at iteration t and non-zeros for the active weights. I.e., y_t constitutes a *compressed* version of the full model x_t.

- IST performs gradient descent on the compressed y_t as in:

$$x_{t+1} = y_t - \eta \nabla f_{i_t}(y_t),$$

  for η being the learning rate, and i_t being selected randomly from $[n]$.

The above setting resembles that of *gradient descent with compressed iterates* (GDCI) in [46]. The analysis differentiates in that this proof considers a different function class.

Compression operators $\mathcal{M}(\cdot)$. Let me describe and prove some properties of the compression operator $\mathcal{M}(\cdot)$.

Property 1 (Unbiasedness of $\mathcal{M}(\cdot)$) The operator $\mathcal{M} : \mathbb{R}^p \to \mathbb{R}^p$ as defined above is unbiased; i.e.,

$$\mathbb{E}_{\mathcal{M}}\left[\mathcal{M}(x) \mid x\right] = x, \quad \forall x \in \mathbb{R}^p$$

Proof 1 To see this, compute:

$$\mathbb{E}_{\mathcal{M}}\left[\mathcal{M}(x) \mid x\right] = \begin{bmatrix} \mathbb{E}_{\mathcal{M}}\left[(\mathcal{M}(x))_1 \mid x\right] \\ \mathbb{E}_{\mathcal{M}}\left[(\mathcal{M}(x))_2 \mid x\right] \\ \vdots \\ \mathbb{E}_{\mathcal{M}}\left[(\mathcal{M}(x))_p \mid x\right] \end{bmatrix} = \begin{bmatrix} \xi \cdot \frac{x_1}{\xi} + (1-\xi)\cdot 0 \\ \xi \cdot \frac{x_2}{\xi} + (1-\xi)\cdot 0 \\ \vdots \\ \xi \cdot \frac{x_p}{\xi} + (1-\xi)\cdot 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} = x$$

Property 2 (Bounded variance of $\mathcal{M}(\cdot)$) The operator $\mathcal{M} : \mathbb{R}^p \to \mathbb{R}^p$ has bounded variance as in:

$$\mathbb{E}_{\mathcal{M}}\left[\|\mathcal{M}(x) - x\|_2^2 \mid x\right] = \tfrac{1-\xi}{\xi}\|x\|_2^2, \quad \forall x \in \mathbb{R}^p$$

Proof 2 First expand the squared term:

$$\begin{aligned} \mathbb{E}_{\mathcal{M}}\left[\|\mathcal{M}(x) - x\|_2^2 \mid x\right] &= \mathbb{E}_{\mathcal{M}}\left[\|\mathcal{M}(x)\|_2^2 + \|x\|_2^2 - 2\langle \mathcal{M}(x), x\rangle \mid x\right] \\ &= \mathbb{E}_{\mathcal{M}}\left[\|\mathcal{M}(x)\|_2^2 \mid x\right] + \|x\|_2^2 - 2\langle \mathbb{E}_{\mathcal{M}}\left[\mathcal{M}(x) \mid x\right], x\rangle \\ &= \mathbb{E}_{\mathcal{M}}\left[\|\mathcal{M}(x)\|_2^2 \mid x\right] - \|x\|_2^2 \end{aligned}$$

Focusing on the first term on the right hand side:

$$\begin{aligned} \mathbb{E}_{\mathcal{M}}\left[\|\mathcal{M}(x)\|_2^2 \mid x\right] &= \mathbb{E}_{\mathcal{M}}\left[\sum_{i=1}^{p} (\mathcal{M}(x))_i^2 \mid x\right] \\ &= \sum_{i=1}^{p} \mathbb{E}_{\mathcal{M}}\left[(\mathcal{M}(x))_i^2 \mid x\right] = \sum_{i=1}^{p} \left(\xi \cdot \tfrac{x_i^2}{\xi^2} + (1-\xi)\cdot 0\right) \\ &= \tfrac{1}{\xi}\sum_{i=1}^{p} x_i^2 = \tfrac{1}{\xi}\|x\|_2^2 \end{aligned}$$

Combining the above to get:

$$\mathbb{E}_{\mathcal{M}}\left[\|\mathcal{M}(x) - x\|_2^2 \mid x\right] = \tfrac{1}{\xi}\|x\|_2^2 - \|x\|_2^2 = \tfrac{1-\xi}{\xi} \cdot \|x\|_2^2$$

Some assumptions for $\mathcal{M}(\cdot)$ with regard to the gradient of f are as follows.

Assumption 5 (Additive gradient error assumption with bounded energy) Let x_t be the current model and let $y_t = \mathcal{M}(x_t)$ be the compressed model. Consider the stochastic gradient term $\nabla f_{i_t}(y_t)$; I assume that, on expectation, the following additive noise assumption holds:

$$\mathbb{E}_{\mathcal{M},i_t}\left[\nabla f_{i_t}(y_t) \mid x_t\right] = \nabla f(x_t) + \varepsilon_t, \quad \text{for } \varepsilon_t \in \mathbb{R}^p \text{ such that } \|\varepsilon_t\|_2 \leq B \text{ for } B > 0.$$

A different assumption that can be made for stochastic gradient $\nabla f_{i_t}(y_t)$ is the *norm condition*, used in derivative free optimization [47, 48]:

Assumption 6 (Norm condition) There is a constant $\theta \in [0, 1)$ such that:

$$\left\|\mathbb{E}_{\mathcal{M},i_t}\left[\nabla f_{i_t}(y_t) \mid x_t\right] - \nabla f(x_t)\right\|_2 = \|\varepsilon_t\|_2 \leq \theta \|\nabla f(x_t)\|_2,$$

where x_t is the current model and $y_t = \mathcal{M}(x_t)$ is the compressed model.

2.2.2 Proof of Sequential IST

This subsection will provide the backbone of the proof; later on I will make different assumptions that will lead to different final results. For this first part, I will only use the basic properties of the compression operator $\mathcal{M}(\cdot)$ and the L-smoothness of f.

Start with the following Lemma.

Lemma 1 Let $y_t = \mathcal{M}(x_t)$. Then, for $x^\star$ the optimal point for f, it holds:

$$\mathbb{E}_{\mathcal{M}}\left[\|y_t - x_t\|_2^2 \mid x_t\right] \leq \tfrac{2(1-\xi)}{\xi}\|x_t - x^\star\|_2^2 + \tfrac{2(1-\xi)}{\xi}\|x^\star\|_2^2.$$

Proof 3 By Property 2:

$$\mathbb{E}_{\mathcal{M}}\left[\|y_t - x_t\|_2^2 \mid x_t\right] = \tfrac{1-\xi}{\xi}\|x_t\|_2^2.$$

Combining this with the property that:

$$\|x_t\|_2^2 = \|x_t - x^\star + x^\star\|_2^2 \leq 2\|x_t - x^\star\|_2^2 + 2\|x^\star\|_2^2,$$

The reported result is obtained.

Moreover, I also state the following Lemma, which is used in most proofs below.

Lemma 2 Let $y_t = \mathcal{M}(x_t)$. Then,

$$\begin{aligned}\mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - x_t - \eta\nabla f_{i_t}(y_t)\|_2^2 \mid x_t\right] \leq & \frac{4(1+\eta^2 L_{\max}^2)(1-\xi)}{\xi}\left(\|x_t - x^\star\|_2^2 + \|x^\star\|_2^2\right) \\ & + 2\eta^2\mathbb{E}_{i_t}\left[\|\nabla f_{i_t}(x_t)\|_2^2 \mid x_t\right].\end{aligned}$$

Proof 4 For any $z \in \mathbb{R}^p$:

$$\begin{aligned}&\mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - x_t - \eta\nabla f_{i_t}(y_t)\|_2^2 \mid x_t\right] \\ =&\mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - x_t - \eta\nabla f_{i_t}(z) + \eta\nabla f_{i_t}(z) - \eta\nabla f_{i_t}(y_t)\|_2^2 \mid x_t\right] \\ \leq& 2\cdot\mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - x_t - \eta\nabla f_{i_t}(z)\|_2^2 \mid x_t\right] + 2\eta^2\cdot\mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f_{i_t}(z) - \nabla f_{i_t}(y_t)\|_2^2 \mid x_t\right] \\ =&2\cdot\mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - x_t\|_2^2 + \eta^2\|\nabla f_{i_t}(z)\|_2^2 - 2\eta\cdot\langle\nabla f_{i_t}(z), y_t - x_t\rangle \mid x_t\right] \\ &+ 2\eta^2\cdot\mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f_{i_t}(z) - \nabla f_{i_t}(y_t)\|_2^2 \mid x_t\right]\end{aligned}$$

Observe that

$$\begin{aligned}\mathbb{E}_{\mathcal{M},i_t}\left[\langle\nabla f_{i_t}(z), y_t - x_t\rangle \mid x_t\right] &= \langle\mathbb{E}_{i_t}\left[\nabla f_{i_t}(z)\right], \mathbb{E}_{\mathcal{M}}\left[y_t \mid x_t\right] - x_t\rangle \\ &= \langle\nabla f(z), x_t - x_t\rangle = 0\end{aligned}$$

due to Property 1. Then:

$$\begin{aligned}\mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - x_t - \eta\nabla f(y_t)\|_2^2 \mid x_t\right] \leq & 2\cdot\mathbb{E}_{\mathcal{M}}\left[\|y_t - x_t\|_2^2 \mid x_t\right] + 2\eta^2\mathbb{E}_{i_t}\left[\|\nabla f_{i_t}(z)\|_2^2 \mid x_t\right] \\ & + 2\eta^2\cdot\mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f_{i_t}(z) - \nabla f_{i_t}(y_t)\|_2^2 \mid x_t\right]\end{aligned}$$

By L_i-smoothness:

$$\begin{aligned}
&\mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(z)-\nabla f(y_t)\|_2^2 \mid x_t\right] \\
\leq\ &\mathbb{E}_{\mathcal{M},i_t}\left[L_{i_t}^2\cdot\|z-y_t\|_2^2 \mid x_t\right] \\
\leq\ &L_{\max}^2\cdot\mathbb{E}_{\mathcal{M},i_t}\left[\|z-y_t\|_2^2 \mid x_t\right] \\
\leq\ &L_{\max}^2\cdot\mathbb{E}_{\mathcal{M}}\left[\|z-x_t+x_t-y_t\|_2^2 \mid x_t\right] \\
=\ &L_{\max}^2\cdot\mathbb{E}_{\mathcal{M}}\left[\|z-x_t\|_2^2+\|x_t-y_t\|_2^2+2\langle z-x_t,x_t-y_t\rangle \mid x_t\right] \\
=\ &L_{\max}^2\cdot\mathbb{E}_{\mathcal{M}}\left[\|z-x_t\|_2^2 \mid x_t\right]+L_{\max}^2\cdot\mathbb{E}_{\mathcal{M}}\left[\|x_t-y_t\|_2^2 \mid x_t\right]
\end{aligned}$$

In the above, Property 1 is used to get $\mathbb{E}_{\mathcal{M}}[y_t \mid x_t]=x_t$. Using this result in the expression above to get:

$$\begin{aligned}
&\mathbb{E}_{\mathcal{M},i_t}\left[\|y_t-x_t-\eta\nabla f_{i_t}(y_t)\|_2^2 \mid x_t\right] \\
\leq\ &2\cdot\mathbb{E}_{\mathcal{M}}\left[\|y_t-x_t\|_2^2 \mid x_t\right]+2\eta^2\mathbb{E}_{i_t}\left[\|\nabla f_{i_t}(z)\|_2^2 \mid x_t\right] \\
&+2\eta^2\cdot L_{\max}^2\cdot\left(\mathbb{E}_{\mathcal{M}}\left[\|z-x_t\|_2^2 \mid x_t\right]+\mathbb{E}_{\mathcal{M}}\left[\|x_t-y_t\|_2^2 \mid x_t\right]\right) \\
=\ &2(1+\eta^2L_{\max}^2)\cdot\mathbb{E}_{\mathcal{M}}\left[\|y_t-x_t\|_2^2 \mid x_t\right]+2\eta^2\mathbb{E}_{i_t}\left[\|\nabla f_{i_t}(z)\|_2^2 \mid x_t\right] \\
&+2\eta^2\cdot L_{\max}^2\cdot\mathbb{E}_{\mathcal{M}}\left[\|z-x_t\|_2^2 \mid x_t\right]
\end{aligned}$$

The above expression holds for any $z\in\mathbb{R}^p$, and thus it will hold for $z=x_t$. In this case, the above inequality becomes:

$$\begin{aligned}
\mathbb{E}_{\mathcal{M},i_t}\left[\|y_t-x_t-\eta\nabla f_{i_t}(y_t)\|_2^2 \mid x_t\right] \leq\ &2(1+\eta^2L_{\max}^2)\cdot\mathbb{E}_{\mathcal{M}}\left[\|y_t-x_t\|_2^2 \mid x_t\right] \\
&+2\eta^2\mathbb{E}_{i_t}\left[\|\nabla f_{i_t}(x_t)\|_2^2 \mid x_t\right]
\end{aligned}$$

Finally, using Lemma 1, the desired inequality is obtained.

The following lemma leads to a general recursion over function values, that will lead to specific convergence rate guarantees later on, based on additional assumptions.

Lemma 3 Let f be L-smooth, and consider the recursion over compressed iterates:

$$x_{t+1} = y_t - \eta \nabla f_{i_t}(y_t), \quad \text{where } y_t = \mathcal{M}(x_t).$$

Then the following recursion holds:

$$\begin{aligned}\mathbb{E}_{\mathcal{M},i_t}\left[f(x_{t+1}) \mid x_t\right] \leq & f(x_t) - \eta \cdot \langle \nabla f(x_t), \mathbb{E}_{\mathcal{M},i_t}\left[\nabla f_{i_t}(y_t) \mid x_t\right]\rangle \\ & + \tfrac{L_i}{2} \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - \eta \nabla f_{i_t}(y_t) - x_t\|_2^2 \mid x_t\right],\end{aligned}$$

where the expectation is over the random selection on the compression operator $\mathcal{M}(\cdot)$ and the stochasticity of the gradient.

Proof 5 Starting from the L_i-smoothness condition, the expectation with respect to $\mathcal{M}, i_t$ at time t is:

$$\begin{aligned}
&\mathbb{E}_{\mathcal{M},i_t}\left[f(x_{t+1}) \mid x_t\right] \\
\leq & \mathbb{E}_{\mathcal{M},i_t}\left[f(x_t) + \langle \nabla f_{i_t}(x_t), x_{t+1} - x_t\rangle + \tfrac{L_i}{2}\|x_{t+1} - x_t\|_2^2 \mid x_t\right] \\
= & \mathbb{E}_{\mathcal{M},i_t}\left[f(x_t) \mid x_t\right] + \mathbb{E}_{\mathcal{M},i_t}\left[\langle \nabla f_{i_t}(x_t), x_{t+1} - x_t\rangle \mid x_t\right] + \tfrac{L_i}{2} \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\|x_{t+1} - x_t\|_2^2 \mid x_t\right] \\
= & f(x_t) + \mathbb{E}_{\mathcal{M},i_t}\left[\langle \nabla f_{i_t}(x_t), y_t - \eta \nabla f_{i_t}(y_t) - x_t\rangle \mid x_t\right] + \tfrac{L_i}{2} \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - \eta \nabla f_{i_t}(y_t) - x_t\|_2^2 \mid x_t\right] \\
= & f(x_t) + \mathbb{E}_{\mathcal{M},i_t}\left[\langle \nabla f_{i_t}(x_t), y_t - x_t\rangle \mid x_t\right] - \eta \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\langle \nabla f_{i_t}(x_t), \nabla f_{i_t}(y_t)\rangle \mid x_t\right] \\
& + \tfrac{L_i}{2} \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - \eta \nabla f(y_t) - x_t\|_2^2 \mid x_t\right] \\
= & f(x_t) + \langle \mathbb{E}_{i_t}[\nabla f_{i_t}(x_t) \mid x_t], \mathbb{E}_{\mathcal{M}}\left[y_t \mid x_t\right] - x_t\rangle - \eta \cdot \langle \mathbb{E}_{i_t}[\nabla f_{i_t}(x_t) \mid x_t], \mathbb{E}_{\mathcal{M},i_t}\left[\nabla f_{i_t}(y_t) \mid x_t\right]\rangle \\
& + \tfrac{L_i}{2} \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - \eta \nabla f_{i_t}(y_t) - x_t\|_2^2 \mid x_t\right] \\
\overset{\text{Property 1}}{=} & f(x_t) - \eta \cdot \langle \nabla f(x_t), \mathbb{E}_{\mathcal{M},i_t}\left[\nabla f_{i_t}(y_t) \mid x_t\right]\rangle + \tfrac{L_i}{2} \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - \eta \nabla f_{i_t}(y_t) - x_t\|_2^2 \mid x_t\right]
\end{aligned}$$

In the last step, the unbiasedness of the stochastic gradient is applied.

The proof will branch out for different assumptions. I begin with the following "cocktail" of assumptions.

Theorem 1 *Let* $f := \frac{1}{n}\sum_{i=1}^{n} f_i(x)$ *has* L_i*-smooth components* f_i *for* $L_{\max} := \max_i L_i$*, and consider the recursion over compressed iterates:*

$$x_{t+1} = y_t - \eta \nabla f_{i_t}(y_t), \quad \textit{where } y_t = \mathcal{M}(x_t).$$

In additional to Lemma 3, I further assume that f *is* Q*-Lipschitz, and satisfies the Error Bound with parameter* $\mu > 0$*. Further, the recursion should satisfy the gradient boundedness Assumption 4 for* $M > 0$ *and* $0 < M_f < 1$*. Finally, assume that the operator mask, along with* f*, satisfy the additive gradient error assumption with bounded energy such that:*

$$\mathbb{E}_{\mathcal{M},i_t}\left[\nabla f_{i_t}(y_t) \mid x_t\right] = \nabla f(x_t) + \varepsilon_t, \quad \textit{for } \varepsilon_t \in \mathbb{R}^p \textit{ such that } \|\varepsilon_t\|_2 \leq B \textit{ for } B > 0.$$

Then, after running the above recursion for T *iterations for step size* $\eta = \frac{1}{2L_{\max}}$*, one can obtain:*

$$\min_{t\in\{0,\dots,T\}} \mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(x_t)\|_2^2\right] \leq \frac{f(x_0)-f(x^\star)}{\alpha(T+1)} + \frac{1}{\alpha}\cdot\left(\frac{BQ}{2L_{\max}} + \frac{5L_{\max}\omega}{2}\cdot\|x^\star\|_2^2 + \frac{M}{4L_{\max}}\right)$$

where the expectation is over the random selection on the compression operator $\mathcal{M}(\cdot)$ *and the stochastic selection* i_t*,* $\alpha = \frac{1}{2L_{\max}}\left(1 - \frac{M_f}{2}\right) - \frac{5\omega L_{\max}}{2\mu^2}$*, and* $\omega = \frac{1-\xi}{\xi} < \frac{\mu^2}{10L_{\max}^2}$*.*

Proof 6 *By Lemma 3, the following holds:*

$$\begin{aligned}\mathbb{E}_{\mathcal{M},i_t}\left[f(x_{t+1}) \mid x_t\right] \leq & f(x_t) - \eta\cdot\langle\nabla f(x_t), \mathbb{E}_{\mathcal{M},i_t}\left[\nabla f_{i_t}(y_t) \mid x_t\right]\rangle \\ & + \frac{L_i}{2}\cdot\mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - \eta\nabla f_{i_t}(y_t) - x_t\|_2^2 \mid x_t\right],\end{aligned}$$

Using the additive gradient noise assumption $\mathbb{E}_{\mathcal{M},i_t}\left[\nabla f_{i_t}(y_t) \mid x_t\right] = \nabla f(x_t) + \varepsilon_t$ *to have:*

$$
\begin{aligned}
&\mathbb{E}_{\mathcal{M},i_t}\left[f(x_{t+1}) \mid x_t\right] \\
\leq & f(x_t) - \eta \cdot \langle \nabla f(x_t), \nabla f(x_t) + \varepsilon_t \rangle + \tfrac{L_i}{2} \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - \eta \nabla f_{i_t}(y_t) - x_t\|_2^2 \mid x_t\right] \\
= & f(x_t) - \eta \cdot \|\nabla f(x_t)\|_2^2 - \eta \langle \nabla f(x_t), \varepsilon_t \rangle + \tfrac{L_i}{2} \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\|y_t - \eta \nabla f_{i_t}(y_t) - x_t\|_2^2 \mid x_t\right] \\
\overset{\textit{Lemma 2}}{\leq} & f(x_t) - \eta \cdot \|\nabla f(x_t)\|_2^2 - \eta \langle \nabla f(x_t), \varepsilon_t \rangle \\
& + \tfrac{L_i}{2} \cdot \left(\tfrac{4(1+\eta^2 L_{\max}^2)(1-\xi)}{\xi} \left(\|x_t - x^\star\|_2^2 + \|x^\star\|_2^2 \right) + 2\eta^2 \mathbb{E}_{i_t}\left[\|\nabla f_{i_t}(x_t)\|_2^2 \mid x_t \right] \right) \\
\overset{L_i \leq L_{\max}}{\leq} & f(x_t) - \eta \cdot \|\nabla f(x_t)\|_2^2 - \eta \langle \nabla f(x_t), \varepsilon_t \rangle \\
& + \tfrac{L_{\max}}{2} \cdot \left(\tfrac{4(1+\eta^2 L_{\max}^2)(1-\xi)}{\xi} \left(\|x_t - x^\star\|_2^2 + \|x^\star\|_2^2 \right) + 2\eta^2 \mathbb{E}_{i_t}\left[\|\nabla f_{i_t}(x_t)\|_2^2 \mid x_t \right] \right) \\
\overset{\textit{Assumption 4}}{\leq} & f(x_t) - \eta \cdot \|\nabla f(x_t)\|_2^2 - \eta \langle \nabla f(x_t), \varepsilon_t \rangle \\
& + \tfrac{L_{\max}}{2} \cdot \left(\tfrac{4(1+\eta^2 L_{\max}^2)(1-\xi)}{\xi} \left(\|x_t - x^\star\|_2^2 + \|x^\star\|_2^2 \right) + 2\eta^2 \left(M + M_f \|\nabla f(x_t)\|_2^2 \right) \right) \\
= & f(x_t) - \eta(1 - \eta L_{\max} M_f) \cdot \|\nabla f(x_t)\|_2^2 - \eta \langle \nabla f(x_t), \varepsilon_t \rangle \\
& + \tfrac{2L_{\max}(1+\eta^2 L_{\max}^2)(1-\xi)}{\xi} \left(\|x_t - x^\star\|_2^2 + \|x^\star\|_2^2 \right) + L_{\max} \eta^2 M \\
\overset{\textit{Error Bound}}{\leq} & f(x_t) - \eta(1 - \eta L_{\max} M_f) \cdot \|\nabla f(x_t)\|_2^2 - \eta \langle \nabla f(x_t), \varepsilon_t \rangle \\
& + \tfrac{2L_{\max}(1+\eta^2 L_{\max}^2)(1-\xi)}{\xi} \cdot \tfrac{1}{\mu^2} \cdot \|\nabla f(x_t)\|_2^2 + \tfrac{2L_{\max}(1+\eta^2 L_{\max}^2)(1-\xi)}{\xi} \|x^\star\|_2^2 + L_{\max} \eta^2 M
\end{aligned}
$$

To simplify notation, define $\omega = \frac{1-\xi}{\xi}$. *Rearranging the terms in the inequality above to obtain:*

$$
\begin{aligned}
\mathbb{E}_{\mathcal{M},i_t}\left[f(x_{t+1}) \mid x_t\right] \leq f(x_t) - & \left(\eta(1 - \eta L_{\max} M_f) - 2L_{\max}(1 + \eta^2 L_{\max}^2) \cdot \tfrac{\omega}{\mu^2} \right) \cdot \|\nabla f(x_t)\|_2^2 \\
& - \eta \langle \nabla f(x_t), \varepsilon_t \rangle + 2\omega L_{\max}(1 + \eta^2 L_{\max}^2) \|x^\star\|_2^2 + L_{\max} \eta^2 M
\end{aligned}
$$

Define $g(\eta) = \eta(1 - \eta L_{\max} M_f) - 2L_{\max}(1 + \eta^2 L_{\max}^2) \cdot \frac{\omega}{\mu^2}$ *which is a quadratic function. Set the step size as in* $\eta = \frac{1}{2L_{\max}}$, *which is a reasonable assumption based on convex optimization criteria. Then,*

$$
g\left(\tfrac{1}{2L_{\max}}\right) = \tfrac{1}{2L_{\max}} \left(1 - \tfrac{M_f}{2}\right) - \tfrac{5L_{\max}\omega}{2\mu^2} \equiv \alpha.
$$

For the proof, IST requires $\alpha > 0$; according to the assumptions, $1 - \frac{M_f}{2} > \frac{1}{2} \Rightarrow M_f < 1$. This further leads to the requirement that $\frac{1-\xi}{\xi} < \frac{\mu^2}{10L_{\max}^2}$. The above result into:

$$
\begin{aligned}
&\mathbb{E}_{\mathcal{M},i_t}\left[f(x_{t+1}) \mid x_t\right] \\
\leq & f(x_t) - \alpha \cdot \|\nabla f(x_t)\|_2^2 - \eta \langle \nabla f(x_t), \varepsilon_t \rangle + 2\omega L_{\max}(1 + \eta^2 L_{\max}^2)\|x^\star\|_2^2 + L_{\max}\eta^2 M \\
\overset{\textit{Cauchy-Schwarz}}{\leq} & f(x_t) - \alpha \cdot \|\nabla f(x_t)\|_2^2 + \eta \cdot \|\nabla f(x_t)\|_2 \cdot \|\varepsilon_t\|_2 \\
& + 2\omega L_{\max}(1 + \eta^2 L_{\max}^2)\|x^\star\|_2^2 + L_{\max}\eta^2 M \\
\overset{\eta = \frac{1}{2L_{\max}}}{=} & f(x_t) - \alpha \cdot \|\nabla f(x_t)\|_2^2 + \tfrac{1}{2L_{\max}} \cdot \|\nabla f(x_t)\|_2 \cdot \|\varepsilon_t\|_2 \\
& + \tfrac{5L_{\max}\omega}{2} \cdot \|x^\star\|_2^2 + \tfrac{M}{4L_{\max}} \\
\overset{\textit{Q-Lipschitz}}{\leq} & f(x_t) - \alpha \cdot \|\nabla f(x_t)\|_2^2 + \tfrac{Q}{2L_{\max}} \cdot \|\varepsilon_t\|_2 + \tfrac{5L_{\max}\omega}{2} \cdot \|x^\star\|_2^2 + \tfrac{M}{4L_{\max}} \\
\overset{\|\varepsilon_t\|_2 \leq B}{\leq} & f(x_t) - \alpha \cdot \|\nabla f(x_t)\|_2^2 + \tfrac{BQ}{2L_{\max}} + \tfrac{5L_{\max}\omega}{2} \cdot \|x^\star\|_2^2 + \tfrac{M}{4L_{\max}}
\end{aligned}
$$

Using the law of total expectation $\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]]$ to have:

$$
\mathbb{E}\left[f(x_{t+1})\right] = \mathbb{E}\left[\mathbb{E}\left[f(x_{t+1}) \mid x_t\right]\right]
$$

and thus:

$$
\mathbb{E}_{\mathcal{M},i_t}\left[f(x_{t+1})\right] \leq \mathbb{E}_{\mathcal{M},i_t}\left[f(x_t)\right] - \alpha \cdot \mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(x_t)\|_2^2\right] + \tfrac{BQ}{2L} + \tfrac{5L_{\max}\omega}{2} \cdot \|x^\star\|_2^2 + \tfrac{M}{4L_{\max}}
$$

Using the fact that $f(x^\star) \leq \mathbb{E}_{\mathcal{M},i_t}\left[f(x_{T+1})\right]$, and telescoping over T iterations to obtain:

$$
\begin{aligned}
f(x^\star) \leq \mathbb{E}_{\mathcal{M},i_t}\left[f(x_{T+1})\right] \leq & f(x_0) - \alpha \sum_{t=0}^{T} \mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(x_t)\|_2^2\right] \\
& + (T+1) \cdot \left(\tfrac{BQ}{2L} + \tfrac{5L_{\max}\omega}{2} \cdot \|x^\star\|_2^2 + \tfrac{M}{4L_{\max}}\right)
\end{aligned}
$$

which further leads to:

$$
\sum_{t=0}^{T} \mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(x_t)\|_2^2\right] \leq \tfrac{f(x_0) - f(x^\star)}{\alpha} + \tfrac{T+1}{\alpha} \cdot \left(\tfrac{BQ}{2L} + \tfrac{5L_{\max}\omega}{2} \cdot \|x^\star\|_2^2 + \tfrac{M}{4L_{\max}}\right)
$$

Observe that: $(T+1)\cdot \min_{t\in\{0,\dots,T\}} \mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(x_t)\|_2^2\right] \le \sum_{t=0}^{T} \mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(x_t)\|_2^2\right]$, *which leads to the final result:*

$$\min_{t\in\{0,\dots,T\}} \mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(x_t)\|_2^2\right] \le \frac{f(x_0)-f(x^\star)}{\alpha(T+1)} + \frac{1}{\alpha}\cdot\left(\frac{BQ}{2L} + \frac{5L_{\max}\omega}{2}\cdot\|x^\star\|_2^2 + \frac{M}{4L_{\max}}\right)$$

In the following corollary, I exchange the bounded assumption $\|\varepsilon_t\|_2 \le B$ and Q-Lipschitzness, with the norm condition in Assumption 6.

Corollary 1 Let f be L-smooth, and consider the recursion over compressed iterates:

$$x_{t+1} = y_t - \eta\nabla f_{i_t}(y_t), \quad \textit{where } y_t = \mathcal{M}(x_t).$$

In additional to Lemma 3, I further assume that the operator mask, along with f, satisfy the norm condition Assumption 6 with parameter $\theta \in [0,1)$. Then, after running the above recursion for T iterations for step size $\eta = \frac{1}{2L_{\max}}$ to obtain:

$$\min_{t\in\{0,\dots,T\}} \mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(x_t)\|_2^2\right] \le \frac{f(x_0)-f(x^\star)}{\alpha(T+1)} + \frac{1}{\alpha}\cdot\left(\frac{5L_{\max}\omega}{2}\cdot\|x^\star\|_2^2 + \frac{M}{4L_{\max}}\right)$$

where the expectation is over the random selection on the compression operator $\mathcal{M}(\cdot)$, $\alpha = \frac{1}{2L_{\max}}\left(\frac{1}{2}-\theta-\frac{M_f}{2}\right) - \frac{5L_{\max}}{2}\cdot\frac{\omega}{\mu^2}$, and $\omega = \frac{1-\xi}{\xi} < \frac{\mu^2}{5L_{\max}^2\left(\frac{1}{2}-\theta-\frac{M_f}{2}\right)}$.

Proof 7 By Theorem 1:

$$
\begin{aligned}
&\mathbb{E}_{\mathcal{M},i_t}\left[f(x_{t+1}) \mid x_t\right] \\
\leq & f(x_t) - \left(\eta(1-\eta L_{\max} M_f) - 2L_{\max}(1+\eta^2 L_{\max}^2)\cdot\tfrac{\omega}{\mu^2}\right)\cdot\|\nabla f(x_t)\|_2^2 \\
& - \eta\langle\nabla f(x_t), \mathbb{E}_{\mathcal{M},i_t}[\varepsilon_t \mid x_t]\rangle + 2\omega L_{\max}(1+\eta^2 L_{\max}^2)\|x^\star\|_2^2 + L_{\max}\eta^2 M \\
\overset{\textit{Cauchy-Schwarz}}{\leq} & f(x_t) - \left(\eta(1-\eta L_{\max} M_f) - 2L_{\max}(1+\eta^2 L_{\max}^2)\cdot\tfrac{\omega}{\mu^2}\right)\cdot\|\nabla f(x_t)\|_2^2 \\
& + \eta\|\nabla f(x_t)\|_2\cdot\|\mathbb{E}_{\mathcal{M},i_t}[\varepsilon_t \mid x_t]\|_2 + 2\omega L_{\max}(1+\eta^2 L_{\max}^2)\|x^\star\|_2^2 + L_{\max}\eta^2 M \\
= & f(x_t) - \left(\eta(1-\eta L_{\max} M_f) - 2L_{\max}(1+\eta^2 L_{\max}^2)\cdot\tfrac{\omega}{\mu^2}\right)\cdot\|\nabla f(x_t)\|_2^2 \\
& + \eta\|\nabla f(x_t)\|_2\cdot\|\mathbb{E}_{\mathcal{M},i_t}\left[\nabla f_{i_t}(y_t) \mid x_t\right] - \nabla f(x_t)\|_2 \\
& + 2\omega L_{\max}(1+\eta^2 L_{\max}^2)\|x^\star\|_2^2 + L_{\max}\eta^2 M \\
\overset{\textit{Norm condition}}{\leq} & f(x_t) - \left(\eta(1-\eta L_{\max} M_f) - 2L_{\max}(1+\eta^2 L_{\max}^2)\cdot\tfrac{\omega}{\mu^2}\right)\cdot\|\nabla f(x_t)\|_2^2 \\
& + \eta\theta\|\nabla f(x_t)\|_2^2 + 2\omega L_{\max}(1+\eta^2 L_{\max}^2)\|x^\star\|_2^2 + L_{\max}\eta^2 M \\
= & f(x_t) - \left(\eta(1-\theta-\eta L_{\max} M_f) - 2L_{\max}(1+\eta^2 L_{\max}^2)\cdot\tfrac{\omega}{\mu^2}\right)\cdot\|\nabla f(x_t)\|_2^2 \\
& + 2\omega L_{\max}(1+\eta^2 L_{\max}^2)\|x^\star\|_2^2 + L_{\max}\eta^2 M
\end{aligned}
$$

For coherence, assume that $\eta = \frac{1}{2L_{\max}}$. *Following the same procedure I obtain, define* $g\left(\frac{1}{2L_{\max}}\right) \equiv \alpha = \frac{1}{2L_{\max}}\left(\frac{1}{2} - \theta - \frac{M_f}{2}\right) - \frac{5L_{\max}}{2}\cdot\frac{\omega}{\mu^2}$. *Observe that* $\alpha > 0$ *when* $\omega < \frac{\mu^2}{5L_{\max}^2\left(\frac{1}{2}-\theta-\frac{M_f}{2}\right)}$. *Then, with similar reasoning with Theorem 1, telescope the inequality to obtain:*

$$
\min_{t\in\{0,\ldots,T\}} \mathbb{E}_{\mathcal{M},i_t}\left[\|\nabla f(x_t)\|_2^2\right] \leq \frac{f(x_0)-f(x^\star)}{\alpha(T+1)} + \frac{1}{\alpha}\cdot\left(\frac{5L_{\max}\omega}{2}\cdot\|x^\star\|_2^2 + \frac{M}{4L_{\max}}\right)
$$

2.3 Empirical Evaluation

In this section, a set of experiments are designed to show the potential benefit of the IST approach when the goal is distributed training in a public cloud, with a

relatively slow network but where fast CPUs and GPUs are available. To do this, IST is compared with data parallel learning and local SGD [27], on a variety of learning tasks and neural architectures.

2.3.1 Learning Tasks and Environment

The following tasks are applied in empirical evaluation.

- *Google Speech Commands* [49]: a 2-layer network of 4096 neurons and a 3-layer network of 8192 neurons are learned to recognize 35 labeled keywords from audio wave-forms (in contrast to the 12 keywords in prior reports [49]). Each waveform is represented as a 4096-dimensional feature vector [50].

- *Image classification on CIFAR10 and full ImageNet* [51, 52]: the ResNet18 model over CIFAR10, and the VGG12 model over full ImageNet are trained (see Section 2.1.6 for a discussion of IST and non-fully connected architectures). **Note that the complete ImageNet dataset is included with all** $21,841$ **categories and report the top-10 accuracy**. Because it is so difficult to train, there are few reported results over the complete ImageNet data set.

- *Amazon-670k* [53]: a 2-layer, fully-connected neural network is trained, which accepts a $135,909$-dimensional input feature, and generates a prediction over $670,091$ output labels.

The Google speech and ResNet18 on CIFAR10 are trained over three AWS CPU clusters, with 2, 4, and 8 CPU instances (`m5.2xlarge`). The VGG12 model on full ImageNet and Amazon-670k extreme classification network are trained over three AWS GPU clusters, with 2, 4, and 8 GPU machines (`p3.2xlarge`). The choice of

AWS was deliberate, as it is a very common learning platform, and *illustrates the challenge faced by many consumers:* distributed ML without a super-fast interconnect.

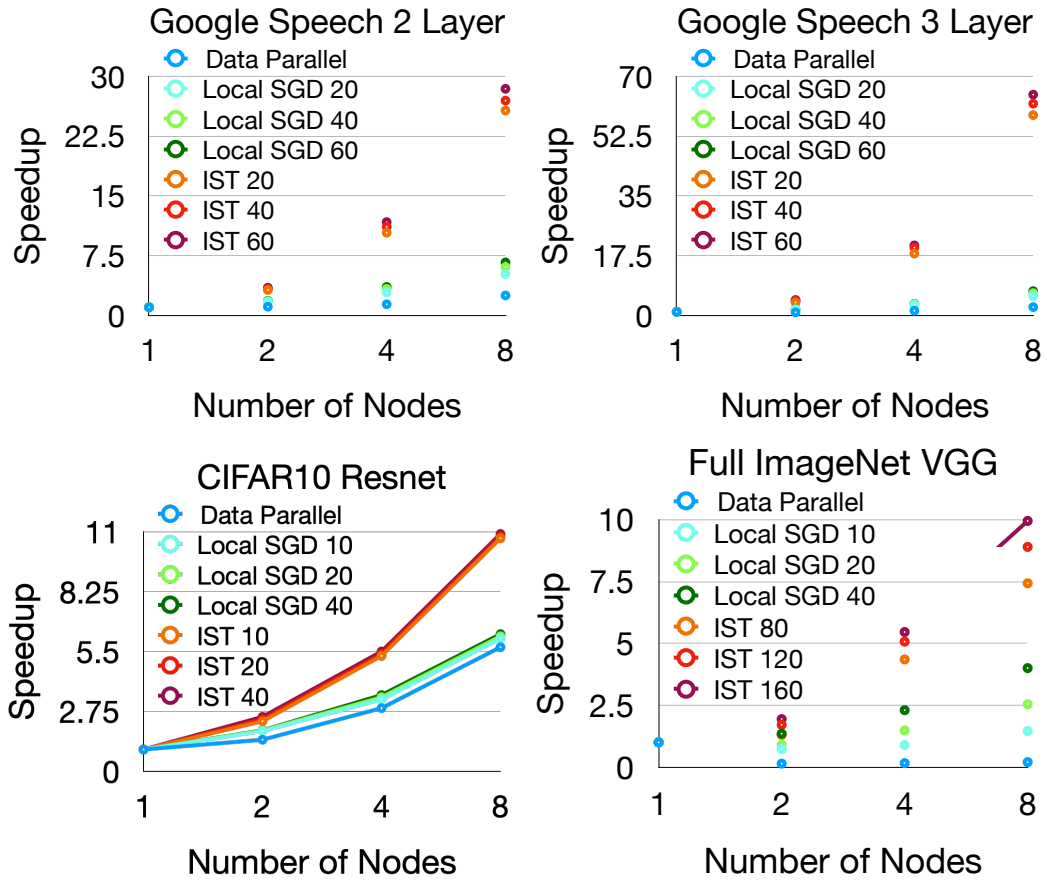

Figure 2.3 : Scaling comparison of data parallel, local SGD and IST.

2.3.2 Distributed Implementation Notes

I implement a distributed parameter server for IST in PyTorch. I compare IST to the PyTorch implementation of data parallel learning. I also adapt the PyTorch data parallel learning to realize local SGD [27], where learning occurs locally for a number of iterations before synchronizing.

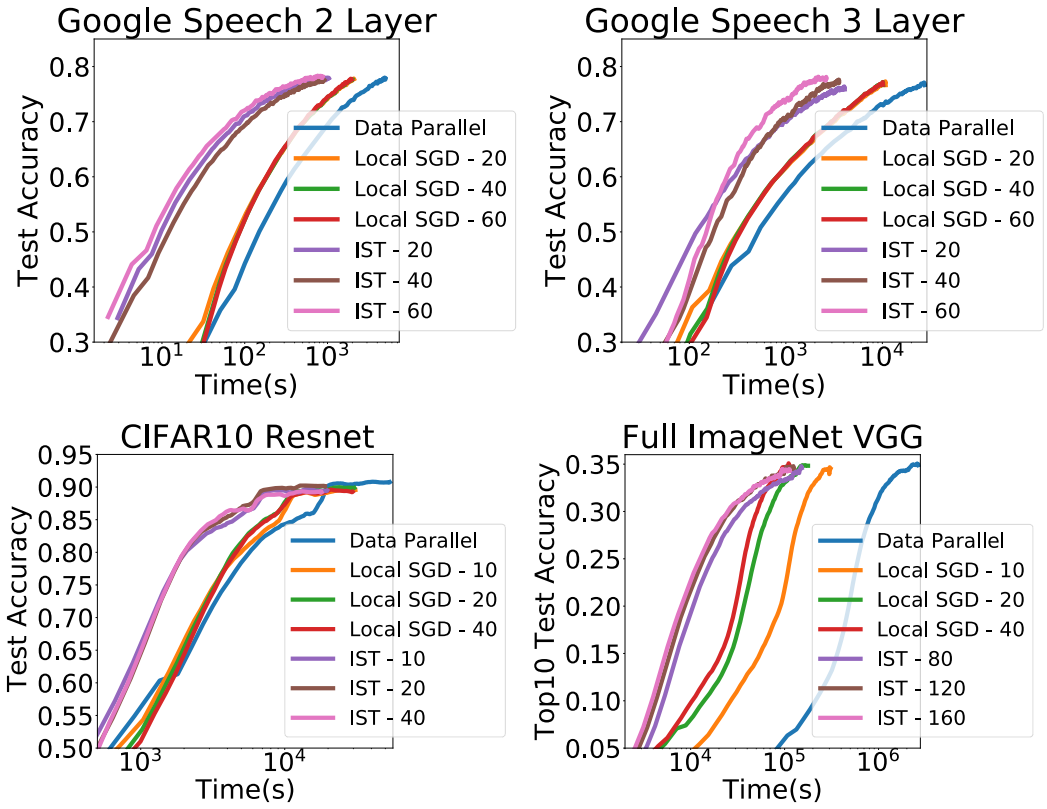

Figure 2.4 : Test accuracy vs. time.

For the CPU experiments, PyTorch's `gloo` backend is used. For the GPU experiments, data parallel learning and local SGD use PyTorch's `nccl` backend, which leverages the most advanced Nvidia collective communication library (the set of high-performance multi-GPU and multi-node collective communication primitives optimized for Nvidia GPUs). `Nccl` implements ring-based all-reduce [26], which is used in well-known distributed learning systems such as Horovod [54].

Unfortunately, IST cannot use the `nccl` backend: it does not support the `scatter` operator required to implement IST, likely because the deep learning community has focused on data parallel learning. As a result, IST must use the `gloo` backend (meant for CPU-based learning). *This is a serious handicap for IST*, though I emphasize that

it is not the result of any intrinsic flaw of the method, it is merely a lack of support for required operations in the high-performance GPU library.

2.3.3 Experimental Results

The experimental results are summarized below:

Scalability. I first investigate the relative scaling of IST compared to the alternatives, with an increasing number of EC2 workers. For various configurations I time how long each of the distributed learning frameworks take to complete one training epoch. Figure 2.3 gives the results.

Convergence speed. While IST can process data quickly, there are questions regarding its statistical efficiency vis-a-vis the other methods, and how this affects convergence. Figure 2.4 plots the hold-out test accuracy for selected benchmarks as a function of time. 2-/3- layer Google speech models are trained using an 8-CPU cluster; ResNet18 on CIFAR10 is trained using 4-CPU cluster; VGG12 on full ImageNet is trained using a 8-GPU cluster. The number after local SGD or IST legend represents the local update iterations.

Table 2.1 shows the training time required for the various methods to reach specified levels of hold-out test accuracy. The speedup is calculated by comparing with the training time for one epoch to 1-worker SGD. The number after local SGD or IST legend represents the local update iterations.

Trained model accuracy. Because IST is inherently a model-parallel training method, it has certain advantages, including the ability to scale to large models. the relationship between the embedding dimensions and the hold-out test performance is studied for the Amazon-670k recommendation task in a 8-GPU cluster. The precisions @1, @3, and @5 are reported in Table 2.2. In Table 2.3 the final accuracy of each

method is reported, trained on a 2-node cluster.

Google Speech 2 Layer									
	Data Parallel			Local SGD			IST		
Accuracy	2 Node	4 Node	8 Node	2 Node	4 Node	8 Node	2 Node	4 Node	8 Node
0.63	118	269	450	68	130	235	35	28	**24**
0.75	759	1708	2417	444	742	1110	231	**167**	192
Google Speech 3 Layer									
	Data Parallel			Local SGD			IST		
Accuracy	2 Node	4 Node	8 Node	2 Node	4 Node	8 Node	2 Node	4 Node	8 Node
0.63	376	1228	1922	182	586	1115	**76**	141	300
0.75	4534	9340	14886	2032	4107	6539	812	**664**	1161
CIFAR10 Resnet18									
	Data Parallel			Local SGD			IST		
Accuracy	2 Node	4 Node	8 Node	2 Node	4 Node	8 Node	2 Node	4 Node	8 Node
0.85	21775	13689	6890	18769	12744	7020	15093	7852	**5241**
0.90	54002	38430	17853	36891	22198	**12157**	33345	16798	13425
Full ImageNet VGG12									
	Data Parallel			Local SGD			IST		
Accuracy	2 Node	4 Node	8 Node	2 Node	4 Node	8 Node	2 Node	4 Node	8 Node
0.20	108040	278542	504805	6900	14698	30441	**3629**	4379	5954
0.26	225911	393279	637188	15053	22055	39439	**6189**	7711	10622

Table 2.1 : The time (in seconds) to reach various levels of accuracy.

2.3.4 Discussion

There are a few takeaways from the experimental results. First, as expected, there are significant advantages to IST in terms of being able to process data quickly. Figure 2.3 shows that IST is able to process far more data in a short amount of time than the other distributed training algorithm. Interestingly, I find that the IST speedups in CPU clusters are more significant than that in GPU clusters. There are two reasons for this. First, for GPU clusters, IST suffers from its use of PyTorch's `gloo` backend,

compared to the `all-reduce` operator provided by `nccl`. Second, since the GPU provides a very high level of computation, there is less benefit to be realized from the reduction in FLOPS per gradient step using IST (as the GPU does not appear to be compute bound).

Figure 2.4 and Table 2.1 generally show that IST is much faster compared to the other frameworks for achieving high levels of accuracy on a hold-out test set. For example, IST exhibits a 4.2× speedup compared to local SGD, and 10.6× speedup compared to classical data parallel for the 2-layer Google speech model to reach 77%. IST exhibits 6.1× speedup compared to local SGD, and a 16.6× speedup comparing to data parallel for the 3-layer model to reach the accuracy of 77%. Note that this was observed even though IST was handicapped by its use of `gloo` for its GPU implementation. Interestingly, for the full ImageNet data set, the communication bottleneck using AWS is so severe that the smaller clusters were always faster; at each cluster size, IST was still the fastest option. For CFAR10, because CPUs were used for training, the network is less of a bottleneck and all methods were able to scale. This negates the IST advantage just a bit. In this case, IST was fastest to 85% accuracy, but was slower to fine-tune to 90% accuracy in 8-CPU cluster.

	Data Parallel			IST		
Dim	P@1	P@3	P@5	P@1	P@3	P@5
512	0.3861	0.3454	0.3164	0.3962	0.3604	0.3313
1024	Fail			0.4089	0.3685	0.3392
1536	Fail			0.4320	0.3907	0.3614
2048	Fail			0.4365	0.3936	0.3637
2560	Fail			**0.4384**	**0.3944**	**0.3658**

Table 2.2 : Precisions @1, @3, @5 on the Amazon 670k benchmark.

Another key advantage of IST is illustrated in Table 2.2; because it is a model-parallel framework and distributes the model to multiple machines, IST is able to scale to virtually unlimited model sizes. In this case, it can compute 2560-dimensional embedding in 8-GPU cluster (and realize the associated, additional accuracy) whereas the data parallel approaches are unable to do this.

	Data Parallel	Local SGD	IST
Speech 2 layer	0.7938	0.7998	**0.8153**
Speech 3 layer	0.7950	0.7992	**0.8327**
CIFAR10 Resnet18	**0.9128**	0.9087	0.9102
Full Imagenet VGG12	0.3688	0.3685	**0.3802**

Table 2.3 : Final accuracy on each benchmark.

It is interesting that *some of the frameworks actually do worse with additional machines, especially with a fast GPU*. This illustrates a significant problem with distributed learning. Unless a super-fast interconnect is used (and such interconnects are not available from typical cloud providers), it can actually be *detrimental* to add additional machines, as the added cost of transferring data can actually result in

slower running times. This is clearly observed in Table 2.1, where the state-of-the-art PyTorch data parallel implementation (and the local SGD variant) does significantly *worse* with more machines. IST shows the best potential to utilize additional machines without actually becoming much slower or slower to reach high accuracy.

Finally, note that various compression schemes can be used to increase the bandwidth of the interconnect (e.g., gradient sparsification [55], quantization [56], sketching [57], and low-rank compression [58]). However, these methods could be used with *any* framework — including IST. It is reasonable to conjecture that while compression may allow effective scaling to larger clusters, it would not affect the efficacy of IST.

2.4 Summary

In this chapter, I introduce *independent subnet training* for distributed training of neural networks. By stochastically partitioning the model into non-overlapping subnets, IST reduces the communication overhead for model synchronization, and the computation workload of forward-backward propagation for a thinner model on each worker. This results in two advances: *i*) IST significantly accelerates the training process comparing with standard data parallel approaches for distributed learning, and *ii*) IST scales to large models that cannot be learned using standard data parallel approaches.

Chapter 3

Tensor Relational Algebra

In this chapter, I propose a simple and concise *tensor relational algebra* over so-called *tensor relations*. At the highest level, a tensor relation is simply a binary relation between keys and multi-dimensional arrays.

There are some key reasons that this makes sense as the implementation abstraction in ML system design. Crucially, just like the relational algebra, it is a set-based abstraction whose operations are easily parallelized in the same way that relational algebra is parallelized. Because it is designed to facilitate "chunking" of tensors into smaller pieces that can be operated over using efficient CPU or GPU kernels, it is by design easy to implement efficiently across multiple machines or ASICs. Second, like the relational algebra, it is simple and concise, so it should be possible to optimize ad infinitum. Finally, it is powerful. It is easy to prove that the tensor relational algebra is at least as powerful as the Einstein notation, a standard tensor calculus. Hence, it can be used to implement anything that can be written in the Einstein notation (including matrix multiplications, convolutions, and so on).

This chapter will discuss the TRA as well as an implementation algebra (IA) which is easy to implement in a distributed system. I consider how computations in the TRA can be transformed into computations in the IA, and propose a set of transformations or equivalences that allow re-writes of computations in the IA. Finally, a prototype implementation of the IA is provided to show that these system design can enable efficient, distributed implementations of several ML computations, which can reach or

even significantly outperform the existing HPC and ML systems, such as ScaLAPACK and PyTorch.

3.1 Tensor Relational Algebra

The TRA operates over *tensor relations*. Tensor relations contain pairs of the form:

`(key, array)`.

Conceptually, these tensor relations store sets of arrays. Each key value serves, not surprisingly, as the key for the pair. Each tensor relation has the following meta-data and constraints:

1. Each tensor relation has a key-arity of dimension k, so that each key value `key` in a relation is in $(\mathbb{Z}^*)^k$.

2. Each relation has an *array type*. Conceptually, each `array` is a multi-dimensional tensor. The relation array type consists of a *rank* $r \in \mathbb{Z}^*$ as well as a *bound* $\mathbf{b} \in (\mathbb{Z}^*)^r$. For two vectors $\mathbf{u} = \langle u_i \rangle$ and $\mathbf{v} = \langle v_i \rangle$, define $\mathbf{u} \leq \mathbf{v} \equiv \wedge_i (u_i \leq v_i)$. Define $\mathbf{u} < \mathbf{v}$ similarly. Each `array` is *bounded* by vector $\mathbf{b}$, so that for any index $\mathbf{i} \in (\mathbb{Z}^*)^r$, $\vec{0} \leq \mathbf{i} < \mathbf{b} \implies \texttt{array}_{\mathbf{i}} \in \mathbb{R}$. However, $\neg(\vec{0} \leq \mathbf{i} < \mathbf{b}) \implies \texttt{array}_{\mathbf{i}} = \perp$. That is, for any index $\mathbf{i}$ outside of the bound, $\texttt{array}_{\mathbf{i}}$ is undefined.

Denote the set of all arrays of rank r and bound $\mathbf{b}$ as $T^{(r,\mathbf{b})}$. Denote the power set of $(\mathbb{Z}^*)^k \times T^{(r,\mathbf{b})}$ as $R^{(k,r,\mathbf{b})}$; this is the set of all possible tensor relations with k-dimensional keys, storing arrays of type $T^{(r,\mathbf{b})}$.

3.1.1 Operations in Tensor Relational Algebra

Given this, the TRA is essentially a set of higher-order functions over tensor relations. That is, each operation takes as input a kernel function defined over multi-dimensional

arrays (in practice, this function is likely be an array-based MKL, CUDA, or Verilog kernel) and returns a function over tensor relations.

The introduction begins by giving an overview of the higher-order functions taking binary functions as input: aggregation (denoted using Σ) and join (denoted using $\bowtie$). (1) *Aggregation* is a function:

$$\Sigma : \left((\mathbb{Z}^*)^g \times \left(T^{(r,\mathbf{b})} \times T^{(r,\mathbf{b})} \rightarrow T^{(r,\mathbf{b})}\right)\right) \rightarrow \left(R^{(k,r,\mathbf{b})} \rightarrow R^{(g,r,\mathbf{b})}\right)$$

$\Sigma_{(\texttt{groupByKeys, aggOp})}$ takes as input a list of key dimensions to aggregate according to `groupByKeys` as well as an array kernel operation `aggOp`, and then returns a function that takes as input a tensor relation, groups the arrays in the relation based upon the indicated key values, and applies `aggOp` to the arrays in the group.

For example, consider the matrix $\mathbf{A}$,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{bmatrix},$$

which may be stored as a tensor relation

$$\mathtt{R}_A = \left\{ \left(\langle 0,0 \rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \right), \left(\langle 0,1 \rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \right), \left(\langle 1,0 \rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \right), \left(\langle 1,1 \rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix} \right) \right\}.$$

One can sum up the individual arrays vertically using

$$\Sigma_{(\langle 1 \rangle, \texttt{matAdd})}(\mathtt{R}_A)$$

which gives:

$$\left\{ \left(\langle 0 \rangle, \begin{bmatrix} 10 & 12 \\ 14 & 16 \end{bmatrix} \right), \left(\langle 1 \rangle, \begin{bmatrix} 18 & 20 \\ 22 & 24 \end{bmatrix} \right) \right\}.$$

Because of the argument $\langle 1 \rangle$, the call $\Sigma_{(\langle 1 \rangle, \texttt{matAdd})}$ constructs an aggregation function that groups all pairs having the same value for the key in position 1, and sums them. Or one can sum up the individual arrays into a single array using:

$$\Sigma_{(\langle\rangle, \texttt{matAdd})}(\mathtt{R}_A)$$

which gives:

$$\left\{ \left(\langle\rangle, \begin{bmatrix} 28 & 32 \\ 36 & 40 \end{bmatrix} \right) \right\}.$$

(2) *Join* is a function:

$$\bowtie: \left((\mathbb{Z}^*)^g \times (\mathbb{Z}^*)^g \times \left(T^{(r_l, \mathbf{b}_l)} \times T^{(r_r, \mathbf{b}_r)} \rightarrow T^{(r_o, \mathbf{b}_o)} \right) \right)$$
$$\rightarrow \left(R^{(k_l, r_l, \mathbf{b}_l)} \times R^{(k_r, r_r, \mathbf{b}_r)} \rightarrow R^{(k_l + k_r - g, r_o, \mathbf{b}_o)} \right)$$

$\bowtie_{(\texttt{joinKeysL, joinKeysR, projOp})}$ takes as input a set of key dimensions to join on from the left and from the right, as well as an operation to run over all `(leftArray, rightArray)` pairs that are created during the join, and returns a function that performs the join and applies `projOp` to all pairs. Similar to a natural join in classical databases systems, the output key is all of the key values from the left input, with all of the key values from the right input appended to them, subject to the constraint that no value in `joinKeysR` is repeated a second time.

With join and aggregation one may implement matrix multiply over two matrices stored as tensor relations. Imagine to implement $\mathbf{A} \times \mathbf{A}$ for the matrix $\mathbf{A}$ defined previously, where $\mathbf{A}$ is stored as a tensor relation $\mathtt{R}_A$. This can be written as:

$$\Sigma_{(\langle 0,2 \rangle, \texttt{matAdd})} \left(\bowtie_{(\langle 1 \rangle, \langle 0 \rangle, \texttt{matMul})} (\mathtt{R}_A, \mathtt{R}_A) \right)$$

This computes a matrix multiply of the matrix $\mathbf{A}$ because all of the pairs in $\mathtt{R}_A$ are first joined on key index 1 from the first instance of $\mathtt{R}_A$ equaling key index 0 from the second instance of $\mathtt{R}_A$. Each pair of arrays are then multiplied using the kernel `matMul`. For example,

$$\left(\langle 0,1\rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right) \text{ and } \left(\langle 1,0\rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right)$$

are joined to produce

$$\left(\langle 0,1,0\rangle, \begin{bmatrix} 111 & 122 \\ 151 & 166 \end{bmatrix}\right).$$

The index $\langle 0,1,0\rangle$ in this output pair is a combination of $\langle 0,1\rangle$ and $\langle 1,0\rangle$ from the two input pairs, with the redundant index entry dropped (redundant because it is known that two of the entries in positions 1 and 0, respectively, are repeated due to the join). Next, the arrays are aggregated using `matAdd`, summing out index 1 (keeping indices $\langle 0,2\rangle$ as `groupByKeys`), to complete the matrix multiply.

In contrast to join and aggregation, rekey, filter and transform are higher-order functions taking a unary function as input.

(3) *ReKey* allows manipulation of keys:

$$\textsc{ReKey} : \left((\mathbb{Z}^*)^{k_i} \rightarrow (\mathbb{Z}^*)^{k_o}\right) \rightarrow \left(R^{(k_i,r,\mathbf{b})} \rightarrow R^{(k_o,r,\mathbf{b})}\right)$$

$\textsc{ReKey}_{\texttt{(keyFunc)}}$ applies the `keyFunc` on every key in the relation and generates a new key.

(4) *Filter* is a function:

$$\sigma : \left((\mathbb{Z}^*)^{k} \rightarrow \{\texttt{true}, \texttt{false}\}\right) \rightarrow \left(R^{(k,r,\mathbf{b})} \rightarrow R^{(k,r,\mathbf{b})}\right)$$

$\sigma_{\texttt{(boolFunc)}}$ returns a function that accepts a tensor relation and filters each of the tuples in the tensor relation by applying `boolFunc` to the keys in the tuples.

(5) *Transform* is a function:

$$\lambda : \left(T^{(r_i,\mathbf{b}_i)} \rightarrow T^{(r_o,\mathbf{b}_o)}\right) \rightarrow \left(R^{(k,r_i,\mathbf{b}_i)} \rightarrow R^{(k,r_o,\mathbf{b}_o)}\right)$$

$\lambda_{\texttt{(transformFunc)}}$ returns a function that accepts a tensor relation and applies the kernel function `transformFunc` to the array in each tuple from the tensor relation.

For an example of the rekey, filter and transform operations, assume one has a kernel operation `diag` that diagonalizes a matrix block, a function $\texttt{isEq}(\langle \texttt{k}_0, \texttt{k}_1 \rangle) \mapsto \texttt{k}_0 = \texttt{k}_1$ that accepts a key and returns true if entries in position 0 and 1 in the key are identical to one another, and a function $\texttt{getKey0}(\langle \texttt{k}_0, \texttt{k}_1 \rangle) \mapsto \langle \texttt{k}_0 \rangle$ that returns the first dimension of a key. One can use these functions along with filter, rekay, and transform to diagonalize a matrix $\mathbf{A}$ represented as a tensor relation $\texttt{R}_A$, by first examining the keys to remove all pairs that do not contain entries along the diagonal, and then diagonalizing the resulting arrays:

$$\lambda_{\texttt{(diag)}} \left(\textsc{ReKey}_{\texttt{(getKey0)}} \left(\sigma_{\texttt{(isEq)}} \left(\texttt{R}_A \right) \right) \right).$$

In addition, there are a number of operations that can be used to alter the organization of arrays within a tensor relation. This allows the manipulation of how a tensor is represented as a tensor relation. For this purpose, tile and concat are defined:
(6) *Tile*:

$$\textsc{Tile} : (\mathbb{Z}^* \times \mathbb{Z}^*) \rightarrow \left(R^{(k,r,\mathbf{b})} \rightarrow R^{(k+1,r,\mathbf{b}')}\right)$$

$\textsc{Tile}_{\texttt{(tDim, tSize)}}$ returns a function that decomposes (or tiles) each array along a dimension `tDim` to arrays of the target `tSize` (by applying the `arrayTileOp` function on the array). As a result, a new key dimension is created, that effectively counts which tile the tuples holds along the tiling dimension.

For example, consider the matrix $\mathbf{B}$,

$$\mathbf{B} = \begin{bmatrix} 1 & 2 & 5 & 6 & 9 & 10 & 13 & 14 \\ 3 & 4 & 7 & 8 & 11 & 12 & 15 & 16 \end{bmatrix},$$

stored in tensor relation:

$$\mathtt{R}_B = \left\{ \left(\langle 0 \rangle, \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{bmatrix} \right), \left(\langle 1 \rangle, \begin{bmatrix} 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{bmatrix} \right) \right\}$$

If one makes the call $\textsc{Tile}_{(1,2)}\left(\mathtt{R}_B\right)$, one will decompose each array along dimension 1, creating one new array for each two columns. In addition, a new key dimension is created, that effectively counts which tile the pair holds along the tiling dimension:

$$\textsc{Tile}_{(1,2)}\left(\mathtt{R}_B\right) = \left\{ \left(\langle 0,0 \rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \right), \left(\langle 0,1 \rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \right), \left(\langle 1,0 \rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \right), \left(\langle 1,1 \rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix} \right) \right\}.$$

Note that the interpretation of a tensor relation as a representation of one or more tensors is not defined; this interpretation is the result of the mapping from the ML system's computational abstraction (such as Einstein notation or Ricci calculus) onto the tensor-relation-based implementation abstraction. Whatever the mapping, one may find himself in a situation where it is necessary to manipulate the key in each pair in a tensor relation so that the key is consistent with the desired interpretation. For example, the tensor relation $\mathtt{R}_B$ defined above represents a matrix with eight columns and two rows, so $\textsc{Tile}_{(1,2)}\left(\mathtt{R}_B\right)$ is inconsistent with this, logically representing a matrix having four columns and four rows. For this purpose, the ReKey operator can be leveraged as defined before.

For example, one can rekey the output of $\textsc{Tile}_{(1,2)}\left(\mathtt{R}_B\right)$ so that logically, it corresponds to a two-by-eight matrix:

$$\textsc{ReKey}_{(\langle \mathtt{k}_0, \mathtt{k}_1 \rangle \rightarrow \langle 2\mathtt{k}_0 + \mathtt{k}_1 \rangle)}\left(\textsc{Tile}_{(1,2)}\left(\mathtt{R}_B\right)\right)$$

This will result in:

$$\left\{ \left(\langle 0 \rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \right), \left(\langle 1 \rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \right), \left(\langle 2 \rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \right), \left(\langle 3 \rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix} \right) \right\}$$

Finally, it is possible to undo a tiling:

(7) *Concat*:

$$\textsc{Concat} : (\mathbb{Z}^* \times \mathbb{Z}^*) \rightarrow \left(R^{(k,r,\mathbf{b})} \rightarrow R^{(k-1,r,\mathbf{b'})}\right)$$

$\textsc{Concat}_{(\texttt{keyDim, arrayDim})}$ is an inverse to tile, which first groups all pairs in the relation using all of the key dimensions *other than* `keyDim`, then concatenates all of the arrays in each group along `arrayDim`, with the concatenation ordering provided by `keyDim`.

A call to $\textsc{Concat}_{(1,1)}\left(\textsc{Tile}_{(1,2)}\left(\mathtt{R}_B\right)\right)$ first groups all pairs in $\textsc{Tile}_{(1,2)}\left(\mathtt{R}_B\right)$ using all of the key dimensions other than key dimension 1, and then concatenates the arrays in each group along array dimension 1, with the ordering provided by key dimension 1. Hence, this computation simply results in the recovery of $\mathtt{R}_B$.

3.1.2 Integrity Constraints and Closedness

There are two important integrity constraints that each tensor relation must follow: uniqueness of keys, and a lack of "holes" in the tensor relation. The primary reason for defining this constraints is facilitating easy, cost-based optimization. With such constraints, cardinality estimation, one of the most vexing problems in relational optimization, goes away — see Section 3.3.3. Further, neither is particularly burdensome

when expressing computations using the TRA. In fact, if the interpretation of a tensor relation of type $R^{(k,r,\mathbf{b})}$ is that it represents a r-dimensional tensor decomposed into chunks, these constraints are quite natural:

- *Uniqueness*: every key should be unique in a tensor relation.
- *Continuity*: there are no "holes". Given a tensor relation R, of key-arity k, define the *frontier* of R as FRONT(R). $\mathbf{f}$ = FRONT(R) is a k-dimensional vector that bounds all keys in R. That is, for each key vector $\mathbf{k}$ in R, $\mathbf{k} < \mathbf{f}$. Further, the frontier is the "smallest" vector bounding R, in that for any other vector $\mathbf{f}'$ bounding R, $\mathbf{f} \leq \mathbf{f}'$. *Continuity* requires that for any vector $\mathbf{k} < \mathbf{f}$, some tuple in R have the key $\mathbf{k}$.

It is easy to show that for the majority of TRA operations — the exceptions being the rekey and filter operations — tensor relations are closed. That is, if the input(s) are tensor relation(s) that obey uniqueness and continuity, then the output must be a tensor relation that obeys these constraints. Filtering a tensor relation or altering the keys can obviously violate the constraints, where the the former probably leads to holes in the resulting relation, and the latter can result in repeated key values. Analyzing a TRA expression to automatically detect whether it can violate these constraints is left as future work; it is reasonable to conjecture that if the filtering predicate (or re-keying computation) are limited to simple arithmetic expressions, it may be possible to check for closedness using an SMT solver [59].

3.1.3 Expressivity

This section argues that the TRA is expressive as the Einstein notation, which is the most common tensor calculus.

Consider the following attribute grammar, with inherited attributes I and J (I and J are ordered sequences of symbols, serving as lists of indices). Let `exp` be an expression produced by recursively applying the following production rules:

$$f(I, J) \rightarrow f(I_L, J_L)\ \texttt{binop}\ f(I_R, J_R)$$

$$f(I, J) \rightarrow \texttt{unop}(f(I, J))$$

$$f(I, J) \rightarrow \texttt{TensorLiteral}_K$$

In the first rule, I_L is a subsequence of I and J_L is a subsequence of J. In the last rule, K is a permutation of $I \circ J$, where $\circ$ is concatenation. `binop`, `unop` are mathematical functions that operate on scalars.

For an example of this, consider the input lists $I = \langle i_1, i_2 \rangle$, $J = \langle j_1 \rangle$. One can fire the first rule to obtain the expression $f(\langle i_1 \rangle, \langle j_1 \rangle) \times f(\langle i_2 \rangle, \langle j_1 \rangle)$. Now one can fire the last rule to obtain $\mathbf{A}_{i_1,j_1} \times f(\langle i_2 \rangle, \langle j_1 \rangle)$, and fire it again to obtain the expression $\mathbf{A}_{i_1,j_1} \times \mathbf{B}_{j_1,i_2}$.

Let an expression `exp` produced in this way refer to the tensor $\mathbf{X}$, where $\mathbf{X}_I = \sum_J \texttt{exp}$. This is known as *Einstein notation*. Note that, strictly speaking, this is not the classical Einstein notation, as the lists of indices I and J are explicitly given. In the classical Einstein notation, I and J are inferred from the expression itself. Classical Einstein notation presents certain complications, such as the introduction of the Kronecker delta symbol, that one can avoid via the use of an explicit I and J. This does not affect the expressibility argument given here. Note that $\mathbf{A}_{i_1,j_1} \times \mathbf{B}_{j_1,i_2}$ is Einstein notation for matrix multiply.

It is possible to show that TRA is as powerful as Einstein notation, in that it can implement any computation that can be specified in Einstein notation. One can begin by using a tensor relation to store a tensor. Assume a tensor relation `R` of pairs `(key,`

`val)` where each `val` is a scalar (array with a single value) corresponds to the tensor $\mathbf{X}$ where if `(key, val)` is in `R`, then $\mathbf{X}_{\texttt{key}} = \texttt{val}$ (note that the proof does not use the fact that tensor relations to store arrays; this does not seem to add to the power of the TRA, though it does mean that it can be implemented efficiently).

The proof of the power of the TRA is by induction on the complexity of an expression exp produced by the grammar given above.

The base case is when `exp` takes the value $\texttt{TensorLiteral}_K$. Assume the tensor referred is stored as a tensor relation `R`. In this case, simply rekey `R`, applying the permutation from K to $I \circ J$ to reorder the key attributes, and then aggregate using the positions indicated by the attributes in J, using addition as the kernel function.

Now, assume that one can implement any computation that is simpler than `exp` (that is, constructed from fewer production rules) and prove that one can implement exp.

Two cases. Assume `exp` takes the form `unop(subexp)`. By induction, it is known that one can compute a tensor relation `R` holding the result of `subexp`, given index lists I and J. Then, simply use the TRA to map `R` to apply the kernel function corresponding to `unop`, to compute the result of `exp` given index lists I and J.

If exp takes the form of $\texttt{exp}_L$ `binop` $\texttt{exp}_R$, let I_L be the subsequence of indices in I referred to by $\texttt{exp}_L$, and let J_L be the subsequence of indices in J referred to by $\texttt{exp}_L$. Define I_R and J_R similarly. To evaluate `exp`, first use the TRA to compute the relation $\texttt{R}_L$ corresponding to $\texttt{exp}_L$ with index lists $I = I_L \circ (J_L \vee J_R)$ and $J = J_L - (J_L \vee J_R)$. Here, $(J_L \vee J_R)$ merges the two lists, removing duplicates, and $-$ removes every index present in the second list from the first. Similarly compute $\texttt{R}_R$. Next join $\texttt{R}_L$ and $\texttt{R}_R$ on the last $|(J_L \vee J_R)|$ key attributes in each relation, applying the kernel function corresponding to `binop`, and then aggregate, grouping on the joined keys, applying

scalar addition. Finally, rekey the result to that the order of key attributes matches the original I. Thus, one can implement any computation expressed in the Einstein notation.

3.2 Implementation Algebra

TRA serves as a reasonable target for a compiler from a high-level language such as the Ricci Calculus or the Einstein Notation. However, it is not suitable for direct implementation. This section now describes the *implementation algebra* (IA) that is suitable for execution in such an environment.

3.2.1 Physical Tensor Relation

In IA, each `(key, array)` tuple in a tensor relation is extended with an additional `site` attribute, so that a *physical tensor relation* R will consist of triples:

$$\texttt{(key, array, site)}.$$

The `site` attribute takes a value in $\{1, 2, ..., s\}$ where s is the number of computation sites. Conceptually, the `site` value indicates the location where the tuple is stored; this could be a machine in a distributed cluster, or a compute unit like a GPU.

Each physical tensor relation can map a particular `(key, array)` pair to one or more sites. There are a few especially important mappings, recognized by the predicates $\textsc{All}()$ and $\textsc{Part}_D()$:

1. If $\textsc{All}(\mathsf{R}) =$ true, it indicates that if the `array` attribute is projected away, the resulting set will take the value:

$$\{\mathbf{k} \text{ s.t. } \mathbf{k} \leq \textsc{Front}(\mathsf{R}) \times \{1...s\}\}$$

where FRONT(R) is the frontier of R (the frontier of a physical tensor relation is defined as in a "regular" tensor relation). In other words, this means that each possible `(key, array)` tuple in R appears at all sites.

2. If $\text{PART}_D(\mathsf{R}) =$ true for some set $D \subseteq \{1...k\}$, it indicates that: (1) for a given `key` value, there is only one tuple in R and (2) two tuples with the same key values for all dimensions in D must be found on the same site. In other words, R is partitioned according to the key dimensions in the set D.

Now, it is ready to describe the IA. Let $\mathcal{R}^{(k,r,\mathbf{b},s)}$ specify the set of all valid physical tensor relations with key-arity of dimension k, storing arrays of type $T^{(r,\mathbf{b})}$, and partitioned across s sites.

3.2.2 Operations in Implementation Algebra

The first two operations are concerned with manipulating the assigning of tuples in a physical relation to sites, while the later four operations operate over the `key` and `array` attributes.

(1) *Broadcast* is defined as:

$$\text{BCAST} : \mathcal{R}^{(k,r,\mathbf{b},s)} \rightarrow \mathcal{R}^{(k,r,\mathbf{b},s)}$$

Given a physical tensor relation, BCAST simply ensures that each tuple takes each site value, so that (a) the set of `(key, array)` pairs is unchanged after BCAST, but (b) in any physical relation R output from a broadcast, ALL(R) = true.

(2) *Shuffle* is is defined as:

$$\text{SHUF} : 2^{\{1...k\}} \rightarrow \left(\mathcal{R}^{(k,r,\mathbf{b},s)} \rightarrow \mathcal{R}^{(k,r,\mathbf{b},s)}\right)$$

$\textsc{Shuf}_{\texttt{(partDims)}}$ is a function that accepts a set of key dimensions, and returns a function that accepts physical tensor relation, and then re-partitions the physical tensor relation, so that (a) the set of `(key, array)` pairs is unchanged after SHUF, but (b) in any physical relation R output from a shuffle, $\textsc{Part}_{\texttt{partDims}}(\mathsf{R}) =$ true.

(3) *Local join* is an extension of the TRA's join operation:

$$\bowtie^L : \Big((\mathbb{Z}^*)^g \times (\mathbb{Z}^*)^g \times \big(T^{(r_l,\mathbf{b}_l)} \times T^{(r_r,\mathbf{b}_r)} \rightarrow T^{(r_o,\mathbf{b}_o)} \big) \Big) \\ \rightarrow \big(\mathcal{R}^{(k_l,r_l,\mathbf{b}_l,s)} \times \mathcal{R}^{(k_r,r_r,\mathbf{b}_r,s)} \rightarrow \mathcal{R}^{(k_l+k_r-g,r_o,\mathbf{b}_o,s)} \big)$$

Similar to TRA join ($\bowtie$), $\bowtie^L{}_{\texttt{(joinKeysL, joinKeysR, projOp)}}$ takes as input a set of key dimensions to join on from the left and from the right, as well as a kernel operation to run over all `(leftArray, rightArray)` pairs that are created during the join. The key difference that the local join combines *only* on pairs from the left and right inputs that have the **same** `site` values. If two tuples successfully join, the corresponding output tuple will have the `site` value as those input tuples.

(4) *Local aggregation* is an extension of TRA aggregation:

$$\Sigma^L : \big((\mathbb{Z}^*)^k \times \big(T^{(r,\mathbf{b})} \times T^{(r,\mathbf{b})} \rightarrow T^{(r,\mathbf{b})} \big) \big) \\ \rightarrow \big(\mathcal{R}^{(k,r,\mathbf{b},s)} \rightarrow \mathcal{R}^{(g,r,\mathbf{b},s)} \big)$$

Like TRA aggregation (Σ), $\Sigma^L{}_{\texttt{(groupByKeys, aggOp)}}$ takes as input a list of key dimensions to aggregate over `groupByKeys` as well as a kernel function `aggOp`. However, it returns a function that takes as input a physical tensor relation, groups the arrays in the relation based upon the indicated key values *and* the site value, and applies `aggOp` to the arrays in the group. Each output tuple in the resulting, physical tensor relation will take its `site` value from the `site` value of the set of input tuples that were aggregated to produce it.

(5) IA has a filter:

$$\sigma^L : \Big((\mathbb{Z}^*)^g \rightarrow \{\texttt{true}, \texttt{false}\} \Big) \rightarrow \left(\mathcal{R}^{(k,r,\mathbf{b},s)} \rightarrow \mathcal{R}^{(k,r,\mathbf{b},s)} \right)$$

The only difference is that each accepted input tuple's `site` value is carried through the filter.

(6) Map provides two functionalities:

$$\lambda^L : \Big(\left((\mathbb{Z}^*)^{k_i} \rightarrow \left((\mathbb{Z}^*)^{k_o} \right)^m \right) \times \left(T^{(r_i,\mathbf{b}_i)} \rightarrow \left(T^{(r_o,\mathbf{b}_o)} \right)^m \right) \Big) \rightarrow \left(\mathcal{R}^{(k_i,r_i,\mathbf{b}_i,s)} \rightarrow \mathcal{R}^{(k_o,r_o,\mathbf{b}_o,s)} \right)$$

$\lambda^L{}_{\texttt{(keyMapFunc,arrayMapFunc)}}$ is a multi-map. It returns a function that applies `keyMapFunc` to each `key` value in the input and applies `arrayMapFunc` to each `array` value in the input. Both `keyMapFunc` and `arrayMapFunc` return m output tuples per input tuple; the `site` value is simply copied from input to output. One can subsequently call m the *arity* of `keyMapFunc`/`arrayMapFunc`. In most cases the arity of these functions will be one, but on some cases (such as replication-based matrix multiply, see Section 4.2.2), the arity will be greater.

3.3 Compilation and Optimization

One of the central hypotheses of this chapter is that layering a set-based abstraction on top of a multi-dimensional array makes it much easy to run tensor-based computations in a distributed environment; after all, it is much easier to distribute set-based computations than it is to distribute array-based computations.

To distribute such computations so that they can run efficiently requires an optimization framework. This section discusses three core questions related to actually distributing a computation specified in the TRA: (1) How is that TRA computation

compiled into an equivalent statement in IA? (2) What are a set of equivalence rules that allow computations in IA to be re-written, so as to produce different implementations that are known to produce the same results, but may run more efficiently? And (3) How to cost those different, equivalent implementations, so that a search strategy can be used to choose the most efficient one?

3.3.1 Compiling the TRA

Compiling a TRA computation is a matter of traversing the program, and applying a simple set of rules to produce an equivalent computation in IA. Here, two computations are equivalent if they are guaranteed to produce a tensor relation and a physical tensor relation that are equivalent. The tensor relation and a physical tensor relation are equivalent if projecting away the `site` attribute from the latter results in a tensor relation whose contents are identical to the former.

Note that though there can be multiple physical implementations for a given TRA computation, the compiler will generate one of such physical implementation as the initial physical computation, and an optimizer will typically be responsible for applying a search algorithm to produce the optimal physical plan represented by IA.

A complete set of rules mapping from TRA operations to IA operations are listed in Table 3.1. The notation in Table 3.1 are denoted as: tensor relations $\mathtt{R}$, $\mathtt{R}_l$ and $\mathtt{R}_r$ are stored as the corresponding physical tensor relations R, R_l and R_r; `idOp` represents an identity map for key or array; `arrayTileOp` is a kernel function splitting the array to chunks of the indicated size on the indicated dimension; `arrayConcatOp` reverses this. `keyTileOp` is similar to `arrayTileOp`, but operates on keys; $\langle \mathtt{keyDim} \rangle^c$ represents the complement set of $\langle \mathtt{keyDim} \rangle$.

TRA expression	**Corresponding IA**
$\Sigma_{\texttt{(groupByKeys, aggOp)}}\left(\mathsf{R}\right)$	$\Sigma^{L}{}_{\texttt{(groupByKeys, aggOp)}}\left(\textsc{Shuf}_{\texttt{(groupByKeys)}}\left(\mathsf{R}\right)\right)$
$\bowtie_{\texttt{(joinKeysL, joinKeysR, projOp)}}\left(\mathsf{R}_l, \mathsf{R}_r\right)$	$\bowtie^{L}{}_{\texttt{(joinKeysL, joinKeysR, projOp)}}\left(\textsc{Bcast}\left(\mathsf{R}_l\right), \mathsf{R}_r\right)$
$\textsc{ReKey}_{\texttt{(keyFunc)}}\left(\mathsf{R}\right)$	$\lambda^{L}{}_{\texttt{(keyFunc,idOp)}}\left(\mathsf{R}\right)$
$\sigma_{\texttt{(boolFunc)}}\left(\mathsf{R}\right)$	$\sigma^{L}{}_{\texttt{(boolFunc)}}\left(\mathsf{R}\right)$
$\lambda_{\texttt{(transformFunc)}}\left(\mathsf{R}\right)$	$\lambda^{L}{}_{\texttt{(idOp,transformFunc)}}\left(\mathsf{R}\right)$
$\textsc{Tile}_{\texttt{(tileDim, tileSize)}}\left(\mathsf{R}\right)$	$\lambda^{L}{}_{\texttt{(keyTileOp(tDim, tSize), arrayTileOp(tDim, tSize))}}\left(\mathsf{R}\right)$
$\textsc{Concat}_{\texttt{(keyDim, arrayDim)}}\left(\mathsf{R}\right)$	$\Sigma^{L}{}_{(\langle\texttt{keyDim}\rangle^{c},\texttt{arrayConcatOp})}\left(\textsc{Shuf}_{\langle\texttt{keyDim}\rangle^{c}}\left(\mathsf{R}\right)\right)$

Table 3.1 : Translation from TRA to IA.

3.3.2 Equivalence Rules

Once a (possibly inefficient) computation in IA is produced, it can be optimized via the application of a set of equivalence rules. An *equivalence rule* for IA expressions holds if, for any input physical tensor relations, the two expressions produce equivalent outputs — two physical tensor relations are said to be equivalent if they contain the same set of `(key, array)` pairs after projecting away the `site` attribute.

There are two classes of equivalence rules: *simple equivalence rules* which are often extensions of classic relational equivalence rules (e.g., commutative property of selections), and *domain-specific equivalence rules* that are more complex transformations that always hold, and tend to be useful for mathematical computations, such as matrix multiplications.

Simple Equivalence Rules

Simple Equivalence Rules fall into two categories: i) those based on kernel function composition, and ii) equivalence rules based on optimization of re-partitions. Kernel function composition targets the order or location of the application of kernel functions in order to reduce the computation load and memory consumption. This is closely related to the idea of ML operator fusion, which has been explored in systems such as TVM [60] (though TVM does not consider the sort of distributed computations considered here). Re-partition rules formalize notions of distributed query optimization over tensor relations, and are primarily designed to reduce communication.

Simple equivalence rules for kernel function composition are listed below:

R1-1. Filter operations can be merged. For a physical relation R:

$$\boxed{\sigma^{L}{}_{(\texttt{boolFunc1})}\left(\sigma^{L}{}_{(\texttt{boolFunc2})}\left(\mathsf{R}\right)\right) \equiv \sigma^{L}{}_{(\texttt{boolFunc1}\wedge\texttt{boolFunc2})}\left(\mathsf{R}\right).}$$

R1-2. Map operations can be merged, if the output arity of the key and array mapping functions is one. For a physical relation R:

$$\boxed{\begin{aligned} & \lambda^{L}{}_{(\texttt{keyMapFunc1, arrayMapFunc1})}\left(\lambda^{L}{}_{(\texttt{keyMapFunc2, arrayMapFunc2})}\left(\mathsf{R}\right)\right) \\ \equiv \quad & \lambda^{L}{}_{(\texttt{keyMapFunc1}\circ\texttt{keyMapFunc2},\texttt{arrayMapFunc1}\circ\texttt{arrayMapFunc2})}\left(\mathsf{R}\right) \end{aligned}}$$

R1-3. Map and filter are commutative if `keyMapFunc` is an identify function (`idOp`). For a physical relation $\mathsf{R} \in \mathcal{R}^{(k,r,\mathbf{b},s)}$, if $\forall$ `key` $\in (\mathbb{Z}^*)^k$, `keyMapFunc`(`key`) = `key`, then:

$$\boxed{\begin{aligned} & \lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\sigma^{L}{}_{(\texttt{boolFunc})}\left(\mathsf{R}\right)\right) \\ \equiv \quad & \sigma^{L}{}_{(\texttt{boolFunc})}\left(\lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\mathsf{R}\right)\right) \end{aligned}}$$

R1-4. The `arrayMapFunc` in map can be composed with local aggregation if `keyMapFunc` is an identify function (`idOp`). For a physical relation $\mathsf{R} \in \mathcal{R}^{(k,r,\mathbf{b},s)}$, if $\forall$ `key` $\in (\mathbb{Z}^*)^k$,

$\texttt{keyMapFunc}(\texttt{key}) = \texttt{key}$, then:

$$\begin{array}{|rc|}\hline & \lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathsf{R}\right)\right) \\ \equiv & \Sigma^{L}{}_{(\texttt{groupByKeys, arrayMapFunc}\circ\texttt{aggOp})}\left(\mathsf{R}\right) \\ \hline\end{array}$$

And if the kernel function $\texttt{arrayMapFunc}$ and $\texttt{aggOp}$ is distributive: $\forall\, \texttt{array}_1, \texttt{array}_2 \in T^{(r,\mathbf{b})}$:

$$\begin{aligned}&\texttt{arrayMapFunc}\left(\texttt{aggOp}\left(\texttt{array}_1, \texttt{array}_2\right)\right) \\ =&\texttt{aggOp}\left(\texttt{arrayMapFunc}\left(\texttt{array}_1\right), \texttt{arrayMapFunc}\left(\texttt{array}_2\right)\right)\end{aligned}$$

then:

$$\begin{array}{|rc|}\hline & \lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathsf{R}\right)\right) \\ \equiv & \Sigma^{L}{}_{(\texttt{groupByKeys, aggOp}\circ\texttt{arrayMapFunc})}\left(\mathsf{R}\right) \\ \hline\end{array}$$

R1-5. The $\texttt{boolFunc}$ in filter can be composed with local aggregation if the kernel function only depends on $\texttt{groupByKeys}$. For a physical relation $\mathsf{R} \in \mathcal{R}^{(k,r,\mathbf{b},s)}$, $\forall\, \texttt{key}_1, \texttt{key}_2 \in (\mathbb{Z}^*)^k$, if

$$\begin{aligned}&\texttt{boolFunc}\left(\Pi_{\texttt{groupByKeys}}\left(\texttt{key}_1\right)\right) = \texttt{boolFunc}\left(\Pi_{\texttt{groupByKeys}}\left(\texttt{key}_2\right)\right) \\ \Rightarrow&\texttt{boolFunc}\left(\texttt{key}_1\right) = \texttt{boolFunc}\left(\texttt{key}_2\right)\end{aligned}$$

then:

$$\boxed{\sigma^{L}{}_{(\texttt{boolFunc})}\left(\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathsf{R}\right)\right) \equiv \Sigma^{L}{}_{(\texttt{boolFunc(groupByKeys), aggOp})}\left(\mathsf{R}\right)}$$

R1-6. The kernel function in local filter can be pushed down with local join if the $\texttt{boolFunc}$ only checks the joined keys. For physical relations R_l and R_r:

$$\begin{array}{|rc|}\hline & \sigma^{L}{}_{(\texttt{boolFunc})}\left(\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathsf{R}_l, \mathsf{R}_r\right)\right) \\ \equiv & \bowtie^{L}{}_{(\texttt{joinKeysL, joinKeys, projOp})}\left(\sigma^{L}{}_{(\texttt{boolFunc})}\left(\mathsf{R}_l\right), \sigma^{L}{}_{(\texttt{boolFunc})}\left(\mathsf{R}_r\right)\right) \\ \hline\end{array}$$

R1-7. The kernel function in local map can be composed with local join, if the output arity of the key and array mapping functions is one. For physical relations R_l and R_r:

$$\begin{aligned} & \lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathsf{R}_l, \mathsf{R}_r\right)\right) \\ \equiv\ & \bowtie^{L}{}_{(\texttt{keyMapFunc(joinKeysL), keyMapFunc(joinKeysR), arrayMapFunc}\circ\texttt{projOp})}\left(\mathsf{R}_l, \mathsf{R}_r\right) \end{aligned}$$

And if the kernel function `arrayMapFunc` and `projOp` is distributive: $\forall\, \texttt{array}_1, \texttt{array}_2 \in T^{(r,\mathbf{b})}$,

$$\begin{aligned} & \texttt{arrayMapFunc} : (\texttt{projOp}\left(\texttt{array}_1, \texttt{array}_2\right)) \\ =& \texttt{projOp}\left(\texttt{arrayMapFunc}\left(\texttt{array}_1\right), \texttt{arrayMapFunc}\left(\texttt{array}_2\right)\right) \end{aligned}$$

then:

$$\begin{aligned} & \lambda^{L}{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathsf{R}_l, \mathsf{R}_r\right)\right) \\ \equiv\ & \bowtie^{L}{}_{(\texttt{keyMapFunc(joinKeysL), keyMapFunc(joinKeysR), projOp}\circ\texttt{arrayMapFunc})}\left(\mathsf{R}_l, \mathsf{R}_r\right) \end{aligned}$$

Simple equivalence rules for optimization of re-partitions are listed below:

R2-1. Only the final broadcast/shuffle in a sequence of broadcast/shuffle operations is needed. For a physical relation R:

$$\textsc{Bcast}\left(\textsc{Bcast}\left(...\textsc{Bcast}\left(\mathsf{R}\right)\right)\right) \equiv \textsc{Bcast}\left(\mathsf{R}\right)$$

$$\textsc{Shuf}_{(\texttt{partDims}_n)}\left(...\textsc{Shuf}_{(\texttt{partDims}_2)}\left(\textsc{Shuf}_{(\texttt{partDims}_1)}\left(\mathsf{R}\right)\right)\right) \equiv \textsc{Shuf}_{(\texttt{partDims}_n)}\left(\mathsf{R}\right)$$

R2-2. The re-partition operations are commutative with the local filter operation. For a physical relation R:

$$\textsc{Bcast}\left(\sigma^{L}{}_{(\texttt{boolFunc})}\left(\mathsf{R}\right)\right) \equiv \sigma^{L}{}_{(\texttt{boolFunc})}\left(\textsc{Bcast}\left(\mathsf{R}\right)\right)$$

$$\textsc{Shuf}_{(\texttt{partDim})}\left(\sigma^{L}{}_{(\texttt{boolFunc})}\left(\mathsf{R}\right)\right) \equiv \sigma^{L}{}_{(\texttt{boolFunc})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathsf{R}\right)\right)$$

R2-3. The re-partition operations are commutative with the local map operation. For a physical relation R:

$$\boxed{\textsc{Bcast}\left(\lambda^{L}{}_{(\texttt{keyMapFunc},\texttt{arrayMapFunc})}\left(\mathsf{R}\right)\right) \equiv \lambda^{L}{}_{(\texttt{keyMapFunc},\texttt{arrayMapFunc})}\left(\textsc{Bcast}\left(\mathsf{R}\right)\right)}$$

And, if `keyMapFunc` is the identity function:

$$\boxed{\begin{aligned} & \textsc{Shuf}_{(\texttt{partDims})}\left(\lambda^{L}{}_{(\texttt{keyMapFunc},\texttt{arrayMapFunc})}\left(\mathsf{R}\right)\right) \\ \equiv\ & \lambda^{L}{}_{(\texttt{keyMapFunc},\texttt{arrayMapFunc})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathsf{R}\right)\right)\end{aligned}}$$

R2-4. A shuffle can be avoided if the physical relation is already partitioned by a local aggregation's `groupByKeys`. For a physical relation R, if `partDims` $\subseteq$ `groupByKeys`:

$$\boxed{\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathsf{R}\right)\right) \equiv \Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathsf{R}\right)}$$

R2-5. An aggregation can be split to two phases, if the physical relation is only partially partitioned. For a physical relation R, if `groupByKeys` $\subset$ `partDims`:

$$\boxed{\begin{aligned} & \Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathsf{R}\right)\right) \\ \equiv\ & \Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}\left(\mathsf{R}\right)\right)\right)\end{aligned}}$$

R2-6. A Join $\bowtie$ defined by the TRA can be implemented in the following equivalent ways. For physical relations R_l and R_r:

$$\boxed{\begin{aligned} & \bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Bcast}\left(\mathsf{R}_l\right), \mathsf{R}_r\right) \\ \equiv\ & \bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathsf{R}_l, \textsc{Bcast}\left(\mathsf{R}_r\right)\right) \\ \equiv\ & \bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Shuf}_{(\texttt{joinKeysL})}\left(\mathsf{R}_l\right), \textsc{Shuf}_{(\texttt{joinKeysR})}\left(\mathsf{R}_r\right)\right)\end{aligned}}$$

R2-7. The local join can be pushed through shuffle. For physical relations $\mathsf{R}_l \in \mathcal{R}^{(k_l,r_l,\mathbf{b}_l,s)}$ and $\mathsf{R}_r \in \mathcal{R}^{(k_r,r_r,\mathbf{b}_r,s)}$, if `partDims` $\subseteq$ `joinKeysL`:

$$\boxed{\begin{aligned} & \textsc{Shuf}_{(\texttt{partDims})}\left(\bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Shuf}_{(\texttt{joinKeysL})}\left(\mathsf{R}_l\right), \textsc{Shuf}_{(\texttt{joinKeysR})}\left(\mathsf{R}_r\right)\right)\right) \\ \equiv\ & \bowtie^{L}{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Shuf}_{(\texttt{joinKeysL})}\left(\mathsf{R}_l\right), \textsc{Shuf}_{(\texttt{joinKeysR})}\left(\mathsf{R}_r\right)\right)\end{aligned}}$$

Such rules are surprisingly effective for optimizing distributed tensor manipulations. Consider the example of extracting the diagonal elements of matrix $\mathbf{X}$ plus matrix $\mathbf{Y}$: $\texttt{diag}(\mathbf{X}+\mathbf{Y})$, where matrix $\mathbf{X}$ and $\mathbf{Y}$ are stored in physical tensor relations R_X and R_Y. This computation can be represented by the following TRA expression, where $\texttt{isEq}(\langle \texttt{k}_0, \texttt{k}_1 \rangle) \mapsto \texttt{k}_0 = \texttt{k}_1$, $\texttt{merge}(\langle \texttt{k}_0, \texttt{k}_1 \rangle) \mapsto \langle \texttt{k}_0 \rangle$, `matAdd` is an element-wise sum of two arrays, and `diag` diagonalizes the array. Then $\text{diag}(\mathbf{X}+\mathbf{Y})$ can be written as:

$$\lambda_{(\texttt{diag})}\left(\text{ReKey}_{(\texttt{merge})}\left(\sigma_{(\texttt{isEq})}\left(\bowtie_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\mathsf{R}_X, \mathsf{R}_Y\right)\right)\right)\right).$$

This TRA expression can be translated to the IA expression:

$$\lambda^L{}_{(\texttt{idOp},\texttt{diag})}\left(\lambda^L{}_{(\texttt{merge},\texttt{idOp})}\left(\sigma^L{}_{(\texttt{isEq})}\left(\bowtie^L{}_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\text{BCast}\left(\mathsf{R}_X\right), \mathsf{R}_Y\right)\right)\right)\right).$$

One can apply the following equivalence rules for the above IA expression:

$$\begin{aligned}
&\lambda^L{}_{(\texttt{idOp},\texttt{diag})}\left(\lambda^L{}_{(\texttt{merge},\texttt{idOp})}\left(\sigma^L{}_{(\texttt{isEq})}\left(\bowtie^L{}_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\text{BCast}\left(\mathsf{R}_X\right), \mathsf{R}_Y\right)\right)\right)\right)\\
&\overset{R1-2}{\equiv}\ \lambda^L{}_{(\texttt{merge},\texttt{diag})}\left(\sigma^L{}_{(\texttt{isEq})}\left(\bowtie^L{}_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\text{BCast}\left(\mathsf{R}_X\right), \mathsf{R}_Y\right)\right)\right)\\
&\overset{R1-6}{\equiv}\ \lambda^L{}_{(\texttt{merge},\texttt{diag})}\left(\bowtie^L{}_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\sigma^L{}_{(\texttt{isEq})}\left(\text{BCast}\left(\mathsf{R}_X\right)\right), \sigma^L{}_{(\texttt{isEq})}\left(\mathsf{R}_Y\right)\right)\right)\\
&\overset{R2-2}{\equiv}\ \lambda^L{}_{(\texttt{merge},\texttt{diag})}\left(\bowtie^L{}_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\text{BCast}\left(\sigma^L{}_{(\texttt{isEq})}\left(\mathsf{R}_X\right)\right), \sigma^L{}_{(\texttt{isEq})}\left(\mathsf{R}_Y\right)\right)\right)\\
&\overset{R1-7}{\equiv}\ \bowtie^L{}_{(\texttt{merge}(\langle 0,1\rangle),\texttt{merge}(\langle 0,1\rangle),\texttt{matAdd}\circ\texttt{diag})}\left(\text{BCast}\left(\sigma^L{}_{(\texttt{isEq})}\left(\mathsf{R}_X\right)\right), \sigma^L{}_{(\texttt{isEq})}\left(\mathsf{R}_Y\right)\right).
\end{aligned}$$

The transformation will significantly reduce both the communication overhead and the computation load: by applying R1-6, the `isEq` functions will be pushed down, this transformation not only reduces the input tuple pairs for the join to execute the `matAdd` function but also the enables reduction of communication overhead where the the filter operation is commuted with the broadcast operation by R2-2; lastly, R1-7 leverages the property that kernel functions `diag` and `matAdd` are distributive, as a result, addition will only be applied for the diagonal elements for the paired blocks after kernel function composition.

Domain-Specific Equivalence Rules

Such rules encode specific knowledge from parallel and distributed computing algorithms. Adding such rules to a system allows IA to have at its disposal common implementation strategies, that it can choose from in a cost-based manner.

This section does not attempt to produce an exhaustive list of such rules, but rather this section considers in detailing one example: distributed matrix multiplication over tensor relations $\mathtt{R}_X$ and $\mathtt{R}_Y$:

$$\Sigma_{(\langle 0,2\rangle,\mathtt{matAdd})}\left(\bowtie_{(\langle 1\rangle,\langle 0\rangle,\mathtt{matMul})}\left(\mathtt{R}_X,\mathtt{R}_Y\right)\right).$$

For physical tensor relations R_X and R_Y, using the rules of Table 1, this would be compiled into:

$$\Sigma^L{}_{(\langle 0,2\rangle,\mathtt{matAdd})}\left(\textsc{Shuf}_{(\langle 0,2\rangle)}\left(\bowtie^L{}_{(\langle 1\rangle,\langle 0\rangle,\mathtt{matMul})}\left(\textsc{Bcast}\left(\mathsf{R}_X\right),\mathsf{R}_Y\right)\right)\right).$$

This is a simple, broadcast-based matrix multiply. Applying simple equivalence rules brings us to cross product-based matrix multiplication, which partitions R_X on columns, and R_Y on rows. The IA program is:

$$\Sigma^L{}_{(\langle 0,2\rangle,\mathtt{matAdd})}\left(\textsc{Shuf}_{(\langle 0,2\rangle)}\left(\bowtie^L{}_{(\langle 1\rangle,\langle 0\rangle,\mathtt{matMul})}\left(\textsc{Shuf}_{(\langle 1\rangle)}\left(\mathsf{R}_X\right),\textsc{Shuf}_{(\langle 0\rangle)}\left(\mathsf{R}_Y\right)\right)\right)\right).$$

However, more complicated schemes are possible, which are expressible in IA, but not derivable using the simple equivalence rules. For example, replication-based matrix multiplication can be viewed as a relational version of the 3D parallel matrix multiplication [61]. The algorithm first replicates matrix $\mathbf{X}$ and $\mathbf{Y}$'s blocks multiple times, viewing the result as a 3-D array, and shuffles them using the index of the corresponding voxel as a key; then each site joins the tuples with the same keys and performs local multiplications, aggregating to obtain the final results. If `xDups`

is defined as $\textsc{Front}(\mathsf{R}_Y)[1]$ and `yDups` is $\textsc{Front}(\mathsf{R}_X)[0]$, the shuffle stage can be implemented in IA as:

$$\mathsf{R}^*_X = \textsc{Shuf}_{(\langle 0,2\rangle)}\left(\lambda^L{}_{(\texttt{insertDim(2,xDups)},\texttt{duplicate(xDups)})}\left(\mathsf{R}_X\right)\right)$$
$$\mathsf{R}^*_Y = \textsc{Shuf}_{(\langle 0,2\rangle)}\left(\lambda^L{}_{(\texttt{insertDim(0,yDups)},\texttt{duplicate(yDups)})}\left(\mathsf{R}_Y\right)\right)$$

where kernel functions `insertDim` and `duplicate` add a new dimension, and duplicate each existing array the specified number of times. For example, applying $\lambda^L{}_{(\texttt{insertDim(2,xDups)},\texttt{duplicate(xDups)})}$ to the tensor relation

$$\left\{ \left(\langle 0,0\rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right), \left(\langle 0,1\rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right), \left(\langle 1,0\rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right), \left(\langle 1,1\rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right) \right\}$$

will produce:

$$\left\{ \left(\langle 0,0,0\rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right), \left(\langle 0,0,1\rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right), \left(\langle 0,1,0\rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right), \left(\langle 0,1,1\rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right), \left(\langle 1,0,0\rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right), \left(\langle 1,0,1\rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right), \left(\langle 1,1,0\rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right), \left(\langle 1,1,1\rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right) \right\}$$

Next one can execute:

$$\Sigma^L{}_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\bowtie^L{}_{(\langle 0,1,2\rangle,\langle 0,1,2\rangle,\texttt{matMul})}\left(\mathsf{R}^*_X, \mathsf{R}^*_Y\right)\right).$$

The equivalence of these three implementations is an example of a set of domain-specific equivalence rules.

3.3.3 Cost Model

One of the beneficial aspects of the TRA is that cost-based optimization is much easier than that for classical relational algebra: If the uniqueness and continuity constraints hold, tuple counts need not be estimated and can be computed exactly.

In the simple cost model presented here, one can use the number of floating point values that must be transferred between sites as the cost metric. The rationale is that for the most part, the number of floating-point math operations in a distributed ML computation is fixed. While this is not a hard and fast rule — it is possible to push the filtering of tuples in a tensor relation past the application of a kernel function, which would change the number of floating-point operations — in many applications, network transfer is the dominant cost, and is a reasonable cost metric.

To compute the network transfer cost for a plan in IA, one need to be able to compute the frontier of each physical relation R: $\mathbf{f} = \textsc{Front}(\mathsf{R})$. The reason is that, assuming that uniqueness and continuity constraints hold, one can compute the number of floating point numbers in R using $\mathbf{f}$. If R is of type $\mathcal{R}^{(k,r,\mathbf{b},s)}$, and $\mathbf{f} = \textsc{Front}(\mathsf{R})$, then the number of tuples in the corresponding tensor relation is $n = \prod_i \mathbf{f}_i$, and the number of floating point numbers in the tensor relation is $n \times \prod_i \mathbf{b}_i$.

Once the frontier is known, it is used to compute the transfer cost for each Bcast and Shuf operation. The cost to broadcast a tensor relation of type $\mathcal{R}^{(k,r,\mathbf{b},s)}$ and having f floating point numbers, is simply $f \times s$. The cost to shuffle a tensor relation of f floating point numbers is simply f.

Note that this section makes the simplifying assumption that there are no automatic,

algorithmic optimizations — optimizations are all realized via the application of the transformation rules from the last section. For example, if there are two broadcasts in a row, the cost model assumes that the second broadcast has exactly the same cost as the first — in reality, the second broadcast is superfluous and could be optimized out, but the cost model assumes that this is done via the application of transformation rules. If the optimized plan is "stupid" enough to ask for such a double-broadcast, both broadcasts are performed.

Thus, the task of costing a physical TRA plan reduces to computing the type of each intermediate relation, as well as its frontier.

Computing the type is relatively easy: one can work up from the leaves to the end result(s) of a physical plan, using the type signature of each of the physical operations (Section 4) to infer the type of each intermediate physical relation.

Computing the frontier in this way, working up from leaves to outputs, is also possible, but it requires a bit more thought. The following part considers how the frontier of an output is computed for each of the various operations in the physical algebra:

1. $\bowtie^{L}_{\texttt{(joinKeysL, joinKeysR, projOp)}}(\mathsf{R}_l, \mathsf{R}_r)$. For local join, assuming that R_l and R_r have an appropriate partitioning to sites, let $\mathbf{f}_l$ and $\mathbf{f}_r$ be the left and right input frontiers of dimensionality k_l and k_r, respectively. Then the output frontier $\mathbf{f}$ is computed as follows. For $k < k_l$ and k not in `joinKeysL`, $\mathbf{f}[k] = \mathbf{f}_l[k]$, as the frontier value for that dimension is inherited from the left. For $k < k_l$ and where $k = \texttt{joinKeysL}[i]$, $\mathbf{f}[k] = \min(\mathbf{f}_l[k], \mathbf{f}_r[i])$, as the frontier value for that dimension results from the join of the two relations. And finally, for all other k, $\mathbf{f}[k]$ is inherited from the corresponding dimension in the right frontier.

2. $\Sigma^{L}{}_{(\texttt{groupByKeys, aggOp})}(\mathsf{R})$. For local aggregation, again assuming an appropriate partitioning, let $\mathbf{f}_i$ denote the input frontier, and let n be the number of key dimensions in `groupByKeys`. In this case, for $k \leq n$, $\mathbf{f}[k] = \mathbf{f}_l[\texttt{groupByKeys}[k]]$.

3. $\sigma^{L}{}_{(\texttt{boolFunc})}(\mathsf{R})$. This performs a filter in the physical tensor relation R. For an n-dimensional input frontier $\mathbf{f}_i$, for any $k \leq n$, by definition:

$$\mathbf{f}[k] = 1 + \max\left\{\mathbf{k}[k] \text{ s. t. } \mathbf{k} < \mathbf{f}_i \text{ and } \texttt{boolFunc}(\mathbf{k}) = \text{ true}\right\}.$$

   That is, the kth dimension in the frontier is inherited from the largest key value in that dimension accepted by `boolFunc`. In many cases, especially if `boolFunc` consists of simple arithmatic expressions any comparisons, symbolic methods can be used to compute this. But in practice, it may simply be easier to use a brute-force approach, where each key value is fed into `boolFunc` to compute the required maximum. Since the size of a tensor relation is typically small — tens of thousands of tuples would be very large — this is a very practical approach.

4. $\lambda^{L}{}_{(\texttt{keyMapFunc,arrayMapFunc})}(\mathsf{R})$. Similarly, for an n-dimensional input frontier $\mathbf{f}_i$, for any $k \leq n$, by definition:

$$\mathbf{f}[k] = 1 + \max\left\{\texttt{keyMapFunc}(\mathbf{k})[k] \text{ s. t. } \mathbf{k} < \mathbf{f}_i\right\}.$$

   Again, a brute force-approach is appropriate for computing the frontier in this case.

3.4 Empirical Study

This section details the empirical evaluation aimed at answering the following key questions:

(1) Given a data set and an associated tensor computation, can the ideas in this paper be used to convert a TRA expression into an expression in the IA which is optimized for, and runs well on, that particular data set?

(2) How does the execution time of such an optimized computation compare to what one might expect for a competing, high-performance engine?

Benchmark Tasks. To answer these questions, three benchmark tasks are developed: (i) distributed matrix multiplication, (ii) distributed nearest neighbor search in a Riemannian metric space, and (iii) distributed stochastic gradient decent (SGD) in a two-layer, feed-forward neural network (FFNN).

TRA Implementation. An prototype execution engine for the IA is implemented in Python. While it may seem surprising that Python is appropriate for implementing a relational engine, for even very large ML problems, the number of tuples in a TRA computation is small; most data are stored in the large arrays. The Python execution engine makes heavy use of PyTorch to handle those arrays. PyTorch is used to actually execute the compute kernels on the various sites in a compute cluster, and the IA implementation uses PyTorch's optimized communication library to move the arrays stored in tensor relations between machines.

3.4.1 Matrix Multiplication

Multiplication of $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{K \times J}$ can be formalized:

$$\Sigma_{(\langle 0,2\rangle, \texttt{matAdd})} \left(\bowtie_{(\langle 1\rangle, \langle 0\rangle, \texttt{matMul})} \left(\mathtt{R}_{\mathbf{A}}, \mathtt{R}_{\mathbf{B}} \right) \right)$$

where matrix $\mathbf{A}$ and $\mathbf{B}$ are stored in tensor relations $\mathtt{R}_{\mathbf{A}}$ and $\mathtt{R}_{\mathbf{B}}$.

To test the effectiveness of IA optimization, as others have done [62, 63], the benchmark considers three different multiplications: (i) general matrices ($I = K =$

$J = 4\times10^4$), (ii) matrices with a common large dimension ($K = 6.4\times10^5$, $I = J = 10^4$), and (iii) matrices with two large dimensions ($I = J = 8 \times 10^4$, $K = 10^4$). Matrices are filled with random data following uniform distribution $\mathcal{U}(-1, 1)$.

As discussed, the above TRA as three equivalent IA plans: broadcast based matrix multiplication (BMM), cross-product based matrix multiplication (CMM), and replication-based matrix multiplication (RMM). These three IA implementations are compared with Intel's version of ScaLAPACK [64] which realizes the classic SUMMA [65] algorithm and is generally considered to be one of the fastest distributed matrix multiplications available from the high performance computing community. All methods are benchmarked over Amazon EC2 clusters with 5, 10 or 15 `r5d.2xlarge` instances (each with 8 vCPU, 64 GB RAM, and connected by up to 10 Gb/s interconnect.) Note that reasonable amount of effort have been made to tune the hyper-parameters of ScaLAPACK (e.g., grid size, thread number) and report the best results. Since ScaLAPACK uses an optimized initial layout, the similar optimal layouts for the IA implementations are initialized as below: BMM has both R_A and R_B partitioned by dimension 0; CMM has R_A partitioned by dimension 1 and R_B partitioned by dimension 0; and RMM has both R_A and R_B partitioned by dimension 0.

The results are listed in Table 3.2. In Table 3.3, the cost (as computed in 4.3) predicted in a 10-node cluster for different matrix multiplication algorithms and data sets are reported.

3.4.2 Nearest Neighbor Search

TRA is used to implement a nearest neighbor search problem in a Riemannian metric space encoded by matrix $\mathbf{A} \in \mathbb{R}^{D\times D}$, where given a query vector $\mathbf{x}_q \in \mathbb{R}^{1\times D}$ and a

candidate set $\mathbf{X} \in \mathbb{R}^{N \times D}$, the goal is to find the i-th row in the matrix that minimizes: $d_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}_q) = (\mathbf{x}_i - \mathbf{x}_q)\,\mathbf{A}\,(\mathbf{x}_i - \mathbf{x}_q)^T$. Suppose $\mathbf{x}_q$, $\mathbf{X}$, $\mathbf{A}$ are stored in tensor relation $\mathtt{R}_{\mathbf{x}_q}$, $\mathtt{R}_{\mathbf{X}}$ and $\mathtt{R}_{\mathbf{A}}$, the corresponding TRA program can be encoded as:

$$
\begin{aligned}
\mathtt{R}_{\mathbf{diff}} &= \bowtie_{(\langle 1\rangle, \langle 1\rangle, \mathtt{matVecSub})} \left(\mathtt{R}_{\mathbf{x}_q}, \mathtt{R}_{\mathbf{X}}\right) \\
\mathtt{R}_{\mathbf{proj}} &= \Sigma_{(\langle 0,2\rangle, \mathtt{matAdd})} \left(\bowtie_{(\langle 1\rangle, \langle 0\rangle, \mathtt{matMul})} \left(\mathtt{R}_{\mathbf{diff}}, \mathtt{R}_{\mathbf{A}}\right)\right) \\
\mathtt{R}_{\mathbf{dist}} &= \lambda_{(\mathtt{rowSum})} \left(\Sigma_{(\langle 0\rangle, \mathtt{matAdd})} \left(\bowtie_{(\langle 0,1\rangle, \langle 0,1\rangle, \mathtt{elemMul})} \left(\mathtt{R}_{\mathbf{proj}}, \mathtt{R}_{\mathbf{diff}}\right)\right)\right) \\
\mathtt{R}_{\mathbf{min}} &= \Sigma_{(\langle\rangle, \mathtt{minIndex})} \left(\mathtt{R}_{\mathbf{dist}}\right).
\end{aligned}
$$

where `matVecSub` is matrix-vector subtraction, `elemMul` is element-wise matrix multiplication (Hadamard product), `minIndex` minindex finds the minimal element's index.

This TRA program is hand-compiled into an expression in the IA, and then use the various equivalence rules to produce two different implementations: `Opt4Horizontal`, and `Opt4Vertical`. `Opt4Horizontal` will broadcast $\mathtt{R}_{\mathbf{x}_q}$ and $\mathtt{R}_{\mathbf{A}}$ to each compute site and partition $\mathtt{R}_{\mathbf{X}}$ by dimension 0; then the computation of $\mathtt{R}_{\mathbf{diff}}$, $\mathtt{R}_{\mathbf{proj}}$, and $\mathtt{R}_{\mathbf{dist}}$ will be conducted by local operations. `Opt4Vertical` will first broadcast $\mathtt{R}_{\mathbf{x}_q}$ to each site and compute $\mathtt{R}_{\mathbf{diff}}$, then partition $\mathtt{R}_{\mathbf{diff}}$ by dimension 1 and partition $\mathtt{R}_{\mathbf{A}}$ by dimension 0 so that $\mathtt{R}_{\mathbf{proj}}$ is computed in a cross-product based matrix multiplication.

For the the `Opt4Horizontal` IA implementation, $\mathtt{R}_{\mathbf{x}_q}$, $\mathtt{R}_{\mathbf{X}}$ and $\mathtt{R}_{\mathbf{A}}$ are initially partitioned by dimension 0. For the the `Opt4Vertical` IA implementation, $\mathtt{R}_{\mathbf{x}_q}$, and $\mathtt{R}_{\mathbf{A}}$ are initially partitioned by dimension 0, while $\mathtt{R}_{\mathbf{X}}$ is initially partitioned by 1.

Two datasets are generated: 1) `Large`, with a large number of data points ($N = 1.5 \times 10^6$) but small feature space ($D = 6 \times 10^3$); and 2) `Wide`, with a small number of data points ($N = 6 \times 10^3$), with a large feature space ($D = 10^5$). This computation is executed on compute clusters with 4, 8 or 12 `r5d.2xlarge` instances.

Surprisingly, it is quite difficult to produce a distributed implementation on a

competing system — the obvious distributed alternatives were not able to run this computation. Thus, as a baseline, the distributed IA implementations are compared by the execution time with a PyTorch implementation that runs on a single site equipped with the same level of computing power: an `r5d.8xlarge` instance (with 32 vCPU, 256 GB RAM), an `r5d.16xlarge` instance (with 64 vCPU, 512 GB RAM) and an `r5d.24xlarge` instance (with 96 vCPU, 768 GB RAM). Since the single-site implementation has zero communication overhead, this should be something of a lower-bound on the time required to run the computation. The results are enumerated in Table 3.4. Again, the predicted costs are given in Table 3.5.

3.4.3 Feed-Forward Neural Network

The last benchmark is a training iteration of a two-layer FFNN for multiple label classification. Suppose a mini-Batch $(\mathbf{X}, \mathbf{Y})$ includes N data samples, where $\mathbf{X} \in \mathbb{R}^{N \times D}$ is a matrix storing the input features, $\mathbf{Y} \in \mathbb{R}^{N \times L}$ is the matrix of one-hot encoded labels. Let $\mathbf{W}_1^{(i)} \in \mathbb{R}^{D \times H}$ and $\mathbf{W}_2^{(i)} \in \mathbb{R}^{H \times L}$ be the weight matrix for layer 1 and layer 2 for iteration i and η be the learning rate. The TRA expression for feed-forward neural network learning (backpropagation) is listed below:

$$\mathtt{R}_{\mathbf{a}_1} = \lambda_{(\mathtt{relu})}\left(\Sigma_{(\langle 0,2\rangle,\mathtt{matAdd})}\left(\bowtie_{(\langle 1\rangle,\langle 0\rangle,\mathtt{matMul})}\left(\mathtt{R}_{\mathbf{X}}, \mathtt{R}_{\mathbf{W}_1^{(i)}}\right)\right)\right);$$

$$\mathtt{R}_{\mathbf{a}_2} = \lambda_{(\mathtt{sigmoid})}\left(\Sigma_{(\langle 0,2\rangle,\mathtt{matAdd})}\left(\bowtie_{(\langle 1\rangle,\langle 0\rangle,\mathtt{matMul})}\left(\mathtt{R}_{\mathbf{a}_2}, \mathtt{R}_{\mathbf{W}_2^{(i)}}\right)\right)\right);$$

$$\mathtt{R}_{\nabla_{\mathbf{a}_2}} = \bowtie_{(\langle 0,1\rangle,\langle 0,1\rangle,\mathtt{matSub})}\left(\mathtt{R}_{\mathbf{a}_2}, \mathtt{R}_{\mathbf{Y}}\right);$$

$$\mathtt{R}_{\nabla_{\mathbf{W}_2}^{(i)}} = \Sigma_{(\langle 0,2\rangle,\mathtt{matAdd})}\left(\bowtie_{\left(\langle 0\rangle,\langle 0\rangle,\mathtt{matTranMul}^L\right)}\left(\mathtt{R}_{\mathbf{a}_1}, \mathtt{R}_{\nabla_{\mathbf{a}_2}}\right)\right);$$

$$\mathtt{R}_{\nabla_{\mathbf{a}_1}-1} = \Sigma_{(\langle 0,2\rangle,\mathtt{matAdd})}\left(\bowtie_{\left(\langle 1\rangle,\langle 1\rangle,\mathtt{matTranMul}^R\right)}\left(\mathtt{R}_{\nabla_{\mathbf{a}_2}}, \mathtt{R}_{\mathbf{W}_2^{(i)}}\right)\right);$$

$$\mathtt{R}_{\nabla_{\mathbf{a}_1}-2} = \bowtie_{(\langle 0,1\rangle,\langle 0,1\rangle,\mathtt{elemMul})}\left(\lambda_{(\mathtt{relu'})}\left(\mathtt{R}_{\mathbf{a}_1}\right), \mathtt{R}_{\nabla_{\mathbf{a}_1}-1}\right);$$

$$\mathtt{R}_{\nabla_{\mathbf{W}_1}^{(i)}} = \Sigma_{(\langle 0,2\rangle,\mathtt{matAdd})}\left(\bowtie_{\left(\langle 0\rangle,\langle 0\rangle,\mathtt{matTranMul}^L\right)}\left(\mathtt{R}_{\mathbf{X}}, \mathtt{R}_{\nabla_{\mathbf{a}_1}-2}\right)\right);$$

$$\mathtt{R}_{\mathbf{W}_2^{(i+1)}} = \bowtie_{(\langle 0,1\rangle,\langle 0,1\rangle,\mathtt{matSub})}\left(\mathtt{R}_{\mathbf{W}_2^{(i)}}, \lambda_{\left(\mathtt{scaleMul}_{(\eta)}\right)}\left(\mathtt{R}_{\nabla_{\mathbf{W}_2}^{(i)}}\right)\right);$$

$$\mathtt{R}_{\mathbf{W}_1^{(i+1)}} = \bowtie_{(\langle 0,1\rangle,\langle 0,1\rangle,\mathtt{matSub})}\left(\mathtt{R}_{\mathbf{W}_1^{(i)}}, \lambda_{\left(\mathtt{scaleMul}_{(\eta)}\right)}\left(\mathtt{R}_{\nabla_{\mathbf{W}_1}^{(i)}}\right)\right).$$

Here, `matTranMul` represents transposed matrix multiplication and $\mathtt{scaleMul}_{(\eta)}$ represents scalar multiplication by η.

Again, this TRA program is compiled by hand into the IA, and use the equivalence rules to produce two implementations. The first, called `TRA-DP`, resembles the classic data parallel implementation. The weights $\mathtt{R}_{\mathbf{W}_1^{(i)}}$ and $\mathtt{R}_{\mathbf{W}_2^{(i)}}$ (initially partitioned by dimension 0) are broadcast to each site. $\mathtt{R}_{\mathbf{X}}$ and $\mathtt{R}_{\mathbf{Y}}$ are initially partitioned using dimension 0. The computation of activation and gradient will be conducted by local operations — until the last step that shuffles the weight's gradients stored in $\mathtt{R}_{\nabla_{\mathbf{W}_1}^{(i)}}$ and $\mathtt{R}_{\nabla_{\mathbf{W}_2}^{(i)}}$ for the weight updates. The second, called `TRA-MP`, corresponds to an intra-operation model parallel plan that initially pre-partitions $\mathtt{R}_{\mathbf{W}_1^{(i)}}$ along dimension 0 and $\mathtt{R}_{\mathbf{W}_1^{(i)}}$ along dimension 1, while the batches stored in $\mathtt{R}_{\mathbf{X}}$ and $\mathtt{R}_{\mathbf{Y}}$ are initially partitioned across dimension 1. In the forward propagation, $\mathtt{R}_{\mathbf{a}_1}$ is computed by a cross-product-based matrix multiplication followed by a local map to implement the

`relu` function. $\mathtt{R}_{\mathbf{a}_2}$ is computed by a broadcast-based matrix multiplication. Note that the generated $\mathtt{R}_{\mathbf{a}_2}$ is partitioned by dimension 1; in the backward propagation, $\mathtt{R}_{\nabla_{\mathbf{a}_2}}$ is computed by a local join; the computation of $\mathtt{R}_{\nabla^{(i)}_{\mathbf{W}_2}}$ is realized by broadcast-based matrix multiplication with zero communication cost since $\mathtt{R}_{\mathbf{a}_1}$ is already broadcast in the forward propagation; the computation of $\nabla_{\mathbf{a}_1}$ is split to two steps, $\mathtt{R}_{\nabla_{\mathbf{a}_1}-1}$ is computed by a cross-product-based matrix multiplication, and $\mathtt{R}_{\nabla_{\mathbf{a}_1}-2}$ via a local join; the computation of $\mathtt{R}_{\nabla^{(i)}_{\mathbf{W}_1}}$ is performed by broadcast-based matrix multiplication (broadcasting $\mathtt{R}_{\nabla_{\mathbf{a}_1}-2}$); finally, the updated weights stored in $\mathtt{R}_{\mathbf{W}_1^{(i+1)}}$ and $\mathtt{R}_{\mathbf{W}_2^{(i+1)}}$ are computed by local joins.

These two IA plans are compared with the state-of-the-art data paralleled implementation provided by PyTorch [66].

Two data sets are considered. First, the data from the Google speech recognition task [49], where a 1600 feature vector is extracted from audio wave-forms; the goal is to identify 10 keywords ($D = 1600$ and $L = 10$); for this task, a very wide hidden layer with large number of neurons is trained, where $H = 1 \times 10^5$, 1.5×10^5, or 2×10^5; a batch size of 10^4 ($N = 10^4$) are used for min-batch SGD. Second, the benchmark considers the AmazonCat-14K [67, 68] task, which is an "extreme classification" dataset. It includes a large number of features ($D = 597540$) and labels ($L = 14588$); in this case, a relatively narrow network with $H = 1 \times 10^3$, 3×10^3, 5×10^3, or 7×10^3 is trained; a batch size of 10^3 ($N = 10^3$) are used for mini-batch SGD.

Each is executed on CPU clusters with 2, 5 or 10 `r5dn.2xlarge` instances connected by up to 25 Gb/s interconnect) and GPU clusters with 2, 5 or 10 `p3.2xlarge` instances (each with a Nvidia Tesla V100 GPU, and connected by 10 Gb/s interconnect).

The results for Google are listed in Table 3.6; for Amazon-14k in Table 3.7. Predicted costs are given in Table 3.8 (Speech-Xk is the Google Speech data set,

XML-Xk is the extreme classification problem, with a hidden layer size of $X \times 10^3$).

3.4.4 Discussion

The TRA enabled very fast computations. For matrix multiplication, for each of the three types of multiplications, one of the TRA implementations was nearly as fast, or significantly faster than Intel's ScaLAPACK. The TRA CMM was twice as fast as ScaLAPACK for larger clusters, for matrices multiplied along one common large dimension. ScaLAPACK is a carefully-implemented, distributed matrix multiply library, and beating ScaLAPACK is significant.

For the nearest neighbor computation, the better TRA implementation on multiple machines nearly matched the speed of PyTorch on one machine, in each case. For example, on 12 machines `Opt4Vertical` was about 6% slower than PyTorch on a single machine for the `Large` data set, and on the `Wide` data set, `Opt4Horizontal` was about 6.6% slower than PyTorch on a single machine. Thus, the TRA implementation was able to nearly zero out the cost of moving from a single, parallel machine to a computation on 12 different machines. That is, the set-based abstraction only results in a 6% overhead when moving from a tightly-coupled, parallel machine to a distributed environment with a slow interconnect.

The TRA implementation also matched or outperformed PyTorch on the FFNN computation. For Google speech, the TRA-DP (data parallel) IA was able to closely match PyTorch's speed — despite the fact that *naturally data parallel computations like this are the computations PyTorch is built to run.* Further, while PyTorch failed on the larger Google computations, the TRA implementation was able to run to completion. On the even larger, extreme classification problem, the TRA-MP (model parallel) IA was much, much faster than PyTorch, and much more scalable. PyTorch

simply cannot handle the huge matrices required to power this computation. It is reasonable to believe that these results do show that the TRA can be the basis for a very high-performance distributed ML systems.

Note that in each case, there was one IA implementation that was suitable for the input data, and one that was not; the difference between the two was often significant. In a system based upon the TRA, it would be crucial to automatically choose the suitable implementation. It is interesting to highlight that in each case, the simple cost model from Section 4.3 would have chosen the correct implementation. For example, consider Table 3.8. In each case, the cost metric correctly assigns the lower cost to the appropriate IA computation: TRA-DP for the smaller, Google problem, and TRA-MP for the larger, extreme classification problem. These results suggest that it should easily be possible to perform cost-based optimization over the IA.

Finally, all implementations failed on the large FFNN extreme classification problems, for the smaller GPU cluster sizes. The reason for this is clear: the data required by the computation on each site could not fit on a GPU and the simple, Python-based TRA implementation lacked a proper memory management system. Using PyTorch to perform the GPU computations requires moving tensors onto the GPU, with no easy way to move them off to make way for other computations, aside from implementing some sort of buffering scheme. Lacking this, the system would fail. This is a crucial area for future work: designing a proper buffering scheme, that is able to manage data at many levels: secondary storage, RAM, and GPU RAM.

3.5 Summary

This chapter introduces novel abstractions necessary for building distributed machine learning systems, which includes a tensor relational algebra expressive enough to en-

code any tensor operation in the form of the Einstein notation, and an implementation algebra targeting at effective optimizations for paralleled and distributed environment. Empirical study shows that IA optimization can enable efficient, distributed implementations of several ML computations, which can reach or even significantly outperform the existing HPC and ML systems. I believe that such computational and implementation abstractions will lead to a flexible declarative ML system design especially for a distributed runtime.

Cluster Size	5	10	15
`Two General Matrices`			
BMM	61.18s	46.48s	38.54s
CMM	63.14s	40.08s	29.38s
RMM	60.71s	43.56s	44.55s
ScaLAPCK	66.11s	37.05s	28.30s
`Two Matrices with a Common Large Dim`			
BMM	106.24s	104.67s	101.63s
CMM	51.52s	30.58s	23.09s
RMM	91.19s	74.40s	68.43s
ScaLAPCK	83.96s	58.17s	35.45s
`Two Matrices with Two Large Dims`			
BMM	57.23s	37.60s	31.64s
CMM	106.82s	82.72s	75.63s
RMM	59.91s	41.12s	33.26s
ScaLAPCK	53.06s	28.13s	22.34s

Table 3.2 : Distributed matrix multiply runtimes.

	BMM	CMM	RMM
`Two General`	1.6×10^{10}	1.6×10^{10}	1.6×10^{10}
`A Common Large Dim`	6.4×10^{10}	1.0×10^{9}	6.4×10^{10}
`Two Large Dims`	8.0×10^{9}	6.4×10^{10}	8.0×10^{9}

Table 3.3 : Predicted costs for MM in a 10-node cluster.

Cluster Size	4	8	12
`Wide data set`			
`Opt4Horizontal`	55.62s	39.24s	24.63s
`Opt4Vertical`	120.09s	112.69s	108.59s
Single big machine	51.54s	35.82s	23.01s
`Large data set`			
`Opt4Horizontal`	159.69s	229.40s	315.17s
`Opt4Vertical`	57.81s	35.61s	26.43s
Single big machine	48.22s	31.12s	24.88s

Table 3.4 : Nearest neighbor search runtimes.

	`Opt4Horizontal`	`Opt4Vertical`
`Wide data set`	2.9×10^8	8.0×10^{10}
`Large data set`	7.2×10^{10}	4.8×10^9

Table 3.5 : Predicted nearest neighbor search costs, 8 machines.

Cluster	CPU			GPU		
Nodes	2	5	10	2	5	10
`100k Neurons`						
PyTorch-DP	11.16s	6.15s	4.75s	0.99s	1.19s	1.27s
TRA-DP	11.62s	6.51s	5.20s	1.49s	1.59s	1.63s
TRA-MP	26.56s	28.71s	29.09s	7.01s	11.56s	fail
`150k Neurons`						
PyTorch-DP	14.28s	9.46s	6.54s	1.18s	1.65s	1.78s
TRA-DP	14.52s	9.68s	7.56s	2.15s	2.22s	2.23s
TRA-MP	33.20s	42.80s	43.10s	fail	fail	fail
`200k Neurons`						
PyTorch-DP	17.25s	11.94s	9.30s	0.99s	1.19s	1.27s
TRA-DP	17.89s	12.51s	9.67s	1.49s	1.59s	1.63s
TRA-MP	37.82s	54.23s	59.84s	fail	fail	fail

Table 3.6 : SGD runtime of 2 layer FFNN for Google Speech Recognition.

Cluster	CPU			GPU		
Nodes	2	5	10	2	5	10
`1k Neurons`						
PyTorch-DP	9.74s	10.29s	10.34s	2.67s	3.76s	4.20s
TRA-DP	12.50s	14.29s	15.68s	4.67s	4.69s	4.73s
TRA-MP	3.86s	2.79s	1.70s	0.40s	0.37s	0.35s
`3k Neurons`						
PyTorch-DP	25.46s	29.04s	30.51s	fail	fail	fail
TRA-DP	26.59s	38.15s	46.06s	fail	12.74s	13.13s
TRA-MP	10.57s	6.36s	3.88s	fail	0.54s	0.44s
`5k Neurons`						
PyTorch-DP	34.05s	46.53s	50.17s	fail	fail	fail
TRA-DP	44.12s	68.54s	75.15s	fail	fail	fail
TRA-MP	18.59s	8.07s	5.75s	fail	0.59s	0.48s
`7k Neurons`						
PyTorch-DP	fail	fail	fail	fail	fail	fail
TRA-DP	60.28s	89.36s	107.86s	fail	fail	fail
TRA-MP	21.35s	12.12s	7.854s	fail	fail	0.73s

Table 3.7 : SGD runtime of 2 layer FFNN for Amazon-14k extreme classification.

	TRA-DP	TRA-MP
Speech-100k	9.7×10^8	1.0×10^{10}
Speech-150k	1.5×10^9	1.5×10^{10}
Speech-200k	1.9×10^9	2.0×10^{10}
XML-1k	3.7×10^9	1.0×10^7
XML-3k	1.1×10^{10}	3.0×10^7
XML-5k	1.8×10^{10}	5.0×10^7
XML-7k	2.6×10^{10}	7.0×10^7

Table 3.8 : Predicted costs for FFNN in a 5-node cluster

Chapter 4

Related Works

The advance of ML, especially deep learning, has led to successes in various applications such as controlling self-driving cars [69], recognizing speech [70], or predicting consumer behavior [71]. This success is usually achieved by designing complex models with large numbers of parameters trained over substantial amounts of training data. The demand for learning such complex models over large training data sets raises the need for parallel/distributed ML, where the computation is allocated to multiple cores/machines in order to learn faster than on a single core or machine. Parallelizing or distributing ML computations is much more difficult than distributing traditional big data workflows, since the stochastic gradient decent (SGD) computation that typically powers modern ML is inherently sequential.

In this chapter, I discuss the state of the art techniques for the efficient parallelization of the training process.* The solutions considered include algorithmic designs that allow efficient distribution of the computation and optimization of the communication patterns, along with the development of ML systems to support these algorithms.

*Note that there are various types of ML computations that can be distributed/parallelized; this chapter considers the application of SGD to minimization of a loss function, typically over a neural network.

4.1 Distributed Learning Algorithms

Before the discussion of the distributed algorithmic approaches, it is worthwhile to first review SGD-based optimization, which is used to minimize a loss function by repeatedly adjusting the model's parameters in the direction of the negative gradient. The gradient descent is called "stochastic", as the gradient is calculated from a randomly sampled subset of the training data. The optimization procedure can be summarized as:

1. Fetch a batch of randomly sampled training data.
2. Compute the loss function based on the model's output and the label, known as the forward propagation.
3. Compute the gradient with respect to the model and the loss function, known as the backward propagation.
4. Update the model along the direction of the negative gradient, where the magnitude of the update is controlled by a learning rate.
5. Repeat this process until satisfied with the value of the loss function.

4.1.1 Data Parallelism vs. Model Parallelism

In order to distribute the computation in SGD, two fundamental approaches are proposed, known as **data parallelism** and **model parallelism**[†].

In the data parallel approach, e.g., [73, 74], the training data are split evenly and stored distributively in each worker node in the cluster. In order to conduct the

[†]It is crucial to note that the two types of parallelism can also be complementary [72].

SGD algorithm, each worker first fetches the current model (usually from a parameter server [10]) and then executes the forward and backward propagation using to its local training data. Finally, the gradients from the workers will be aggregated to update the current model. The technique can be leveraged for any ML algorithm with an independent and identical distribution (i.i.d.) assumption over the data samples. This assumption is valid for most widely applied ML models, from simple linear models to complex deep neural networks.

By contrast, the model parallel approach, e.g., [75], partitions the model and assigns each fragment to the worker nodes in the cluster. For example, to train deep neural networks, the model parallel approach generally assigns different layers to different worker nodes so that in the forward propagation, activations are communicated while in the backward propagation, the gradients of the activations are transferred. Different from data parallelism, model parallelism cannot automatically be applied to any machine learning model, because there can be no effective partitioning of the model parameters in some cases.

Generally, data parallelism is more popular for distributed ML due to its relative simplicity and wide applicability. However, model parallelism has its own advantages: complex models with large amount of parameters may not fit the RAM of a single compute unit or worker node for data parallelism — in this case, there is no enough computational power to execute the forward/backward propagation locally.

4.1.2 Synchronous Mode vs. Asynchronous Mode

In data parallelism, there are different ways of combining gradients. In this way, different data parallel approaches can be categorized into two classes: **synchronous data parallelism** and **asynchronous data parallelism**.

In the fully synchronous mode (also known as bulk synchronous parallel), e.g., [76], the gradients from all worker nodes are averaged at every SGD iteration. This simplest mode is directly inherited from the map-reduce programming model [77, 78], in which programs ensure consistency by synchronizing between each computation and communication phase. In other words, the computation is completely deterministic, and is guaranteed to perform the same computation as a classical, centralized SGD. Due to its simplicity, this is usually the default option in modern distributed ML systems. However, a disadvantage is that there is a synchronization barrier at every iteration, where quick workers have to wait until all other workers are finished, which results in the "straggler problem" — every transient slowdown of any given process in the distributed cluster can delay all other processes so that each iteration proceeds at the pace of the slowest process [79].

By contrast, the fully asynchronous mode, e.g., [80, 81, 82], lets workers communicate without any synchronization barrier. Obviously, this strategy obtains the highest possible speedup with respect to the parallel computational efficiency. However, the asynchronicity can lead to statistical inefficiency — there is no guarantee that each worker computes the gradients with respect to the most recent model parameters, which is known as "staleness." More concretely, given a collection of workers in a cluster, each of which makes additive updates to a shared parameter (e.g., $\mathbf{x} \leftarrow \mathbf{x} + \delta$) at regular intervals called clocks.[‡] Since updates may not be immediately visible to other workers trying to read the parameter (e.g, $\mathbf{x}$), the workers only see parameters from a "stale" subset of updates. The staleness can be defined as the difference of clocks between the fastest and slowest workers in a cluster [73] [§]. Staleness can

[‡]Clocks represent the unit of progress by an ML algorithm owned by each worker.

[§]Note that different definition of staleness does exist, for example, [83] defines staleness for

introduce additional noise and make the training unstable, this causes the statistical inefficiency [83]. To resolve this issue, staleness-aware asynchronous SGD [84] is proposed to categorize the gradients by their staleness. By restricting the impact of the stale gradients, this approach is able to obtain convergence similar to synchronous approaches.

4.1.3 Algorithmic Optimization for Distributed ML Communication

An essential goal for distributed ML is to minimize the communication cost, since the communication between workers in a cluster can be a bottleneck to limit the system's scalability [85]. At the algorithmic level, this goal can be achieved by reducing the communication frequency or reducing the amount of traffic for each synchronization.

A straightforward way to reduce the communication frequency is to adopt a large batch so that the per-worker workload can be fully leveraged. However, some tricks are necessary to make this scheme work. For example, the Facebook team increases the learning rate proportional to the batch size, and use special learning rate with "warm-up" policy to overcome initial optimization difficulty, which makes it possible to train ImageNet parallelly within one hour [17]. Further, layer-wise adaptive rate scaling (LARS) is proposed to increase batch size more aggressively [19] — LARS uses different learning rates for different layers based on the norm of the weights and the norm of the gradients since they observe that the ratio of weights and gradients varies significantly for different layers. However, there are also some disadvantages w.r.t large batch SGD. For example, larger batch tends to cause a degradation in the quality of the model measured by its ability to generalize. [86] presents numerical

---

each worker as the difference of clocks that have occurred between when the worker conduct the corresponding read and update operations for a SGD iteration.

evidence to show that large batch methods tend to converge to sharp minimizers of the training and testing functions — as is well known, sharp minima lead to poorer generalization.

Local SGD [27, 87] is a more effective method to reduce the communication frequency, where each worker in the cluster runs multiple iterations of forward and backward propagation before synchronization; the algorithm then aggregates all the local models into a coherent global model. Theoretical analysis and empirical evaluation suggest that local SGD is more efficient than large batch distributed SGD.

In order to reduce the synchronization traffic, compression-based methods are proposed to compress the gradients or model parameters that need to be transmitted among instances in a cluster. These compression techniques include quantization [56, 88] and sparsification [89, 90, 91].

Quantization-based methods propose a compression scheme that a small number of bits (possibly as small as a single bit [88]) to represent a floating point number, which is classically represented by 4 or 8 bytes. The idea of quantization compression for gradient comes naturally from the idea of running of deep learning algorithms with low precision [92]. This research revealed that low precision gradients can still guarantee the convergence of training, and, as a result, quantized communication is feasible for distributed ML.

Sparsification-based methods aim to reduce the number of elements that are communicated in each synchronization. One key observation that motivates sparsification compression is that in the update phase of SGD, only "significant" gradients must be used to update the model parameter to guarantee the convergence of SGD [91]. In order to induce sparsity, a natural approach is to round small coordinates of a gradient to zero [93]. More advanced strategies to introduce sparsity include random

sparsification [90], where entries to communicate and update are selected randomly; and deterministic sparsification according to fixed threshold [94] (by selecting elements with absolute values greater than a pre-defined threshold) or adaptive threshold [89, 95] (by selecting top-k percent of elements).

Note that in practice, all the above approaches to reduce the communication can be combined, e.g., [96].

4.2 Distributed Learning Systems

The advance of ML systems has revolutionized the practice of ML, especially deep learning. In this section, I begin by introducing some popular ML systems for deep learning, and then discuss some systematic design to support data parallel ML communications and model parallel integration adopted in such systems. With the demand of processing ML applications, popular general purpose big data systems are modified to support distributed ML — such extensions are also presented in this section. Last but not least, I cover some recent attempts to apply database design principles for distributed ML.

4.2.1 Specialized ML Systems

As a result of the rising popularity of deep learning in many applications, specialized ML frameworks have been developed, among which TensorFlow [2], PyTorch [3], and MXNet [97] are the top-3 most widely adopted systems according to the survey of ML frameworks and libraries [98].

Generally speaking, these ML frameworks consist of three main components:

- A group of declarative APIs for the user to specify the model by defining the forward propagation;

- An automatic differentiation engine to generate the corresponding backward propagation and to construct the computation graph of a SGD iteration;
- A runtime engine to map the computation graph to the computational units and execute the graph locally or distributively.

I personally believe that these systems deserve credit for the advance of deep learning in both industry and academia, since the engineering effort to implement gradient decent for learning ML models has been greatly reduce through their use. In fact, difficult gradient computations that would have been impossible to get right "by hand" are generated quickly by these systems without any programmer involvement. As a result, data scientists and researchers can fully devote their effort in exploring novel neural network architectures, and need to spend relatively little time implementing them.

On the other hand, the distributed ML runtime engines are perhaps not as well-designed and flexible as the first two system components (the API and auto-differentiation engine). Nevertheless, there has been a lot of effort aimed at designing better distributed ML runtime engines.

4.2.2 System Designs for Data Parallel Communication

As the discussion in section 4.1.1 illustrates, data parallelism is more widely adopted than model parallelism, due to its simplicity and applicability. Multiple systems have been proposed for supporting data parallel computations.

The parameter server (PS) [10] is a popular architecture designed for data parallel training of ML models. A PS framework consists of two components: a PS (or key-value store) that is responsible for saving the latest global model and collecting

the updating information from the workers; and a set of workers who pull the latest global model from the PS, and then compute the updates, which will be transmitted to the PS. Traditional PS architecture is generally referred as a centralized framework, where the PS node becomes the bottleneck due to the communication congestion. To alleviate the congestion, distributed PS framework has been proposed, e.g., each node in the cluster runs both a PS process and a worker process. This strategy is adopted in the release of TensorFlow [2].

A fundamentally different method to implement gradient aggregation is the all-reduce communication protocol [99]. In this framework, all nodes in the cluster communicate without a central server. After the communication, every node acquires all gradients from all the other nodes and then updates the local model, which makes all the local models consistent. Among the all-reduce implementations, ring based all-reduce has been shown to be efficient for distributed ML, where the framework structures the cluster of nodes as a ring[¶] and cascades the reduction operation. According to the study of collective reduction operations [100], the ring layout is able to utilize all bandwidth optimally in general cases, where the number of nodes in a cluster is non-power-of-two[‖]. Horovod [54] is a well-known system that modifies TensorFlow by leveraging ring-based all-reduce framework to replace the parameter server for model synchronization. PyTorch [66] also utilizes the Nvidia Collective Communications Library (NCCL) [26] — Nvidia's release of the ring-based all-reduce implementation, to realize data parallelism.

[¶]Each node has and only has two neighbors in the ring.

[‖]Tree based layout can be the optimal choice when the number of nodes is power-of-two in the cluster [100].

4.2.3 System Designs for Model Parallel Integration

Comparing to data parallelism, model parallelism is less frequently used to accelerate deep neural network training usually due to two main limitations: i) the burden of partitioning a model across multiple nodes is left to the programmer; and ii) the computation units stall when communicating activations/gradients. There are some recent effort attempting to address these issues:

An extension to support model parallel over TensorFlow is Mesh-TensorFlow [101]. Mesh-TensorFlow is an abstraction for specifying a general class of distributed tensor computations. Based on this abstraction, data parallelism can be viewed as splitting tensors and corresponding operations along the batch dimension; further, users can also specify any tensor-dimensions to be split across any dimensions of a multi-dimensional mesh of processors and operations are then split accordingly. The idea of splitting tensors is similar to TRA introduced in Chapter 3 — this abstraction can provide convenience for organizing tensor computations without staleness. Nevertheless, the user has to manually layout the computations for model parallelism.

DeepMind also releases their distributed ML framework on top of TensorFlow called TF-Replicator [102], which includes an interface to declare model parallel computations. The notion of replica (a computation designed to be run in a device, e.g., one iteration of SGD) is extended to support computation graphs that span multiple devices. An API is provided to manage the mapping of replica partitions with device IDs automatically.

A system called PipeDream [103] is proposed to overcome both two challenges: the system versions model parameters for numerically correct gradient computations, and schedules forward and backward passes of different batches concurrently on different workers in order to construct a very efficient pipeline for model parallelism; the system

also introduces an algorithm to automatically partition neural network layers among workers to balance workload and minimize communication.

It is also worthwhile to mention Microsoft's attempt to aggressively reduce residual memory consumptions for model parallelism in their system called ZeRO [104]. ZeRO adopts a similar tensor partition schema as Mesh-TensorFlow. It removes the memory redundancies in model parallelism by partitioning the activations across compute devices, and uses all-gather to reconstruct these activations on demand — in this way, the activation memory footprint is reduced proportional to the parallel degree.

4.2.4 Extension of General-Purpose Big Data Systems to ML

To process large volumes of data in parallel in a cluster is not a unique problem restricted to ML — this has been studied for a decades in distributed systems.

Classic data-flow systems using the map-reduce programming model [105] have been modified to support distributed ML, e.g., Spark [106] provides various extensions for ML [107, 108, 109]. I list a few extensions of well-known map-reduce systems to support ML now:

Spark MLlib [110] contains optimized libraries for general ML in Spark. Spark MLlib contains various ML models that are convenient to use in development and production. Three families of functions are included by Spark MLlib: ML algorithms for analyzing data; feature transformers for manipulating individual features; functions for manipulating Spark DataFrames.

Deeplearning4j [107] is a distributed deep learning library implemented in Java. The system aims to provide an industrial-strength Java development ecosystems for ML. The Deeplearning4j framework provides built-in GPU support, 3rd party model import, and native libraries support for quick matrix data processing on the CPU and

GPU.

FlinkML [111] is a part of Apache Flink, a distributed stream and batch data processing system. FlinkML provides a set of scalable ML models and API adopted to Flink distributed framework. Since Flink is focused on data processing with very low latency and high fault tolerance on distributed systems, FlinkML is capable of distributed learning for real time streaming data.

4.2.5 Databases for Distributed ML

Distributed computations in relational database management systems (RDBMSs) have been studied for more than thirty years, and are fast and robust. The query optimizer, shipped with an RDBMS, is highly effective for optimizing distributed computations. Plenty of research has focused on supporting numerical computations and ML in database systems.

Declarative and efficient numerical array manipulation has long been studied over relational systems. It is worthwhile to mention this topic here since numerical array manipulation can be viewed as the cornerstone for ML. Some approaches attempt to integrate linear algebra operations into the relational model and eliminates the dichotomy between matrices and relations for declarativeness and high-performance:

AIDA [112] integrates NumPy [113] into MonetDB [114] by exploiting the fact that both systems use C arrays as an internal data structure — to avoid copying NumPy data to MonetDB, AIDA shares pointers to arrays to transfer between the two systems. On the other hand, data copying is still required when passing MonetDB results to NumPy since MonetDB cannot guarantee that multiple columns are contiguous in memory, which is required by NumPy. AIDA also provides a Python-style language for relational and linear algebra operations. Sequences of relational operations are

evaluated lazily, which enables AIDA to conduct cross query optimizations. Note that this optimization does not include linear algebra operations.

LARA [115] proposes an algebra with tuple-wise operators, attribute-wise operators, and tuple extensions, then defines linear and relational algebra operations using these primitives. LARA offers a theoretical analysis of the properties of its operators, which allows some inter-algebra optimizations that span linear and relational operations. However, it is worth mentioning that LARA attempts to implement ML computations as algebraic expressions (e.g., a join followed by an aggregation) over relations of `(key, value)` pairs. This requires pushing a huge number of pairs through the system, which introduces significant overhead.

Further, systems are proposed to handle numerical array manipulation in a relational style parallelly and distributively in clusters:

SystemML [116, 117, 118] is a well-known system designed by IBM. The system provides a set of linear algebra primitives that are expressed in a high-level, declarative language, implemented on Hadoop [116] and Spark [117]. SystemML conducts the linear algebra optimizations in a fashion that is very similar to relational optimizations, e.g., determining the matrix chain multiplication execution order. Further, SystemML is equipped with compressed linear algebra [118], where lightweight database compression techniques are applied to matrices and then linear algebra operations such as matrix-vector multiplication are executed directly on the compressed representations. The speedup is achieved by effective column compression schemes, cache conscious operations, and an efficient sampling based compression algorithm. Note that the compressed linear algebra is super effective to handle sparse matrices/tensors originally stored as dense arrays. While the computation is usually dense in SGD optimization for general deep learning models, sparse features do exists as well. In this case it is

reasonable to expect the solution proposed in Chapter 3 can be jointly optimized by a similar compression — this can be an interesting future work.

SciDB [119] is a database system that is specialized for scientific array-based computations. Matrices and relations are implemented as nested arrays. SciDB offers efficient array processing over parallel scientific computations. SciDB also includes element-wise operations and selected linear algebra operations, such as SVD.

Multi-dimensional-recursion has been built on top of SimSQL [120], a distributed analytic database system, to support linear algebra computations [121]. This system introduces matrices as ordered numeric-only attribute types and efficient relational optimization, which leads to competitive performance.

The declarative matrix multiplication in database is further optimized for GPUs. DistME [63] is such a system built over Spark. DistME proposes a distributed elastic matrix multiplication method called CuboidMM, where matrices are partitioned into cuboids to optimize the network communication w.r.t memory usage per process; and cuboid are further split into subcuboids for GPU acceleration by optimizing the PCI-E communication with the limitation of GPU memory usage.

Relational systems have also been directly used for ML. The initial attempt is to make the database system an efficient data loader, to generate training batches for SGD iterations. This is an very interesting idea since the training data are usually stored — not surprisingly — in the database for real-world industrial applications. MLog [122] is such a declarative relational system managing data movement, data persistency, and training batch generation. MLog provides tensor-based views of the relational data for efficient batch generation. However, MLog does not view the RDBMS as a compute engine for the forward and backward propagations; instead, MLog simply utilizes TensorFlow for the ML computations. Similar ideas have been

applied in [123] for feature extraction queries over multi-relation databases, and [124] for optimizing sparse tensor computations constructed from relational tables.

Recently, relational systems have also been considered as runtime engines (instead of limiting itself as an efficient data loader) for distributed ML. DB4ML [125] proposes user-defined iterative transactions for ML computations. Their key observation is that database transactions are able to provide an execution environment that allows RDBMSs to efficiently mimic the execution model of modern parallel ML algorithms. Based on this observation, DB4ML is designed in a way that it can support user-defined ML algorithms inside a RDBMS with the competitive efficiency.

The follow-up work of multi-dimensional-recursion on top of SimSQL [121] attempts to argue that one should use a database system for distributed ML [126]. This work extends the previous optimizations for linear algebra to model parallelism ML optimizations and achieves competitive scalabilty compared to TensorFlow. It is worth mentioning that this work leverages intra-layer model parallel optimization where the computation within a layer, e.g., a huge fully connected layer, is partitioned. This is different from the model parallelism introduced in Section 4.1.1. I note the previous model parallelism as inter-layer model parallelism here. In fact, the inter-layer level model parallelism, where the large models are partitioned spatially among compute sites [127, 128, 129, 103], is very similar to the concept of pipelined parallelism in the database community, which has long been used in relational systems [130].

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

This thesis proposes algorithmic and system-based approaches to speed up and scale out distributed ML. The algorithmic approach is called independent subnet training (IST), and facilitates distributed training of neural networks. By stochastically partitioning the model into non-overlapping subnets, IST reduces the communication overhead for model synchronization, and reduces the computation required for forward-backward propagation via the use of a thinner model on each worker. This results in two advances: i) IST significantly accelerates the training process compared with standard data parallel approaches for distributed learning, and ii) IST scales to large models that cannot be learned using standard data parallel approaches.

At the systems level, I introduce novel abstractions necessary for building distributed ML systems. I propose the tensor relational algebra (TRA), which is expressive enough to encode any tensor operation written using Einstein notation, and propose an implementation algebra (IA) targeted at effective optimizations for a parallel/distributed environment. Optimization of IA expressions can enable efficient, distributed implementations of ML computations, and I show that such computations can match or even significantly outperform existing HPC and ML systems. Such computational and implementation abstractions could form the basis for the design of future declarative ML systems.

5.2 Future Research Opportunities

The ideas proposed in this thesis bring plenty of new research opportunities, among which I would discuss two in depth.

5.2.1 Efficient Training with IST at the Edge

While IST greatly alleviates the computational burden in a single compute site in a distributed cluster, the subnetwork's complexity can still be too high to handle in edge-computing scenarios. Edge computing devices can often use limited energy, and have constraints on available storage and computational resources. For resource-constrained applications in edge devices, even performing inference is a challenging task. Compared to inference, distributed training on edge devices is more difficult still, for many reasons. For example, the actual per-iteration of the training algorithm is more expensive than that of the inference procedure; the optimization of the hardware has to guarantee the empirical convergence rate of the training and integrate with existing hardware architecture [131].

Toward this end, it is important study techniques to aggressively reduce the training cost of subnetworks without compromising their algorithmic accuracy. Specifically, some techniques can be combined with IST for this purpose: i) an end-to-end mixed low-precision model via progressive and dynamic training precision scheduling and ii) lazy parameter updates via dynamic fractional skipping of layers. The aim of such an IST extension is to reduce network latency and energy consumption during IST training. I conjecture that it will be possible to continuously adapt the subnetwork training parameters at the worker nodes, so as to maximize performance without incurring a drop in accuracy.

5.2.2 Reinforcement Learning for TRA Optimization

As the empirical study shows, the proposed cost model is an effective means of estimating the real-life runtime of tensor-based computations in a distributed cluster. Classical relational optimizers face significant problems in accurate cardinality estimation [132, 133]. However, in the TRA it is possible to exactly characterize the size and other statistical characteristics of every tensor relations in a computation.

Still, TRA optimization is not a solved problem. Optimizing the computation represented by a calculation in TRA still presents a unique set of challenges. There are hard constraints for TRA computations: individual arrays must fit in the RAM of a device (the RAM of one GPU, or the RAM of a machine); the communication cost can be heterogeneous — shuffling between GPUs on the same machine is costly, shuffling across machines through networks are devastating.

In response to this, it would be interesting to investigate an entirely new paradigm for optimization, where rather than being statically optimized, one can view the process of executing a tensor-relational algebra statement repeatedly as a Markov decision process (MDP) [134]*. In the context of optimizing and executing TRA algebra computations, one can consider a potential mapping to an MDP, where states represent a current plan and a set of atomic portions of the plan that have been executed thus far; actions include altering of the current plan according to the re-write rules; transition probabilities describe the probability of getting the next plan(s) in execution time when taking an action for the current plan(s); and the reward function can be the reverse of the expected cost to run the whole plan. As a few

*The TRA statement must be optimized over its lifetime of executions, so as to achieve minimum cost over all executions. Note that ML computations are usually run again and again; e.g., during gradient descent, the same TRA statement will be executed potentially thousands of times.

recent interesting works leverage deep reinforcement learning (RL) to solve MDPs for database research [135, 136, 137, 138, 139], it is reasonable to believe RL would also be a good option for optimizing TRA computations formalized as a MDP problem.

Bibliography

[1] J. Wang, "A new system for distributed machine learning," *National Science Review*, vol. 5, no. 3, pp. 303–304, 2018.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[3] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[4] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, 2018.

[5] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 5595–5637, 2017.

[6] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–33, 2020.

[7] A. Ratner, D. Alistarh, G. Alonso, P. Bailis, S. Bird, N. Carlini, B. Catanzaro, E. Chung, B. Dally, J. Dean, *et al.*, "SysML: The new frontier of machine learning systems," *arXiv preprint arXiv:1904.03257*, 2019.

[8] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, pp. 1223–1231, 2012.

[9] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system.," in *OSDI*, vol. 14, pp. 571–582, 2014.

[10] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server.," in *OSDI*, vol. 14, pp. 583–598, 2014.

[11] S. Hadjis, C. Zhang, I. Mitliagkas, D. Iter, and C. Ré, "Omnivore: An optimizer for multi-device deep learning on CPUs and GPUs," *arXiv preprint arXiv:1606.04487*, 2016.

[12] X. Zhang, M. Mckenna, J. P. Mesirov, and D. L. Waltz, "An efficient implementation of the back-propagation algorithm on the connection machine CM-2," in *Advances in neural information processing systems*, pp. 801–809, 1990.

[13] P. Farber and K. Asanovic, "Parallel neural network training on multi-spert," in *Algorithms and Architectures for Parallel Processing, 1997. ICAPP 97., 1997 3rd International Conference on*, pp. 659–666, IEEE, 1997.

[14] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning

using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*, pp. 873–880, ACM, 2009.

[15] L. Ma, G. Montague, J. Ye, Z. Yao, A. Gholami, K. Keutzer, and M. W. Mahoney, "Inefficiency of K-FAC for large batch size training," *arXiv preprint arXiv:1903.06237*, 2019.

[16] N. Golmant, N. Vemuri, Z. Yao, V. Feinberg, A. Gholami, K. Rothauge, M. W. Mahoney, and J. Gonzalez, "On the computational inefficiency of large batch sizes for stochastic gradient descent," *arXiv preprint arXiv:1811.12941*, 2018.

[17] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training ImageNet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[18] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato, "Multi-GPU training of convnets," *arXiv preprint arXiv:1312.5853*, 2013.

[19] Y. You, I. Gitman, and B. Ginsburg, "Scaling SGD batch size to 32K for ImageNet training," *arXiv preprint arXiv:1708.03888*, 2017.

[20] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," *arXiv preprint arXiv:1711.00489*, 2017.

[21] V. Codreanu, D. Podareanu, and V. Saletore, "Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train," *arXiv preprint arXiv:1711.04291*, 2017.

[22] Y. You, J. Li, J. Hseu, X. Song, J. Demmel, and C.-J. Hsieh, "Reducing BERT pre-training time from 3 days to 76 minutes," *arXiv preprint arXiv:1904.00962*,

2019.

[23] Y. You, J. Hseu, C. Ying, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large-batch training for LSTM and beyond," *arXiv preprint arXiv:1901.08256*, 2019.

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[25] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast monte carlo algorithms for matrices i: Approximating matrix multiplication," *SIAM Journal on Computing*, vol. 36, no. 1, pp. 132–157, 2006.

[26] "Nccl based multi-gpu training." http://on-demand.gputechconf.com/gtc-cn/2018/pdf/CH8209.pdf, 2018. Accessed: 2020-02-06.

[27] T. Lin, S. U. Stich, and M. Jaggi, "Don't use large mini-batches, use local SGD," *arXiv preprint arXiv:1808.07217*, 2018.

[28] E. F. Codd, "A relational model of data for large shared data banks," *CACM*, vol. 13, no. 6, pp. 377–387, 1970.

[29] A. V. Aho and J. D. Ullman, "Universality of data retrieval languages," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 110–119, ACM, 1979.

[30] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *European Conference on Parallel Processing*, pp. 90–109, Springer, 2011.

[31] S. Wright and J. Nocedal, "Numerical optimization," *Springer Science*, vol. 35, no. 67-68, p. 7, 1999.

[32] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

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

[34] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[35] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[36] A. Defazio and L. Bottou, "On the ineffectiveness of variance reduced optimization for deep learning," *arXiv preprint arXiv:1812.04529*, 2018.

[37] A. S. Berahas, R. Bollapragada, and J. Nocedal, "An investigation of Newton-sketch and subsampled Newton methods," *arXiv preprint arXiv:1705.06211*, 2017.

[38] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *Siam Review*, vol. 60, no. 2, pp. 223–311, 2018.

[39] S. B. Kylasa, F. Roosta-Khorasani, M. W. Mahoney, and A. Grama, "GPU accelerated sub-sampled Newton's method," *arXiv preprint arXiv:1802.09113*, 2018.

[40] P. Xu, F. Roosta-Khorasani, and M. W. Mahoney, "Newton-type methods for non-convex optimization under inexact hessian information," *arXiv preprint arXiv:1708.07164*, 2017.

[41] A. S. Berahas, M. Jahani, and M. Takáč, "Quasi-Newton methods for deep learning: Forget the past, just sample," *arXiv preprint arXiv:1901.09997*, 2019.

[42] J. Martens and R. Grosse, "Optimizing neural networks with Kronecker-factored approximate curvature," in *International conference on machine learning*, pp. 2408–2417, 2015.

[43] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, pp. 448–456, 2015.

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

[45] H. Karimi, J. Nutini, and M. Schmidt, "Linear convergence of gradient and proximal-gradient methods under the Polyak-łojasiewicz condition," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 795–811, Springer, 2016.

[46] A. Khaled and P. Richtárik, "Gradient descent with compressed iterates," *arXiv preprint arXiv:1909.04716*, 2019.

[47] R. Carter, "On the global convergence of trust region algorithms using inexact gradient information," *SIAM Journal on Numerical Analysis*, vol. 28, no. 1, pp. 251–265, 1991.

[48] A. Berahas, L. Cao, K. Choromanski, and K. Scheinberg, "A theoretical and empirical comparison of gradient approximations in derivative-free optimization," *arXiv preprint arXiv:1905.01332*, 2019.

[49] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[50] S. S. Stevens, J. Volkmann, and E. B. Newman, "A scale for the measurement of the psychological magnitude pitch," *The Journal of the Acoustical Society of America*, vol. 8, no. 3, pp. 185–190, 1937.

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

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

[53] K. Bhatia, K. Dahiya, H. Jain, Y. Prabhu, and M. Varma, *The Extreme Classification Repository: Multi-label Datasets and Code*. http://manikvarma.org/downloads/XC/XMLRepository.html.

[54] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[55] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent," *arXiv preprint arXiv:1704.05021*, 2017.

[56] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD:

Communication-efficient SGD via gradient quantization and encoding," in *Advances in Neural Information Processing Systems*, pp. 1709–1720, 2017.

[57] N. Ivkin, D. Rothchild, E. Ullah, I. Stoica, R. Arora, *et al.*, "Communication-efficient distributed sgd with sketching," in *Advances in Neural Information Processing Systems*, pp. 13144–13154, 2019.

[58] T. Vogels, S. P. Karimireddy, and M. Jaggi, "Powersgd: Practical low-rank gradient compression for distributed optimization," in *Advances in Neural Information Processing Systems*, pp. 14236–14245, 2019.

[59] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[60] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 578–594, 2018.

[61] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[62] R. Gu, Y. Tang, C. Tian, H. Zhou, G. Li, X. Zheng, and Y. Huang, "Improving execution concurrency of large-scale matrix multiplication on distributed data-parallel platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2539–2552, 2017.

[63] D. Han, Y.-M. Nam, J. Lee, K. Park, H. Kim, and M.-S. Kim, "Distme: A fast and elastic distributed matrix computation engine using gpus," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 759–774, 2019.

[64] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: A scalable linear algebra library for distributed memory concurrent computers," in *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 120–121, IEEE Computer Society, 1992.

[65] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[66] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[67] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel, "Image-based recommendations on styles and substitutes," in *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pp. 43–52, 2015.

[68] J. McAuley, R. Pandey, and J. Leskovec, "Inferring networks of substitutable and complementary products," in *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 785–794, 2015.

[69] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for

self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[70] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[71] A. E. Khandani, A. J. Kim, and A. W. Lo, "Consumer credit-risk models via machine-learning algorithms," *Journal of Banking & Finance*, vol. 34, no. 11, pp. 2767–2787, 2010.

[72] E. P. Xing, Q. Ho, P. Xie, and D. Wei, "Strategies and principles of distributed machine learning on big data," *Engineering*, vol. 2, no. 2, pp. 179–195, 2016.

[73] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, pp. 1223–1231, 2013.

[74] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, "High-performance distributed ml at scale through parameter server consistency models," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pp. 79–87, 2015.

[75] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *Advances in neural information processing systems*, pp. 2834–2842, 2014.

[76] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[77] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1–10, Ieee, 2010.

[78] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pp. 15–28, 2012.

[79] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing, "Solving the straggler problem with bounded staleness," in *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[80] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, pp. 693–701, 2011.

[81] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[82] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching {LAN} speeds," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 629–647, 2017.

[83] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," *arXiv preprint arXiv:1604.00981*, 2016.

[84] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," *arXiv preprint arXiv:1511.05950*, 2015.

[85] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, "Communication-efficient distributed deep learning: A comprehensive survey," *arXiv preprint arXiv:2003.06307*, 2020.

[86] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[87] S. U. Stich, "Local sgd converges fast and communicates little," *arXiv preprint arXiv:1805.09767*, 2018.

[88] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[89] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified sgd with memory," in *Advances in Neural Information Processing Systems*, pp. 4447–4458, 2018.

[90] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," in *Advances in Neural Information Processing Systems*, pp. 1299–1309, 2018.

[91] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[92] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, pp. 1737–1746, 2015.

[93] J. Langford, L. Li, and T. Zhang, "Sparse online learning via truncated gradient," in *Advances in neural information processing systems*, pp. 905–912, 2009.

[94] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[95] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli, "The convergence of sparsified gradient methods," in *Advances in Neural Information Processing Systems*, pp. 5973–5983, 2018.

[96] D. Basu, D. Data, C. Karakus, and S. Diggavi, "Qsparse-local-sgd: Distributed sgd with quantization, sparsification and local computations," in *Advances in Neural Information Processing Systems*, pp. 14695–14706, 2019.

[97] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[98] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. L. García, I. Heredia, P. Malík, and L. Hluchỳ, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.

[99] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters

of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[100] R. Rabenseifner, "Optimization of collective reduction operations," in *International Conference on Computational Science*, pp. 1–9, Springer, 2004.

[101] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," in *Advances in Neural Information Processing Systems*, pp. 10414–10423, 2018.

[102] P. Buchlovsky, D. Budden, D. Grewe, C. Jones, J. Aslanides, F. Besse, A. Brock, A. Clark, S. G. Colmenarejo, A. Pope, *et al.*, "Tf-replicator: Distributed machine learning for researchers," *arXiv preprint arXiv:1902.00465*, 2019.

[103] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

[104] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimization towards training a trillion parameter models," *arXiv preprint arXiv:1910.02054*, 2019.

[105] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[106] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *USENIX HotCloud*, pp. 1–10, 2010.

[107] D. Team *et al.*, "Deeplearning4j: Open-source distributed deep learning for the jvm," *Apache Software Foundation License*, vol. 2, p. 2, 2016.

[108] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "Sparknet: Training deep networks in spark," *arXiv preprint arXiv:1511.06051*, 2015.

[109] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li, *et al.*, "Bigdl: A distributed deep learning framework for big data," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 50–60, 2019.

[110] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[111] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[112] J. V. D'silva, F. De Moor, and B. Kemme, "Aida: Abstraction for advanced in-database analytics," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1400–1413, 2018.

[113] "Numpy." https://numpy.org/.

[114] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution.," in *Cidr*, vol. 5, pp. 225–237, 2005.

[115] D. Hutchison, B. Howe, and D. Suciu, "Laradb: A minimalist kernel for linear and relational algebra computation," in *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, p. 2, ACM, 2017.

[116] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on mapreduce," in *ICDE*, pp. 231–242, 2011.

[117] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, *et al.*, "Systemml: Declarative machine learning on spark," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1425–1436, 2016.

[118] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for large-scale machine learning," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 960–971, 2016.

[119] P. G. Brown, "Overview of SciDB: large scale array storage, processing and analysis," in *SIGMOD*, pp. 963–968, 2010.

[120] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine, "Simulation of database-valued markov chains using simsql," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 637–648, 2013.

[121] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine, "Scalable linear algebra on a relational database system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 7, pp. 1224–1238, 2018.

[122] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang, "Mlog: Towards declarative in-database machine learning," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1933–1936, 2017.

[123] M. Abo Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich, "In-database learning with sparse tensors," in *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pp. 325–340, 2018.

[124] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich, "Ac/dc: in-database learning thunderstruck," in *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, pp. 1–10, 2018.

[125] M. Jasny, T. Ziegler, T. Kraska, U. Roehm, and C. Binnig, "Db4ml-an in-memory database kernel with machine learning support," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 159–173, 2020.

[126] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao, "Declarative recursive computation on an rdbms: or, why you should use a database for distributed machine learning," *Proceedings of the VLDB Endowment*, vol. 12, no. 7, pp. 822–835, 2019.

[127] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in neural information processing systems*, pp. 103–112, 2019.

[128] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," *arXiv preprint arXiv:2006.09503*, 2020.

[129] B. Yang, J. Zhang, J. Li, C. Ré, C. R. Aberger, and C. De Sa, "Pipemare: Asynchronous pipeline parallel dnn training," *arXiv preprint arXiv:1910.05124*, 2019.

[130] W. Hong, "Exploiting inter-operation parallelism in xprs," *ACM SIGMOD Record*, vol. 21, no. 2, pp. 19–28, 1992.

[131] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[132] W. Rjaibi, G. M. Lohman, and P. J. Haas, "Estimation of column cardinality in a partitioned relational database," May 4 2004. US Patent 6,732,110.

[133] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins," in *2011 IEEE 27th International Conference on Data Engineering*, pp. 984–994, IEEE, 2011.

[134] M. L. Littman, T. L. Dean, and L. P. Kaelbling, "On the complexity of solving markov decision problems," in *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pp. 394–402, Morgan Kaufmann Publishers Inc., 1995.

[135] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi, "Learning state representations for query optimization with deep reinforcement learning," in *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, pp. 1–4, 2018.

[136] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, *et al.*, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 415–432, 2019.

[137] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2118–2130, 2019.

[138] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis, "Skinnerdb: Regret-bounded query evaluation via reinforcement learning," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 1153–1170, 2019.

[139] R. Marcus and O. Papaemmanouil, "Deep reinforcement learning for join order enumeration," in *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pp. 1–4, 2018.