Exploiting Application Characteristics for Efficient System Support of Data-Parallel Machine Learning

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Henggang Cui

B.S., Electronic Information Science and Technology, Tsinghua University
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May, 2017

Keywords: Large-Scale Machine Learning, Big Data Analytics

For careful readers.

Acknowledgments

I never expect that this acknowledgments section could be the toughest paragraphs to write in my Ph.D. study. There is so much that I want to thank, but, unfortunately, this type of writing is not my strength.

The most important person who made all of this come true is my advisor, Greg Ganger. I feel so fortunate to be advised by Greg because he is so smart and so helpful (especially the last week before paper deadlines). Our weekly meetings keep me always on the right track, and his jokes are funny. Many important ideas in this dissertation would not have been so well baked without his magic.

Phil Gibbons also deserves his own paragraph, because I think Phil is basically my unofficial co-advisor (at least I put his emails under the advisor label in my mailbox). Phil, Greg, and I have a one-on-two meeting every week, and Phil's feedback is equally helpful for making my work solid.

The work in this dissertation is part of the BigLearning project, and I really appreciate the feedback and suggestions from professor Garth Gibson and Eric Xing throughout my Ph.D. research work. I am also very fortunate to work with a group of wonderful collaborators in the BigLearning project. Jim Cipar and Qirong Ho were the students who started the LazyTable and Stale Synchronous Parallel project, and I have learned a lot from them. I also want to thank Aaron Harlap for having lunch with me every day. I have learned a lot from Aaron, including setting up TV antenna for the NFL games on free TV channels and playing softball.

I appreciate the insightful discussions with the BigLearning people, including Wei (David) Dai, Kevin Hsieh, Jin Kyu Kim, Jinliang Wei, and Hao Zhang, as well as the other people in the BigLearning project, including Abutalib Aghayev, Will Crichton, Andrew Chung, Dan DeCapria, Drew Gifford, John R. Klein, Dimitris Konomis, Michael Kuchnik, Lisa Lee, Seunghak Lee, Xiaodan Liang, Scott McMillan, Alex Poms, Aurick Qiao, David Shepard, Brandon Wolfe, Pengtao Xie, and Shizhen Xu.

I have been having a good time in Greg Ganger's Gang (G^3), with Jim Cipar, Jesse Haber-Kucharsky, Aaron Harlap, Angela Jiang, Rajat Kateja, Elie Krevat, Michelle Mazurek, Jun Woo Park, Raja Sambasivan, Ilari Shafer, Alexey Tumanov, and Lianghong Xu, and I hope the name "G^3" is still in use.

I am glad that Derek Murray joined my thesis committee. Derek was the shepherd of my GeePS work, and he has provided a lot of valuable feedback to the work in my dissertation. I did a wonderful internship at HP Labs in the summer of 2014, where I was fortunate to have Kimberly Keeton, Indrajit Roy, Krishnamurthy Viswanathan, and Haris Volos as my mentors.

My research and life has been made much easier with the help of an amazing group of administrative and technical staff at Parallel Data Lab (PDL). Karen Lindenfelser is always so helpful with all kinds of miscellaneous stuff at PDL and prepares delicious fruit and snacks for the weekly PDL meeting. Joan

Digney does a wonderful job of making posters for all my work. Jason Boles, Bill Courtright, Chuck Cranor, Chad Dougherty, Zis Economou, Mitch Franzos, and Xiaolin Zang provide wonderful technical support for the clusters that I use for my experiments.

I also want to thank my friends at PDL for the technical discussions or random chats, including David Andersen, Dana Van Aken, George Amvrosiadis, Joy Arulraj, Rachata Ausavarungnirun, Ben Blum, Sol Boucher, Christopher Canel, Lei Cao, Kevin Chang, Debanshu Das, Utsav Drolia, Chris Fallin, Bin Fan, Kayvon Fatahalian, Nitin Gupta, Kiryong Ha, Mor Harchol-Balter, Junchen Jiang, Wesley Jin, Anuj Kalia, Michael Kaminsky, Atsushi Kawamura, Mike Kozuch, Akira Kuroda, Conglong Li, Mu Li, Hyeontaek Lim, Yixin Luo, Lin Ma, Priya Narasimhan, Andy Pavlo, Kai Ren, Majd Sakr, Vivek Seshadri, Alex Smola, Joshua Tan, Thomas Tauber-Marshall, Iulian Moraru, Jiri Simsa, Nandita Vijaykumar, Lin Xiao, Hongyi Xin, Huanchen Zhang, Michael (Tieying) Zhang, Jie Zhang, Qing Zheng, Dong Zhou, and Timothy Zhu.

CIC 2nd floor is a great working environment, except for being freezing in the summer, and I have many good friends here, including Tiffany (Youzhi) Bao, Chen Chen, Yanli Li, Jiaqi Liu, Mahmood Sharif, Janos Szurdi, Miao Yu, and Tianlong Yu.

I also want to thank the members and companies of the PDL Consortium (including Broadcom, Citadel, Dell EMC, Google, Hewlett-Packard Labs, Hitachi, Intel Corporation, Microsoft Research, MongoDB, NetApp, Oracle Corporation, Samsung Information Systems America, Seagate Technology, Tintri, Two Sigma, Toshiba, Uber, Veritas, and Western Digital) for their interest, insights, feedback, and support. The work in this dissertation is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE).

Finally, I want to thank my wife for accompanying me through the tough times, my parents for always backing me up. They have made all of this happen.

Abstract

Large scale machine learning has many characteristics that can be exploited in the system designs to improve its efficiency. This dissertation demonstrates that the characteristics of the ML computations can be exploited in the design and implementation of parameter server systems, to greatly improve the efficiency by an order of magnitude or more. We support this thesis statement with three case study systems, IterStore, GeePS, and MLtuner. IterStore is an optimized parameter server system design that exploits the repeated data access pattern characteristic of ML computations. The designed optimizations allow IterStore to reduce the total run time of our ML benchmarks by up to 50×. GeePS is a parameter server that is specialized for deep learning on distributed GPUs. By exploiting the layer-by-layer data access and computation pattern of deep learning, GeePS provides almost linear scalability from single-machine baselines (13× more training throughput with 16 machines), and also supports neural networks that do not fit in GPU memory. MLtuner is a system for automatically tuning the training tunables of ML tasks. It exploits the characteristic that the best tunable settings can often be decided quickly with just a short trial time. By making use of optimization-guided online trial-and-error, MLtuner can robustly find and re-tune tunable settings for a variety of machine learning applications, including image classification, video classification, and matrix factorization, and is over an order of magnitude faster than traditional hyperparameter tuning approaches.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large scale machine learning has emerged as a primary computing activity in business, science, and services, attempting to extract insight from quantities of observation data. A machine learning task assumes a particular mathematical *model* will describe the observed training data and use an algorithm to identify *model parameter* values that make it fit the input data most closely. Depending on the application, such models can expose relationships among data items (e.g., for grouping documents into topics), predict outcomes for new data items based on selected characteristics (e.g., for classification tasks and recommendation systems), correlate effects with causes (e.g., for genomic analyses of diseases), and so on.

We focus on one major subset of these ML tasks that we refer to as iterative data-parallel machine learning. Examples include deep learning for image and video classification, matrix factorization for movie recommendation, and topic modeling. For those ML tasks, the ML algorithm (such as SGD) starts with some initial parameter value guesses, and then performs a number of *iterations* to refine them. Each iteration evaluates each input datum, one by one, against current model parameters and adjusts parameters to better fit that datum. The expensive computation required often makes it desirable to run these ML tasks distributedly on a cluster of machines. We focus on a *data-parallel* approach of parallelizing such tasks, in which the training data is partitioned among workers, and the workers iteratively make changes to the shared model parameter data, based on their local training data.

While other designs can be used, an increasingly popular design for maintaining the distributed shared parameter data is to use a so-called *parameter server* architecture [4, 73, 59, 29, 20, 28, 27, 25, 92]. The parameter server manages the shared parameter data for the application and takes care of all the distributed system details, such as propagating parameter updates and synchronizing parameter data among workers. Designing a parameter server that can scale efficiently is challenging. The challenges include overcoming the overhead of communication and synchronization, and supporting specialized hardware, such as GPUs.

1.1 Thesis statement

This dissertation describes our parameter system designs that address those challenges by exploiting the characteristics of ML computations. In particular, we make the following thesis statement:

The characteristics of large-scale data-parallel machine learning computations can be exploited in the implementation of a parameter server to increase their efficiency by an order of magnitude or more.

To support this thesis statement, we will show three case studies of specializing parameter server designs that exploit different characteristics of ML computations.

- **IterStore: exploiting repeated parameter data access pattern (Chapter 3).** Many iterative ML algorithms have the property that the same (or nearly the same) sequence of accesses is applied to the parameter server every iteration. This repeating sequence can be exploited to improve the performance of parameter servers. We designed a *virtual iteration* method to collect this repeating access sequence from the application, and employed five parameter server specializations using the collected information, including prefetching, contiguous marshalling-free data storage, locality and NUMA-aware data placement, and specialized caching policy. Our experiments show that these specializations greatly reduce the total run time of our application benchmarks by up to 50×.
- **GeePS: exploiting layer-by-layer computation of deep learning (Chapter 4).** Deep neural networks are often trained using GPUs, but scaling GPU applications on multiple machines is challenging, because of the limited GPU memory size and expensive data movement overheads between GPU and CPU memory. We have designed a parameter server system, called GeePS, that is specialized for GPU deep learning applications. We identify an important characteristic of the deep learning application that their data access and computation is performed layer-by-layer. By exploiting this characteristic, GeePS is able to hide communication latency from the application, and support neural networks that do not fit in GPU memory, by swapping data from/to CPU memory in the background. We have linked GeePS with Caffe, a state-of-art single-machine deep learning system, and our experiments show that GeePS provides almost linear scalability from single-machine Caffe (13× more training throughput with 16 machines).
- **MLtuner: exploiting quick decision and dynamism of training tunable selection (Chapter 5).** Training a machine learning model involves the selection and tuning of many *training tunables*, such as the mini-batch size, learning rate, and synchronization policy. The best setting of these tunables depends on many factors, such as the problem, the dataset, and the hardware environment, and is often hard to decide offline. Moreover, the best settings often change during the training. A bad choice of the tunables can make the model converge very slowly, converge to suboptimal solution, or even not converge. We designed a system, called MLtuner, that is able to automatically select and tune the training tunables for machine learning systems. It can be integrated with existing parameter server systems, such

as IterStore and GeePS. MLtuner exploit the characteristics of the ML tasks that the best tunable setting can often be decided quickly, with a short trial time. It uses efficient snapshotting and optimization-guided online trial-and-error to find good initial tunable settings as well as to re-tune settings during training. Our experiments with five real ML tasks, including deep learning and matrix factorization, show that MLtuner can robustly find and re-tune tunable settings for them, and is over an order of magnitude faster than traditional hyperparameter tuning approaches.

1.2 Contributions

This dissertation makes the following key contributions.

IterStore:

- It identifies iterative-ness as a property of many data-parallel ML applications that can and should be exploited to improve efficiency significantly.
- It describes and evaluates a broad collection of specializations that exploit iterativeness effectively, showing their overall and individual values for several real ML applications.
- It describes and evaluates the concept of a virtual iteration and compares it with detection of patterns during the first iteration.

GeePS:

- It describes the first GPU-specialized parameter server design and the changes needed to achieve efficient data-parallel multi-machine deep learning with GPUs.
- It reports on large-scale experiments showing that GeePS indeed supports scalable data parallel execution via a parameter server, in contrast to previous expectations.
- It introduces new parameter server support for enabling such data-parallel deep learning on GPUs even when models are too big to fit in GPU memory, by explicitly managing GPU memory as a cache for parameters and intermediate layer state.

MLtuner:

- It introduces the first approach for automatically tuning the multiple tunables associated with nearly any ML application within the context of a single execution of that application.
- It describes a tool (MLtuner) that implements the approach, overcoming various challenges, and how MLtuner was integrated with two different ML training systems.
- It presents results from experiments with real ML applications, demonstrating the efficacy of this new approach in removing the black art of tuning from ML application training.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 motivates our works with more background on data-parallel machine learning, the parameter server architecture,

and some popular machine learning applications. Chapter 3 describes IterStore [27], our parameter server design that exploits the iterativeness characteristic of ML computations. Chapter 4 describes GeePS [28], our parameter server design that exploits the layer-by-layer computation pattern of GPU-based deep learning applications. Chapter 5 describes MLtuner, a system for automatically picking and tuning training tunables. Chapter 6 concludes the dissertation and discusses future research directions.

Chapter 2

Background: Data-Parallel ML and Parameter Servers

This section describes some additional background of our work, including data-parallel machine learning, consistency models, the parameter server architecture, and popular ML application benchmarks.

2.1 Data-parallel machine learning

The goal of a ML task is to train a ML model on training (input) data, so that the trained model can be used to make predictions on unseen data. A model has trainable *model parameters*, and the ML task tries to determine the values of these model parameters that best fit the training data. The fitness error of the set of model parameters to the training data is defined mathematically with an *objective function*, and the current value of the objective function is often called the *objective value* or *training loss*. Thus, the machine learning problem is an optimization problem, whose goal is to minimize the objective function.

The ML task often optimizes the objective function with an iterative convergent algorithm. For example, with the gradient descent algorithm, the model parameters are first initialized to some random values, and in every step, the gradient descent algorithm calculates the gradient of the objective function, w.r.t. the model parameters. The model parameter change will be the opposite direction of the gradient (if we want to minimize the objective function), multiplied by a *learning rate*. The algorithm will repeat the above steps for multiple *iterations*, until the model converges. For large-scale machine learning problems, where the size of the training data is large, people often use the stochastic gradient descent (SGD) algorithm, where in each iteration, the gradients are calculated on just one batch of the training data, rather than all the training data.

To speed up ML tasks, people often distribute the ML computations across a cluster of machines. The *data-parallel* approach is a popular approach of doing that, where the training data is partitioned across multiple ML workers on different machines. Each ML worker keeps a local copy of the model parameters and computes model gradients based

on their local model parameter copy and training data partition. The ML workers will propagate the computed model gradients and refresh their model parameter copies every *clock*, which is often logically defined as some unit of work (e.g., every training data batch).

2.2 Consistency models for data-parallel ML

When the model is trained in the distributed manner, the ML workers will have temporarily inconsistent model parameter copies, causing *data staleness errors* [42, 25]. To guarantee model convergence, it is usually necessary to enforce some *consistency model* that bounds the data staleness in some way. There are generally three types of consistency models to be considered.

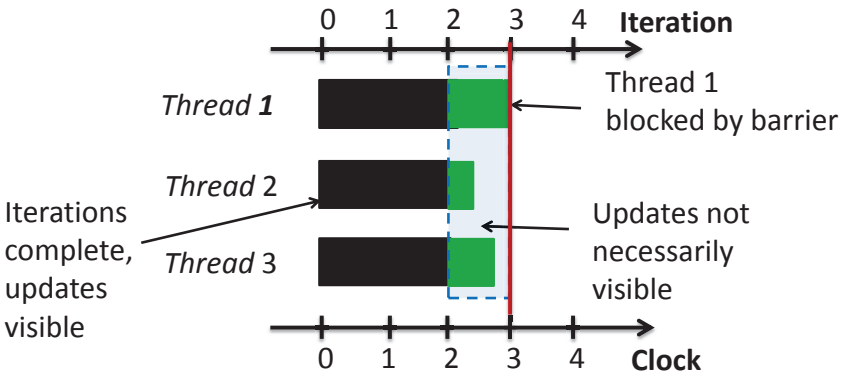


Figure 2.1: Bulk Synchronous Parallel (BSP).

The Bulk Synchronous Parallel (BSP) model, as illustrated in Figure 2.1, requires that all workers must see the parameter updates from all other workers from all previous clocks, before proceeding to the next clock. BSP guarantees the workers a fresh view of the model parameters every clock, which is often defined as one iteration. The potential problem with the BSP model is that the time waiting for communication might block the computation, and moreover, a slower worker (i.e., a straggler) can slow down the whole computation. At the other end of the spectrum, is the Asynchrony model [74, 20], where the workers never wait for each other, and the amount of data staleness error depends on the best-effort communication throughput and latency. The potential problem of the Asynchrony model is that the model might need more iterations to converge or even diverge.

The Stale Synchronous Parallel (SSP) model [42, 25] is a middle ground between BSP and Asynchrony. It provides a looser but still bounded data staleness guarantee. As is illustrated in Figure 2.2, SSP allows the fastest worker to be ahead of the slowest worker by up to *slack* clocks. For example, with a slack of s, a worker at clock t is guaranteed to see all updates from clocks 1 to $t - s - 1$, and it may see (not guaranteed) the updates after clock $t - s - 1$. SSP generalizes BSP and Asynchrony, making BSP a special case with zero slack and Asynchrony a special case with infinite slack. By introducing this slack parameter, SSP allows people to explicitly balance data freshness and system throughput.

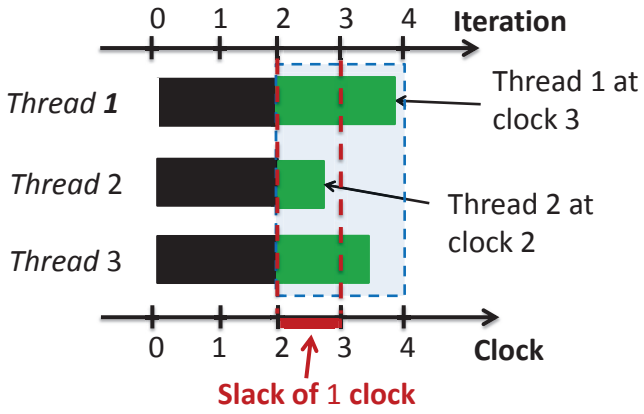


Figure 2.2: Stale Synchronous Parallel (SSP), with a slack of one clock.

2.3 Parameter server architecture

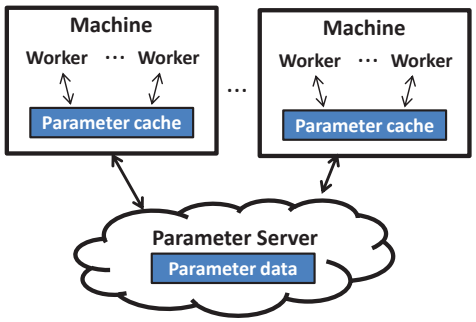


Figure 2.3: Parallel ML with parameter server.

For data-parallel ML, the shared model parameters are often managed with a *parameter server* system [59, 25, 28, 92, 27, 20, 29, 4, 73]. Figure 2.3 illustrates the basic parameter server architecture. The ML application launches multiple workers, and each of them process their assigned training data, and communicates with the parameter server to read and update the shared model parameter data.

Key-value interface. The parameter server is essentially a distributed key-value store and provides a simple key-value interface for the ML application to to read and update the shared model parameters, and synchronize their progress. This interface usually includes:

- `Read`: reads the parameter data. To support the Stale Synchronous Parallel model, some parameter servers (e.g., [25, 92]) allows the application to specify the slack parameter for each `Read` operation.
- `Update`: updates the parameter data by *delta*. The user will need to define an update function to merge the parameter update *delta* with the current parameter data. The update function is often required to be associative and commutative, so that the

updates from different ML workers can be applied in any order. For most of our applications, this update function is simply an addition-assignment operation ($+=$).

- `Clock`: signals the completion of one clock of work. Each clock of work is usually logically defined as processing one batch of the training data.

Sharding. The parameter server is usually a distributed key-value store, and the parameter data is sharded across multiple parameter server shards. While Figure2.3 illustrates the parameter server as separate from the machines executing ML workers, the state-of-art parameter server systems (e.g., [25, 92, 27, 28]) often places the shards across the same machines as the ML workers. This approach avoids dedicating extra machines to run the parameter server and allows some degree of data locality to be explored, as is shown in our IterStore work described in Chapter 3.

Caching. To avoid constant remote communication, a parameter server system often includes client-side caches that serve most operations locally. For example, when the application calls an `Update` operation, the parameter updates will be first coalesced in the update caches, and will then be propagated to the parameter servers when the `Clock` operation is called.

Consistency guarantees. The parameter server often supports the Staleness Synchronous Parallel model, which generalizes BSP and Asynchrony. It maintains the specified data staleness bound by attaching a *data age* field with each parameter data entry. When a parameter data entry has a data age of t, it means that the data includes all the parameter updates from all workers up to clock t. The parameter server will block the `Read` operations, when the data is not fresh enough. For example, when an application worker at clock T reads the parameter data with a slack of s clocks, the `Read` operation will be blocked, if the parameter data age is less than $T - s - 1$.

Fault tolerance. Because training an ML model is often an offline task, which requires high throughput rather than high availability from the parameter server, most of the state-of-art parameter sever implementations (e.g., [25, 92, 27, 28]) do not replicate the parameter data. Instead, they handle the fault tolerance by checkpointing (e.g., taking a checkpoint every hour). To make a consistent checkpoint, the client can just broadcast the checkpointing command to all the parameter server shards, telling them at which clock the checkpoint needs to be made, and the parameter server shards will take the checkpoint independently when they reach that clock.

2.4 Example ML applications and algorithms

This section describes the popular ML applications that are frequently used in our work, including their algorithms, and how they are parallelized with parameter server systems.

2.4.1 Matrix factorization for movie recommendation

Matrix factorization is a technique commonly used in recommendation systems, such as recommending movies to users on Netflix (a.k.a. collaborative filtering). The key idea is to discover latent interactions between the two entities (e.g., users and movies) via

matrix factorization. Given a partially filled rating matrix X of size $N \times M$, where each known entry (i, j) is user i's rating of movie j, matrix factorization factorizes X into two lower-ranked factor matrices L with size $N \times K$ and R with size $K \times M$ (usually $K << N$ and $K << M$), such that their product can be used to predict the missing entries of X (i.e., $X \approx L \cdot R$). For this application, the known entries of X is the training data, and L and R are the model parameters.

For this application, the goal is to minimize the objective function, which is often defined as the the squared difference between each known entry X_{ij} and its prediction $L_{i:} \cdot R_{:j}$:

$$\min_{L,R} f(L, R) = \sum_{(i,j)\in \text{Data}} \left\| X_{ij} - \sum_{k=1}^{K} L_{ik} R_{ik} \right\|^2 . \tag{2.1}$$

This objective function is often optimized with the stochastic gradient descent (SGD) algorithm [35, 55], and the gradient can be calculated as follows:

$$\begin{aligned} \frac{\partial f}{\partial L_{ik}} &= \sum_{(a,b)\in \text{Data}} \delta(a = i) \left[-2X_{ab} R_{kb} + 2L_{a\cdot} R_{\cdot b} R_{kb}\right] \\ \frac{\partial f}{\partial R_{kj}} &= \sum_{(a,b)\in \text{Data}} \delta(b = j) \left[-2X_{ab} L_{ak} + 2L_{a\cdot} R_{\cdot b} L_{ak}\right] \end{aligned} \tag{2.2}$$

where f is the objective function in Eq(2.1), and $\delta(a = i)$ equals 1 if $a = i$, 0 otherwise.

In every iteration, it computes the gradients $\frac{\partial f}{\partial L_{ik}}$ and $\frac{\partial f}{\partial R_{kj}}$, according to the current model parameter values L and R, and updates the model parameters with the SGD algorithm. It keeps doing that, until the model converges. The MF model can be trained in a data-parallel manner with a parameter server. To do that, we can distribute the data points X_{ij} evenly over multiple machines, and use the parameter server to manage the shared model parameters L and R. [1] Algorithm 1 shows the basic structure of a data-parallel MF implementation. In each iteration, each worker iterates over all its assigned matrix entries, and reads and updates the parameter data accordingly.

2.4.2 Deep learning

Deep learning is a popular task for learning deep neural network models. In deep learning, the ML programmer/user does not specify which specific features of the raw input data correlate with the outcomes being associated. Instead, the ML algorithm determines which features correlate most strongly by training a *deep neural network* with a large number of hidden layers [6]. A deep neural network, as depicted in Figure 2.4, is a multi-layer network, that encodes the connections between the input (e.g., raw images) and the output (e.g., image labels).

[1] If we partition the X matrix in such a way that each row is assigned to a particular worker, the corresponding L matrix rows can be kept locally by the workers, and only the R matrix needs to be kept in the parameter server.

Algorithm 1 Data-parallel matrix factorization with a parameter server.

```
while not converged do
    for each known entry X_ij do
        Li = ps.Read(i-th row of L matrix)
        Rj = ps.Read(j-th row of R matrix)
        Compute the gradients LiGrad and RjGrad from Eq(2.2)
        Scale the gradients with the learning rate
        ps.Update(i-th row of L matrix, LiGrad)
        ps.Update(j-th row of R matrix, RjGrad)
    end for
    ps.Clock()
end while
```

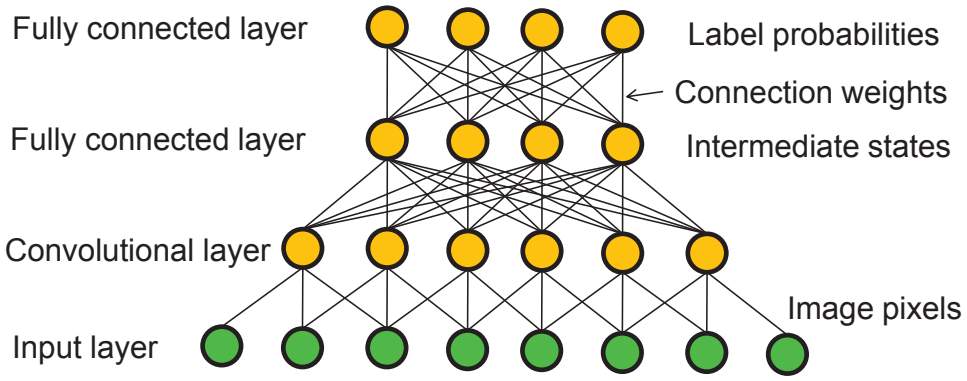


Figure 2.4: A convolutional neural network, with one convolutional layer and two fully connected layers.

Our deep learning benchmarks include two applications, image classification [20, 52, 29, 82] and video classification [31, 94]. An image classification network classifies images (raw pixel maps) into pre-defined labels. Its network is trained with a large set of training images with known labels. The video classification network classifies videos (a sequence of image frames) into pre-defined labels. It often uses an image classification network as a submodule and stacks recurrent layers on it to capture the sequence information. This subsection will describe the image classification application first and then describe how a video classification network can be built on it.

The image classification task often uses a type of model called a *convolutional neural network* (CNN). The first layer of the nodes (input of the network) are the raw pixels of the input image, and the last layer of the nodes (output of the network) are the probabilities that this image should be assigned to each label. The nodes in the middle are intermediate states. To classify an image with such a neural network, the image pixels will be assigned as the values for the first layer of nodes, and these nodes will *activate* their connected nodes of the next layer. There is a *weight* associated with each connection, and the value of each node at the next layer is a prespecified function of the weighted values of its connected nodes. Each layer of nodes is activated, one by one, by the setting of the node values for

the layer below. This procedure is called a *forward pass*.

There several types of layers. For example, a *fully connected* layer activates its neurons as the weighted sum of all the node values at the prior level. A *convolutional* layer activates its neuron with a convolution function over a (typically small) subset of the node values.

A common way of training a neural network is to use the SGD algorithm. For each training image, a forward pass is done to activate all nodes using the current weights. The values computed for each node are retained as intermediate states. At the last layer, an *error term* is calculated by comparing the predicted label probabilities with the true label. Then, the error terms are propagated back through the network with a *backward pass*. During the backward pass, the gradient of each connection weight is calculated from the error terms and the retained node values, and the connection weights (i.e., the model parameters) are updated using these gradients.

For efficiency, most training applications do each forward and backward pass with a batch of images (called a *mini-batch*) instead of just one image. For each image inside the mini-batch, there will be one set of node activation values during the forward pass and one set of error terms during the backward pass. Convolutional layers tend to have far fewer weights than fully connected layers, both because there are far fewer connections and because, by design, many connections share the same weight.

Video classification tasks often use a structure called a *recurrent neural network* (RNN). An RNN is made up of multiple layers with recurrent (i.e. feedback) connections, called recurrent layers, such that a static unrolling of the RNN would be a very deep network with shared weights between some of the layers. The *Long-Short Term Memory* (LSTM) layer [43] is one popular type of recurrent layers that is frequently used for vision and speech tasks to capture sequence information of the data [31, 90, 94]. The LSTM layer contains memory cells that "remember" knowledge of previous timestamps, and the memory is updated selectively at each timestamp, controlled by special gate functions. A common approach for using RNNs on vision tasks, such as video classification [31, 94] and image captioning [90], is to stack LSTM layers on top of CNN layers. The CNN layers serve as an encoder that converts each frame of a video into a feature vector and feeds the video as a sequence of feature vectors into the LSTM layers.

2.4.3 Latent Dirichlet allocation (LDA) for topic modeling

LDA is an unsupervised method for discovering hidden semantic structures (*topics*) in an unstructured collection of *documents*, each consisting of a bag (multi-set) of *words* [40]. LDA discovers the topics via word co-occurrence. For example, "Obama" is more likely to co-occur with "Congress" than "super-nova", and thus "Obama" and "Congress" are categorized to the same topic associated with political terms, and "super-nova" to another topic associated with scientific terms. Further, a document with many instances of "Obama" would be assigned a topic distribution that peaks for the politics topics. LDA learns the hidden topics and the documents' associations with those topics jointly. It is often used for news categorization, visual pattern discovery in images, ancestral grouping from genetics data, and community detection in social networks.

The LDA model is often trained with the collapsed Gibbs sampling (CGS) algorithm [40],

which is a type of Markov Chain Monte Carlo (MCMC) method that is used to produce random samples from a probability distribution that would otherwise be difficult to solve analytically.

Formally, let w_{ij} be the term in the i-th document's j-th word position, let θ_i be document i's K-dimensional topic vector, and let β_{kv} be the probability of term v occuring in topic $k \in \{1, \ldots, K\}$. Finally, let $z_{ij} \in \{1, \ldots, K\}$ be the (single) topic assigned to w_{ij}, e.g. "Obama" could be assigned to the "Politics" topic. We are given w_{ij} as input, and the goal is to find the posterior distribution of θ_i, β_{kv} under the LDA model

$$\prod_{i=1}^{N} \prod_{j=1}^{N_i} \mathbb{P}(w_{ij} \mid z_{ij}, \beta) \mathbb{P}(z_{ij} \mid \theta_i) \mathbb{P}(\theta_i) \mathbb{P}(\beta), \tag{2.3}$$

where $\mathbb{P}(\theta_i)$, $\mathbb{P}(\beta)$ are prior distributions over θ_i, β. When CGS is applied, the variables θ_i, β are integrated out, leaving z_{ij} as the only random variables. The z_{ij} are sampled from the K-category distribution

$$\mathbb{P}(z_{ij} \mid \mathbf{z} \setminus z_{ij}, \mathbf{w}),$$

where $\mathbf{z} \setminus z_{ij}$ is all other z's apart from z_{ij}, and $\mathbf{w}$ is the set of all terms. To compute this distribution, we need to maintain two integer tables as the parameter data, a `document-topic` table and a `word-topic` table that count how many z_{ij} belong to each (document,topic) pair and each (word,topic) pair respectively.

To implement a data-parallel LDA, we can partition all the documents evenly across multiple worker machines. In every iteration, each worker goes through its assigned documents and makes adjustments to the `document-topic` table and the `word-topic` table accordingly. Algorithm 2 shows the basic structure of data-parallel LDA.

Algorithm 2 Data-parallel LDA with a parameter server.

```
while not converged do
    for each word w in document d do
        WordTopicVector = ps.Read(w-th row of WordTopicTable)
        DocumentTopicVector = ps.Read(d-th row of DocumentTopicTable)
        Compute the NewTopic assignment of this word occurrence with CGS
        OldTopic ← previous topic assignment
        TopicVectorUpdate[NewTopic] = 1
        TopicVectorUpdate[OldTopic] = -1
        ps.Update(w-th row of WordTopicTable, TopicVectorUpdate)
        ps.Update(d-th row of DocumentTopicTable, TopicVectorUpdate)
        Store the NewTopic assignment as local state
    end for
    ps.Clock()
end while
```

2.4.4 PageRank

PageRank is a link analysis algorithm, and it assigns a numerical weighting (i.e., *rank*) to each node of a hyperlinked set of documents (e.g., pages in the World Wide Web), with the purpose of "measuring" its relative importance within the set [11]. The rank of each node is decided as:

$$Rank(i) \quad = \quad (1-d) + d(\sum_{j \in Neighbor(i)} \frac{Rank(j)}{Degree(j)}) \tag{2.4}$$

where d is a damping factor and is usually set to a constant number of 0.85.

The PageRank algorithm is frequently used as the benchmark application by many graph computing engines, such as GraphLab [60], Pregel [62], GraphX [38], as well as parameter server systems, such as LazyTable [25]. To implement a data-parallel PageRank application with the parameter server model, we can partition the edges of the input graph evenly among all workers and keep the page-ranks in the parameter server. The structure of the PageRank application is shown in Algorithm 3. In every iteration, each worker goes through all edges in its sub-graph and updates the page-rank of the destination node according to the current page-rank of the source node. We repeat this procedure until the model converges.

Algorithm 3 PageRank implementation on a parameter server.

```
while not converged do
    for each edge from src to dst do
        SrcRank = ps.Read(src)
        DstRankContributionFromSrc = d × SrcRank / SrcDegree
        DstRankUpdate = DstRankContributionFromSrc − PrevDstRankContributionFromSrc
        ps.Update(dst, DstRankUpdate)
        Store the new DstRankContributionFromSrc as local state
    end for
    ps.Clock()
end while
```

Chapter 3

IterStore: Exploiting Iterativeness for Efficient Parallel ML

This chapter explores an opportunity created by the iterative-ness characteristic of the machine learning tasks: knowable repeating patterns of access to the shared state. Often, each ML worker processes its portion of the input data in the same order in each iteration, and the same subset of parameters is read and updated any time a particular data item is processed. So, each iteration involves the same pattern of reads and writes to the shared state.

Knowledge of these patterns can be exploited to improve efficiency, both within a multi-core machine and for communication across machines. For example, within a machine, state used primarily by one thread can be placed in the memory NUMA zone closest to the core on which it runs, while significantly write-shared state can be replicated in thread-private memory to reduce lock contention and synchronized only when required. Naturally, cross-machine overheads can be reduced by appropriate partitioning and prefetching of parameter state. But, one can gain further efficiency by constructing static structures for both the servers' and workers' copies of the shared state, rather than the general dynamic structures that would typically be used, organized according to the prefetch and update batches that will occur each iteration so as to minimize marshaling costs. Indeed, even those batch prefetch and update requests can be constructed statically.

Using three real and oft-studied ML applications (matrix factorization, LDA, and PageRank), we experiment extensively with a broad collection of iteration-aware specializations in a parameter server (called IterStore) that was originally designed assuming arbitrary access patterns. Altogether, we find that the specializations reduce per-iteration runtimes by 33–98%. Measuring their individual impacts shows that different specializations are most significant for different ML applications, based on their use of shared state, and that there can be synergistic dependencies among them. For example, informed prefetching is crucial for all of the applications, but must be coupled with static pre-marshaled cache structures to achieve high performance on PageRank.

We also evaluate costs associated with attempting to exploit iterative-ness. Naturally, the work of detecting and processing per-iteration patterns is part of an application's overall execution time. We find that most of the specializations compensate for those costs, though

cross-machine partitioning surprisingly sometimes does not. We also find that, although capturing the patterns during execution of the first iteration does work, it is more efficient if the application instead does an explicit *virtual iteration* that performs the reads and updates without affecting any state. Finally, we find that the benefits of the specializations are robust to imperfect information about the pattern, as might occur when converged values are no longer modified or responsibility for processing some input data items is shifted to another thread.

3.1 Background and prior approaches

In prior state-of-art approaches, as is described in Section 2.3, a parameter server system is most similar to a generic distributed key-value store, where the parameter data is stored as a collection of key-indexed rows, managed by a distributed cluster of servers. This design, however, is often an overkill for iterative ML applications. We find many iterative ML applications, including all four applications described in Section 2.4, have the property that the same (or nearly the same) sequence of accesses is applied to the parameter data every iteration, as is illustrated in Figure 3.1. This is because, for these applications, each worker processes its portion of the input data in the same order in each iteration, and the same subset of parameter data is read and updated any time a particular data item is processed. So, each iteration involves the same pattern of reads and writes to the parameter data. We call this property *iterative-ness*.

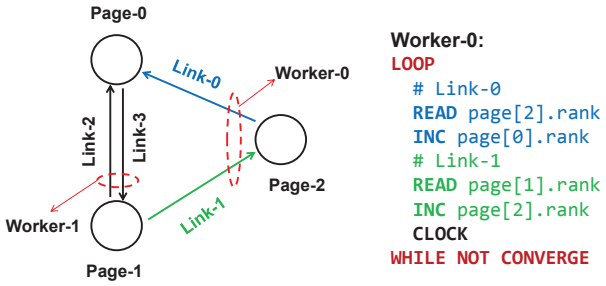


Figure 3.1: Iterativeness in PageRank. The graph contains three pages and four links. Each of the two workers is assigned with two links. In every iteration, each worker goes through its assigned links. For each link, the worker reads the rank of the source page and updates the rank of the destination page. The same sequence of accesses repeats every iteration.

The iterative-ness property creates an opportunity for efficiency: when per-worker sequences of reads and updates repeat every iteration, they can be known in advance and used to reduce the overheads associated with maintaining the data in the parameter server. The rest of this section will discuss approaches to identifying the sequences and a variety of ways that they can be exploited.

3.2 Exploiting iterative-ness for performance

The iterative-ness discussed above creates an opportunity: when per-worker sequences of reads and updates repeat each iteration, they can be known in advance and used to reduce the overheads associated with maintaining the shared state. This section discusses approaches to identifying the sequences and a variety of ways that they can be exploited.

3.2.1 Obtaining per-iteration access sequence

There are several ways that a parameter server can obtain the access sequences, with different overheads and degrees of help from the application writer. Of course, an ideal solution would have no overhead and need no help, but realistic options are non-optimal on one or both of these axes.

At one end of the spectrum would be completely transparent detection, in which the parameter server gathers the pattern between pairs of `Clock` calls. Although this seems straightforward, it is not for two primary reasons. First, many ML applications use the parameter server before beginning to iterate, such as to initialize parameter values appropriately. Because the initialization access pattern likely does not match the per-iteration pattern, and may involve several calls to `Clock`, identifying the right pattern would require comparing inter-`Clock` patterns until a repeat is found. Second, not every iterative ML application is perfectly repetitive, so such a repeat may never be found, either because there are no exact matches or perhaps even no significant repetitiveness at all. Third, exploitation of repeating patterns can only begin after they are known, so a significant portion of the application may be executed inefficiently until then. And, of course, the shared state must be retained in any conversion of the parameter server to a more efficient configuration. These three issues make the fully transparent approach high overhead and not robust; we do not consider it further.

Instead, we explore two options that involve some amount of assistance from the application programmer, illustrated in Figure 3.2: explicit reporting of the sequence (right-most pseudo-code) and explicit reporting of the iteration boundaries (middle pseudo-code). Both options are described in terms of the access sequence being reported once, at the beginning of the application. But, detecting and specializing can be repeated multiple times in an execution, if the access pattern changes dramatically. Section 3.4.6 shows that doing so is unnecessary for moderate access pattern changes.

Explicit virtual iteration. The first, and most efficient, option involves having the application execute what we call a *virtual iteration*. In a virtual iteration, each application worker reports their sequence of parameter server operations (`Read`, `Update`, and `Clock`) for an iteration. The parameter server logs the operations and returns success, without doing any reads or writes. Naturally, because no real values are involved, the application code cannot have any internal side-effects or modify any state of its own when performing the virtual iteration. So, ideally, the code involved in doing an iteration would be side-effect free (at least optionally) with respect to its local state; our example applications accommodate this need. If the per-iteration code normally updates local state, but still has repeating patterns, then a second side-effect free version of the code would be needed for executing

```
// Original
InitParams()
ps.Clock()
do {
  DoIteration()
  ps.Clock()
} while (not stop)
```

```
// Gather in first iter
InitParams()
ps.Clock()
do {
  if (first iteration)
    ps.StartGather(real)
  DoIteration()
  if (first iteration)
    ps.FinishGather()
  ps.Clock()
} while (not stop)
```

```
// Gather in virtual iter
ps.StartGather(virtual)
DoIteration()
ps.FinishGather()
InitParams()
ps.Clock()
do {
  DoIteration()
  ps.Clock()
} while (not stop)
```

Figure 3.2: Two ways of collecting access information. The left-most pseudo-code illustrates a simple iterative ML program flow, for the case where there is a `Clock` after each iteration. The middle pseudo-code adds code for informing the parameter server of the start and end of the first iteration, so that it can record the access pattern and then reorganize (during `ps.FinishGather`) to exploit it. The right-most pseudo-code adds a virtual iteration to do the same, re-using the same `DoIteration` code as the real processing.

the virtual iteration to expose them. Moreover, because no real values are involved, the application's sequence of parameter server requests must be independent of any parameter values read. Note that if the sequence were to depend on the parameter values, then the sequence would likely vary from iteration to iteration. Thus, the independence property is expected in applications that meet our overall requirement that the request sequence is (roughly) the same from iteration to iteration, and indeed, our example applications satisfy this property.

A virtual iteration can be very fast, since operations are simply logged, and does not require any inefficient shared state maintenance. In particular, the virtual iteration can be done before even the initialization of the shared state. So, not only is every iteration able to benefit from iterative-ness specializations, no transfer of state from an inefficient to an efficient configuration is required. Moreover, the burden of adding a virtual iteration is modest—only ≈10 lines of annotation code for our ML applications.

Explicit identification of iteration boundaries. If a virtual iteration would require too much coding effort, an application writer can instead add start and end breadcrumb calls to identify the start and end of an iteration. Doing so removes the need for pattern recognition and allows the parameter server to transition to more efficient operation after just one iteration. This option does involve some overheads, as the initialization and first iteration are not iterative-ness specialized, and the state must be retained and converted as specializations are applied. But, it involves minimal programmer effort.

3.2.2 Exploiting access information

This section describes the detailed parameter server specializations afforded by the knowledge of repeating per-iteration access patterns.

Data placement across machines. When parameter state is sharded among multiple machines, both communication demands and latency can be reduced if parameters are

co-located with the computation that uses them. As others have observed, the processing of each input data item usually involves only a subset of the parameters, and different workers may access any given parameter with different frequencies. Systems like GraphLab [60, 37] exploit this property aggressively, partitioning both input data and state according to programmer-provided graphs of these relationships. Even without such a graph, knowledge of per-iteration access patterns allows a subset of this benefit. Specifically, given the access sequences, the system can decide in which machine each parameter would best be stored by looking at the access frequency of the workers in each machine.

Data placement inside a machine. Modern multi-core machines, particularly larger machines with multiple sockets, have multiple NUMA memory zones. That is, a memory access from a worker thread running on given core will be faster or slower depending on the "distance" to the corresponding physical memory. For example, in the machines used in our experiments (see Section 3.4), we observe that an access to memory attached to a different socket from the core can be as much as 2.4× slower than an access to the memory attached the local socket. Similar to the partitioning of parameters across machines, knowledge of the access sequences can be exploited to co-locate worker threads and data that they access frequently to the same NUMA memory zone.

Static per-thread caches. Caching usually improves performance. Beyond caching state from remote server shards, per-worker-thread caching can improve performance in two ways: reducing contention between worker threads (and thus locking overheads on a shared client cache) and reducing accesses to remote NUMA memory zones. But, when cache capacity is insufficient to store the whole working set, requiring use of a cache replacement policy (e.g., LRU), we have observed that per-thread caches hurt performance rather than help. The problem we observe is that doing eviction (including propagating updates) slows progress significantly, by resulting in much more data propagation between data structures than would otherwise be necessary. Given the access patterns, one can employ a *static cache policy* that determines beforehand the best set of entries to be cached and never evicts them. The size of this per-thread cache can also be optimized—see Section 3.3.6.

Efficient data structures. The client library is usually multi-threaded, with enough application worker threads to use all cores as well as background threads for communication, and the parameter server is expected to store arbitrary keys as they are inserted, used, and deleted by the application. As a result, a general-purpose implementation must use thread-safe data structures, such as concurrent hash maps [45] for the index. However, given knowledge of the access patterns, one can know the full set of entries that each data structure needs to store, allowing use of more efficient less-general data structures. For example, one can instead use non-thread-safe data structures for the index and construct all the entries in a preprocessing stage, as opposed to inserting the entries dynamically. Moreover, a data structure that does not require support for insertion and deletion can be organized in contiguous memory in a format that can be copied directly to other machines, reducing marshaling overhead by eliminating the need to extract and marshal each value one-by-one. As noted earlier, the first iteration may not provide perfect information about the pattern in all subsequent iterations. To retain the above performance benefits while preserving correctness, one can fall back to using a thread-safe dynamic data structure solely for the part of the pattern that deviates from the first iteration, as discussed in

Section 3.3.3.

Prefetching. Under BSP, each worker thread must use updated values after each `clock`, requiring that each cached value be refreshed before use in the new iteration. Naturally, read miss latencies that require fetching values from remote server shards can have significant performance impact. Prefetching can help mask the high latency, and of course knowing the access pattern maximizes the potential value of prefetching. One can go beyond simply fetching all currently cached values by constructing large batch prefetch requests once and using them each iteration, with multiple prefetch requests used to pipeline the communication and computation work. So, for example, a first prefetch request can get values used at the beginning of the iteration, while a second prefetch request gets the values used in the remainder of the iteration.

3.2.3 Limitations

Many of the IterStore specializations reply on the knowledge of the application's access information. Since the collected access information is used only as a hint, knowing the exact operation sequence is not a requirement for correctness, but a performance optimization. For all of our example applications, including image classification, video classification, matrix factorization, LDA, and PageRank, their parameter data access repeats every iteration and can be easily collected via a virtual iteration. But for some applications with sparse training data that is sampled randomly every clock, the parameter data access of each clock might depend on the specific training data samples. For those applications, users can either report all the parameter data accesses that could possibly happen (i.e., report extra information), or report only the parameter data access that would definitely happen (i.e., report incomplete information). Both approaches will provide correct training result, but the performance might differ. Section 3.4.6 evaluates the impact of reporting extra or incomplete information on the performance of IterStore, and we find usually it is better to report extra information.

3.3 Implementation

IterStore is a distributed parameter server that maintains globally shared data for applications. It employs our optimizations (optionally) when informed of the per-iteration access patterns. Although our descriptions and experiments in this paper focus on the traditional BSP execution model, the same optimizations apply to the more flexible Stale Synchronous Parallel (SSP) [22, 42] model supported by LazyTable, and we observe approximately the same relative improvements; that is, the benefits of exploiting iterative-ness apply equally to SSP.

3.3.1 System architecture

The design of IterStore follows the standard parameter server architecture described in Section 2.3. The parameter data in IterStore is managed as a collection of *rows* indexed

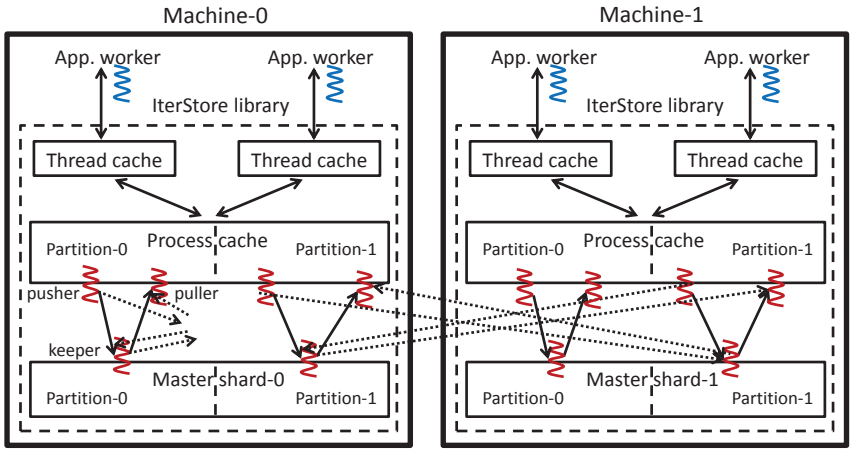


Figure 3.3: IterStore with two partitions, running on two machines with two application threads each.

by *keys*. A row is a user-defined data type that should be defined with an associative and commutative aggregation operation, allowing updates to be applied in any order. In most applications, the aggregation is simply an addition-assignment operation ($+=$). Some of the IterStore specializations (such as the "efficient data structures" one) requires the row to be a dense type, such as an array, so that the values of multiple rows can be easily packed in a single message, without any extra marshalling work.

The IterStore architecture is depicted in Figure 3.3. The distributed application program usually creates one process on each machine and each process links to one instance of the IterStore library. Each IterStore instance stores one shard of the master version of the data in its *master store*. The data is not replicated, and fault tolerance is handled by checkpointing [25]. The application threads don't directly access the master data. Instead, they access the *process cache* that is shared by all threads in the same instance. One level above the process cache, each application thread has a private *thread cache*. For simplicity, the current implementation assumes that each physical machine has a single IterStore instance, and that each process cache is large enough to cache all values used by local worker threads. To accommodate cases where there is not that much memory capacity available, the process cache would store only a subset of said values, exploiting the static cache approach used for thread caches; the iteration-aware specializations discussed in this section would still be effective.

IterStore supports both the BSP model and the SSP model. To reduce communication overhead, IterStore batches updates in thread caches and process caches, and propagates them to master stores at the edge of one clock. We attach a data age field to each process cache (and also thread cache) row, so that readers can detect out-of-date data and refresh from master stores.

IterStore performs communication asynchronously with three types of background threads: *keeper* threads manage the data in master stores; *pusher* threads move data from process caches to master stores by sending messages to keeper threads; *puller* threads move data from master stores to process caches by receiving messages from keeper threads.

To achieve high communication throughput, IterStore globally divides the key space (by the hash of the key) into M partitions; each of the N IterStore machines manages the rows falling in different partitions with different threads in different data structures. For each of the M partitions, each machine launches one keeper thread, one pusher thread, and one puller thread dedicated to the rows of that partition. Each machine's shard of the master data is divided into M pieces in its master store, thus $N \times M$ pieces in total. (The sharding scheme is discussed in Section 3.3.4.) The keeper thread of `partition-j` of `machine-i` exchanges messages with the pusher thread and puller thread of `partition-j` of all N machines.

3.3.2 Access information gathering

IterStore supports both ways of gathering access information described in Section 3.2.1, allowing the application to report the access information either in a virtual iteration or in a real iteration (e.g., the first). When `finish_gather()` is called, IterStore summarizes the gathered information, including the generation of per-thread and machine-wide access rates of all rows, and applies the specializations accordingly.

3.3.3 Efficient data structures

Because the application can use arbitrary keys to index the stored data, IterStore uses hash maps to store the rows (or their pointers) in master stores, process caches, and thread caches. If not using the virtual iteration approach, IterStore has to start with empty data structures and have the worker threads insert rows dynamically. For master stores and thread caches, dynamic row insertion might not be a big problem, because these data structures are only accessed by one thread. Process caches, however, are shared. Each process cache partition is accessed by the application worker threads, pusher thread, and puller thread. To support dynamic row insertion, we have to either always grab a global lock on the index before accessing or use a thread-safe data structure, and both approaches are expensive. Moreover, when the background threads exchange updates between process caches and master stores, they need to go through every updated row in their partitions, incurring significant marshaling and update costs.

When IterStore is informed of the set of rows to be stored, it can construct a static data structure with all the rows in it and an immutable index, so that only per-row locks are needed. When the size of each row is fixed (true for all our benchmarks), IterStore will allocate contiguous memory to store the rows for each storage module. Row keys are stored together with the values, making the data structure self-explanatory, so that it can be sent in bulk without per-row marshaling work. The receiving sides still need to unmarshall the message row-by-row, because the layout of the receiver might not be the same as the sender. For random key access, IterStore uses the hash maps to store the mapping from row keys to memory indices, but these hash maps are immutable and can be non-thread-safe.

To deal with the situation where the application needs to access unreported rows (i.e., the current iteration has deviated from the first iteration's pattern), IterStore uses two sets of data structures for each process cache partition, a static part and a dynamic part. The

static part is what we described above, and the dynamic part uses a thread-safe hash map to store the additional unreported rows. When a thread fails to find a desired row in the static part, it checks the dynamic part, creating a new row if not found there either.

3.3.4 Data placement across machines

IterStore determines the partition ID of a row by the hash of its key, but the master version of each row can still be stored in any of the master stores among N machines. Without the access information from the application, IterStore determines the machine ID of each row using another hash, so that all rows will be stored uniformly on all $N \times M$ master stores. With the access information, IterStore assigns each row to machines in a way that minimizes cross-machine traffic, decided in the preprocessing stage, as described next.

Because of the process caches, each machine sends to master stores at most one read and one update request for each row, per clock. Since the *batched access frequency* is either one or zero,[1] IterStore simply places each row on any one of the machines that has access to it. When there are multiple machines accessing the same row, IterStore greedily chooses the machine that has the least number of rows.

The placement decision is accomplished in a distributed manner, by a metadata storage module of the master stores. The master stores of `partition-i` decide and keep the machine placement of rows that are hashed to `partition-i`. For a particular `row-k` in `partition-i`, we choose which master store makes the decision by hashing its key. Suppose the hash is p, in the preprocessing stage, all machines send their batched access frequency of `row-k` to `machine-p`, which chooses the machine to store it based on the frequencies. Suppose the chosen one is `machine-q`, it would inform all machines to send further `READ` and `UPDATE` requests to `machine-q`. Each machine maintains a local mapping of these placement decisions. Since each row is usually an array of parameter values, the storage for the mapping will be much smaller than the size of the parameter data.

3.3.5 Data placement inside a machine

In our implementation, most of the memory accesses are by application worker threads to thread caches, pusher/puller threads to process caches, and keeper threads to master stores. While many systems assume that access latencies to all memory regions are the same and allocate memory blindly, it is beneficial to allocate memory close to the execution in modern multi-core systems [5]. Because the data structures are allocated statically in the preprocessing stage, IterStore can co-locate the data structures in the same NUMA zones with the thread accessing them most.

Suppose each machine has C CPU cores and Z NUMA zones. We encourage the application writer to create one process per machine and C application threads per process. IterStore will divide the key space into $\frac{C}{2}$ partitions, so that with three background threads per partition, IterStore can (empirically) fully utilize all CPU cores when the application

[1] In case of iterations with multiple clocks, where the work done varies between the clocks of an iteration, the batched access frequency of `machine-i` to `row-j` is the number of clocks in which it accesses `row-j`, divided by the total number of clocks.

threads are blocked for communication. For each set of $\frac{C}{Z}$ cores in a NUMA zone, we have $\frac{C}{Z}$ application threads and the background threads of $\frac{C}{2Z}$ partitions run on these cores and only allocate memory in the local NUMA zone. Note that each thread cache is allocated by its corresponding application thread, each process cache partition by its pusher thread, and each master store partition by its keeper thread. As a result, all data structures can be accessed locally by the threads accessing them most often.

3.3.6 Contention and locality-aware thread caches

Thread caches, in addition to a shared process cache, can improve parameter server performance in two ways: reducing contention and increasing memory access locality. To assist with experimentation, the capacity of each thread cache is specified as an input parameter, which might otherwise be determined by dividing the total amount of memory remaining for the application (after initialization, process cache allocation, and master store allocation) by the number of threads per machine.

When access information is not provided, IterStore defaults to an LRU policy for cache eviction. However, we find there is sufficient overhead in doing row eviction and LRU list maintenance that dynamic thread caches often hurt performance rather than help. To avoid this overhead, IterStore uses a static cache policy when it is provided with the access information of the application; IterStore determines the set of rows to be cached in the preprocessing stage and never evicts them. A first small number of rows address contention and the remainder address locality. The remainder of this section describes the selection process for each.

As described in Section 3.3.1, each process cache partition is accessed by all application threads, one pusher thread, and one puller thread. IterStore explicitly uses thread caches to reduce contention on the process cache. In the preprocessing stage, IterStore estimates the contention probability of each row accessed by each application thread. If the estimated probability is higher than a predetermined threshold, IterStore caches that row in the thread cache.

We use the following model to estimate the contention probability of a row. We define the *access frequency* $\texttt{AF}_i^{(j)}$ as the number of times (0 or 1) that `thread-i` accesses `row-j`, divided by the number of times that `thread-i` accesses any row, in one iteration. If we assume the time it takes to access each row is the same, access frequency equals *access probability* $\texttt{AP}_i^{(j)}$, which is the probability that at a given point of time, `thread-i` is accessing `row-j`. Consider a `row-j` that is accessed by `thread-i`, and let $n_j \geq 2$ be the number of threads that access `row-j`. Let $\texttt{CP}_i^{(j)}$ be the probability that the access of `thread-i` to `row-j` overlaps with one or more accesses to `row-j` by other threads. Our goal is to use thread caches to reduce all $\texttt{CP}_i^{(j)}$ to below a target bound `CPB`. To do this, we calculate a threshold $\texttt{AFT}_i^{(j)}$ for each access frequency such that if some access frequency $\texttt{AF}_i^{(j)}$ is larger than $\texttt{AFT}_i^{(j)}$ we will have `thread-i` cache `row-j` in its thread cache. Under the model, it suffices to set $\texttt{AFT}_i^{(j)} = \frac{\texttt{CPB}}{n_j - 1}$ for all i and j, as shown in Equation 3.1.

$$
\begin{aligned}
\mathtt{CP}_i^{(j)} = & \quad 1 - \prod_{i' \neq i} (1 - \mathtt{AP}_{i'}^{(j)}) = 1 - \prod_{i' \neq i} (1 - \mathtt{AF}_{i'}^{(j)}) \\
\leq & \quad 1 - (1 - \max_{i' \neq i} \mathtt{AF}_{i'}^{(j)})^{n_j - 1} \leq 1 - (1 - \max_{i' \neq i} \mathtt{AFT}_{i'}^{(j)})^{n_j - 1} \qquad (3.1) \\
= & \quad 1 - (1 - \frac{\mathtt{CPB}}{n_j - 1})^{n_j - 1} \approx (n_j - 1) \times \frac{\mathtt{CPB}}{n_j - 1} = \mathtt{CPB}
\end{aligned}
$$

The number of rows cached as a result of these thresholds is bounded by $\max_j(1/\mathtt{AFT}_i^{(j)})$, which is no greater than $\frac{n-1}{\mathtt{CPB}}$, where n is the total number of threads in this machine. The remainder of available IterStore thread cache capacity is used to reduce the amount of remote memory access. As described in Section 3.3.5, memory access from application threads to process caches is not necessarily local. The static cache selection policy picks additional rows in decreasing order of access frequency, skipping those stored in the local memory NUMA zone at the process cache level.

3.3.7 Prefetching

In the preprocessing stage, each IterStore process cache partition summarizes the union set of rows that application worker threads on that machine read and uses it to construct prefetch requests. At the beginning of each clock, a prefetch module sends these pre-constructed prefetch messages to master stores, requesting a refresh of the rows.

To better pipeline the prefetch with computation work, we can do *pipelined prefetch* when we know the ordering that each row is read. Each IterStore process cache partition creates two prefetch requests for each clock, controlled by an *early-ratio* parameter. The *early prefetch* requests contain the rows that satisfy the first early-ratio `READ` operations, and the *normal prefetch* requests contain the other rows used. The early prefetch requests are prioritized over the normal ones, so that we can reduce the waiting time at the beginning of each clock. We generally observe that an early-ratio of 10% works well, and Section 3.4.5 describes some experimental results.

3.4 Evaluation

This section describes our experiment results, showing that exploiting iterative-ness significantly improves efficiency, quantifying individual benefits of the specializations, and showing that they are robust to some deviations from expected per-iteration access patterns.

3.4.1 Experimental setup

Hardware Information. All experiments use an 8-node cluster of 64-core machines. Each node has four 2-die 2.1 GHz 16-core AMD® Opteron 6272 packages, with a total of 128GB of RAM and eight memory NUMA zones. The nodes run Ubuntu 12.04 and are connected via an Infiniband network interface (40Gbps spec; ≈13Gbps observed).

Application benchmarks. We use the following three example applications: PageRank, matrix factorization (MF), and latent Dirichlet allocation (LDA). For MF, we use the

Netflix dataset, which is a 480k-by-18k sparse matrix with 100m known elements. The application is configured to factor it into the product of two matrices with rank 1000. For LDA, we use the *NYTimes* dataset [88], containing 100m tokens in 300k documents with a vocabulary size of 100k, and we configure the application to generate 1000 topics. For PageRank, we use the *twitter-graph* dataset [53], which is a web graph with 40m nodes and 1.5b edges. In order to create more locality, for the PageRank benchmark, we group the links going to the same destination node together, so that they are likely to be assigned to the same worker. Similarly, we group the elements in the same row together for MF, and group the tokens in the same document together for LDA. For each application, the virtual iteration annotation requires only ≈10 lines of code.

IterStore setup. We run one application process on each machine. Each machine creates 64 computation threads and is linked to one instance of IterStore library with 32 partitions. We assume each machine has enough memory to not need replacement in its process cache.

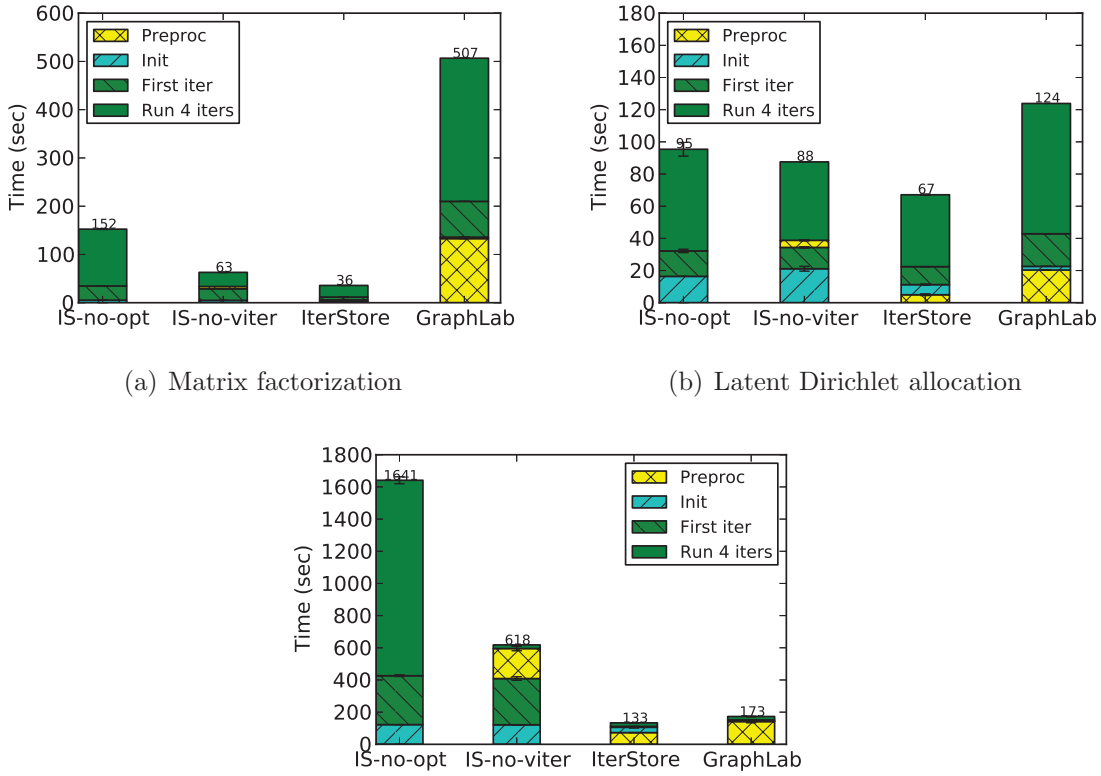
Experiment methodology. Note that our specializations do not change the per-iteration convergence rate, but only the per-iteration execution time is affected. Therefore, we report on the time required to complete a given number of iterations in the bulk of our analysis. We run each experiment at least thrice (except for Figure 3.5), report arithmetic means, and use error bars (often too small to see) to show standard deviations.

3.4.2 Overall performance

Figure 3.4 shows performance for each of the three ML applications running on four system setups: IterStore without any iterative-ness specializations ("IS-no-opt"), IterStore with all of the specializations and obtaining the access pattern from the first real iteration ("IS-no-viter"), IterStore with all specializations and use of a virtual iteration ("IterStore"), and GraphLab [60, 37].[2] Comparing different ones of the four setups allow evaluation of different key aspects, including how much benefit IterStore realizes from iterative-ness specializations, how much more efficient using a virtual iteration is than obtaining the patterns in the first iteration, and, to put the numbers in a broader context, how IterStore compares to a popular efficient system with and without the specializations.

Each bar shows the time required for the application to initialize its data structures and execute 5 iterations, broken into four parts: preprocessing, initialization, first iteration, and next four iterations. Preprocessing time includes gathering the access information and setting up data structures according to them, which is zero for IS-no-opt; GraphLab's preprocessing time is its graph finalization step, during which it uses the application-supplied graph of dependencies between data-items to partition those data-items across machines and construct data structures. Initialization includes setting initial parameter values as well as some other application-level initialization work. The first iteration is shown separately, because it is slower for IS-no-viter and because that setup performs preprocessing after the first iteration; all five iterations run at approximately the same speed for the other three

[2]The GraphLab code was downloaded from `https://github.com/graphlab-code/graphlab/`, with the last commit on Jan 27, 2014.



(a) Matrix factorization

(b) Latent Dirichlet allocation

(c) PageRank

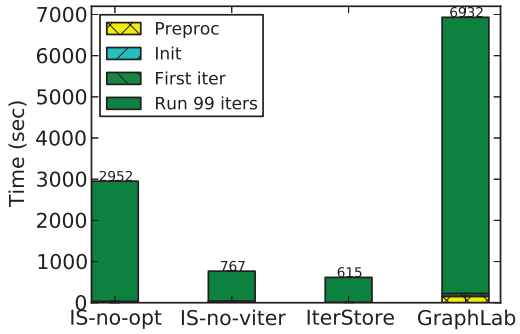
Figure 3.4: Performance comparison, running 5 iterations.

systems.

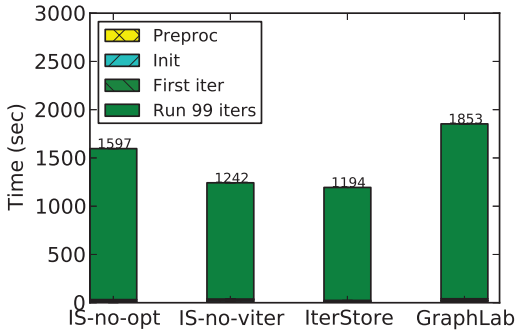
The results show that the specializations decrease per-iteration times substantially (by 33–98%). Even when accounting for preprocessing times, exploiting iterative-ness reduces the time to complete five iterations by 29–82%. As more iterations are run, the improvement can approach the per-iteration improvement, although early convergence of some parameters can change the access pattern over time and thereby reduce the effectiveness of exploiting iterative-ness. Figure 3.5 shows performance for the same workloads completing 100 iterations, instead of just five. For these applications, greater benefit is achieved with more iterations, despite some changes in the pattern, because the preprocessing work is better amortized. Section 3.4.6 explores further the sensitivity of exploiting iterative-ness to such changes. Note that the entire range of few to 100+ iterations can be of interest in practice, depending on the available time budget and the convergence rate.

The results (more easily seen in Figure 3.4) also show that using a virtual-iteration is more efficient than collecting patterns in the first real iteration, because the latter causes the initialization and first iteration to be inefficient. Moreover, doing preprocessing after the first iteration requires copying the parameter server state from the original dynamic data structures to the new static ones, making the preprocessing time longer.

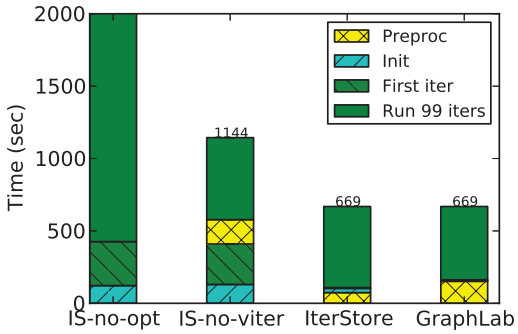
With optimizations and virtual-iteration turned on, IterStore out-performs GraphLab



(a) Matrix factorization



(b) Latent Dirichlet allocation



(c) PageRank

Figure 3.5: Performance comparison, running 100 iterations. The "IS-no-opt" bar in the PageRank figure is cut off at 2000 sec, because it's well over an order of magnitude worse than the other three.

for all of the three benchmarks, even PageRank which fits GraphLab's graph-oriented execution style very well. For MF and LDA, IterStore out-performs GraphLab even without the optimizations, and by more with them. Note that we under-estimate GraphLab's initialization and preprocessing time in these results, because GraphLab does some of its preprocessing work together with its data loading. We are not showing the data loading time of these systems, because it is huge for GraphLab, due to the fact that GraphLab uses unpartitioned text file as input, while IterStore uses partitioned binary input.

Table 3.1 summarizes some features of the three benchmarks. We show the total number of rows, size of each row, and the average degree of each node when we express them with a sparse graph.[3] A graph is more sparse when it has a smaller average node degree. The PageRank benchmark gets huge benefit from our optimizations because its corresponding graph is very sparse, with a huge number of tiny rows, and both features cause our previous IterStore implementation to be inefficient.

[3]Though MF and LDA are not graphical applications, some frameworks, such as GraphLab, will still express the computation using graphical structures, and here we use the same graphical structures as those used in GraphLab toolkits to calculate the average node degree.

Application	# of rows	Row size (bytes)	Average node degree
MF	500k	8k	200
LDA	400k	8k	250
PageRank	40m	8	75

Table 3.1: Benchmarks characteristics.

3.4.3 Optimization effectiveness break down

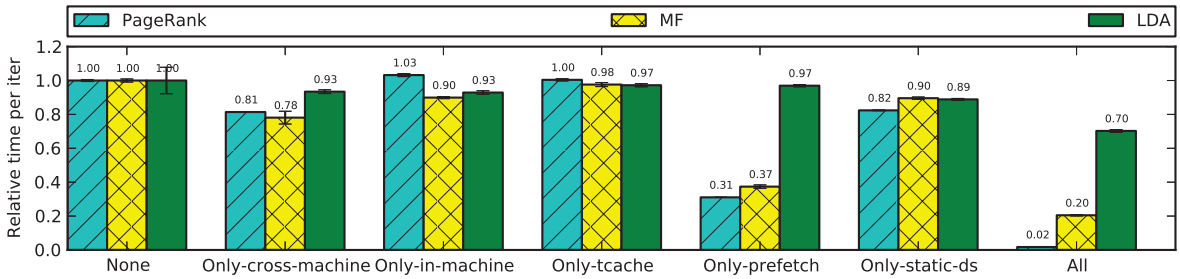


Figure 3.6: Turn on one of the optimizations.

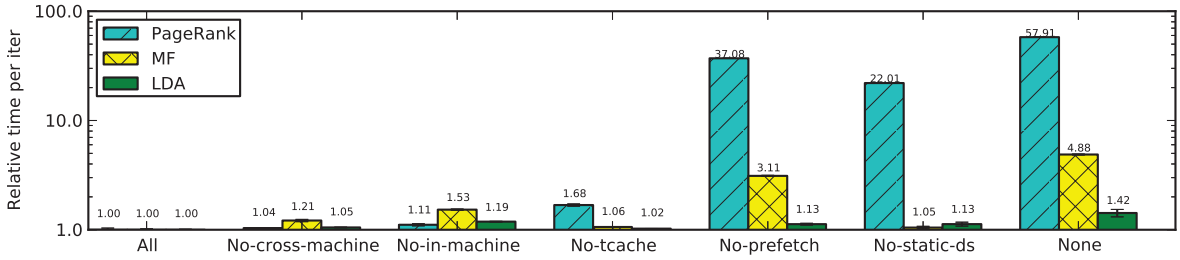


Figure 3.7: Turn off one of the optimizations.

The optimizations proposed in this paper can work independently. In this experiment, we break down the contribution coming from each of them. We employ two approaches to investigate that. First, we turn on each of the individual optimizations alone (Figure 3.6) and measure speed up compared with having none of the optimizations. Second, we turn off each of the optimizations (Figure 3.7), with the others on, and measure how much the execution is slowed down compared with having all of the optimizations. The results show that most of the optimizations are effective when deployed independently of others, and we also have several interesting findings. First, we find different optimizations are most significant for different applications, based on their use of shared states. PageRank benefits most from prefetching and static data structures, while MF benefits most from

prefetching and in-machine memory management. Second, we find there can be synergistic dependencies among these optimizations. Comparing the "cross-machine" speed up in both figures, we find its benefit becomes less when we have other optimizations. This is because there is less cross-machine communication overhead in the presence of other optimizations. We also find "prefetch" and "static-ds" offer much more speed up when applied collaboratively than individually. This is because, when both of them are present, IterStore local stores and master stores can exchange data in a batched form, with lower marshaling and unmarshaling overhead.

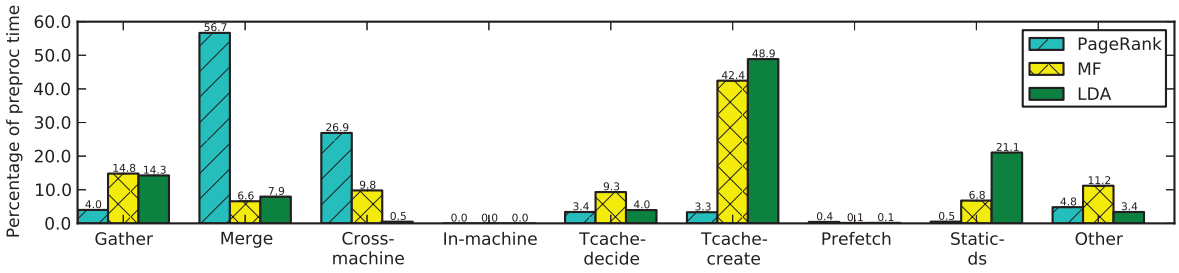
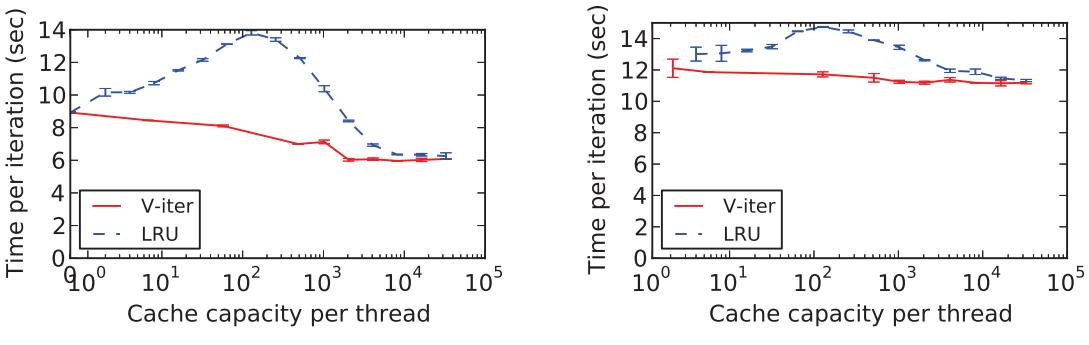


Figure 3.8: Preprocessing time break down.

Figure 3.8 shows the costs of each of these optimizations in the preprocessing stage, together with some costs shared by all of them. "Gather" stands for collecting information via virtual iteration, "Merge" stands for merging information from all threads in each machine, "Cross-machine" to "Static-ds" refer to the respective optimizations, and "Other" stands for the rest of time spent, mostly on synchronization inside the procedure. We break the time for thread cache preparation into a decision part and a creation part. The creation part is the cost of creating the cache data structures, which is inevitable in order to use the cache, and all "Static-ds" costs are for creating data structures for local stores. We find in MF and LDA, these two bars account for most of the preprocessing time.

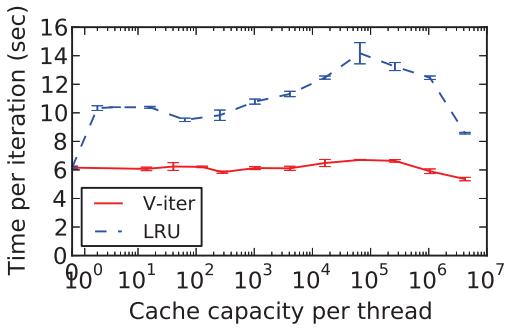
3.4.4 Contention and locality-aware caching

This set of experiments compares the performance of IterStore's static thread-caching policy with the LRU policy. We increase the capacity of the thread cache from zero to the size that is large enough to store all values accessed by a thread, and compare time per iteration, as shown in Figure 3.9. We are not showing the data point for cache capacity being zero in Figure 3.9(b), because it is too high. LDA has one summation row that needs to be updated whenever the topic assignment of any word is changed; when the cache capacity is zero, each iteration takes 493 seconds. The results show that IterStore's cache policy based on the known access pattern outperforms LRU. For all three benchmarks, time per iteration curves for LRU first go up and then go down, which can be explained as follows. When the cache capacity is small, there are few cache hits because most entries are already evicted before being accessed a second time, and the time per iteration increases with



(a) Matrix factorization

(b) Latent Dirichlet allocation



(c) PageRank

Figure 3.9: Comparing IterStore's static cache to LRU, varying the cache size (log scale).

increasing cache sizes due to the increasing overheads for cache insertions and evictions. On the other hand, when the cache capacity is sufficiently large, the benefit from increased hit rates outweighs these overheads, and the time per iteration decreases. For our static cache policy, even when we allocate only a tiny amount of memory for the cache, it almost always gives positive benefits from reduced contention and remote memory accesses. Moreover, it performs slightly, or in the case of PageRank significantly, better than LRU even when the thread cache capacity is large enough to store all values, for two reasons. First, even when there is enough capacity, IterStore's cache policy does not cache entries that have low contention probability and are stored in the local memory bank, because we find caching these entries results in unnecessary work for negligible benefit. Second, LRU cache needs to update its LRU list whenever an entry is accessed, so it has additional bookkeeping overhead.

3.4.5 Pipelined prefetching

Pipelined prefetch helps reduce the waiting time at the beginning of each clock by fetching the entries that are used by the first *early-ratio* `READ` operations. We evaluate the effectiveness of pipelined prefetch by comparing the time per iteration with different early-prefetch ratios

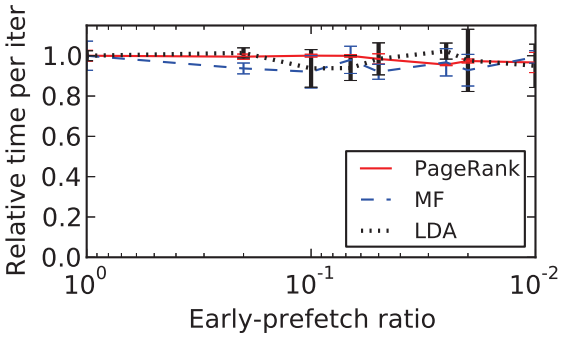
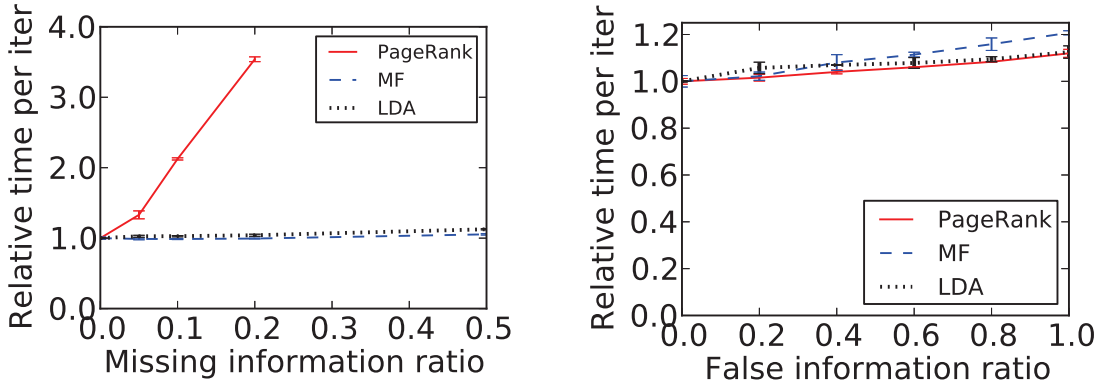


Figure 3.10: Pipelined prefetching.

(Figure 3.10). To emphasize the effect, we use a one Gigabit Ethernet network, which has less bandwidth than our default setting. The results show that pipelined prefetch reduces time per iteration in general, but there is high variance across different runs. That is because each client sends one early prefetch request and one normal prefetch request for every master store partition, and the replies containing entries used for the first few reads can be delayed by non-early-prefetch requests from other machines.

3.4.6 Inaccurate information



(a) Missing Information

(b) False Information

Figure 3.11: Influence of inaccurate information. Data point "missing information ratio is 0.2%" means that, the number of the missing unreported accesses is 20% of the number of the actual accesses (i.e., report 20% less). Data point "false information ratio is 0.2%" means that, the number of the false accesses is 20% of the number of the actual accesses (i.e., report 20% more).

This section investigates the sensitivity of IterStore's specializations when the parameter data accesses of the application differ from those initially reported, such as can occur

when an application's access pattern changes over time due to early convergence of some parameters or due to work migration for load balancing.

In a first set of experiments (Figure 3.11(a)), we have each application worker thread report an incomplete set of their accesses, which emulates the situation where these workers get the work of others during the run due to work reassignment. We fix the actual parameter data accesses the same, and have each setup omit a fraction of the accesses (chosen randomly) in the virtual iteration. We use the case when there is no missing access as the baseline, and compare the per-iteration times with different amounts of missing information. Results show that IterStore can work well with small amounts of missing information. PageRank is slowed down by 16% when we have 5% information missing. For MF and LDA, we observe almost no slow down even with only 50% of accesses reported. That is because most of the achieved speedups of these two applications come from informed prefetching, and because of the large number of application threads (64) in each machine that access overlapping subsets of values; as a result, reporting 50% of accesses from each thread is enough for IterStore to know most of the rows accessed by the machine.

In a second set of experiments (Figure 3.11(b)), we investigate the situation where the reported accesses are more than the actual ones, which can be caused by either work reassignment or by convergence of some parameters. We call the reported accesses that do not actually occur as *false information*. In this experiment, we use the accesses of another thread in a different machine as the false information. Again, we fix the actual accesses the same, and have each setup report different amounts of false information. The results show that reporting extra false information has minimal influence on the performance. The additional overhead comes from extra communication and occupation of thread caches.

3.4.7 Comparison with single thread baselines

Application	**Single-threaded**	**IterStore (512 threads)**	**Speedup**
MF	374.7 sec	6.02 sec	62×
LDA	1105 sec	11.2 sec	99×
PageRank	41 sec	5.13 sec	8×

Table 3.2: Time per iteration comparison of single-threaded non-distributed implementations using one machine versus IterStore using 8 machines, 512 cores and 512 threads.

We compare our parameter server implementation of MF, LDA, and PageRank against single thread baselines. We use open source GibbsLDA++[4] for LDA, which uses the same standard Gibbs sampling algorithm as our implementation on IterStore. For MF and PageRank, we could not find fast open source implementations using the exact same model and algorithm and thus implemented our own. These single-threaded implementations take full advantage of having all data in memory and require no locking, no network communication, and no inter-processor communication.

[4]http://gibbslda.sourceforge.net/

Table 3.2 compares the per-iteration time on our benchmarks. [5] As expected, IterStore does not show speedups linear in the number of cores because of the overhead of parallelization and communication. Perhaps surprisingly, however, it does show speeds at least linear in the number of machines. This argues for the use of distributed ML implementations with IterStore techniques, when quick completion times are important, even if the problem size is small enough to fit in the memory of a single node and good single-threaded code is available. The speed up for PageRank is smaller, because our single-threaded implementation assumes the webpage IDs are dense and stores their ranks in a vector instead of a hash map.

3.5 Additional Related Work

Frameworks for distributed ML have become an active area of systems research. Some rely on repeated executions of MapReduce for iterations [21, 13]. Some mix MapReduce iterations with multi-iteration in-memory state [98, 95]. Pregel applies the core BSP technique [36] to a long running iterative ML framework [63]. Percolator applies iterative refinement to a distributed key-value store with transactional updates and value triggers [72]. GraphLab [60] and PowerGraph [37] combine these abstractions with a graph-centric programming model, flexible dependency enforcement and optional asynchronous execution. And, several frameworks based on parameter servers have been developed [4, 73, 42]. Many of these systems could use the ideas explored in this paper, exploiting repeating per-iteration access patterns to improve their efficiency.

Other machine learning frameworks emphasize features outside the scope of this paper, such as out-of-core disk scheduling [54, 75]. Naiad generalizes iterative machine learning into a Dryad-like dataflow model for iterative and streaming computation of many forms [67].

Informed or application-aware caching and prefetching have been explored in file systems and databases [15, 70, 39, 57, 14]. In addition to exploring use of future access knowledge, researchers have explored a range of approaches to obtaining it, including explicit application calls [15, 70], compiler analysis [12], speculative execution [17, 34], and dynamic pattern detection [39, 57]. Some of our detection and exploitation of per-iteration patterns build on these ideas, adapting them to the specific characteristics of parameter servers for supporting parallel and distributed ML.

Data placement in NUMA systems was well-studied two decades ago for multiprocessor systems [10, 16]. The re-emergence of NUMA issues in multi-socket systems is well-known [5], bringing back the value of carefully placing data and threads to increase locality. Similar access latency asymmetry has also been noted in single-socket many-core chips [87, 86]. Exploiting iterative-ness, as explored in this paper, allows one to orchestrate such locality without the dynamic identification and re-allocation overheads usually found in general-purpose solutions.

[5] We admit that there are some better single-threaded performance numbers reported in the literature, which were potentially achieved on different hardware setups and/or different input data orders. For example, McSherry et al. [65] reported that their single-threaded PageRank implementation has a per-iteration time of 14 seconds with default input data order and 5.5 seconds with input data in Hilbert order, and we believe the Hilbert ordering optimization can also be applied to IterStore.

Chapter 4

GeePS: Exploiting Layered Computation for Efficient Deep Learning on Distributed GPUs

Neural network training is known to map well to GPUs [23, 52], but it has been argued that this approach is only efficient for smaller scale neural networks that can fit on GPUs attached to a single machine [20]. The challenges of limited GPU memory and inter-machine communication have been identified as major limitations.

This chapter describes GeePS, a parameter server system specialized for scaling deep learning applications across GPUs distributed among multiple server machines. Like previous CPU-based parameter servers [59, 25], GeePS handles the synchronization and communication complexities associated with sharing the model parameters being learned (the weights on the connections, for neural networks) across parallel workers. Unlike such previous systems, GeePS performs a number of optimizations specially tailored to making efficient use of GPUs, including pre-built indexes for "gathering" the parameter values being updated in order to enable parallel updates of many model parameters in the GPU, along with GPU-friendly caching, data staging, and memory management techniques.

GeePS supports data-parallel model training, in which the input data is partitioned among workers on different machines that collectively update shared model parameters (that themselves are sharded across machines). This avoids the excessive communication delays that would arise in model-parallel approaches, in which the model parameters are partitioned among the workers on different machines, given the rich dependency structure of neural networks [93]. Data-parallel approaches are limited by the desire to fit the entire model in each worker's memory, and, as observed by prior work [20, 29, 24, 93, 50, 91], this would seem to imply that GPU-based systems (with their limited GPU memory) are suited only for relatively small neural networks. GeePS overcomes this apparent limitation by assuming control over memory management and placement, and carefully orchestrating data movement between CPU and GPU memory based on its observation of the access patterns at each layer of the neural network.

Experiments show that single-GPU codes can be easily modified to run with GeePS and obtain good scalable performance across multiple GPUs. For example, by modifying

Caffe [47], a state-of-the-art open-source system for deep learning on a single GPU, to store its data in GeePS, we can improve Caffe's training throughput (images per second) by 13× using 16 machines. Using GeePS, less than 8% of the GPU's time is lost to stalls (e.g., for communication, synchronization, and data movement), as compared to 65% when using an efficient CPU-based parameter server implementation. In terms of image classification accuracy, GeePS's rate of improvement on 16 machines is 8× faster than the single-GPU optimized Caffe's. The training throughput achieved with just four GPU machines beats that reported recently for a state-of-the-art 108-machine CPU-only system (ProjectAdam) [20], and the accuracy improvement with just 16 GPU machines is 4× faster than what was reported for a 58-machine ProjectAdam configuration. Experiments with video classification via recurrent neural networks show similar results. Interestingly, in contrast with recent work [42, 59, 20], we find that for deep learning on GPUs, BSP-style execution leads to faster accuracy improvements than more asynchronous parameter consistency models, because the negative impact of staleness outweighs the benefits of reduced communication delays.

Experiments also confirm the efficacy of GeePS's support for data-parallel training of very large neural networks on GPUs. For example, results are shown for a 20 GB neural network (5.6 billion connections) trained on GPUs with only 5 GB memory, with the larger CPU memory holding most of the parameters and intermediate layer state most of the time. By moving data between CPU memory and GPU memory in the background, GeePS is able to keep the GPU engines busy without suffering a significant decrease in training throughput relative to the case of all data fitting into GPU memory.

4.1 High performance deep learning

4.1.1 Deep learning using GPUs

GPUs are often used to train deep neural networks, because the primary computational steps match their single-instruction-multiple-data (SIMD) nature and they provide much more raw computing capability than traditional CPU cores. Most high end GPUs are on self-contained devices that can be inserted into a server machine, as illustrated in Figure 4.1. One key aspect of GPU devices is that they have dedicated local memory, which we will refer to as "GPU memory," and their computing elements are only efficient when working on data in that GPU memory. Data stored outside the device, in CPU memory, must first be brought into the GPU memory (e.g., via PCI DMA) for it to be accessed efficiently.

Neural network training is an excellent match to the GPU computing model. For example, the forward pass of a fully connected layer, for which the value of each output node is calculated as the weighted sum of all input nodes, can be expressed as a matrix-matrix multiplication for a whole mini-batch. During the backward pass, the error terms and gradients can also be computed with similar matrix-matrix multiplications. These computations can be easily decomposed into SIMD operations and be performed efficiently with the GPU cores. Computations of other layers, such as convolution, have similar SIMD properties and are also efficient on GPUs. NVIDIA provides libraries for launching these

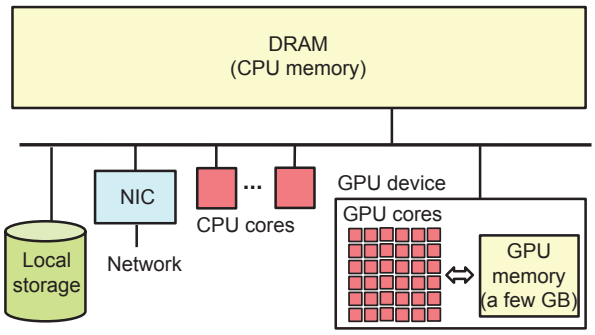


Figure 4.1: A machine with a GPU device.

computations on GPUs, such as the cuBLAS library [1] for basic linear algebra computations and the cuDNN library [2] for neural-network specific computations (e.g., convolution).

Caffe [47] is an open-source deep learning system that uses GPUs. In Caffe, a single-threaded worker launches and joins with GPU computations, by calling NVIDIA cuBLAS and cuDNN libraries, as well as some customized CUDA kernels. Each mini-batch of training data is read from an input file via the CPU, moved to GPU memory, and then processed as described above. For efficiency, Caffe keeps all model parameters and intermediate states in the GPU memory. As such, it is effective only for models and mini-batches small enough to be fully held in GPU memory. Figure 4.2 illustrates the CPU and GPU memory usage for a basic Caffe scenario.

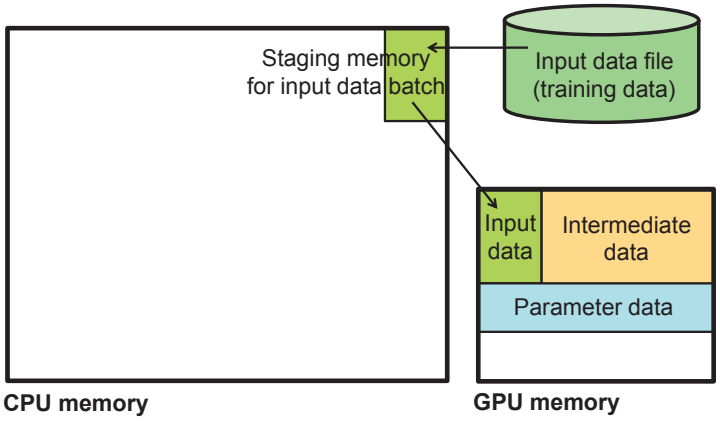


Figure 4.2: Single GPU ML, such as with default Caffe.

4.1.2 Scaling distributed GPU ML with a parameter server

Given the proven value in CPU-based distributed ML, it is natural to use a traditional parameter server architecture, such as IterStore, with distributed ML on GPUs. To explore its effectiveness, we ported two applications (the Caffe system discussed above and a

multi-class logistic regression (MLR) program) to a state-of-the-art parameter server system (IterStore [27]). Doing so was straightforward and immediately enabled distributed deep learning on GPUs, confirming the application programmability benefits of the data-parallel parameter server approach. Figure 4.3 illustrates what sits where in memory, to allow comparison to Figure 4.2 and designs described later.

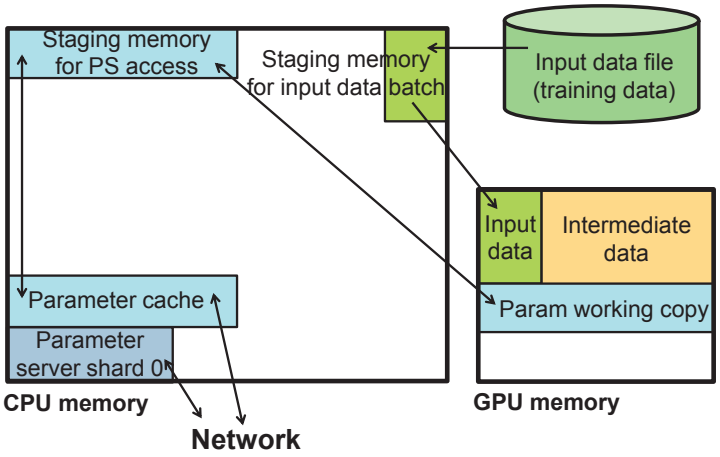


Figure 4.3: Distributed ML on GPUs using a CPU-based parameter server. The right side of the picture is much like the single-GPU illustration in Figure 4.2. But, a parameter server shard and client-side parameter cache are added to the CPU memory, and the parameter data originally only in the GPU memory is replaced in GPU memory by a local working copy of the parameter data. Parameter updates must be moved between CPU memory and GPU memory, in both directions, which requires an additional application-level staging area since the CPU-based parameter server is unaware of the separate memories.

While it was easy to get working, the performance was not acceptable. As noted by Chilimbi et al. [20], the GPU's computing structure makes it "extremely difficult to support data parallelism via a parameter server" using current implementations, because of GPU stalls, insufficient synchronization/consistency, or both. Also as noted by them and others [91, 93], the need to fit the full model, as well as a mini-batch of input data and intermediate neural network states, in the GPU memory limits the size of models that can be trained. The next section describes our design for overcoming these obstacles.

4.2 GPU-specialized parameter server design

This section describes three primary specializations to a parameter server to enable efficient support of parallel ML applications running on distributed GPUs: explicit use of GPU memory for the parameter cache, batch-based parameter access methods, and parameter server management of GPU memory on behalf of the application. The first two address performance, and the third expands the range of problem sizes that can be addressed with data-parallel execution on GPUs. Also discussed is the topic of execution model synchrony,

which empirically involves a different choice for data-parallel GPU-based training than for CPU-based training.

4.2.1 Maintaining the parameter cache in GPU memory

One important change needed to improve parameter server performance for GPUs is to keep the parameter cache in GPU memory, as shown in Figure 4.4. (Section 4.2.3 discusses the case where everything does not fit.) Perhaps counter-intuitively, this change is not about reducing data movement between CPU memory and GPU memory—the updates from the local GPU must still be moved to CPU memory to be sent to other machines, and the updates from other machines must still be moved from CPU memory to GPU memory. [1] Rather, moving the parameter cache into GPU memory enables the parameter server client library to perform these data movement steps in the background, overlapping them with GPU computing activity. Then, when the application uses the read or update functions, they proceed within the GPU memory. Putting the parameter cache in GPU memory also enables updating of the parameter cache state using GPU parallelism.

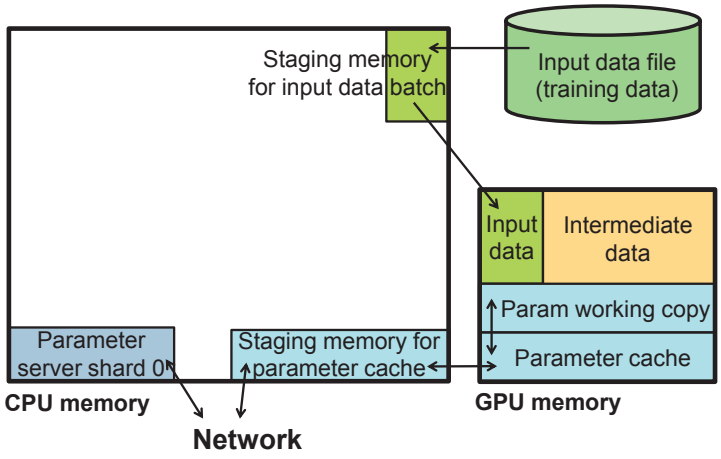


Figure 4.4: Parameter cache in GPU memory. In addition to the movement of the parameter cache box from CPU memory to GPU memory, this illustration differs from Figure 4.3 in that the associated staging memory is now inside the parameter server library. It is used for staging updates between the network and the parameter cache, rather than between the parameter cache and the GPU portion of the application.

[1] If we keep the global version of the parameter data in the GPU memory of the servers and use GPUDirect for cross-machine communication, we can avoid the GPU/CPU data transfer. We did not choose this design, because (1) GPUDirect needs special hardware support; (2) the bandwidth for GPU/CPU data transfer is often higher than the network bandwidth, so as long as the GPU/CPU data transfer is done in the background, it will not become the performance bottleneck.

Method name	Input	Description	Blocking
`Read`	list of keys data staleness bound	request a buffer, filled with parameter data	yes
`PostRead`	buffer from `Read` call	release the buffer	no
`PreUpdate`	list of keys	request an empty buffer, structured for parameter data	yes
`Update`	buffer from `PreUpdate` call	release the buffer and save the updates	no
`LocalAccess`	list of keys for local data	request a buffer, (by default) filled with local data	yes
`PostLocalAccess`	buffer from `LocalAccess` call	release the buffer, and (by default) save the data in it	no
`TableClock`	table ID	commit all updates of a table	no

Table 4.1: GeePS API calls used for access to parameter data and GeePS-managed local data.

4.2.2 Pre-built indexes and batch operations

For effective sharding across parameter server shards, parameter data is often organized as rows (e.g., 128 value entries per row), but, given the SIMD-style parallelism of GPU devices, per-row read and update operations of arbitrary model parameter rows can significantly slow execution. In particular, performance problems arise from per-value locking, index lookups, and data movement. To realize sufficient performance, our GPU-specialized parameter server supports batch-based interfaces for reads and updates. Moreover, GeePS exploits the iterative nature of model training [27] to provide batch-wide optimizations, such as pre-built indexes for an entire batch that enable GPU-efficient parallel "gathering" and updating of the set of parameters accessed in a batch. These changes make parameter server accesses much more efficient for GPU-based training.

4.2.3 Managing limited GPU device memory

As noted earlier, the limited size of GPU device memory was viewed as a serious impediment to data-parallel CNN implementations, limiting the size of the model to what could fit in a single device memory. Our parameter server design addresses this problem by managing the GPU memory for the application and swapping the data that is not currently being used to CPU memory. It moves the data between GPU and CPU memory in the background, minimizing overhead by overlapping the transfers with the training computation, and our results demonstrate that the two do not interfere with one another.

Managing GPU memory inside the parameter server. Our GPU-specialized parameter server design provides read and update interfaces with parameter-server-managed buffers. When the application reads parameter data, the parameter server client library will *allocate* (user-level allocation implemented by the parameter server) a buffer in GPU memory for it and return the pointer to this buffer to the application, instead of copying the parameter data to an application-provided buffer. When the application finishes using the parameter data, it returns the buffer to the parameter server. We call those two interfaces

`Read` and `PostRead`. When the application wants to update parameter data, it will first request a buffer from the parameter server using `PreUpdate` and use this buffer to store its updates. The application calls `Update` to pass that buffer back, and the parameter server library will apply the updates stored in the buffer and reclaim the buffer memory.

The application can also store their local non-parameter data (e.g., intermediate states) in the parameter server using similar interfaces. The local data will not be shared with the other application workers, so accessing the local data will be much faster than accessing the parameter data. For example, when the application reads the local data, the parameter server will just return a pointer that points to the stored local data, without copying it to a separate buffer. Similarly, the application can directly modify the requested local data, without needing to issue an explicit `Update` operation.

Swapping data to CPU memory when it does not fit. The parameter server client library will be able to manage all the GPU memory on a machine, if the application keeps all its local data in the parameter server and uses the PS-managed buffers. When the GPU memory of a machine is not big enough to host all data, the parameter server will store part of the data in the CPU memory. The application still accesses everything through GPU memory, as before, and the parameter server library will do the data movement for it. When the application `Read`s parameter data that is stored in CPU memory, the parameter server will perform this read using CPU cores and copy the data from CPU memory to an allocated GPU buffer, likewise for local data `Read`s. Figure 4.5 illustrates the resulting data layout in the GPU and CPU memories.

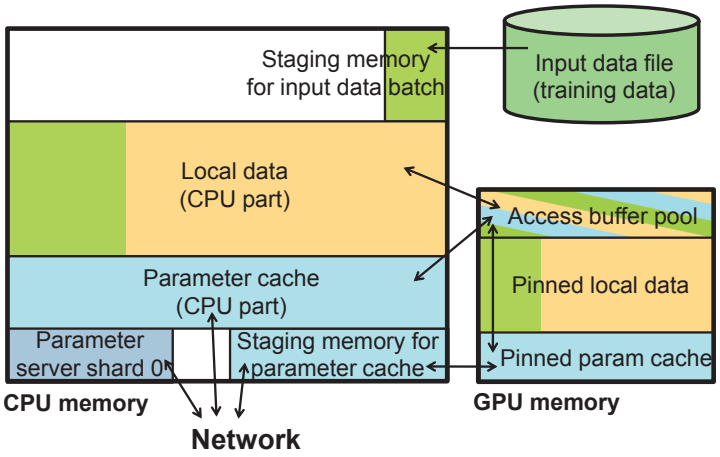


Figure 4.5: Parameter cache and local data partitioned across CPU and GPU memories. When all parameter and local data (input data and intermediate data) cannot fit within GPU memory, our parameter server can use CPU memory to hold the excess. Whatever amount fits can be pinned in GPU memory, while the remainder is transferred to and from buffers that the application can use, as needed.

GPU/CPU data movement in the background. Copying data between GPU and CPU memory could significantly slow down data access. To minimize slowdowns, our parameter server uses separate threads to perform the `Read` and `Update` operations in the

background. For an `Update` operation, because the parameter server owns the update buffer, it can apply the updates in the background and reclaim the update buffer after it finishes. In order to perform the `Read` operations in the background, the parameter server will need to know in advance the sets of parameter data that the application will access. Fortunately, iterative applications like neural network training typically apply the same parameter data accesses every iteration [27], so the parameter server can easily predict the `Read` operations and perform them in advance in the background.

4.2.4 Eschewing asynchrony

Many recent ML model training systems, including for neural network training, use a parameter server architecture to share state among data-parallel workers executing on CPUs. Consistent reports indicate that, in such an architecture, some degree of asynchrony (bounded or not) in parameter update exchanges among workers leads to significantly faster convergence than when using BSP [42, 59, 20, 29, 25, 4, 92]. We observe the opposite with data-parallel workers executing on GPUs—while synchronization delays can be largely eliminated, as expected, convergence is much slower with the more asynchronous models because of reduced training quality. This somewhat surprising observation is supported and discussed further in Section 4.4.4.

4.3 GeePS implementation

This section describes GeePS, a GPU-specialized parameter server system that implements the design aspects described in Section 4.2.

4.3.1 GeePS data model and API

GeePS is a C++ library that manages both the parameter data and local data for GPU-based machine learning applications (such as Caffe). The distributed application program usually creates one ML worker process on each machine and each of them links to one instance of the GeePS library. Algorithm 4 gives an example structure of a deep learning application using GeePS. The ML application worker often runs in a single CPU thread that launches NVIDIA library calls or customized CUDA kernels to perform computations on GPUs, and it calls GeePS functions to access and release GeePS-managed data. The GeePS APIs are summarized in Table 4.1.

GeePS manages all data as a collection of *rows* indexed by keys. The rows are then logically grouped into *tables*, and rows in the same table share the same attributes (e.g., data age). In our current implementation, each row is defined as a fixed sized array of float values, allowing efficient cross-machine communication without any marshalling. In our deep learning application, because the model parameters (i.e., connection weights of each layer) can have different sizes, we store each model parameter as multiple rows in the same table.

GeePS implements the read and update operations with PS-managed buffers for parameter data access, and a pair of operations for local data access, with which the application can directly modify the accessed local data without an explicit update operation. GeePS also provides a `TableClock` operation for application workers to signal the completion of per-table updates, and the *data age* of a table (and the rows in it) is defined as the number of times that the `TableClock` operation is called on that table by all workers. Among all the API calls, `Read`, `PreUpdate`, and `LocalAccess` are blocking, forcing the application worker to wait when data or buffer space is not ready, and the other calls are all asynchronous and return immediately. By making the application worker wait on `Read`, GeePS supports three execution synchrony models: BSP, SSP [42], and Asynchrony.

Some of the GeePS specializations (pre-built indices, background `Read`, and data placement decisions) exploit knowledge of the operation sequence of the application. GeePS uses the virtual iteration method described in Chapter 3 to collect the operation sequence information, and uses the collected information as a hint to build the data structures, build the access indices, make GPU/CPU data placement decisions, and perform prefetching. Since the gathered access information is used only as a hint, knowing the exact operation sequence is not a requirement for correctness, but a performance optimization. [2]

4.3.2 GeePS architecture

Storing data. GeePS shards the parameter data across all instances, and each GeePS instance stores one shard of the parameter data in its *parameter server shard*. The parameter server shards are not replicated, and fault tolerance is handled by checkpointing. In order to reduce communication traffic, each instance has a *parameter cache* that stores a local snapshot of the parameter data, and the parameter cache is refreshed from the parameter server shards, such as at every clock for BSP. When the application applies updates to the parameter data, those updates are also stored in the parameter cache (a write-back cache) and will be submitted to the parameter server shards at the end of every clock (when a `TableClock` is called). The parameter cache has two parts, a GPU-pinned parameter cache and a CPU parameter cache. If everything fits in GPU memory, only the GPU parameter cache is used. But, if the GPU memory is not big enough, GeePS will keep some parameter data in the CPU parameter cache. (The data placement policies are described in Section 4.3.4.) Each GeePS instance also has an *access buffer pool* in GPU memory, and GeePS allocates GPU buffers for `Read` and `PreUpdate` operations from the buffer pool. When `PostRead` or `Update` operations are called, the memory will be reclaimed by the buffer pool. GeePS manages application's input data and intermediate states as *local data*. The local data also has a GPU-pinned part and a CPU part, with the CPU part only used if necessary. GeePS divides the key space into multiple *partitions*, and the rows in different

[2]For most DNN applications (including CNN and RNN), the application accesses all model parameters every mini-batch, so the gathered information is exact. For some applications with sparse training data (e.g., BOW representation for NLP tasks), the bottom layer and the sampled softmax layer of the network might just use a subset of the weights. Even for these tasks, the operation sequence of a whole epoch still repeats. The operation sequence only changes when the training data is shuffled across epochs, and, for this case, GeePS will have to work without some of its specializations.

Algorithm 4 A DNN application with GeePS

```
L ← number of layers in the network
paramDataKeys ← decide row keys for param data
localDataKeys ← decide row keys for local data
# Report access information with a virtual iteration
TrainMiniBatch(null, virtual = yes)
# Real training iterations
while not done do
    TrainMiniBatch(nextTrainData, virtual = false)
end while
function TrainMiniBatch(trainData, virtual)
    # Forward pass
    for i = 0 ∼ (L − 1) do
        paramDataPtr ← geeps.Read(paramDataKeys_i, virtual)
        localDataPtr ← geeps.LocalAccess(localDataKeys_i, virtual)
        if not virtual then
            Setup layer_i with data pointers
            Forward computation of layer_i
        end if
        geeps.PostRead(paramDataPtr)
        geeps.PostLocalAccess(localDataPtr)
    end for
    # Backward pass
    for i = (L − 1) ∼ 0 do
        paramDataPtr ← geeps.Read(paramDataKeys_i, virtual)
        paramUpdatePtr ← geeps.PreUpdate(paramDataKeys_i, virtual)
        localDataPtr ← geeps.LocalAccess(localDataKeys_i, virtual)
        if not virtual then
            Setup layer_i with data pointers
            Backward computation of layer_i
        end if
        geeps.PostRead(paramDataPtr)
        geeps.Update(paramUpdatePtr)
        geeps.PostLocalAccess(localDataPtr)
        geeps.TableClock(table = i, virtual)
    end for
end function
```

partitions are physically managed in different data structures and with different sets of communication threads.

Data movement across machines. GeePS performs communication across machines asynchronously with three types of background threads: *keeper* threads manage the parameter data in parameter server shards; *pusher* threads send parameter data updates from parameter caches to parameter server shards, by sending messages to keeper threads; *puller* threads receive parameter data from parameter server shards to parameter caches, by receiving messages from keeper threads.

The communication is implemented using sockets, so the data needs to be copied to some CPU staging memory before being sent through the network, and the received data will also be in the CPU staging memory. The pusher/puller threads perform data movement between CPU memory and GPU memory using CUDA APIs.

Data movement inside a machine. GeePS uses two background threads to perform the data access operations for the application workers. The *allocator* thread performs the `Read`, `PreUpdate`, and `LocalAccess` operations by allocating buffers from the buffer pool and copying the requested data to the buffers. The *reclaimer* thread performs the `PostRead`, `Update`, and `PostLocalAccess` operations by saving the data to parameter cache or local store and reclaiming the buffers back to the buffer pool. These threads assign and update parameter data in large batches with pre-built indices by launching CUDA kernels on GPUs, as described in Section 4.3.3.

Synchronization and data freshness guarantees. GeePS supports BSP, asynchrony, and the Staleness Synchronous Parallel (SSP) model [42], wherein a worker at clock t is guaranteed to see all updates from all workers up to clock $t - 1 - slack$, where the slack parameter controls the data freshness. SSP with a slack of zero is the same as BSP.

To enforce SSP bounds, each parameter server shard keeps a vector clock for each table, where each vector clock entry stores the number of times each worker calls the `TableClock` operation on that table. The *data age* of each table in a parameter server shard is the minimal value of the corresponding vector clock. The parameter cache also keeps the data age information with the cached data, and the allocator thread is blocked when the data is not fresh enough.

Locking. GeePS's background threads synchronize with each other, as well as the application threads, using mutex locks and condition variables. Unlike some other CPU-based parameter servers that use per-row locks [25, 27, 92], we employ a coarse-grained locking design, where one set of mutex lock and condition variable is used for a whole key partition. We make this design decision for two reasons. First, with coarse-grained locking, batched data operations can be easily performed on a whole partition of rows. Second, unlike CPU applications, where one application thread is launched for each CPU core, a GPU application often has just one CPU host thread interacting with each GeePS instance, making lock contention less of an issue.

4.3.3 Parallelizing batched access

GeePS provides a key-value store interface to the application, where each parameter row is named by a unique key. When the application issues a read or update operation (for

accessing a set of model parameters), it will provide a list of keys for the target rows. GeePS could use a hash map to map the row keys to the locations where the rows are stored. But, in order to make the batched access be executed by all GPU cores, GeePS will use the following mechanism. Suppose the application update n rows, each with m floating point values, in one `Update` operation, it will provide an array of n parameter row updates $\{\{updates[i][j]\}_{j=1}^{m}\}_{i=1}^{n}$, and (provided in `PreUpdate`) an array of n keys $\{keys[i]\}_{i=1}^{n}$. GeePS will use an index with n entries, where each of $\{index[i]\}_{i=1}^{n}$ stores the location of the cached parameter update. Then, it will do the following data operation for this `Update`: $\{\{parameters[index[i]][j]$ += $updates[i][j]\}_{j=1}^{m}\}_{i=1}^{n}$. This operation can be executed with all the GPU cores. Moreover, the index can be built just once for each batch of keys, based on the operation sequence gathered as described earlier, and re-used for each instance of the given batch access.

4.3.4 GPU memory management

GeePS keeps the GPU-pinned parameter cache, GPU-pinned local data, and access buffer pool in GPU memory. They will be all the GPU memory allocated in a machine if the application keeps all its input data and intermediate states in GeePS and uses the GeePS-managed buffers. GeePS will pin as much parameter data and local data in GPU memory as possible. But, if the GPU memory is not large enough, GeePS will keep some of the data in CPU memory (the CPU part of the parameter cache and/or CPU part of the local data).

In the extreme case, GeePS can keep all parameter data and local data in the CPU memory. But, it will still need the buffer pool to be in the GPU memory, and the buffer pool needs to be large enough to keep all the *actively used data* even for the largest layer. In order to perform the GPU/CPU data movement in the background, GeePS does double buffering by making the access buffer pool twice the size of the largest layer.

Data placement policy. We will now describe our policy for choosing which data to pin in GPU memory. In our implementation, any local data that is pinned in GPU memory does not need to use any access buffer space. The allocator thread will just give the pointer to the pinned GPU local data to the application, without copying the data. For the parameter data, even though it is pinned in GPU memory, the allocator thread still needs to copy it from the parameter cache to an access buffer, because the parameter cache could be modified by the background communication thread (the puller thread) while the application is doing computation. As a result, pinning local data in GPU memory gives us more benefit than pinning parameter cache data. Moreover, if we pin the local data that is used at the largest layer (i.e., layer with the most memory usage), we will reduce the peak access buffer usage and thus reserve less memory for the access buffer.

Algorithm 5 illustrates our GPU/CPU data placement policy, and it only runs at the setup stage, after the access information is gathered. The algorithm chooses the entries to pin in GPU memory based on the gathered access information and a given GPU memory budget. While keeping the access buffer pool twice the size of the largest layer for double buffering, our policy will first try to pin the local data that is used at the largest layer in GPU memory, so that it can reserve less memory for the access buffer pool. Then, it will try to use the available capacity to pin more local data and parameter cache data in GPU

Algorithm 5 GPU/CPU data placement policy

Input: $\{paramData\}, \{localData\} \leftarrow$ entries of all parameter data and local data accessed at each layer
Input: $totalMem \leftarrow$ the amount of GPU memory to use
 # Start with everything in CPU memory
$\{cpuMem\} \leftarrow \{paramData\} \cup \{localData\}$
$\{gpuMem\} \leftarrow \emptyset$
 # Set access buffer twice the size of the largest layer for double buffering
$largestLayer \leftarrow$ layer with most data
$accessBufferSize \leftarrow 2 \times size(largestLayer)$
$availMem \leftarrow totalMem - accessBufferSize$
 # First pin local data used by the largest layer
while $\exists data \in \{localData\} \cap \{cpuMem\} \cap \{largestLayer\}$ **do**
    $largestLayerSizeReduction \leftarrow$
        reduction of the size of the largest layer, if *data* is moved to $\{gpuMem\}$
    $memSizeDelta \leftarrow size(data) - 2 \times largestLayerSizeReduction$
    if $availMem < memSizeDelta$ **then**
        break
    end if
    Move *data* from $\{cpuMem\}$ to $\{gpuMem\}$
    $availMem \leftarrow availMem - memSizeDelta$
    $largestLayer \leftarrow$ layer with most data, excluding GPU local data
end while
 # Pin more local data using the available memory
for each $data \in \{localData\} \cap \{cpuMem\}$ **do**
    if $availMem \geq size(data)$ **then**
        Move *data* from $\{cpuMem\}$ to $\{gpuMem\}$
        $availMem \leftarrow availMem - size(data)$
    end if
end for
 # Pin parameter data using the available memory
for each $data \in \{paramData\} \cap \{cpuMem\}$ **do**
    if $availMem \geq size(paramData)$ **then**
        Move *data* from $\{cpuMem\}$ to $\{gpuMem\}$
        $availMem \leftarrow availMem - size(data)$
    end if
end for
 # Dedicate the remaining available memory to the access buffer
Increase $accessBufferSize$ by $availMem$

memory. Finally, it will add any remaining available GPU memory to the access buffer. The orders that each of the loops use to iterate over the data entries do not matter here, but our implementation iterates over the data entries from the bottom layer to the top layer.

Avoiding unnecessary data movement. When the application accesses/post-accesses the local data that is stored in CPU memory, by default, the allocator/reclaimer thread will need to copy the data between the CPU memory and the allocated GPU memory. However, sometimes this data movement is not necessary. For example, when we train a deep neural network, the input data and intermediate data are overwritten every new mini-batch, and the old values from the last mini-batch can be safely thrown away. To avoid this unnecessary data movement, we allow the application to specify a `no-fetch` flag when calling `LocalAccess`, and it tells GeePS to just allocate an uninitialized piece of GPU memory, without fetching the data from CPU memory. Similarly, when the application calls `PostLocalAccess` with a `no-save` flag, GeePS will just free the GPU memory, without saving the data to CPU memory.

4.4 Evaluation

This section evaluates GeePS's support for parallel deep learning over distributed GPUs, using two recent image classification models and a video classification model executed in the original and modified Caffe application. The evaluation confirms four main findings: (1) GeePS provides effective data-parallel scaling of training throughput and training convergence rate, at least up to 16 machines with GPUs. (2) GeePS's efficiency is much higher, for GPU-based training, than a traditional CPU-based parameter server and also much higher than parallel CPU-based training performance reported in the literature. (3) GeePS's dynamic management of GPU memory allows data-parallel GPU-based training on models that are much larger than used in state-of-the-art deep learning for image classification and video classification. (4) For GPU-based training, unlike for CPU-based training, loose consistency models (e.g., SSP and asynchronous) significantly reduce convergence rate compared to BSP. Fortunately, GeePS's efficiency enables significant scaling benefits even with larger BSP-induced communication delays.

A specific non-goal of our evaluation is comparing the classification accuracies of the different models. Our focus is on enabling faster training of whichever model is being used, which is why we measure performance for several.

4.4.1 Experimental setup

Application setup. We use Caffe [47], the open-source single-GPU convolutional neural network application discussed earlier.[3] Our experiments use unmodified Caffe to represent

[3]For the image classification application, we used the version of Caffe from `https://github.com/BVLC/caffe` as of June 19, 2015. Since their master branch version does not support RNN, for the video classification application, we used the version from `https://github.com/LisaAnne/lisa-caffe-public/tree/lstm_video_deploy` as of Nov 9, 2015.

the optimized single-GPU case and a minimally modified instance (GeePS-Caffe) that uses GeePS for data-parallel execution. GeePS-Caffe uses GeePS to manage all its parameter data and local data (including input data and intermediate data), using the same structure as illustrated in Algorithm 4. The parameter data of each layer is stored as rows of a distinct GeePS table, allowing GeePS to propagate each layer's updates during the computations of other layers, as suggested by Zhang et al. [97]. Each GeePS row is configured to be an array of 128 float values (512 bytes).

Cluster setup. Each machine in our cluster has one NVIDIA Tesla K20C GPU, which has 5 GB of GPU device memory. In addition to the GPU, each machine has four 2-die 2.1 GHz 16-core AMD® Opteron 6272 packages and 128 GB of RAM. Each machine is installed with 64-bit Ubuntu 14.04, CUDA toolkit 7.5, and cuDNN v2. The machines are inter-connected via a 40 Gbps Ethernet interface (12 Gbps measured via iperf), and Caffe reads the input training data from remote file servers via a separate 1 Gbps Ethernet interface.

Image classification datasets and models. The experiments use two datasets for image classification. The first one is the ImageNet22K dataset [30], which contains 14 million images labeled to 22,000 classes. Because of the computation work required to train such a large dataset, CPU-based systems running on this dataset typically need a hundred or more machines and spend over a week to reach convergence [20]. We use half of the images (7 million images) as the training set and the other half as the testing set, which is the same setup as described by Chilimbi et al. [20]. For the ImageNet22K dataset, we use a similar model to the one used to evaluate ProjectAdam [20], which we refer to as the *AdamLike* model.[4] The AdamLike model has five convolutional layers and three fully connected layers. It contains 2.4 billion connections for each image, and the model parameters are 470 MB in size.

The second dataset we used is Large Scale Visual Recognition Challenge 2012 (ILSVRC12) [76]. It is a subset of the ImageNet22K dataset, with 1.3 million images labeled to 1000 classes. For this dataset, we use the *GoogLeNet* model [82], a recent inception model from Google. The network has about 100 layers, and 22 of them have model parameters. Though the number of layers is large, the model parameters are only 57 MB in size, because they use mostly convolutional layers.

Video classification datasets and models. We use the UCF-101 dataset [79] for our video classification experiments. UCF-101 has about 8,000 training videos and 4,000 testing videos categorized into 101 human action classes. We use a recurrent neural network model for this application, following the approach described by Donahue et al. [31]. We use the GoogLeNet network as the CNN layers and stack LSTM layers with 256 hidden units on top of them. The weights of the GoogLeNet layers have been pre-trained with the single frames of the training videos. Following the same approach as is described by Donahue et al. [31], we extract the video frames at a rate of 30 frames per second and train the model with randomly selected video clips of 32 frames each.

[4]We were not able to obtain the exact model that ProjectAdam used, so we emulated it based on the descriptions in the paper. Our emulated model has the same number and types of layers and connections, and we believe our training performance evaluations are representative even if the resulting model accuracy may not be.

Training algorithm setup. Both the unmodified and the GeePS-hosted Caffe train the models using the SGD algorithm with a momentum of 0.9. Unless otherwise specified, we use the configurations listed in Table 4.2. Our experiments in Section 4.4.3 evaluate performance with different mini-batch sizes. For AdamLike and GoogLeNet network, we used the same learning rate for both single-machine training and distributed training, because we empirically found that this learning rate is the best setting for both. For the RNN model, we used a different learning rate for single-machine training, because it leads to faster convergence.

Model	Mini-batch size (per machine)	Learning rate
AdamLike	200 images	0.0036, divided by 10 every 3 epochs
GoogLeNet	32 images	0.0036, divided by 10 every 150 epochs
RNN	1 video, 32 frames each	0.0000125 for 8 machines, 0.0001 for single-machine, divided by 10 every 20 epochs

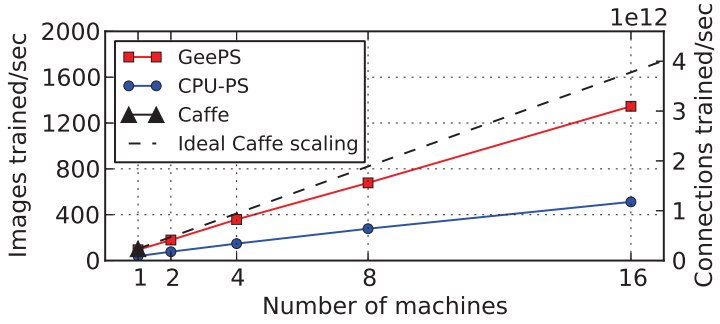
Table 4.2: Model training configurations.

GeePS setup. Unless otherwise specified, we let GeePS keep the parameter cache and local data in GPU memory for our experiments, since it all fits for all of the models used; Section 4.4.3 evaluates performance when keeping part of the data in CPU memory, including for a very large model scenario. Unless otherwise specified, the BSP mode is used; Section 4.4.4 analyzes the effect of looser synchronization models.

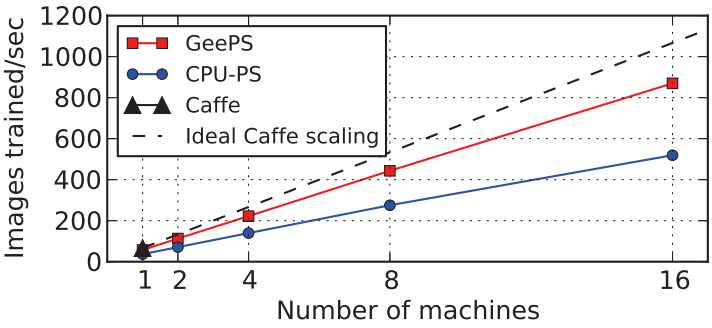
4.4.2 Scaling deep learning with GeePS

This section evaluates how well GeePS supports data-parallel scaling of GPU-based training on both image classification and video classification application. We compare GeePS with three classes of systems: (1) *Single-GPU optimized training*: the original unmodified Caffe system (referred to as "Caffe") represents training optimized for execution on a single GPU. (2) *GPU workers with CPU-based parameter server*: multiple instances of the modified Caffe linked via IterStore [27] a state-of-the-art CPU-based parameter server ("CPU-PS"). (3) *CPU workers with CPU-based parameter server*: reported performance numbers from recent literature are used to put the GPU-based performance into context relative to state-of-the-art CPU-based deep learning.

Image classification. Figure 4.6 shows the training throughput scalability of the image classification application, in terms of both the number of images trained per second and the number of network connections trained per second. Note that there is a linear relationship between those two metrics. GeePS scales almost linearly when we add more machines. Compared to the single-machine optimized Caffe, GeePS achieves 13× speedups on both GoogLeNet and AdamLike model with 16 machines. Compared to CPU-PS, GeePS achieves over 2× more throughput. The GPU stall time of GeePS is only 8% for both



(a) AdamLike model on ImageNet22K dataset.



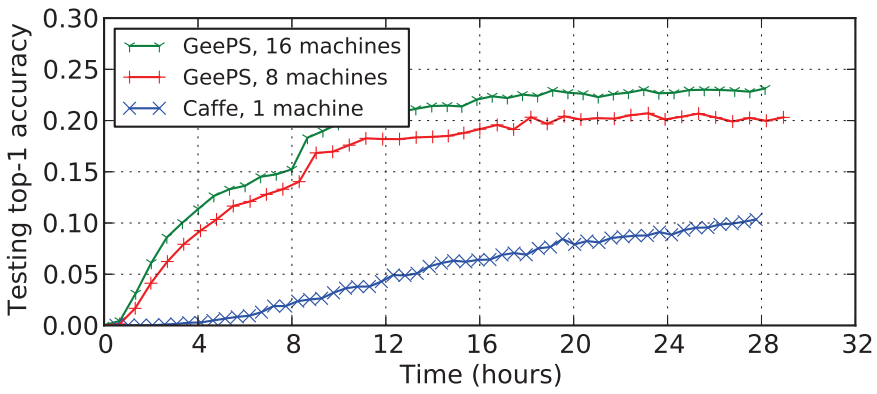
(b) GoogLeNet model on ILSVRC12 dataset.

Figure 4.6: Image classification throughput scalability. Both GeePS and CPU-PS run in the BSP mode.

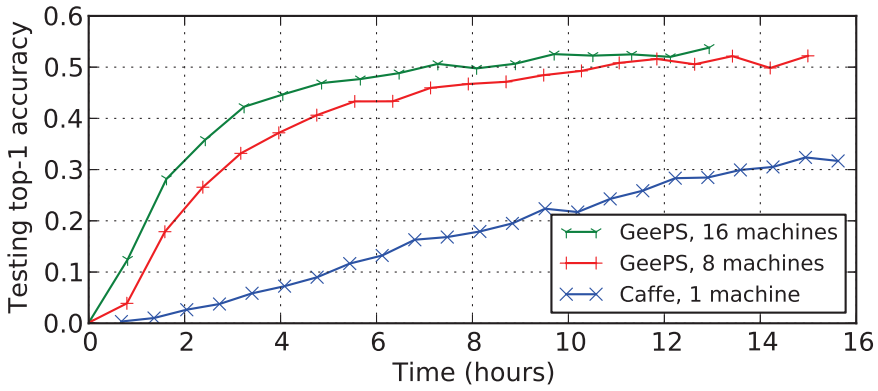
GoogLeNet and AdamLike model, so 92% of the total runtime is spent on the application's computational work. While using CPU-PS, the GPU stall time is 51% and 65% respectively.

Chilimbi et al. [20] report that ProjectAdam trains 570 billion connections per second on the ImageNet22K dataset when with 108 machines (88 CPU-based worker machines with 20 parameter server machines) [20]. Figure 4.6(a) shows that GeePS achieves higher throughput with only 4 GPU machines, because of its efficient data-parallel execution on GPUs.

Figure 4.7 shows the image classification top-1 testing accuracies of the trained models. The top-1 classification accuracy is defined as the fraction of the testing images that are correctly classified. To evaluate convergence speed, we will compare the amount of time required to reach a given level of accuracy, which is a combination of image training throughput and model convergence per trained image. For the AdamLike model on the ImageNet22K dataset, Caffe needs 26.9 hours to reach 10% accuracy, while GeePS needs only 4.6 hours with 8 machines (6× speedup) or 3.3 hours with 16 machines (8× speedup). For the GoogLeNet model on the ILSVRC12 dataset, Caffe needs 13.7 hours to reach 30% accuracy, while GeePS needs only 2.8 hours with 8 machines (5× speedup) or 1.8 hours with 16 machines (8× speedup). The model training time speedups compared to the single-GPU optimized Caffe are lower than the image training throughput speedups, as expected, because each machine determines gradients independently. Even using BSP, more



(a) AdamLike model on ImageNet22K dataset.



(b) GoogLeNet model on ILSVRC12 dataset.

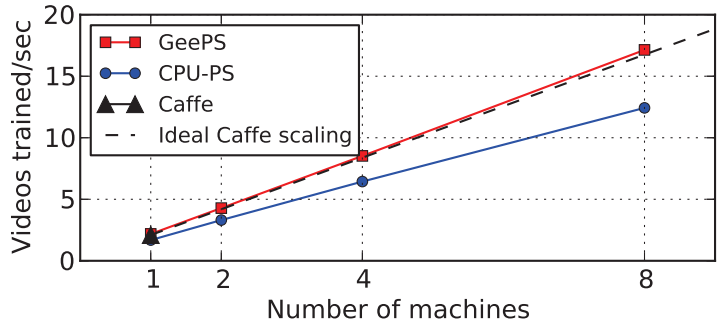
Figure 4.7: Image classification top-1 accuracies.

training is needed than with a single worker to make the model converge. But, the speedups are still substantial.

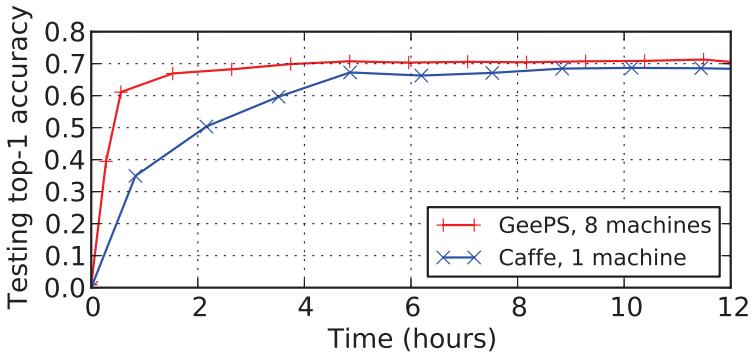
For the AdamLike model on the ImageNet22K dataset, Chilimbi et al. [20] report that ProjectAdam needs one day to reach 13.6% accuracy with 58 machines (48 CPU-based worker machines with 10 parameter server machines). GeePS needs only 6 hours to reach the same accuracy with 16 machines (about 4× speedup). To reach 13.6% accuracy, the DistBelief system trained (a different model) with 2,000 machines for a week [56].

Because both GeePS and CPU-PS run in the BSP mode, with the same number of machines, the accuracy improvement speedups of GeePS over CPU-PS are the same as the throughput speedups, so we leave them out of the graphs.

Video classification. Figure 4.8(a) shows the training throughput of the video classification application. GeePS scales linearly from Caffe (8× throughput with 8 machines). Figure 4.8(b) shows the top-1 testing accuracies. To reach 60% accuracy, Caffe used 3.6 hours, while GeePS with 8 machines used 0.5 hours (7× speedup); to reach 68% accuracy, Caffe used 8.4 hours, while GeePS with 8 machines used 2.4 hours (3.5× speedup).



(a) Training throughput



(b) Top-1 accuracies.

Figure 4.8: Video classification scalability.

4.4.3 Dealing with limited GPU memory

An oft-mentioned concern with data-parallel deep learning on GPUs is that it can only be used when the entire model, as well as all intermediate state and the input mini-batch, fit in GPU memory. GeePS eliminates this limitation with its support for managing GPU memory and using it to buffer data from the much larger CPU memory. Although all of the models we experiment with (and most state-of-the-art models) fit in our GPUs' 5 GB memories, we demonstrate the efficacy of GeePS's mechanisms in two ways: by using only a fraction of the GPU memory for the largest case (AdamLike) and by experimenting with a much larger synthetic model. We also show that GeePS's memory management support allows us to do video classification on longer videos.

Artificially shrinking available GPU memory. With a mini-batch of 200 images per machine, training the AdamLike model on the ImageNet22K dataset requires only 3.67 GB memory per machine, with 123 MB for input data, 2.6 GB for intermediate states, and 474 MB each for parameter data and computed parameter updates. Note that the sizes of the parameter data and parameter updates are determined by the model, while the input data and intermediate states grow linearly with the mini-batch size. For best throughput, GeePS also requires use of an access buffer that is large enough to keep the actively used parameter data and parameter updates at the peak usage, which is 528 MB minimal and

1.06 GB for double buffering (the default) to maximize overlapping of data movement with computation. So, in order to keep everything in GPU memory, the GeePS-based training needs 4.73 GB of GPU memory.

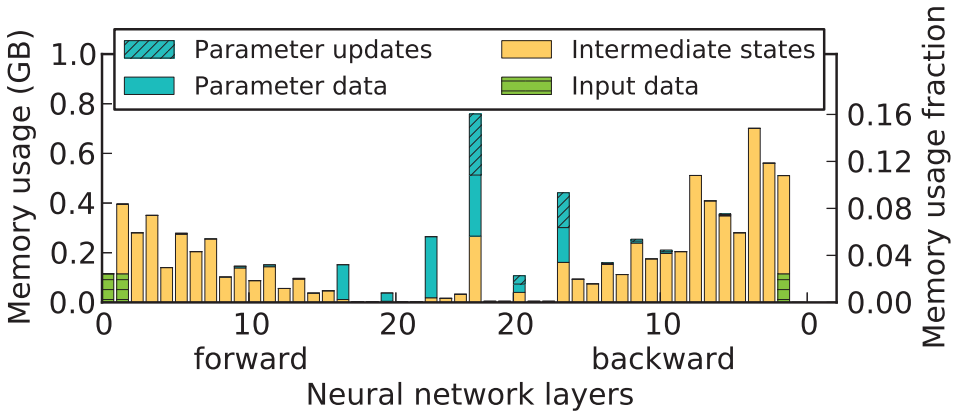


Figure 4.9: Per-layer memory usage of AdamLike model on ImageNet22K dataset.

Recall, however, that GeePS can manage GPU memory usage such that only the data needed for the layers being processed at a given point need to be in GPU memory. Figure 4.9 shows the per-layer memory usage for training the AdamLike model, showing that it is consistently much smaller than the total memory usage. The left Y axis shows the absolute size (in GB) for a given layer, and the right Y axis shows the fraction of the absolute size over the total size of 4.73 GB. Each bar is partitioned into the sizes of input data, intermediate states, parameter data, and parameter updates for the given layer. Most layers have little or no parameter data, and most of the memory is consumed by the intermediate states for neuron activations and error terms. The layer that consumes the most memory uses about 17% of the total memory usage, meaning that about 35% of the 4.73 GB is needed for full double buffering.

Figure 4.10 shows data-parallel training throughput using 8 machines, when we restrict GeePS to using different amounts of GPU memory to emulate GPUs with smaller memories. When there is not enough GPU memory to fit everything, GeePS must swap data to CPU memory. For the case of 200 images per batch, when we swap all data in CPU memory, we need only 35% of the GPU memory compared to keeping all data in GPU memory, but we are still able to get 73% of the throughput.

When the GPU memory limits the scale, people are often forced to use smaller mini-batch sizes to let everything fit in GPU memory. Our results in Figure 4.10 also shows that using our memory management mechanism is more efficient than shrinking the mini-batch size. For the three mini-batch sizes compared, we keep the inter-machine communication the same by doing multiple mini-batches per clock as needed (e.g., four 50-image mini-batches per clock). For the case of 100 images per batch and 50 images per batch, 3.7 GB and 3.3 GB respectively are needed to keep everything in GPU memory (including access buffers for double buffering). While smaller mini-batches reduce the total memory requirement, they perform significantly less well for two primary reasons: (1) the GPU computation is

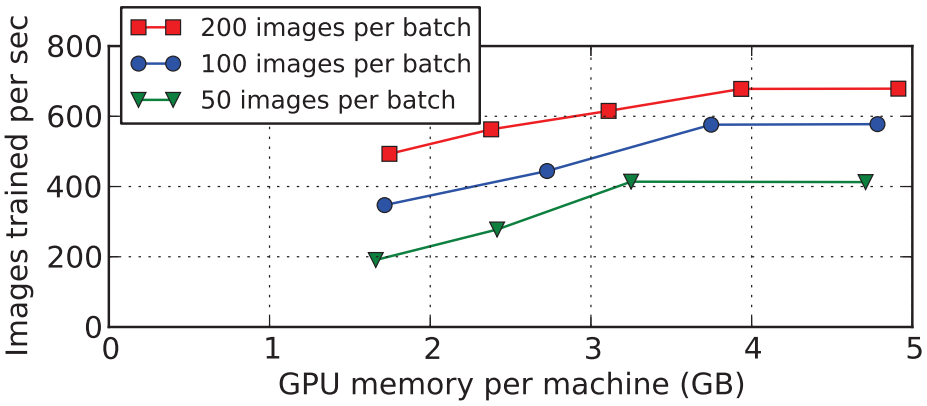


Figure 4.10: Throughput of AdamLike model on ImageNet22K dataset with different GPU memory budgets.

more efficient with a larger mini-batch size, and (2) the time for reading and updating the parameter data locally, which does not shrink with mini-batch size, is amortized over more data.

Training a very large neural network. To evaluate performance for much larger neural networks, we create and train huge synthetic models. Each such neural network contains only fully connected layers with no weight sharing, so there is one model parameter (weight) for every connection. The model parameters of each layer is about 373 MB. We create multiple such layers and measure the throughput (in terms of # connections trained per second) of training different sized networks, as shown in Figure 4.11. For all sizes tested, up to a 20 GB model (56 layers) that requires over 70 GB total (including local data), GeePS is able to train the neural network without excessive overhead. The overall result is that GeePS's GPU memory management mechanisms allows data-parallel training of very large neural networks, bounded by the largest layer rather than the overall model size.

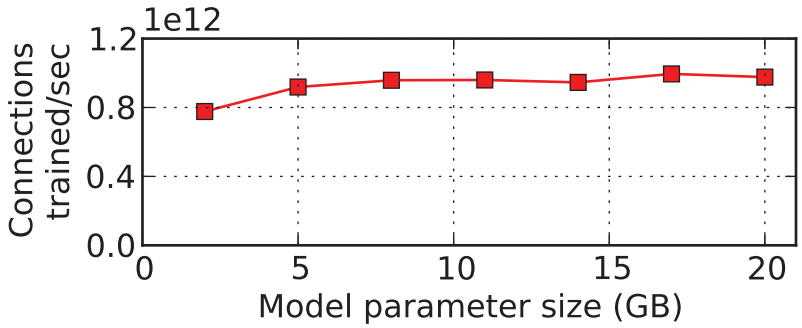


Figure 4.11: Training throughput on very large models. Note that the number of connections increases linearly with model size, so the per-image training time grows with model size because the per-connection training time stays relatively constant.

Video classification on longer videos. As is described in Section 2.4.2, the video classification application requires complete sequences of image frames to be in the same mini-batch, so the GPU memory size will either limit the maximum number of frames per video or force the model to be split across multiple machines, incurring extra complexity and network communication overhead. Using unmodified Caffe, for example, our RNN can support a maximum video length of 48 frames.[5] Because the videos are often sampled at a rate of 30 frames per second, a 48-frame video is less than 2 seconds in length. Ng et al. [94] find that using more frames in a video improves the classification accuracy. In order to use a video length of 120 frames, Ng et al. used a model-parallel approach to split the model across four machines, which incurs extra network communication overhead. By contrast, with the memory management support of GeePS, we are able to train videos with up to 192 frames, using solely data parallelism.

4.4.4 The effects of looser synchronization

Use of looser synchronization models, such as Stale Synchronous Parallel (SSP) or even unbounded asynchronous, has been shown to provide significantly faster convergence rates in data-parallel CPU-based model training systems [42, 59, 20, 29, 25, 4, 92]. This section shows results confirming our experience that this does not hold true with GPU-based deep learning.

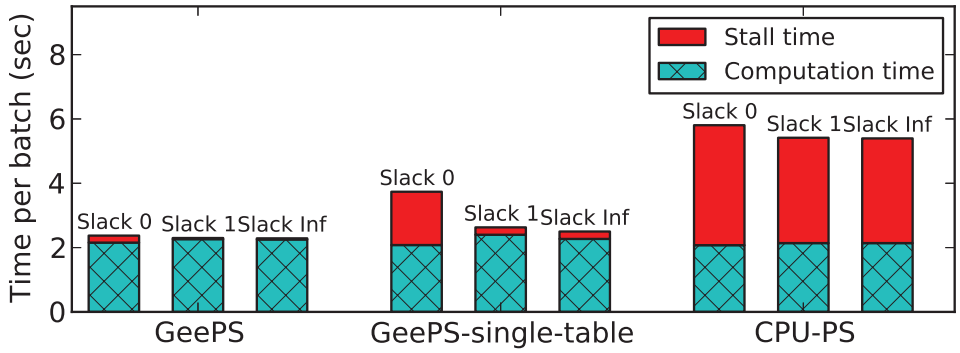


Figure 4.12: Data-parallel per-mini-batch training time for AdamLike model under different configurations. We refer to "stall time" as any part of the runtime when GPUs are not doing computations.

Figure 4.12 compares the per-mini-batch AdamLike model training time with 8 machines, when using GeePS, GeePS-single-table, and CPU-PS, and each of three synchronization models: BSP ("Slack 0"), SSP ("Slack 1"), and Asynchronous ("Slack Inf"). The GeePS-single-table setup has the application store all its parameter data in a single table, so that the parameter updates of all layers are sent to the server together as a whole batch. While in the default GeePS setup, where the parameter data of each layer is stored in a

[5]Here, we are just considering the memory consumption of the training stage. If we further consider the memory used for testing, the supported maximum video length will be shorter.

distinct table, the parameter updates of a layer can be sent to the server (and propagated to other workers) before the computation of other layers finish. Showing the performance of GeePS-single-table helps us understand the performance improvement coming from decoupling the parameter data of different layers. Each bar in Figure 4.12 divides the total into two parts. First, the computation time is the time that the application spends on training, which is mostly unaffected by the parameter server artifact and synchronization model; the non-BSP GeePS cases show a slight increase because of minor GPU-internal resource contention when there is great overlap between computation and background data movement. Second, the stall time includes any additional time spent on reading and updating parameter data, such as waiting time for slow workers to catch up or updated parameters to arrive, and time for moving data between GPU and CPU memory.

There are two interesting observations here. First, even for the Slack 0 (BSP) case, GeePS has little stall time. That is because, with our specializations, the per-layer updates can be propagated in the background, before the computations of other layers finish. The second observation is that expensive GPU/CPU data movement stalls CPU-PS execution, even with asynchronous communication (Slack Inf). Though CPU-PS also keeps parameter data of different layers in separate tables, the application has to perform expensive GPU/CPU data movement in the foreground each time it accesses the parameter data.

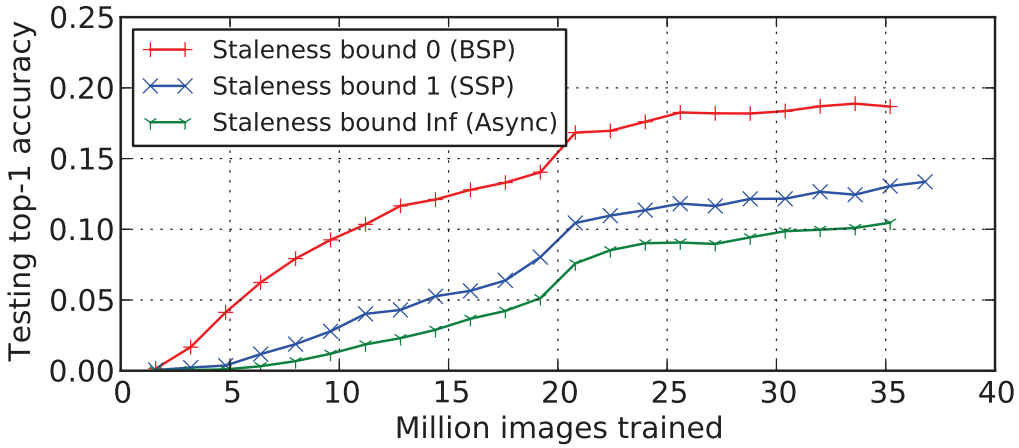


Figure 4.13: AdamLike model top-1 accuracy as a function of the number of training images processed, for BSP, SSP with slack 1, and Async.

Figure 4.13 compares the classification accuracy as a function of the number of training images processed, for GeePS with BSP, SSP (Slack 1), and Async. The result shows that, when using SSP (Slack 1) or Async, many more images must be processed to reach the same accuracy as with BSP (e.g., 2× more for Slack 1 and 3× more for Async to reach 10% accuracy). The training throughput speedups achieved with Slack 1 or Async are not sufficient to overcome the reduced training quality per image processed—using BSP leads to much faster convergence. We believe there are two reasons causing this outcome. First, with our specializations, there is little to no communication delay for DNN applications, so adding data staleness does not increase the throughput much. Second, the conditions in

the SSP proofs of previous literatures [42] do not apply to DNN, because training a DNN is a non-convex problem. Interestingly, our observation is consistent with the concern about using parameter servers for data-parallel GPU-based training expressed by Chilimbi et al. [20]: "Either the GPU must constantly stall while waiting for model parameter updates or the models will likely diverge due to insufficient synchronization." Fortunately, GeePS's GPU-specialized design greatly reduces the former effect, allowing BSP-based execution to scale well.

4.5 Additional related work

This section augments the background related work discussed in Section 4.1, which covered use of parameter servers and individual GPUs for deep learning.

Coates et al. [24] describe a specialized multi-machine GPU-based system for parallel deep learning. The architecture used is very different than GeePS. Most notably, it relies on model parallelism to partition work across GPUs, rather than the simpler data-parallel model. It also uses specialized MPI-based communication over Infiniband, rather than a general parameter server architecture, regular sockets, and Ethernet.

Deep Image [93] is a custom-built supercomputer for deep learning via GPUs. The GPUs used have large memory capacity (12 GB), and their image classification network fits within it, allowing use of data-parallel execution. They also support for model-parallel execution, with ideas borrowed from Krizhevsky et al. [50], by partitioning the model on fully connected layers. The machines are interconnected by Infiniband with GPUDirect RDMA, so no CPU involvement is required, and they do not use the CPU cores or CPU memory to enhance scalability like GeePS does. Deep Image exploits its low latency GPU-direct networking for specialized parameter state exchanges rather than using a general parameter server architecture like GeePS.

MXNet [18] and Poseidon [97] are two concurrently developed ([26]) systems for multi-GPU deep learning. Both systems take the data-parallel approach by making use of CPU-based parameter servers (Poseidon uses Bosen [92] and MXNet uses ParameterServer [59]). GeePS differs from MXNet and Poseidon in two primary ways. First, in order to overcome the inefficiency of using CPU-based parameter servers, both MXNet and Poseidon rely on optimizations to their specific GPU application system (Poseidon is built on Caffe and MXNet has its own). GeePS, on the other hand, specializes its reusable parameter server module, providing efficiency for all GPU deep learning applications it hosts. Indeed, the application improvements made for MXNet and Poseidon would further improve our reported performance numbers. Second, both MXNet and Poseidon require that each of their GPU machines has enough GPU memory to store all model parameters and intermediate states, limiting the size of their neural networks. GeePS's explicit GPU memory management support, on the other hand, allows the training of neural networks that are much bigger than the available GPU memory.

Chen et al. proposed a method for reducing the memory usage of deep neural network training that will recompute, rather than keep, the intermediate data [19]. Their approach explicitly trades extra computation for reduced memory usage, whereas GeePS reduces

memory usage by doing more data transfer and overlaps data transfer with computation. Hence, we believe the two approaches can be combined.

Chapter 5

MLtuner: Exploiting Quick Tunable Decision for Automatic ML Tuning

For training of large, complex models, parallel execution over multiple cluster nodes is warranted. The algorithms and frameworks used generally have multiple tunables that have significant impact on the execution and convergence rates. For example, the learning rate is a key tunable when using stochastic gradient descent (SGD) for training. As another example, the data staleness bound is a key tunable when using frameworks that explicitly balance asynchrony benefits with inconsistency costs [25, 59].

These tunables are usually manually configured or left to broad defaults. Unfortunately, knowing the right settings is often quite challenging. The best tunable settings can depend on the chosen model, model hyperparameters (e.g., number and types of layers in a neural network), the algorithm, the framework, and the resources on which the ML application executes. As a result, manual approaches not surprisingly involve considerable effort by domain experts or yield (often highly) suboptimal training times and solution quality. Our interactions with both experienced and novice ML users comport with this characterization.

MLtuner is a tool for automatically tuning ML application training tunables. It hooks into and guides a training system in trying different settings. MLtuner determines initial tunable settings based on rapid trial-and-error search, wherein each option tested runs for a small (automatically determined) amount of time, to find good settings based on the convergence rate. It repeats this process when convergence slows, to see if different settings provide faster convergence and/or better solution. This paper describes MLtuner's design and how it addresses challenges in auto-tuning ML applications, such as large search spaces, noisy convergence progress, variations in effective trial times, best tunable settings changing over time, when to re-tune, etc.

We have integrated MLtuner with two different state-of-the-art training systems and experimented with several real ML applications, including a recommendation application on a CPU-based parameter server system and both image classification and video classification on a GPU-based parameter server system. For increased breadth, we also experimented with three different popular models and two datasets for image classification. The results show MLtuner's effectiveness: MLtuner consistently zeroes in on good tunables, in each case, guiding training to match and exceed the best settings we have found for the given ap-

plication/model/dataset. Comparing to state-of-the-art hyperparameter tuning approaches, such as Spearmint [78] and Hyperband [58], MLtuner completes over an order of magnitude faster and does not exhibit the same robustness issues for large models/datasets.

This paper makes the following primary contributions. First, it introduces the first approach for automatically tuning the multiple tunables associated with an ML application within the context of a single execution of that application. Second, it describes a tool (MLtuner) that implements the approach, overcoming various challenges, and how MLtuner was integrated with two different ML training systems. Third, it presents results from experiments with real ML applications, including several models and datasets, demonstrating the efficacy of this new approach in removing the "black art" of tuning from ML application training without the orders of magnitude runtime increases of existing auto-tuning approaches.

5.1 Background and related work

5.1.1 Machine learning tunables

ML training often requires the selection and tuning of many *training hyperparameters*. For example, the SGD algorithm has a *learning rate* (a.k.a. step size) hyperparameter that controls the magnitude of the model parameter updates. The *training batch size* hyperparameter controls the size of the training data mini-batch that each worker processes each clock. Many deep learning applications use the momentum technique [81] with SGD, which exhibit a *momentum* hyperparameter, to smooth updates across different training batches. In data-parallel training, ML workers can have temporarily inconsistent parameter copies, and in order to guarantee model convergence, consistency models (such as SSP [42, 25] or bounded staleness [59]) are often used, which provide tunable *data staleness* bounds.

Many practitioners (as well as our own experiments) have found that the settings of the training hyperparameters have a big impact on the completion time of an ML task (e.g., orders of magnitude slower with bad settings) [77, 69, 32, 64, 96, 48, 42, 25] and even the quality of the converged model (e.g., lower classification accuracy with bad settings) [77, 69]. To emphasize that training hyperparameters need to be tuned, we call them *training tunables* in this paper.

The training tunables should be distinguished from another class of ML hyperparameters, called *model hyperparameters*. The training tunables control the training procedure but do not change the model (i.e., they are not in the objective function), whereas the model hyperparameters define the model and appear in the objective function. Example model hyperparameters include model type (e.g., logistic regression or SVM), neural network depth and width, and regularization method and magnitude. MLtuner focuses on improving the efficiency of training tunable tuning, and could potentially be used to select training tunables in an inner loop of existing approaches that tune model hyperparameters.

5.1.2 Related work on machine learning tuning

Manual tuning by domain experts

The most common tuning approach is to do it manually (e.g., [83, 46, 82, 52, 41, 97]). Practitioners often either use some uilt-in defaults or pick training tunable settings via trial-and-error. Manual tuning is inefficient, and the tunable settings chosen are often suboptimal.

For some tasks, such as training a deep neural network for image classification, practitioners find it is important to decrease the learning rate during training in order to get a model with good classification accuracy [83, 46, 82, 52, 41, 97], and there are typically two approaches of doing that. The first approach (taken by [52, 97]) is to manually change the learning rate when the classification accuracy plateaus (i.e., stops increasing), which requires considerable user efforts for monitoring the training. The second approach (taken by [83, 46, 82]) is to decay the learning rate η over time t, with a function of $\eta = \eta_0 \times \gamma^t$. The learning rate decaying factor γ, as a training tunable, is even harder to tune than the learning rate, because it affects the future learning rate. To decide the best γ setting for a training task, practitioners often need to train the model to completion several times, with different γ settings.

Traditional hyperparameter tuning approaches

There is prior work on automatic ML hyperparameter tuning (sometimes also called model search), including [78, 58, 80, 89, 7, 49, 8, 61, 71, 33, 84]. However, none of the prior work distinguishes training tunables from model hyperparameters; instead, they tune both of them together in a combined search space. Because many of their design choices are made for model hyperparameter tuning, we find them inefficient and insufficient for training tunable tuning.

To find good model hyperparameters, many traditional tuning approaches train models *from initialization to completion* with different hyperparameter settings and pick the model with the best quality (e.g., in terms of its cross-validation accuracy for a classification task). The hyperparameter settings to be evaluated are often decided with *bandit optimization algorithms*, such as Bayesian optimization [66] or HyperOpt [9]. Some tuning approaches, such as Hyperband [58] and TuPAQ [80], reduce the tuning cost by stopping the lower-performing settings early, based on the model qualities achieved in the early stage of training.

MLtuner differs from existing approaches in several ways. First, MLtuner trains the model to completion only once, with the automatically decided best tunable settings, because training tunables do not change the model, whereas existing approaches train multiple models to completion multiple times, incurring considerable tuning cost. Second, MLtuner uses training loss, rather than cross-validation model qualities, as the feedback to evaluate tunable settings. Training loss can be obtained for every training batch at no extra cost, because SGD-based training algorithms use training loss to compute parameter updates, whereas the model quality is evaluated by testing the model on validation data, and the associated cost does not allow it to be frequently evaluated (often every thousands

of training batches). Hence, MLtuner can use more frequent feedback to find good tunable settings in less time than traditional approaches. This option is enabled by the fact that, unlike model hyperparameters, training tunables do not change the mathematical formula of the objective function, so just comparing the training loss is sufficient. Third, MLtuner automatically decides the amount of resource (i.e., training time) to use for evaluating each tunable setting, based on the noisiness of the training loss, while existing approaches either hard-code the trial effort (e.g., TuPAQ always uses 10 iterations) or decide it via a grid search (e.g., Hyperband iterates over each of the possible resource allocation plans). Fourth, MLtuner is able to re-tune tunables during training, while existing approaches use the same hyperparameter setting for the whole training. Unlike model hyperparameters, training tunables can (and often should) be dynamically changed during training, as discussed above.

Adaptive SGD learning rate tuning algorithms

Because the SGD algorithm is well-known for being sensitive to the learning rate (LR) setting, experts have designed many adaptive SGD learning rate tuning algorithms, including AdaRevision [64], RMSProp [85], Nesterov [68], Adam [48], AdaDelta [96], and AdaGrad [32]. These algorithms adaptively adjust the LR for individual model parameters based on the magnitude of their gradients. For example, they often use relatively large LRs for parameters with small gradients and relatively small LRs for parameters with large gradients. However, these algorithms still require users to set the initial LR. Even though they are less sensitive to the initial LR settings than the original SGD algorithm, our experiment results in Section 5.4.3 show that bad initial LR settings can cause the training time to be orders of magnitude longer (e.g., Figure 5.7) and/or cause the model to converge to suboptimal solutions (e.g., Figure 5.6). Hence, MLtuner complements these adaptive LR algorithms, in that users can use MLtuner to pick the initial LR for more robust performance. Moreover, practitioners also find that sometimes using these adaptive LR tuning algorithms alone is not enough to achieve the optimal model solution, especially for complex models such as deep neural networks. For example, Szegedy et al. [83] reported that they used LR decaying together with RMSProp to train their Inception-v3 model.

5.2 MLtuner: more efficient automatic tuning

This section describes the high level design of our *MLtuner* approach.

5.2.1 MLtuner overview

MLtuner automatically tunes training tunables with low overhead, and will dynamically re-tune tunables during the training. MLtuner is a light-weight system that can be connected to existing training systems, such as a parameter server. MLtuner sends the tunable setting trial instructions to the training system and receives training feedback (e.g., per-clock training losses) from the training system. The detailed training system interfaces will be

described in Section 5.3.5. Similar to the other hyperparameter tuning approaches, such as Spearmint [78], Hyperband [58], and TuPAQ [80], MLtuner requires users to specify the tunables to be tuned, with the type—either discrete, continuous in linear scale, or continuous in log scale—and range of valid values.

5.2.2 Trying and evaluating tunable settings

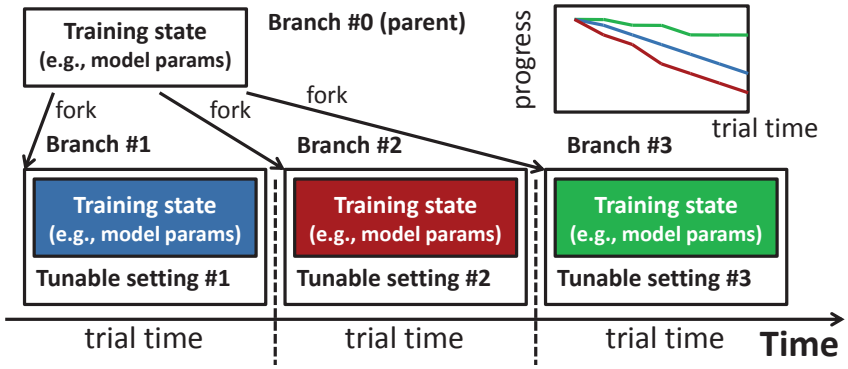


Figure 5.1: Trying tunable settings in training branches. The red branch with tunable setting #2 has the fastest convergence speed.

MLtuner evaluates tunable settings by trying them in forked *training branches*. The training branches are forked from the same consistent snapshot of some initial training state (e.g., model parameters, worker-local state, and training data), but are assigned with different tunable settings to train the model. As is illustrated in Figure 5.1, MLtuner schedules each training branch to run for some automatically decided amount of *trial time*, and collects their *training progress* to measure their *convergence speed*. MLtuner will fork multiple branches to try different settings, and then pick only the branch with the fastest convergence speed to keep training, and kill the other branches. In our example applications, such as deep learning and matrix factorization, the training system reports the per-clock training losses to MLtuner as the training progress.

The training branches are scheduled by MLtuner in a *time-sharing* manner, running in the same training system instance on the same set of machines. We made this design choice, rather than running multiple training branches in parallel on different sets of machines, for three reasons. First, this design avoids the use of extra machines that are just for the trials; otherwise, the extra machines will be wasted when the trials are not running (which is most of the time). Second, this design allows us to use the same hardware setup (e.g., number of machines) for both the tuning and the actual training; otherwise, the setting found on a different hardware setup would be suboptimal for the actual training. Third, running all branches in the same training system instance helps us achieve low overhead for forking and switching between branches, which are now simply memory copying of training state within the same process and choosing the right copy to use. Also, some resources, such as cache memory and immutable training data, can be shared among branches without duplication.

5.2.3 Tunable tuning procedure

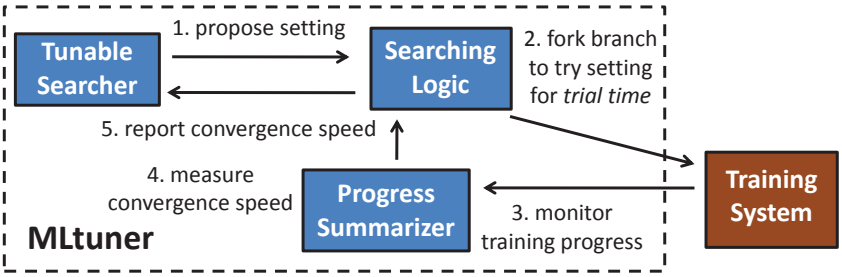


Figure 5.2: MLtuner tuning procedure.

Figure 5.2 illustrates the tuning procedure of MLtuner. MLtuner first tags the current training state as the parent branch. Then, inside the tuning loop, MLtuner uses a *tunable searcher* module (described in Section 5.3.3) to decide the next tunable setting to evaluate. For each proposed trial setting, MLtuner will instruct the training system to fork a trial branch from the parent branch to train the model for some amount of *trial time* with the trial setting. Section 5.3.2 will describe how MLtuner automatically decides the trial time. Then, MLtuner will collect the training progress of the trial branch from the training system, and use the *progress summarizer* module (described in Section 5.3.1) to summarize its *convergence speed*. The convergence speed will be reported back to the tunable searcher to guide its future tunable setting proposals. MLtuner uses this tuning procedure to tune tunables at the beginning of the training, as well as to re-tune tunables during training. Re-tuning will be described in Section 5.3.4.

5.2.4 Assumptions and limitations

The design of MLtuner relies on the assumption that the good tunable settings (in terms of completion time and converged model quality) can be decided based on their convergence speeds measured with a relatively short period of trial time. The same assumption has also been made by many of the state-of-the-art hyperparameter tuning approaches. For example, both Hyperband and TuPAQ stop some of the trial hyperparameter settings early, based on the model qualities achieved in the early stage of the training. This assumption does not always hold for model hyperparameter tuning. For example, a more complex model often takes more time to converge but will eventually converge to a better solution. For most of the training tunables, we find this assumption holds for all the applications that we have experimented with so far, including image classification on two different datasets with three different deep neural networks, video classification, and matrix factorization. That is because the training tunables only control the training procedure but do not change the model. That is also the reason why we do not suggest using MLtuner to tune the model hyperparameters. For use cases where both training tunables and model hyperparameters need to be tuned, users can use MLtuner in the inner loop to tune the training tunables, and use the existing model hyperparameter tuning approaches in the outer loop to tune the model hyperparameters.

5.3 MLtuner implementation details

This section describes the design and implementation details of MLtuner.

5.3.1 Measuring convergence speed

The progress summarizer module takes in the training progress trace (e.g., a series of training loss) of each trial branch, and outputs the convergence speed. The progress trace has the form of $\{(t_i, x_i)\}_{i=1}^{N}$, where t_i is the timestamp, and x_i is the progress. In this section, we assume x is the training loss, and a smaller x value means better convergence.

Downsampling the progress trace. The most straightforward way of measuring the convergence speed is to use the slope of the progress trace: $s = \frac{|x_N - x_1|}{t_N - t_1}$. However, in many ML applications, such as deep neural network training with SGD, the progress trace is often quite noisy, because the training loss points are computed on different batches of the training data. We find the convergence speed measured with just the first and last point of the progress trace is often inaccurate. To deal with the noisiness, the progress summarizer will *downsample* the progress trace of each branch, by uniformly dividing the progress trace into K non-overlapping windows. The value of each window will be calculated as the average of all data points in it. For a downsampled progress trace of $\{(\tilde{t}_i, \tilde{x}_i)\}_{i=1}^{K}$, its slope will be $\tilde{s} = \frac{-range(\tilde{x})}{range(\tilde{t})}$, where $range(\tilde{x}) = \tilde{x}_K - \tilde{x}_1$ and $range(\tilde{t}) = \tilde{t}_K - \tilde{t}_1$. We will describe how we decide K later in this section.

Penalizing unstable branches. Even with downsampling, calculating the slope by simply looking at the first and last downsampled points might still treat branches with unstable jumpy loss as good converging branches. To deal with this problem, the summarizer module will adjust the convergence speed estimation of each branch according to its noisiness. Ideally, the loss of a noise-free trace $\tilde{x}^{(nf)}$ should be monotonically decreasing, so we estimate the noisiness of a trace $\tilde{x}$ as $noise(\tilde{x}) = \max\,(\max_{1 \le i \le K-1} (\tilde{x}_{i+1} - \tilde{x}_i), 0)$, which is the maximum magnitude that a point goes up from the previous point. In order to make a conservative estimation of convergence speeds, our progress summarizer will penalize the convergence speed of each branch with its noise:
$speed = \max\,(\frac{-range(\tilde{x}) - noise(\tilde{x})}{range(\tilde{t})}, 0)$. which is a positive value for a converging branch and zero for a diverged branch. We report zero as the convergence speed of a diverged branch, rather than reporting it as a negative value, because we find it is usually wrong to treat a diverged branch with smaller diverged loss as a better branch than other diverged branches. We treat diverged branches as of the same quality.

Convergence and stability checks. The progress summarizer will check the convergence and stability of each branch, and assign one of the three labels to them: `converging`, `diverged`, or `unstable`. It labels a branch as `converging`, if $range(\tilde{x}) < 0$ and $noise(\tilde{x}) < \epsilon \times |range(\tilde{x})|$. We will describe how we decide ϵ later in this section. It labels a branch as `diverged`, if the training encounters numerically overflowed numbers. Finally, it labels all the other branches as `unstable`, meaning that their convergence speeds might need longer trials to evaluate. With a longer trial time, an unstable branch might become stable, because its $|range(\tilde{x})|$ is likely to increase because of the longer training, and its $noise(\tilde{x})$

is likely to decrease because of more points being averaged in each downsampling window.

Deciding number of samples and stability threshold. The previously described progress summarizer module has two knobs, the number of samples K and the stability threshold ϵ. Since the goal of MLtuner is to free users from tuning, K and ϵ do not need to be tuned by users either. To decide K, we will consider an extreme case, where the true convergence progress $|range(\tilde{x}^{(nf)})|$ is zero, and $\tilde{x}$ is all white noise, following a normal distribution. This trace will be falsely labelled as `converging`, if $\{\tilde{x}_i\}_{i=1}^{K}$ is monotonically decreasing, and this probability is less than $(\frac{1}{2})^K$. Hence, we need a large enough K to bound this false positive probability. We decide to set $K = 10$ to counter the noisiness, so that the false positive probability is less than 0.1%. The ϵ configuration bounds the magnitude (relative to $|range(\tilde{x})|$) that each point in the progress trace is allowed to go up from the previous point. On average, if we approximate the noise-free progress trace $\tilde{x}^{(nf)}$ as a straight line, each point is expected to go down from the previous point by $\approx \frac{|range(\tilde{x})|}{K}$. Hence, MLtuner sets ϵ to $\frac{1}{K}$, so that a `converging` trace will have no point going up by more than it is expected to go down. Our experiments in Section 5.4 show that the same settings of K and ϵ work robustly for all of our application benchmarks.

5.3.2 Deciding tunable trial time

Unlike traditional tuning approaches, MLtuner automatically decides tunable trial time, based on the noisiness of training progress, so that the trial time is just long enough for good tunable settings to have stable converging progress. Algorithm 6 illustrates the trial time decision procedure. MLtuner first initializes the trial time to a small value, such as making it as long as the decision time of the tunable searcher, so that the decision time will not dominate. While MLtuner tries tunable settings, if none of the settings tried so far is labelled as `converging` with the current trial time, MLtuner will double the trial time and use the doubled trial time to try new settings as well as the previously tried settings for longer. When MLtuner successfully finds a stable converging setting, the trial time is decided and will be used to evaluate future settings.

5.3.3 Tunable searcher

The tunable searcher is a replaceable module that searches for a best tunable setting that maximizes the convergence speed. It can be modeled as a black-box function optimization problem (i.e., bandit optimization), where the function input is a tunable setting, and the function output is the achieved convergence speed. MLtuner allows users to choose from a variety of optimization algorithms, with a general tunable searcher interface. In our current implementation, we have implemented and explored four types of searchers, including RandomSearcher, GridSearcher, BayesianOptSearcher, and HyperOptSearcher.

The simplest RandomSearcher just samples settings uniformly from the search space, without considering the convergence speeds of previous trials. GridSearcher is similar to RandomSearcher, except that it discretizes the continuous search space into a grid, and proposes each of the discretized settings in the grid. Despite its simplicity, we find GridSearcher works surprisingly well for low-dimensional cases, such as when there is only

Algorithm 6 MLtuner trial time decision.

```
trialTime ← 0
Parent branch ← current model state
while none of the settings is converging do
    Get tunableSetting from tunable searcher
    trialTime ← max(trialTime, searcherDecisionTime)
    if tunableSetting is not empty then
        Fork a branch from the parent branch with tunableSetting
        Append the new branch to trialBranches
    end if
    for each branch in trialBranches do
        Schedule branch to run for trialTime − branch.runTime
    end for
    Summarize the progress of all trialBranches
    Remove diverged branches from trialBranches
    if any branch in trialBranches is converging then
        bestSetting ← tunable setting that has the best convergence
        Free the non-best branches
        Trial time decided and break out the loop
    else
        trialTime ← trialTime × 2
    end if
end while
Keep searching with trialTime
```

one tunable to be searched. For high-dimensional cases, where there are many tunables to be searched, we find it is better to use bandit optimization algorithms, which spend more searching efforts on the more promising part of the search space. Our BayesianOptSearcher uses the Bayesian optimization algorithm, implemented in the Spearmint [78] package, and our HyperOptSearcher uses the HyperOpt [9] algorithm. Through our experiments, we find HyperOptSearcher works best among all the searcher choices for most use cases, and MLtuner uses it as its default searcher.

The tunable searcher (except for GridSearcher) need a stopping condition to decide when to stop searching. Generally, it can just use the default stopping condition that comes with the optimization packages. Unfortunately, neither the Spearmint nor HyperOpt package provides a stopping condition. They all rely on users to decide when to stop. After discussing with many experienced ML practitioners, we used a rule-of-thumb stopping condition for hyperparameter optimization, which is to stop searching when the top five best (non-zero) convergence speeds differ by less than 10%.

5.3.4 Re-tuning tunables during training

MLtuner re-tunes tunables, when the training stops making further converging progress (i.e., considered as converged) with the current tunable setting. We have also explored designs that re-tune tunables more aggressively, before the converging progress stops, but we did not choose those designs for two reasons. First, we find the cost of re-tuning usually outweighs the increased convergence rate coming from the re-tuned setting. Second, we

Method name	Input	Description
	Messages sent from MLtuner	
`ForkBranch`	(clock, branchId,\ parentBranchId, tunable, [type])	fork a branch by taking a\ consistent snapshot at clock
`FreeBranch`	(clock, branchId)	free a branch at clock
`ScheduleBranch`	(clock, branchId)	schedule the branch to run at clock
	Messages sent to MLtuner	
`ReportProgress`	(clock, progress)	report per-clock training progress

Table 5.1: MLtuner message signatures.

find, for some complex deep neural network models, re-tuning too aggressively might cause them to converge to suboptimal local minimas.

To re-tune, the most straightforward approach is to use exactly the same tuning procedure as is used for initial tuning, which was our initial design. However, some practical issues were found, when we deployed it in practice. For example, re-tuning happens when the training stops making converging progress, but, if the training has indeed converged to the optimal solution and no further converging progress can be achieved with any tunable setting, the original tuning procedure will be stuck in the searching loop for ever.

To address this problem, we find it is necessary to bound both the per-setting trial time and the number of trials to be performed for each re-tuning. For the deep learning applications used in Section 5.4, MLtuner will bound the per-setting trial time to be at most one epoch (i.e., one whole pass over the training data), and we find, in practice, this bound usually will not be reached, unless the model has indeed converged. MLtuner also bounds the number of tunable trials of each re-tuning to be no more than the number of trials of the previous re-tuning. The intuition is that, as more re-tunings are performed, the likelihood that a better setting is yet to be found decreases. These two bounds together will guarantee that the searching procedure can successfully stop for a converged model.

5.3.5 Training system interface

MLtuner works as a separate process that communicates with the training system via messages. Table 5.1 lists the message signatures. MLtuner identifies each branch with a unique *branch ID*, and uses *clock* to indicate logical time. The clock is unique and totally ordered across all branches. When MLtuner forks a branch, it expects the training system to create a new training branch by taking a consistent snapshot of all state (e.g., model parameters) from the parent branch and use the provided tunable setting for the new branch. When MLtuner frees a branch, the training system can then reclaim all the resources (e.g., memory for model parameters) associated with that branch. MLtuner sends the branch operations in clock order, and it sends exactly one `ScheduleBranch` message for every clock. The training system is expected to report its training progress with the `ReportProgress` message every clock.

Although in our MLtuner design, the branches are scheduled based on time, rather than clocks, our MLtuner implementation actually sends the per-clock branch schedules to the

training system. We made this implementation choice, in order to ease the modification of the training systems. To make sure that a trial branch runs for (approximately) the amount of scheduled trial time, MLtuner will first schedule that branch to run for some small number of clocks (e.g., three) to measure its per-clock time, and then schedule it to run for more clocks, based on the measured per-clock time. Also, because MLtuner consumes very few CPU cycles and little network bandwidth, users do not need to dedicate a separate machine for it. Instead, users can just run MLtuner on one of the training machines.

Distributed training support. Large-scale machine learning tasks are often trained with distributed training systems (e.g., with a parameter server architecture). For a distributed training system with multiple training workers, MLtuner will broadcast the branch operations to all the training workers, with the operations in the same order. MLtuner also allows each of the training workers to report their training progress separately, and MLtuner will aggregate the training progress with a user-defined aggregation function. For all the SGD-based applications in this paper, where the training progress is the loss computed as the sum of the training loss from all the workers, this aggregation function just does the sum.

Evaluating the model on validation set. For some applications, such as the classification tasks, the model quality (e.g., classification accuracy) is often periodically evaluated on a set of validation data during the training. This can be easily achieved with the branching support of MLtuner. To test the model, MLtuner will fork a branch with a special `TESTING` flag as the *branch type*, telling the training system to use this branch to test the model on the validation data, and MLtuner will interpret the reported progress of the testing branch as the validation accuracy.

5.3.6 Training system modifications

This section describes the possible modifications to be made, for a training system to work with MLtuner. The modified training system needs to keep multiple versions of its training state (e.g., model parameters and local state) as multiple training branches, and switch between them during training.

We have modified two state-of-the-art training systems to work with MLtuner: IterStore (Chapter 3), a generic parameter server system. and GeePS (Chapter 4), a parameter server with specialized support for GPU deep learning. Both parameter server implementations keep the parameter data as key-value pairs in memory, sharded across all worker machines in the cluster. To make them work with MLtuner, we modified their parameter data storage modules to keep multiple versions of the parameter data, by adding branch ID as an additional field in the index. When a new branch is forked, the modified systems will allocate the corresponding data storage for it (from a user-level memory pool managed by the parameter server) and copy the data from its parent branch. When a branch is freed, all its memory will be reclaimed to the memory pool for future branches. The extra memory overhead depends on the maximum number of co-existing active branches, and MLtuner is designed to keep as few active branches as possible. Except when exploring the trial time (with Algorithm 6), MLtuner needs only three active branches to be kept, the parent branch, the current best branch, and the current trial branch. Because the parameter server

Application	Model	Supervised or not	Clock size	Hardware
Image classification	Convolutional NN	Supervised learning	One mini-batch	GPU
Video classification	Recurrent NN	Supervised learning	One mini-batch	GPU
Movie recommendation	Matrix factorization	Unsupervised learning	Whole data pass	CPU

Table 5.2: Applications used in the experiments. They have distinct characteristics.

system shards its parameter data across all machines, it is usually not an issue to keep those extra copies of parameter data in memory. For example, the Inception-BN [46] model, which is the state-of-art convolutional deep neural network for image classification, has less than 100 MB of model parameters. When we train this model on an 8-machine cluster, the parameter server shard on each machine only needs to keep 12.5 MB of the parameter data (in CPU memory rather than GPU memory). A machine with 50 GB of CPU memory will be able to keep 4000 copies of the parameter data in memory.

Those parameter server implementations also have multiple levels of caches. For example, both parameter server implementations cache parameter data locally at each worker machine. In addition to the machine-level cache, IterStore also provides a distinct thread-level cache for each worker thread, in order to avoid lock contention. GeePS has a GPU cache that keeps data in GPU memory for GPU computations. Since MLtuner runs only one branch at a time, the caches do not need to be duplicated. Instead, all branches can share the same cache memory, and the shared caches will be cleared each time MLtuner switches to a different branch. In fact, sharing the cache memory is critical for GeePS to work with MLtuner, because there is usually not enough GPU memory for GeePS to allocate multiple GPU caches for different branches.

5.4 Evaluation

This section evaluates MLtuner on several real machine learning benchmarks, including image classification with three different models on two different datasets, video classification, and matrix factorization. Table 5.2 summarizes the distinct characteristics of these applications. The results confirm that MLtuner can robustly tune and re-tune the tunables for ML training, and is over an order of magnitude faster than state-of-the-art ML tuning approaches.

5.4.1 Experimental setup

Application setup

Image classification using convolutional neural networks. Image classification is a supervised learning task that trains a deep convolutional neural network (CNN) from many labeled training images. The first layer of neurons (input of the network) is the raw pixels of the input image, and the last layer (output of the network) is the predicted probabilities that the image should be assigned to each of the labels. There is a *weight* associated with each neuron connection, and those weights are the model parameters that will be trained

from the input (training) data. Deep neural networks are often trained with the SGD algorithm, which samples one *mini-batch* of the training data every clock and computes gradients and parameter updates based on that mini-batch [29, 20, 28, 52, 82, 46, 83, 41]. As an optimization, gradients are often smoothed across mini-batches with the *momentum* method [81].

We used two datasets and three models for the image classification experiments. Most of our experiments used the Large Scale Visual Recognition Challenge 2012 (ILSVRC12) dataset [76], which has 1.3 million training images and 5000 validation images, labeled to 1000 classes. For this dataset, we experimented with two popular convolutional neural network models, Inception-BN [46] and GoogLeNet [82].[1] Some of our experiments also used a smaller Cifar10 dataset [51], which has 50,000 training images and 10,000 validation images, labeled to 10 classes. We used AlexNet [52] for the Cifar10 experiments.

Video classification using recurrent neural networks. To capture the sequence information of videos, a video classification task often uses a *recurrent neural network* (RNN), and the RNN network is often implemented with a special type of recurrent neuron layer called *Long-Short Term Memory* (LSTM) [43] as the building block [31, 90, 94]. A common approach for using RNNs for video classification is to first encode each image frame of the videos with a convolutional neural network (such as GoogLeNet), and then feed the sequences of the encoded image feature vectors into the LSTM layers.

Our video classification experiments used the UCF-101 dataset [79], with about 8,000 training videos and 4,000 testing videos, categorized into 101 human action classes. Similar to the approach described by Donahue et al. [31], we used the GoogLeNet [82] model, trained with the ILSVRC12 image data, to encode the image frames, and fed the feature vector sequences into the LSTM layers. We extracted the video frames at a rate of 30 frames per second and trained the LSTM layers with randomly selected video clips of 32 frames each.

Movie recommendation using matrix factorization. The movie recommendation task tries to predict unknown user-movie ratings, based on a collection of known ratings. This task is often modeled as a *sparse matrix factorization* problem, where we have a partially filled matrix X, with entry (i, j) being user i's rating of movie j, and we want to factorize X into two low ranked matrices L and R, such that their product approximates X (i.e., $X \approx L \times R$) [35]. The matrix factorization model is often trained with the SGD algorithm [35], and because the model parameter values are updated with uneven frequency, practitioners often use AdaGrad [32] or AdaRevision [64] to adaptively decide the per-parameter learning rate adjustment from a specified initial learning rate [92]. Our matrix factorization (MF) experiments used the Netflix dataset, which has 100 million known ratings from 480 thousand users for 18 thousand movies, and we factorize the rating matrix with a rank of 500.

Training methodology and performance metrics. Unless otherwise specified, we train the image classification and video classification models using the standard SGD algorithm with momentum, and shuffle the training data every *epoch* (i.e., a whole pass over the training data). The gradients of each training worker are normalized with the training

[1]The original papers did not release some minor details of their models, so we used the open-sourced versions of those models from the Caffe [47] and MXNet [18] repositories.

Tunable	Valid range
Learning rate	10^x, where $x \in [-5, 0]$
Momentum	DNN apps: $x \in [0.0, 1.0]$ Matrix factorization: N/A
Per-machine batch size	Inception-BN/GoogLeNet: $x \in \{2, 4, 8, 16, 32\}$ AlexNet: $x \in \{4, 16, 64, 256\}$ RNN: $x \in \{1\}$ Matrix factorization: N/A
Data staleness	$x \in \{0, 1, 3, 7\}$

Table 5.3: Tunable setups in the experiments.

batch size before sending to the parameter server, where the learning rate and momentum are applied. For those supervised classification tasks, the quality of the trained model is defined as the classification accuracy on a set of validation data, and our experiments will focus on both the convergence time and the converged validation accuracy as the performance metrics. Generally, users will need to specify the convergence condition, and in our experiments, we followed the common practice of other ML practitioners, which is to test the validation accuracy every epoch and consider the model as converged when the validation accuracy plateaus (i.e., does not increase any more) [41, 52, 83]. Because of the noisiness of the validation accuracy traces, we consider the ILSVRC12 and video classification benchmarks as converged when the accuracy does not increase over the last 5 epochs, and considered the Cifar10 benchmark as converged when the accuracy does not increase over the last 20 epochs. Because MLtuner trains the model for one more epoch after each re-tuning, we configure MLtuner to start re-tuning one epoch before the model reaches the convergence condition in order to be fair to the other setups. Note that, even though the convergence condition is defined in terms of the validation accuracy, MLtuner still evaluates tunable settings with the reported training loss, because the training loss can be obtained every clock, whereas the validation accuracy is only measured every epoch (usually thousands of clocks for DNN training).

For the MF task, we define one clock as one whole pass over all training data, without mini-batching. Because MF is an unsupervised learning task, we define its convergence condition as a fixed training loss value (i.e., the model is considered as converged when it reaches that loss value), and use the convergence time as a single performance metric, with no re-tuning. Based on guidance from ML experts and related work using the same benchmark (e.g., [44]), we decided the convergence loss threshold as follows: We first picked a relatively good tunable setting via grid search, and kept training the model until the loss change was less than 1% over the last 10 iterations. The achieved loss value is set as the convergence loss threshold, which is 8.32×10^6 for our MF setup.

MLtuner setup

Table 5.2 summarizes the tunables to be tuned in our experiments. The tunable value ranges (except for the batch size) are the same for all benchmarks, because we assume little prior knowledge from users about the tunable settings. The (per-machine) batch size

ranges are different for each model, decided based on the maximum batch size that can fit in the GPU memory. For the video classification task, we can only fit one video in a batch, so the batch size is fixed to one.

Except for specifying the tunables, MLtuner does not require any other user configurations, and we used the same default configurations (e.g., HyperOpt as the tunable searcher and 10 samples for downsampling noisy progress) for all experiments. An application reports its training loss as the training progress to MLtuner every clock.

Training system and cluster setup

For the deep neural network experiments, we use GeePS [28] connected with Caffe [47] as the training system, running distributed on 8 GPU machines (8 ML workers + 8 server shards). Each machine has one NVIDIA Titan X GPU, with 12 GB of GPU device memory. In addition to the GPU, each machine has one E5-2698Bv3 Xeon CPU (2.0 GHz, 16 cores with 2 hardware threads each) and 64 GB of RAM, running 64-bit Ubuntu 16.04, CUDA toolkit 8.0, and cuDNN v5. The machines are inter-connected via 40 Gbps Ethernet.

The matrix factorization experiments use IterStore [27] as the training system, running distributed on 32 CPU machines (32 ML workers + 32 server shards). Each machine has four quad-core AMD Opteron 8354 CPUs (16 physical cores in total) and 32 GB of RAM, running 64-bit Ubuntu 14.04. The machines are inter-connected via 20 Gb Infiniband.

5.4.2 MLtuner vs. state-of-the-art auto-tuning approaches

This section experimentally compares our MLtuner approach with the state-of-the-art hyperparameter tuning approaches, Spearmint [78] and Hyperband [58]. To control for other performance factors, we implemented the tuning logics of those state-of-the-art approaches in our MLtuner system. All setups tune the same four tunables listed in Table 5.3. The Spearmint approach samples tunable settings with the Bayesian optimization algorithm and trains the model to completion to evaluate each tunable setting.[2] For the Hyperband approach, we followed the "Hyperband (Infinite horizon)" algorithm [58], because the total number of epochs for the model to converge is unknown. The Infinite horizon Hyperband algorithm starts the searching with a small budget and doubles the budget over time. For each given budget, Hyperband samples tunable settings randomly from the search space, and every few iterations, it will stop the half of configurations being tried that have lower validation accuracies.

Figure 5.3 shows the runtime and achieved validation accuracies of Inception-BN on ILSVRC12 and AlexNet on Cifar10. For the larger ILSVRC12 benchmark, MLtuner performs much better than Hyperband and Spearmint. After 5 days, Spearmint reached only 6% accuracy, and Hyperband reached only 49% accuracy, while MLtuner converged to 71.4% accuracy in just 2 days. The Spearmint approach performs so badly because the first tunable setting that it samples sets all tunables to their minimum values (learning rate=1e-5, momentum=0, batch size=2, data staleness=0), and the small learning rate

[2]We used Spearmint's open-sourced Bayesian optimization implementation from `https://github.com/HIPS/\unhbox\voidb@x\hbox{Spearmint}` as of September 14, 2016.

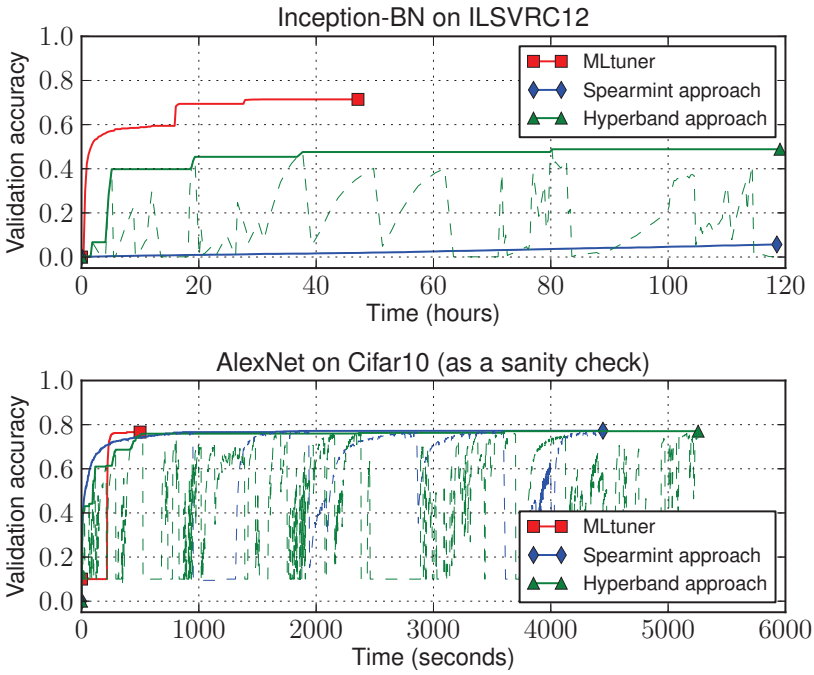


Figure 5.3: Runtime and accuracies of MLtuner and the state-of-the-art approaches. For Spearmint and Hyperband, the dashed curves show the accuracies of each configuration tried, and the bold curves show the maximum accuracies achieved over time.

and batch size cause the model to converge at an extremely slow rate. We have tried running Spearmint multiple times, and found their Bayesian optimization algorithm always proposes this setting as the first one to try. We also show the results on the smaller Cifar10 benchmark as a sanity check, because previous hyperparameter tuning work only reports results on this small benchmark. For the Cifar10 benchmark, all three approaches converged to approximately the same validation accuracy, but MLtuner is 9× faster than Hyperband and 3× faster than Spearmint.[3]

Compared to previous approaches, MLtuner converges to much higher accuracies in much less time. The accuracy jumps in the MLtuner curves are caused by re-tunings. Figure 5.4 gives a more detailed view of MLtuner's tuning/re-tuning behavior. MLtuner re-tunes tunables when the validation accuracy plateaus, and the results shows that the accuracy usually increases after the re-tunings. This behavior echoes experts' findings that, when training deep neural networks, it is necessary to change (usually decrease) the learning rate during training, in order to get good validation accuracies [83, 46, 82, 52, 41, 97]. For the larger ILSVRC12 and RNN benchmarks, there is little overhead (2% to 6%) from the initial tuning stage, but there is considerable overhead from re-tuning, especially from the last re-tuning, when the model has already converged. That is because MLtuner assumes

[3]Since Spearmint and Hyperband do not have stopping conditions of deciding when to quit the searching, we measured the convergence time as the time for each setup to reach 76% validation accuracy. If we set the stopping condition of Spearmint as when the best 5 validation accuracies differ by less than 10%, MLtuner finished the training in 90% less time than Spearmint.

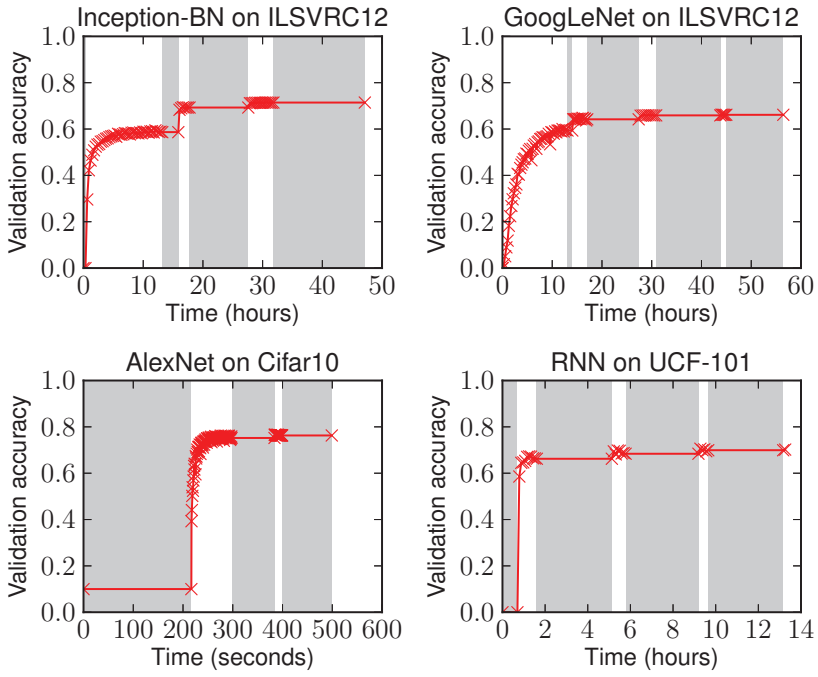


Figure 5.4: MLtuner tuning/re-tuning behavior on four deep learning benchmarks. The markers represent the validation accuracies measured at each epoch. The shaded time ranges are when MLtuner tunes/re-tunes tunables.

no knowledge of the optimal model accuracy that it is expected to achieve. Instead, it automatically finds the best achievable model accuracy via re-tuning.

Figure 5.5 shows the MLtuner results of multiple runs (10 runs for Cifar10 and 3 runs each for the other benchmarks). For each benchmark, MLtuner consistently converges to nearly the same validation accuracy. The number of re-tunings and convergence time are different for different runs. This variance is caused by the randomness of the HyperOpt algorithm used by MLtuner, as well as the inherent behavior of floating-point arithmetic when the values to be reduced arrive in a non-deterministic order. We observe similar behavior when not using MLtuner, due to the latter effect, which is discussed more in Section 5.4.4 (e.g., see Figure 5.9).

5.4.3 Tuning initial LR for adaptive LR algorithms

As we have pointed out in Section 5.1.2, the adaptive learning rate tuning algorithms, including AdaRevision [64], RMSProp [85], Nesterov [68], Adam [48], AdaDelta [96], and AdaGrad [32], still require users to pick the initial learning rate. This section will show that the initial learning rate settings of those adaptive LR algorithms still greatly impact the converged model quality and convergence time, and that MLtuner can be used to tune the initial learning rate for them. For this set of experiments, MLtuner only tunes the initial learning rate, and does not re-tune, so that MLtuner will not affect the behaviors of

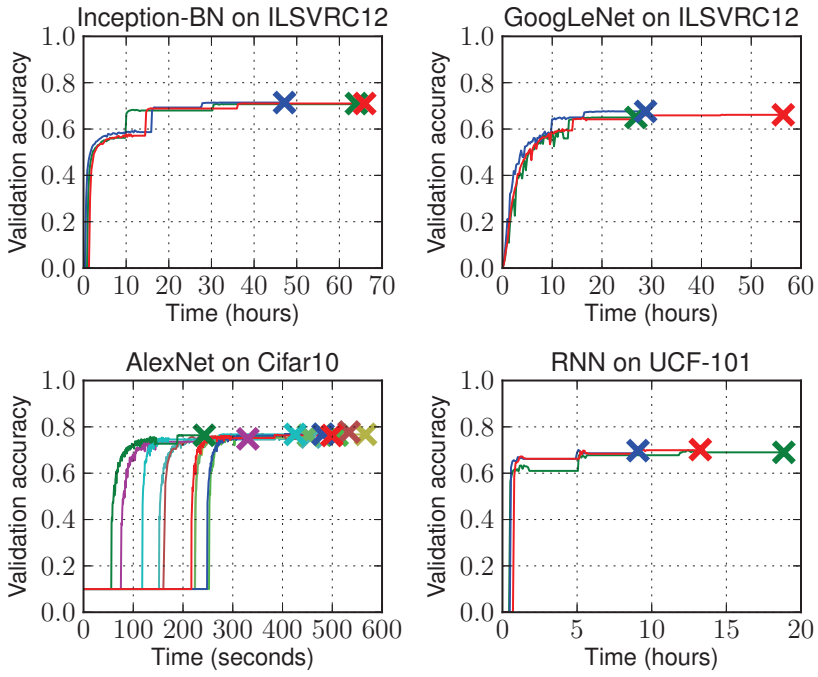


Figure 5.5: MLtuner results of multiple runs. The larger "x" markers mark the end of each run. The runs with the median performance are shown as the red curves in this figure and as the "MLtuner" curves in the other figures.

the adaptive LR algorithms.

Tuning initial LR improves solution quality

Figure 5.6 shows the converged validation accuracies of AlexNet on Cifar10 with different adaptive LR algorithms and different initial learning rate settings. We used the smaller Cifar10 benchmark, so that we can afford to train the model to convergence with many different initial LR settings. For the other tunables, we used the common default values (momentum=0.9, batch size=256, data staleness=0) that are frequently suggested in the literature [52, 82, 46, 28]. The results show that the initial LR setting greatly affects the converged accuracy, that the best initial LR settings differ across adaptive LR algorithms, and that the optimal accuracy for a given algorithm can only be achieved with one or two settings in the range. The result also shows that MLtuner can effectively pick good initial LRs for those adaptive LR algorithms, achieving close-to-ideal validation accuracy. The graph shows only the tuning result for RMSProp because of limited space, but for all the 6 adaptive LR algorithms, the accuracies achieved by MLtuner differ from those with the optimal setting by less than 2%.

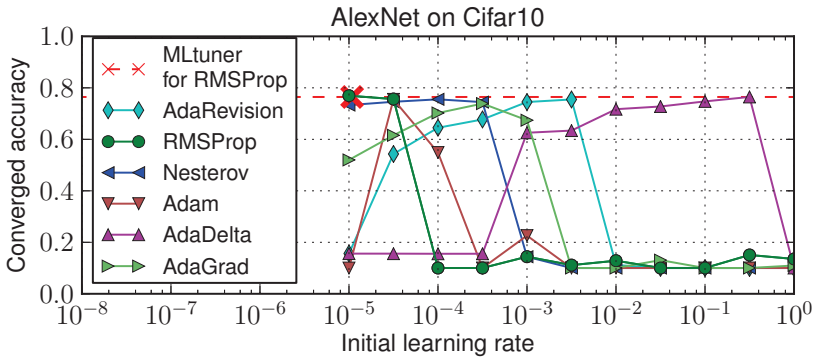


Figure 5.6: Converged validation accuracies when using different initial learning rates. The "x" marker marks the LR picked by MLtuner for RMSProp.

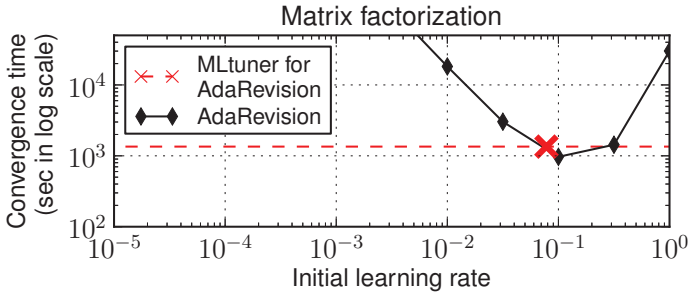


Figure 5.7: Convergence time when using different initial learning rates. The "x" marker marks the LR picked by MLtuner.

Tuning initial LR improves convergence time

Figure 5.7 shows the convergence time when using different initial AdaRevision learning rate settings, for the matrix factorization benchmark. Because the model parameters of the MF task have uneven update frequency, practitioners often use AdaRevision [64] to adjust its per-parameter learning rates [92]. Among all settings, more than 40% of them caused the model to converge over an order of magnitude slower than the optimal setting. We also show that, when tuning the initial LR with MLtuner, the convergence time (including the MLtuner tuning time) is close to ideal and is over an order of magnitude faster than leaving the initial LR un-tuned.

5.4.4 MLtuner vs. idealized manually-tuned settings

Figure 5.8 compares the performance of MLtuner, automatically tuning all four tunables listed in Table 5.3, with an idealized "manually tuned" configuration of the tunable settings. The intention is to evaluate MLtuner's overhead relative to what an expert who already figured out the best settings (e.g., by extensive previous experimentation) might use.

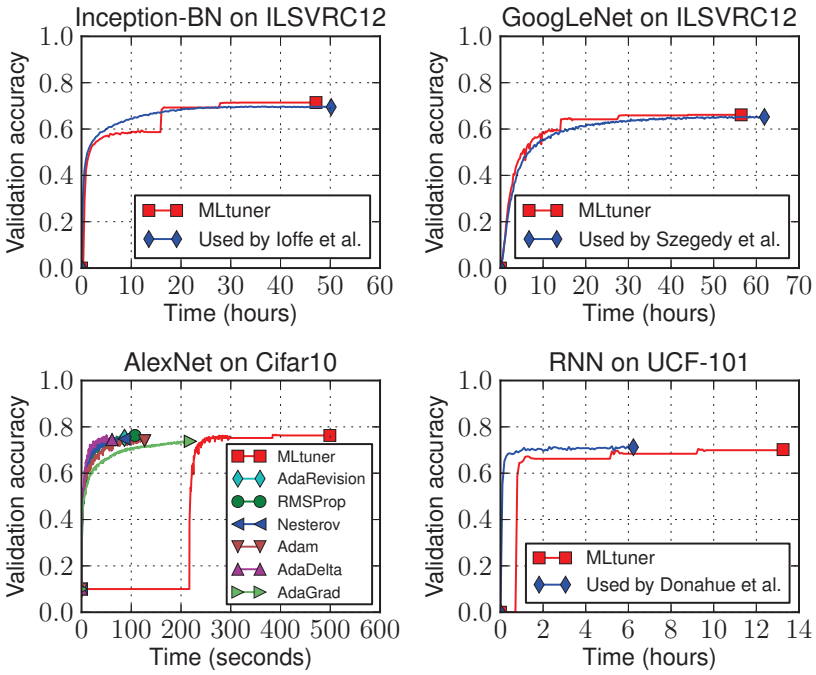


Figure 5.8: MLtuner compared with manually tuned settings. For comparison purpose, we have run the manually tuned settings (except for Cifar10) for long enough to ensure that their accuracies will not increase any more, rather than stopping them according to the convergence condition.

For the Cifar10 benchmark, we used the optimal initial LRs for the adaptive algorithms, found via running all possible settings to completion, and used effective default values for the other tunables (m=0.9, bs=256, ds=0). The results show that, among all the adaptive algorithms, RMSProp has the best performance. Compared to the best RMSProp configuration, MLtuner reaches the same accuracy, but requires about 5× more time.

For the other benchmarks, our budget does not allow us to run all the possible settings to completion to find the optimal ones. Instead, we compared with manually tuned settings suggested in the literature. For Inception-BN, we compared with the manually tuned setting suggested by Ioffe et al. in the original Inception-BN paper [46], which uses an initial LR of 0.045 and decreases it by 3% every epoch. For GoogLeNet, we compared with the manually tuned setting suggested by Szegedy et al. in the original GoogLeNet paper [82], which uses an initial LR of 0.0015, and decreases it by 4% every 8 epochs. For RNN, we compared with the manually tuned setting suggested by Donahue et al. [31], which uses an initial LR of 0.001, and decreases it by 7.4% every epoch. [4] All those manually tuned settings set momentum=0.9 and data staleness=0, and Inception-BN and GoogLeNet set batch size=32. Compared to the manually tuned settings, MLtuner achieved the same accuracies for Cifar10 and RNN, and higher accuracies for Inception-BN (71.4% vs. 69.8%)

[4] [31] does not specify the tunable settings, but we found their settings in their released source code at `https://github.com/LisaAnne/lisa-caffe-public` as of April 16, 2017.

and GoogLeNet (66.2% vs. 64.4%). The higher MLtuner accuracies might be because of two reasons. First, those reported settings were tuned for potentially different hardware setups (e.g., number of machines), so they might be suboptimal for our setup. Second, those reported settings used fixed learning rate decaying rates, while MLtuner is more flexible and can use any learning rate via re-tuning.

As expected, MLtuner requires more time to train than when an expert knows the best settings to use. The difference is 5× for the small Cifar10 benchmark, but is much smaller for the larger ILSVRC12 benchmarks, because the tuning times are amortized over much more training work. We view these results to be very positive, since knowing the ML task specific settings traditionally requires extensive experimentation that significantly exceeds MLtuner's overhead or even the much higher overheads for previous approaches like Spearmint and Hyperband.

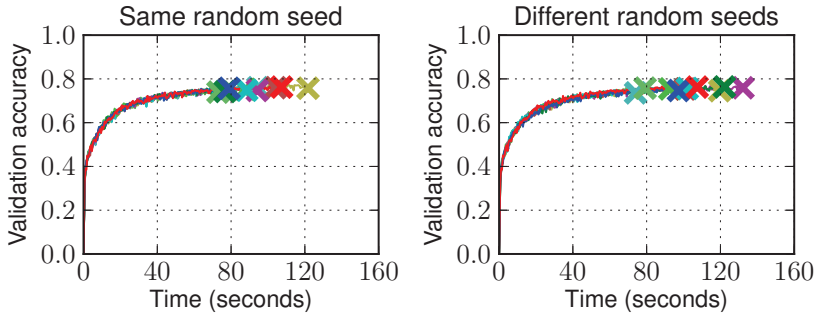


Figure 5.9: Performance for multiple training runs of AlexNet on Cifar10 with RMSProp and the optimal initial LR setting.

Figure 5.9 shows training performance for multiple runs of AlexNet with RMSProp using the same (optimal) initial LR. In the left graph, all runs initialize model parameters and shuffle training data with the same random seed. In the right graph, a distinct random seed is used for each run. We did 10 runs for each case and stopped each run when it reached the convergence condition. The result shows considerable variation in their convergence times across runs, which is caused by random initialization of parameters, training data shuffling, and non-deterministic order of floating-point arithmetic. The coefficients of variation (CoVs = standard deviation divided by average) of their convergence times are 0.16 and 0.18, respectively, and the CoVs of their converged accuracies are both 0.01. For the 10 MLtuner runs on the same benchmark shown in Figure 5.5, the CoV of the convergence time is 0.22, and the CoV of the converged accuracy is 0.01.

5.4.5 Robustness to suboptimal initial settings

This set of experiments studies the robustness of MLtuner. In particular, we turned off the initial tuning stage of MLtuner and had MLtuner use a hard-coded suboptimal tunable setting (picked randomly) as the initial setting. The result in Figure 5.10 shows that,

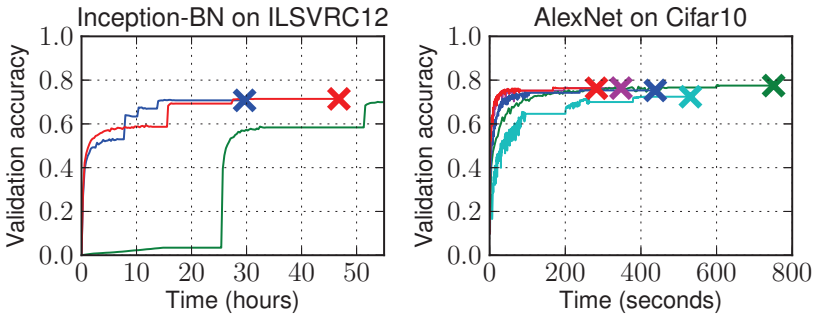


Figure 5.10: MLtuner performance with hard-coded initial tunable settings. The red curves used the tuned initial settings, and the other curves used randomly selected suboptimal initial settings.

even with suboptimal initial settings, MLtuner is still able to robustly converge to good validation accuracies via re-tuning.

5.4.6 Scalability with more tunables

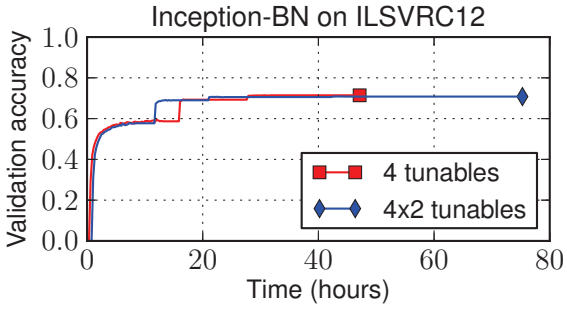


Figure 5.11: MLtuner performance with more tunables.

Figure 5.11 shows MLtuner's scalability to the number of tunables. For the "4×2 tunables" setup, we duplicated the 4 tunables listed in Table 5.3, making it a search space of 8 tunables. Except for making the search space larger, the added 4 tunables are transparent to the training system and do not control any other aspects of the training. The result shows that, with 8 tunables to be tuned, MLtuner still successfully converges to the same validation accuracy. The tuning time increases by about 2×, which is caused by the increased number of settings tried by HyperOpt before it reaches the stopping condition.

Chapter 6

Conclusion and Future Directions

6.1 Conclusion

This dissertation demonstrates that the characteristics of the ML tasks can be exploited in the design and implementation of parameter servers, to greatly improve the efficiency by an order of magnitude or more. To support this thesis statement, we designed three case study systems.

First, we designed IterStore, a parameter server system that exploits repeated parameter data access pattern of many ML applications. IterStore uses an efficient virtual iteration method to collect the repeating access sequence from the application with almost no overhead, and employs five parameter server specializations exploiting this information, including prefetching, contiguous marshalling-free data storage, locality and NUMA-aware data placement, and specialized caching policy. The experiments on matrix factorization, LDA, and PageRank show that these specializations decrease the training time by 33–98%.

Second, we designed GeePS, a parameter server that is specialized for deep learning applications on distributed GPUs. GeePS allows the application to access parameter data directly through GPU memory. By further exploiting the per-layer computation pattern of GPU applications, GeePS is able to efficiently hide data transfer latencies (both the ones between GPU memory and CPU memory in the same machine and the ones across different machines) from the application. GeePS is able to scale a state-of-art single-machine GPU-based deep learning system, Caffe, almost linearly, achieving 13× more training throughput with 16 machines. GeePS is also able to support training neural networks that do not fit in GPU memory, by efficiently swapping data to/from CPU memory in the background.

Third, we designed MLtuner, a system for automatically tuning settings for the machine learning training tunables, such as the learning rate, the mini-batch size, and the data staleness bound. MLtuner uses efficient snapshotting and optimization-guided online trial-and-error to find good initial tunable settings as well as to re-tune settings during execution. It can be easily linked with existing parameter server systems, such as IterStore and GeePS. Our experiments with five real ML tasks, including deep learning and matrix factorization, show that MLtuner can robustly find and re-tune tunable settings for them, and is over an order of magnitude faster than traditional hyperparameter tuning approaches.

6.2 Future Directions

This section discusses several future research directions of applying or extending our thesis work.

6.2.1 Detecting and adapting to access pattern changes

Chapter 3 shows that ML systems can exploit the characteristic of repeating data access pattern of many ML applications, to greatly improve their performance. While our results (Section 3.4.6) show that the performance gain is tolerant to certain amount of information accuracy, the case of dramatic access pattern changes still needs to be tackled. Dramatic access pattern changes can be caused by adding or removing workers, or training data changes. One interesting future research direction is to study ways of detecting the access pattern change and re-adapting the system to the new access pattern. Considering the possible overhead, it might make sense to only restructure the data layout, after the accumulated access pattern change becomes significant enough.

6.2.2 Time-aware iterativeness exploitation

The iterativeness characteristic can be further exploited beyond just repeating access patterns. The iterative ML applications execute the same (or nearly the same) computational routines every iteration. This means that they not only have the same data access sequence, but the computational time between two data accesses is also predictable. This characteristic is based on the assumption that the time spent on each computational routine is independent of the input values, which is often true (except for unlikely cases where the inputs are zero). The first research question is how the system can be specialized with the knowledge of the time-aware iterativeness information. We believe there are many possible specializations. For example, when we know the time between each data access, the system can do better prefetching to better overlap data transfer with computation. Another research question is how to collect the time information. The virtual iteration method proposed in Chapter 3 does not directly work, because the ML application does no computational work during the virtual iteration, so the time between each data access is unknown. One possible solution is to collect the iterativeness information in the first iteration, but, as Chapter 3 points out, the performance will be penalized because of the use of unoptimized data layout and the overhead of data migration. Another solution is to run the virtual iteration with computations, but with random data values. During the virtual iteration, when the application updates the parameter data, we discard the updates (to avoid the use of unoptimized data layout), and when the application reads data, we return random values.

6.2.3 Exploiting layer heterogeneity for deep learning

Deep neural networks often consist of multiple types of layers. Some layers (such as the fully-connected layers) have huge number of model parameters, but very little computational work, while some layers (such as the convolutional layers) have very few model parameters,

but consume the most computational time. There are also some layers (such as the ReLU and pooling layers) that apply static transformation from the input neurons to the output neurons, with no model parameters. We believe this layer heterogeneity characteristic can also be exploited to improve system performance. For example, if we know some data is accessed after a fully-connected layer, we should try to pin it in GPU memory, for faster access, and if some data is accessed after a convolutional layer, we can afford to put it in CPU memory, because there will be plenty of time for us to prefetch it. Moreover, different types of layers might have different tolerance to data staleness, and this can also be exploited.

6.2.4 Combing MLtuner with non-convex optimization techniques

Due to the non-convex nature of the deep neural network model, the converged DNN model solution often turns out to be a local minima rather than the global minima. As a result, people often do simulated annealing or random restarts to explore better model solutions in the non-convex space. As the future work, it is interesting to find out how MLtuner's auto-tuning support can be combined with those non-convex optimization techniques. In particular, it remains a question to see whether the tuning makes it easier or harder to explore the solution space. For example, because MLtuner controls the tunable settings, it might be possible that, even with different starting points, MLtuner might always take the model to some local minima solution, because of its choice of tunable settings. Future work needs to be done to study whether this happens and how to avoid it.

6.2.5 Extending MLtuner to model hyperparameter searching

The MLtuner system is able to efficiently tune the training tunables, with the intuition that a short trial time is often enough to evaluate a tunable choice. We have intentionally left model hyperparameters searching out of our scope, because of the uncertainty of whether the short trial times are enough to evaluate model hyperparameters. However, some recently work on this topic (e.g.,Hyperband [58]) suggests that model hyperparameter tuning with MLtuner is possible. They find that it usually not necessary to train the model to completion to evaluate a model hyperparameter choice, and early-stopping can be very effective. So it is interesting to see how we can extend our MLtuner approach to the model searching problem.

6.3 More Future Directions

This section discusses more interesting future research directions in the broader scope of system support for large-scale machine learning.

6.3.1 Dynamic data and/or models

Our thesis work has been focused on training an ML model *offline* from a static set of training data. However, in many real world applications, the training data often changes dynamically over time. For example, when we train a matrix factorization model for a movie recommendation service, the users might keep providing us new movie ratings, as they watch more movies. Moreover, the set of users and the set of movies might also grow over time. So we need to adapt our trained ML model to the new training data.

It will be interesting to explore the designs for ML system architectures and computational models that can adapt to the dynamic data and/or models. The research questions include:

- How to select the subset of training data that should be sampled and processed at every computational step? For dynamic use cases, each training datum is collected at different point of time. Simply processing the training data with the same probability is not feasible, because the data that was collected earlier would be sampled more often and have greater impact on the model. The newly coming data should probably be sampled with higher probability.
- When to re-train the model? For the case of static data/model, the training finishes when the model is converged. But for the dynamic data/model case, there is no such thing as convergence, because the data/model keeps changing. We will need to keep training the model as more data comes. However, it might not be necessary to always keep running the machine learning algorithms without any stop. For example, if the rate of the new data coming is relatively low, compared to the size of the already collected data, we can batch the newly coming data, and re-train the model, only after a large enough batch is collected.

6.3.2 Systems for both model training and model serving

People use machine learning models to predict the unseen data with the trained model, and the whole ML pipeline includes a model training stage, and a model serving stage (sometimes also called inference). The model serving stage answers queries according to the trained model, such as predicting the rating of a movie, or classifying an image. Our thesis work has been focused on the model training stage, and one interesting future research direction here is to explore the system designs for serving the models.

For the static data/model case, the model serving stage can be decoupled from the training stage. A straightforward approach is to train the model with a model training system (e.g., a parameter server), and send the trained ML model to a separate model serving system. The dynamic data/model case is more interesting, because the trained model keeps changing. It might be inefficient for a model training system to frequently send the trained model to a separate model serving system. It might make more sense to integrate the model training and model serving functionalities into a single system. The research questions regarding to designing such as system include:

- Consistency models. The ML applications usually do not need to use the most up-to-dated ML models for inference. Instead, they might be able to use a *slightly*

stale model, which allows us to trade model freshness for efficiency. One interesting research question is to quantify the amount of model staleness, and design consistency models to bound it.

- Resource allocation. The system will use computational resources for two types of computations, training the model and answering queries with the model. The most straightforward way of doing that is to partition all the resources into two disjoint groups, one for each type of computation. However, the model training computation and the model serving computation might have different bursts, so it might be interesting to exploit the potentials of resource sharing.

Bibliography

[1] NVIDIA cuBLAS `https://developer.nvidia.com/cublas`.

[2] NVIDIA cuDNN `https://developer.nvidia.com/cudnn`.

[3] 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 *OSDI*, 2016.

[4] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, 2012.

[5] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *PACT*, 2010.

[6] Y. Bengio, Y. LeCun, et al. Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5), 2007.

[7] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *ICML (1)*, 28:115–123, 2013.

[8] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.

[9] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.

[10] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *SOSP*, 1989.

[11] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 1998.

[12] A. Brown, T. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. In *ACM Transactions on Computer Systems (TOCS)*, 2001.

[13] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proc. VLDB Endow*, 2010.

[14] S. Byna, Y. Chen, X. Sun, R. Thakur, and W. Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *ACM/IEEE Supercomputing*, 2008.

[15] P. Cao, E. Felton, and K. Li. Implementation and performance of application-controlled file caching. In *OSDI*, 1994.

[16] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *ASPLOS*, 1994.

[17] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *OSDI*, 1999.

[18] 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.

[19] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[20] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[21] C.-T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2006.

[22] 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 *HotOS*, 2013.

[23] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *CVPR*, 2012.

[24] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *ICML*, 2013.

[25] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, 2014.

[26] H. Cui, G. R. Ganger, and P. B. Gibbons. Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. CMU PDL Technical Report (CMU-PDL-15-107), 2015.

[27] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting iterative-ness for parallel ML computations. In *SoCC*, 2014.

[28] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed GPUs with a gpu-specialized parameter server. In *EuroSys*, 2016.

[29] 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 *NIPS*, 2012.

[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[31] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and

description. *arXiv preprint arXiv:1411.4389*, 2014.

[32] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, (Jul), 2011.

[33] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[34] K. Fraser and F. Chang. Operating system I/O speculation: How two invocations are faster than one. In *USENIX Annual Technical Conference*, 2003.

[35] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, 2011.

[36] A. Gerbessiotis and L. Valiant. Direct bulk-synchronous parallel algorithms. In *Scandinavian Workshop on Algorithm Theory*, 1992.

[37] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[38] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[39] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Summer USENIX*, 1994.

[40] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences of the United States of America*, 2004.

[41] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[42] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a Stale Synchronous Parallel parameter server. In *NIPS*, 2013.

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

[44] 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 *NSDI*, 2017.

[45] Intel® Threading Building Blocks. `https://www.threadingbuildingblocks.org`.

[46] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[47] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

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

[49] B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-sklearn: automatic hyperparameter

configuration for scikit-learn. In *ICML workshop on AutoML*, 2014.

[50] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[51] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. 2009.

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[53] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[54] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[55] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *NIPS*, 2009.

[56] Q. V. Le, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng. Building high-level features using large scale unsupervised learning. In *ICML*, 2012.

[57] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *USENIX Annual Technical Conference*, 1997.

[58] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[59] 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*, 2014.

[60] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI*, 2010.

[61] D. Maclaurin, D. Duvenaud, and R. P. Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[62] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*. ACM, 2010.

[63] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[64] B. McMahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. In *NIPS*, 2014.

[65] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *HotOS*. Citeseer, 2015.

[66] J. Močkus. On bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference*. Springer, 1975.

[67] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A

timely dataflow system. In *SOSP*, 2013.

[68] Y. Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, 1983.

[69] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 265–272, 2011.

[70] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP*, 1995.

[71] F. Pedregosa. Hyperparameter optimization with approximate gradient. *arXiv preprint arXiv:1602.02355*, 2016.

[72] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[73] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[74] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[75] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, 2013.

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

[77] A. Senior, G. Heigold, K. Yang, et al. An empirical study of learning rates in deep neural networks for speech recognition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6724–6728. IEEE, 2013.

[78] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[79] K. Soomro, A. R. Zamir, and M. Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[80] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *SoCC*. ACM, 2015.

[81] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton. On the importance of initialization and momentum in deep learning. *ICML*, 2013.

[82] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[83] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015.

[84] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings*

of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 847–855. ACM, 2013.

[85] T. TielemanWang and G. Hinton. Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[86] TILEPro processor family: TILEPro64 overview. `http://www.tilera.com/products/processors/TILEPro_Family`, 2013.

[87] A. Tumanov, J. Wise, O. Mutlu, and G. R. Ganger. Asymmetry-aware execution placement on manycore chips. In *Workshop on Systems for Future Multicore Architectures (SFMA)*, 2013.

[88] UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml/datasets/Bag+of+Words`.

[89] M. Vartak, P. Ortiz, K. Siegel, H. Subramanyam, S. Madden, and M. Zaharia. Supporting fast iteration in model building. *NIPS ML Systems Workshop*, 2015.

[90] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

[91] M. Wang, T. Xiao, J. Li, J. Zhang, C. Hong, and Z. Zhang. Minerva: A scalable and highly efficient training platform for deep learning. NIPS 2014 Workshop of Distributed Matrix Computations, 2014.

[92] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, 2015.

[93] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876*, 2015.

[94] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici. Beyond short snippets: Deep networks for video classification. In *CVPR*, 2015.

[95] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *SOSP*, 2013.

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

[97] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216*, 2015.

[98] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. In *SoCC*, 2011.