# Exploiting characteristics of machine learning applications for efficient parameter servers

Henggang Cui
hengganc@ece.cmu.edu

October 4, 2016

## 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. That is, the computation attempts to optimize the model by minimizing an objective function, which generally describes the error. 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 machine learning. In iterative ML, the algorithm 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. Various stopping conditions may be used, such as when the objective function improvement slows sufficiently or just a given amount of time has been spent refining.

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 [1, 25, 21, 7, 3, 6, 5, 4, 34]. 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. The efficiency of the parameter server system is critical to the efficiency of such ML tasks, and a better parameter server design can sometimes improvement the efficiency by an order of magnitude or more.

### 1.1 Thesis statement

In our finished and ongoing work, 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, I will describe three case studies of specializing parameter server designs to exploit different characteristics of ML computations.

- **Exploiting repeated parameter data access pattern (Section 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 implemented and evaluated two methods to collect this repeating access sequence from the

application, as well as five specializations to speed up parameter servers using the collected information. Our experiments show that these optimizations greatly reduce the total run time of our application benchmarks by up to 98%.

- **Exploiting layer-by-layer computation of deep learning (Section 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. It exploits two characteristics of deep learning applications. First, they also have repeated access patterns. Second, the computation is performed layer-by-layer, providing an opportunity for the parameter server to hide communication latency from the application, as well as an opportunity to support neural networks that do not fit in GPU memory, by moving data 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 (13x more training throughput with 16 machines).
- **Exploiting dynamism of hyperparameter choice (ongoing work, Section 5).** Training a machine learning model involves the choice of many training hyperparameters, such as mini-batch size and learning rate, and synchronization policy. The best choice of these hyperparameters is often hard to decide offline and often changes during the training. A bad choice of the hyperparameters can make the model converge very slowly, converge to suboptimal solution, or even not converge. In this ongoing work, we hope to address these problems by adding support for adaptive training hyperparameters tuning to the parameter server. The approach we take is to have the parameter server actively try and compare different training hyperparameters during the training, and focus resources on the best one to further train the model, so that we can train a model to convergence in much less time and with less human effort.

# 2 Background: data-parallel ML and parameter servers

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

## 2.1 Data parallel machine learning and consistency models

Data-parallelism is a common approach for parallelizing machine learning across multiple workers. Each of the workers is assigned a partition of the training data, and they concurrently make adjustments (*updates*) to the shared model parameter data. In order to synchronize their parameter data updates, the workers are often restricted to some *consistency model*.

The three commonly used consistency models are Bulk Synchronous Parallel (BSP), Stale Synchronous Parallel (SSP) [13], and Asynchrony. In all three models, the workers work on a local copy of (possibly stale) model parameter data and cache their parameter updates. To synchronize the workers, each worker's work is divided into *clocks*, which represent a fixed amount of work such as a whole pass over all its training data (an epoch) or a fixed number of training data samples processed (a mini-batch).

In BSP, a barrier is placed after every clock. At each barrier, each worker will propagate its parameter updates to all other workers and refresh its parameter data with the updates of all other workers, so that updates from previous clocks are always visible to all workers. The Asynchrony model is at the other end of the spectrum, with no restrictions on the worker progress, so it only works on problems that tolerate data staleness well and in situations where progress difference among workers do not occur.

The SSP model is a middle ground between BSP andd Asynchrony. It eliminates the barriers of BSP and instead defines an explicit *slack* configuration for coordinating progress among the workers. The slack specifies how many clocks out-of-date a worker's view of the parameter data can be, which implicitly also dictates how far ahead of the slowest worker that any worker is allowed to progress. 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 from clocks $t-s$ to $t-1$.

## 2.2 Parameter server architectures for data-parallel ML

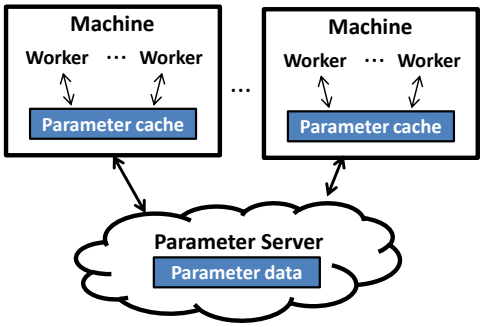



Figure 1: Parallel ML with parameter server.

The *parameter server* (PS) architecture [1, 25, 21, 7, 3, 6, 5, 4, 34] is often used to scale data-parallel ML efficiently. Figure 1 illustrates the basic parameter server architecture. The shared model parameter data (i.e., the model parameters being learned) is kept in the parameter server. An ML application launches multiple workers, and each of them process their assigned training data and use simple key-value interface with `Read-param` and `Update-param` methods to fetch or apply a delta to the parameter data from the PS, leaving the communication and consistency issues to the PS. The parameter data type is often application defined, but most PS implementations require the parameter data to be serializable and be defined with an associative and commutative aggregation function, such as plus or multiply, so that updates from different workers can be applied in any order. In our example machine learning applications (described in Section 2.3), the value type could be an array of floating point values and the aggregation function could be plus.

The prototype systems of our work are implemented on top of LazyTable [4], our parameter server implementation that supports Stale Synchronous Parallel consistency model, which generalizes BSP and Asynchrony. To avoid constant remote communication, LazyTable includes client-side caches that serve most operations locally. While some systems rely entirely on best-effort asynchronous propagation of parameter updates, LazyTable includes an explicit `Clock` method to identify a point (e.g., the end of an iteration or mini-batch) at which a worker's cached updates should be pushed to the shared key-value store and its local cache state should be refreshed.

While Figure1 illustrates the parameter server as separate from the machines executing ML workers, and some systems do work that way, LazyTable shards the server-side parameter server state across the same machines as the ML workers. This approach allows some degree of data locality to be explored, as is shown in our IterStore work described in Section 3.

## 2.3 Example applications

This section describes four real ML applications that are often parallelized using parameter servers. In the following sections, we will use these applications to evaluate our designs.

**Matrix factorization (MF)** 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 matrix $X$ (e.g., a rating matrix where entry $(i, j)$ is user $i$'s rating of movie $j$), matrix factorization factorizes $X$ into factor matrices $L$ and $R$ such that their product approximates $X$ (i.e., $X \approx LR$).

Like many other systems [10, 20], we implement MF using the stochastic gradient descent (SGD) algorithm. Each worker is assigned a subset of the observed entries in $X$; in every iteration, each worker processes every

element of its assigned subset and updates the corresponding row of $L$ and column of $R$ based on the gradient. $L$ and $R$ are stored in the parameter server.

**Latent Dirichlet allocation (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*. 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.

Our LDA solver implements collapsed Gibbs sampling [12]. In every iteration, each worker goes through its assigned documents and makes adjustments to the topic assignment of the documents and the words.

**PageRank (PR)** assigns a weighted score (PageRank) to every vertex in a graph [2]. A vertex's score measures its importance in the graph, with higher scores indicating higher importance. To implement parallel PageRank using a parameter server, we will partition the edges of the input graph evenly among all workers and store the page-ranks in the parameter server. In every iteration, each worker passes 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.

**Deep neural networks (DNN)** is a multi-layer network with interconnected *neurons*, as shown in Figure 2. It is often used for vision and speech tasks, such as image classification [3, 19, 7, 31]. Here we will describe DNN in the context of image classification. For this task, a neural network is trained to classify images (raw pixel maps) into pre-defined labels, using a set of training images with known labels. The first layer of the nodes (input of the network) are the 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 using 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. The connection weights are the model parameters. A common way of learning the values of these model parameters is to train the model using a set of training images, and update model parameters using the SGD algorithm. To parallelize this using a parameter, we will partition the training images into all workers and store the model parameters to be learned in the parameter server.

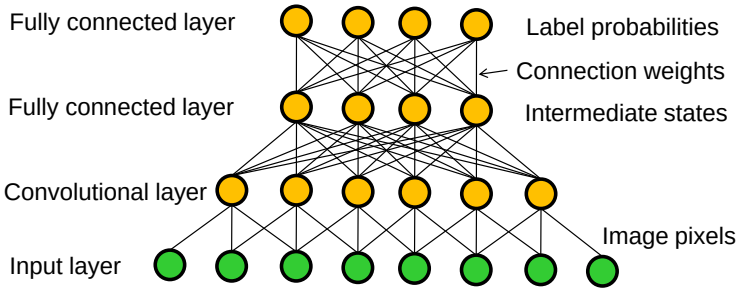



Figure 2: A neural network example for image classification.

# 3 Exploiting iterativeness for efficient data-parallel ML

This sections describes IterStore [5], a parameter server design that exploits the repeating parameter access patterns of iterative applications.

## 3.1 Problem and motivations

In prior state-of-art approaches, 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 distributed by a 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 2.3, 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. 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*.

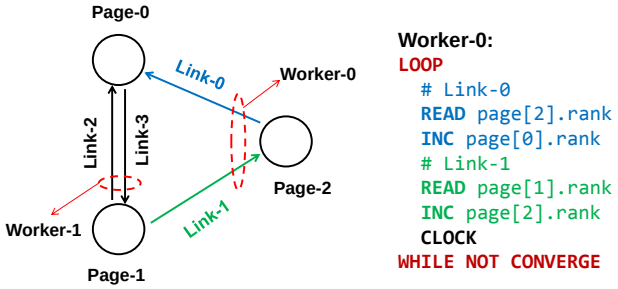



Figure 3: Iterativeness in PageRank. The graph contains three pages and four links. Each of the two workers is assigned with two links. In every iteration, each worker goes through its assigned links. For each link, the worker reads the rank of the source page and updates the rank of the destination 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 System design

### 3.2.1 Obtaining access sequence

There are several ways that a parameter server can obtain the access sequences from the application workers, with different overheads and degrees of help from the writer.

In this work, we explore two options that involve some amount of assistance from the application programmer, illustrated in Figure 4: 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.

**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 thread 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

```
// Original
init_params()
ps.clock()
do {
  do_iteration()
  ps.clock()
} while (not stop)
```

```
// Gather in first iter
init_params()
ps.clock()
do {
  if (first iteration)
    ps.start_gather(real)
  do_iteration()
  if (first iteration)
    ps.finish_gather()
  ps.clock()
} while (not stop)
```

```
// Gather in virtual iter
ps.start_gather(virtual)
do_iteration()
ps.finish_gather()
init_params()
ps.clock()
do {
  do_iteration()
  ps.clock()
} while (not stop)
```

Figure 4: 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.finish_gather) to exploit it. The right-most pseudo-code adds a virtual iteration to do the same, re-using the same do_iteration code as the real processing.

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 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.

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

In this section we detail parameter server specializations afforded by the knowledge of repeating per-iteration access patterns.

**Data placement across machines.** When parameter data is sharded among multiple machines, both communication demands and latency can be reduced if parameters are co-located with 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 [22, 11] 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 memory NUMA zones. That is, a memory access from a 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, 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-worker caches.** Caching usually improves performance. Beyond caching state from remote server shards, per-worker caching can improve performance in two ways: reducing contention between workers in the same process (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.

**Efficient hash maps.** The client library of the parameter server is usually multi-threaded, with enough application workers 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 [14] 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.

**Prefetching.** Under many consistency models (e.g., BSP and SSP), each worker must fresh their cached parameter data from the servers when it gets stale. 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.

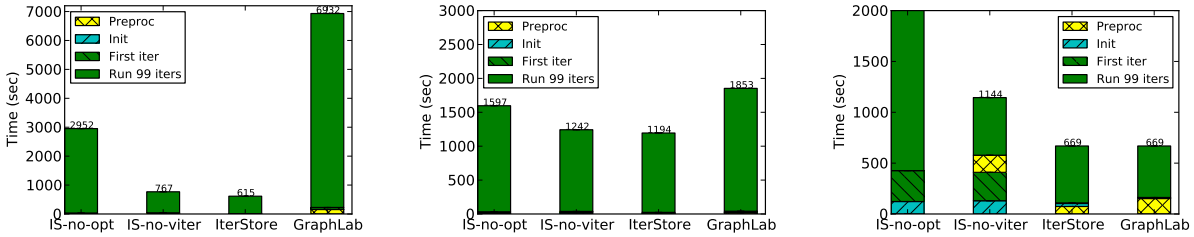
## 3.3 Evaluation

This section presents some key results from [5], with more details available in that paper. We use an 8-node cluster of 64-core machines and evaluate the systems on three ML benchmarks: matrix factorization (MF), latent Dirichlet allocation (LDA), and PageRank. Experimental setup details can be found in our paper [5].

Figure 5 shows performance for each of the three ML benchmarks 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 [22, 11][1] using its synchronous engine.

Each bar shows the time required for the application to initialize its data structures and execute 100 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

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




(a) Collaborative Filtering (b) Topic Modeling (c) PageRank

Figure 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.

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 systems.

The results show that the specializations decrease per-iteration times substantially (by 33–98%) on all three ML benchmarks.

The results 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 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. IterStore's performance advantages have two sources. First, the GraphLab abstraction couples parameter data with computation making it less suitable for MF and LDA. Second, though the GraphLab implementation implicitly uses some of our proposed specializations, it does not use NUMA-aware data placement or contention-aware thread caches.

# 4 Exploiting layered computation for efficient GPU deep learning

This sections describes GeePS [6], a parameter server design that is specialized for GPU-based deep learning applications GeePS addresses the unique challenges of hosting GPU applications by exploiting the characteristics of deep learning applications.

## 4.1 Problem and motivations

GPUs are often used to train deep neural networks, because the primary computational steps match their SIMD-style nature and they provide much more raw computing capability than traditional CPU cores. Most high end GPUs are on self-contained GPU devices that can be inserted into a server machine. 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.

Caffe [15] 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.

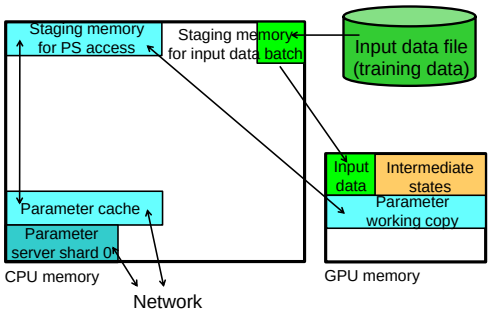



Figure 6: Distributed ML on GPUs using a CPU-based parameter server. Parameter updates must be moved between CPU memory and GPU memory, in both directions, which requires an additional application-level staging area since the CPU-based parameter server is unaware of the separate memories.

Given its proven value in CPU-based distributed ML, it is natural to use the same basic architecture and programming model with distributed ML on GPUs. To explore its effectiveness, we ported the Caffe system to our IterStore system. 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 6 illustrates what sits where in memory, to allow designs described later.

While it was easy to get working, the performance was not acceptable. As noted by Chilimbi et al. [3], 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 [33, 35], 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 rest of this section will describe our designs for overcoming these obstacles.

## 4.2 System 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 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 problems sizes that can be addressed with data-parallel execution on GPUs.

### 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 7. (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. 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.

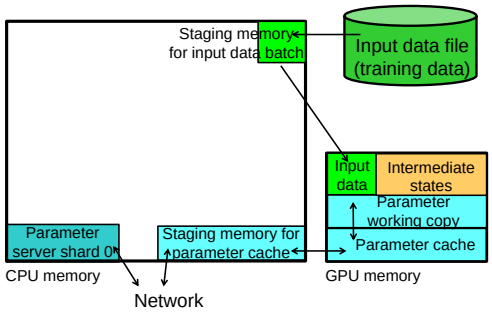



Figure 7: Parameter cache in GPU memory. In addition to the movement of the parameter cache box from CPU memory to GPU memory, this illustration differs from Figure 6 in that the associated staging memory is now inside the parameter server library. It is used for staging updates between the network and the parameter cache, rather than between the parameter cache and the GPU portion of the application.

### 4.2.2 Pre-built indexes and batch operations

Given the SIMD-style parallelism of GPU devices, per-value read and update operations of arbitrary model parameter values can significantly slow execution. In particular, performance problems arise from locking, index lookups, and one-by-one data movement. To realize sufficient performance, our GPU-specialized parameter server supports batch-based interfaces for reads (`READ-BATCH`) and updates (`UPDATE-BATCH`). Moreover, GeePS exploits the repeating nature of iterative model training [5] 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 servers much more efficient for GPU-based training.

### 4.2.3 Managing limited GPU device memory

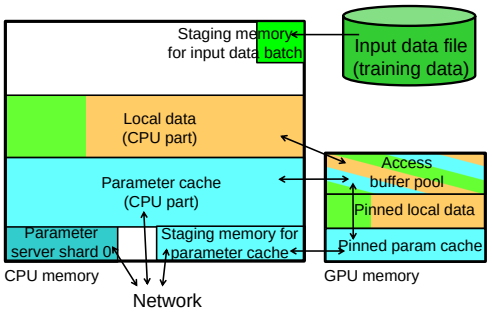



Figure 8: Parameter cache and local data partitioned across CPU and GPU memories. When all parameter and local data (input data and intermediate states) cannot fit within GPU memory, our parameter server can use CPU memory to hold the excess. Whatever amount fits can be pinned in GPU memory, while the remainder is transferred to and from buffers that the application can use, as needed.

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 can move 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* a buffer in GPU memory for it and return the pointer to this buffer to the application, instead of copying the parameter data to a buffer provided by the application. When the application finishes using the parameter data, it returns the buffer to the parameter server. We call those two interfaces `BUFFER-READ-BATCH` and `POST-BUFFER-READ-BATCH`. When the application wants to update parameter data, it will first request a buffer from the parameter server using `PRE-BUFFER-UPDATE-BATCH` and use this buffer to store its updates. The application calls `BUFFER-UPDATE-BATCH` to pass that buffer back, and the parameter server library will apply the updates stored in the buffer and reclaim the buffer memory. To make it concise, in the rest of this paper, we will refer to the batched interfaces using PS-managed buffers as `READ`, `POST-READ`, `PRE-UPDATE`, and `UPDATE`.

The application can also store their local non-parameter data (e.g., intermediate states) in the parameter server using the same interfaces, but with a `LOCAL` flag. 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 `UPDATE` operation.

**Swapping data to CPU memory when it does not fit.** By storing the local data in the parameter server, almost all GPU memory can be managed by the parameter server client library. 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 a CPU core and copy the data from CPU memory to an allocated GPU buffer. Likewise, when the application `READ`s local data that is stored in CPU memory, the parameter server will copy the local data from CPU memory to an allocated GPU buffer. Figure 8 illustrates the resulting data layout in the GPU and CPU memories.

**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 library will use 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 apply the same sequence of parameter server operations every clock [5], so the parameter server can easily predict the `READ` operations and perform them in advance in the background.

## 4.3 Evaluation

This section evaluates GeePS's support for parallel deep learning over distributed GPUs, using image classification as one example.

**Application setup.** We use Caffe [15], the open-source singe-GPU convolutional neural network application discussed earlier.[2]

**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. 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.

**Dataset.** We use the ImageNet22K dataset [8], 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 [3]. We use

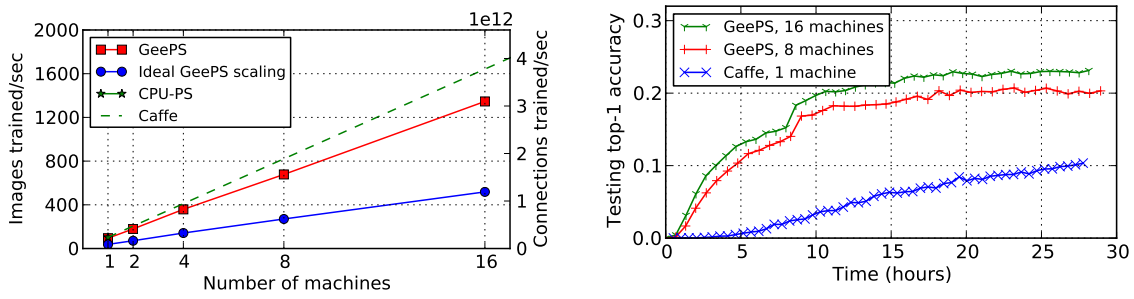
[2]We used the version of Caffe from `https://github.com/BVLC/caffe` as of June 19, 2015.

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. [3].

**Neural network model.** We use a similar model to the one used to evaluate ProjectAdam [3], which we refer to as the *AdamLike* model.[3] The AdamLike model is a variant of AlexNet [19] and 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.

### 4.3.1 Scaling deep learning with GeePS

This section evaluates how well GeePS supports data-parallel scaling of GPU-based deep learning. 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 a state-of-the-art CPU-based parameter server ("CPU-PS"). [4] (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.




(a) Training throughput.

(b) Testing top-1 accuracy.

Figure 9: Scaling image classification.

Figure 9(a) shows model training throughput, in terms of both number of images trained per second and number of neural 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 using 16 machines.

Chilimbi et al. [3] report that ProjectAdam can train 570 billion connections per second on the ImageNet22K dataset when using 108 machines (88 CPU-based worker machines with 20 parameter server machines) [3]. Figure 9 shows the GeePS achieves higher throughput using only 4 GPU machines, because of efficient data-parallel execution on GPUs.

Figure 9(b) shows the top-1 image classification accuracies of our 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 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. Caffe needs 26.9 hours to reach 10% accuracy, while GeePS needs only 4.6 hours using 8 machines (6× speedup) or 3.3 hours using 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 training is needed than with a single worker to make the model converge. But, the speedups are still substantial.

[3]We were not able to obtain the exact model that ProjectAdam uses, 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.

[4]This "CPU-PS" is actually our IterStore system.

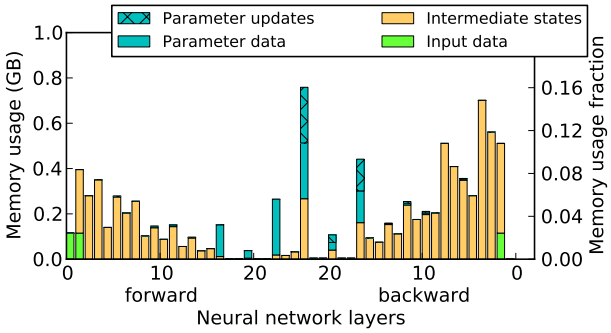



Figure 10: Per-layer memory usage of AdamLike model on ImageNet22K dataset.

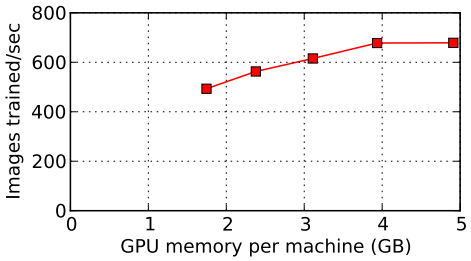



Figure 11: Training throughput with different GPU memory budgets.

Chilimbi et al. [3] report that ProjectAdam needs one day to reach 13.6% accuracy using 58 machines (48 CPU-based worker machines with 10 parameter server machines). GeePS needs only 6 hours to reach the same accuracy using 16 machines (about 4× speedup). To reach 13.6% accuracy, the DistBelief system trained (a different model) using 2,000 machines for a week [7].

#### 4.3.2 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 and by experimenting with a much larger synthetic model.

**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, 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.

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 10 shows the per-layer memory usage for the AdamLike model training, 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 11 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.

**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 per second) of training different sized networks. Figure 12 shows the results. 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.

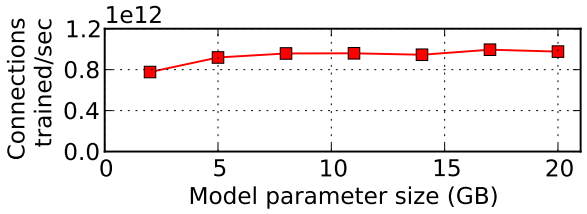



Figure 12: Training throughput on very large models. Note that the number of connections increases linearly with model size, so the per-image training time grows with model size because the per-connection training time stays relatively constant.

# 5 Ongoing work: support for adaptive training

This section describes our ongoing work that tries to speedup the convergence of machine learning tasks by adaptively tuning the training hyperparameters (e.g. learning rate) at runtime.

## 5.1 Problem and motivations

As we have described earlier, the goal of the machine learning tasks on which we focus is to decide the model parameters of a pre-defined model by *training* the model to fit the training data. The training procedure is accomplished with algorithms, such as SGD, which involve many tunable training hyperparameters to the user. For example, to train deep neural networks using SGD, people need to tune hyperparameters like learning rate, momentum magnitude, and mini-batch size. When the model is trained in a distributed setting, there are more training hyperparameters, such as data staleness bound (for SSP). Although the same model is being trained, different choices of the training hyperparameters can affect the convergence speed of the model a lot. A bad choice of the training hyperparameters can make the model converge very slowly, converge to suboptimal solution, or even not converge.

The best training hyperparameters are often problem-specific and are affected by many factors, such as the application, model, algorithm, cluster size, per-node computation power, and network bandwidth. Usually, there are no good ways of deciding the best hyperparameters offline. Moreover, the best hyperparameter might change throughout the training. For example, it often helps to use a larger learning rate and looser consistency bounds at the beginning stage, but as the model approaches convergence, might be more appropriate to use smaller learning rate and BSP. The rest of this subsection discusses two case studies to demonstrate that choosing training hyperparameters is hard, and the next subsection describes the approaches we plan to explore for addressing the challenge.

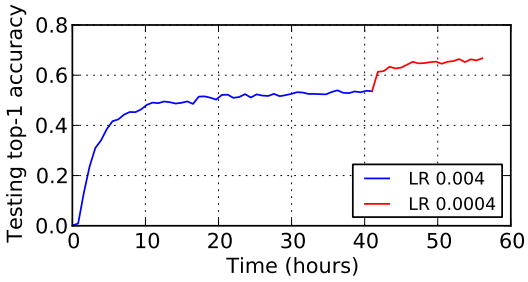



Figure 13: Manual learning rate tuning for image classification with DNN. A learning rate of 0.004 is used at the beginning. When the accuracy stops improving at hour 41, the learning rate is decreased to 0.0004, and the accuracy continues to improve.

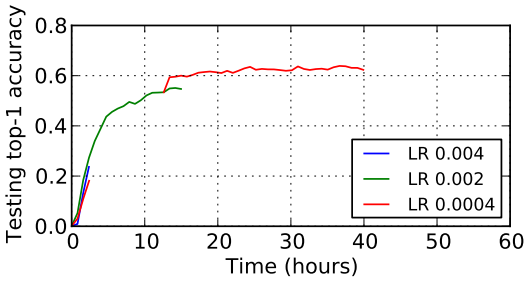



Figure 14: Adaptive branching. Three branches are created at the beginning, `branch-0` with LR 0.004, `branch-1` with LR 0.002, and `branch-2` with LR 0.0004. Because `branch-1` shows the fastest convergence, `branch-0` and `branch-2` are killed at hour 3. At hour 12, `branch-3` is forked from `branch-1` with LR 0.0004 and shows better convergence, so `branch-1` is killed at hour 15.

#### 5.1.1 Case study #1: learning rate for DNN

Here, we will use image classification using deep neural networks (DNN) as an example. DNN models are often trained using the SGD algorithm, and because of the complexity of DNN models, tuning the SGD learning rate for DNNs is notoriously hard [26, 36, 16, 27, 30, 24, 19, 37]. There are three primary ways of tuning the learning rate for DNNs, but none of them completely solves the problem.

**Manual tuning.** One common approach is to pick a learning rate that makes the model accuracy improve and keep training with it. When the model accuracy stops improving, the user then tries changing the learning rate (often decrease by a factor of 10) to make the accuracy keep improving [19, 37]. This approach is illustrated in Figure 13.

There are two problems with this manual tuning approach. First, people need to "babysit" the model training, pausing and restarting the training manually. Second, since the learning rate is only tuned when the model stops improving, the total training time can be significantly longer than optimum.

**Scheduled learning rate decreasing.** In order to avoid the human effort of manually tuning the learning rate, so users use an approach in which the learning rate decreases as a function of iterations completed. Two commonly used learning rate schedules are exponential scheduling and power scheduling. In exponential scheduling, an initial learning rate $lr(0)$ is set, and the learning rate at iteration $t$ has the form of $lr(t) = lr(0) \times 10^{(-t/\gamma)}$. In power scheduling, the learning rate decreases as $lr(t) = lr(0) \times (t/\gamma)^{-c}$. These scheduled learning rate hyperparameter methods reduce the human effort, but only makes it harder to achieve optimum convergence time.

**Adaptive learning rate tuning algorithms.** There are also some adaptive gradient-based optimization algorithms that makes the initial learning rate choice less of an issue, such as AdaGrad [9], AdaDelta [36], Adam [16], and NAG [23, 30]. Even though these algorithms are less sensitive to the choice of hyperparameters, people find that, with proper learning rate tuning, the standard momentum-enabled SGD algorithm actually outperforms these adaptive algorithms [36, 27]. These results suggest that better hyperparameter tuning methods should be designed.

#### 5.1.2 Case study #2: data staleness bound

Allowing more data staleness, such as by using SSP, sometimes improves performance. For example, previous work [13, 4] has shown that for problems such as matrix factorization and LDA, SSP gives us faster model converge than BSP (Figure 15). However, when training DNNs on GPUs, Cui et al. [6] find that allowing

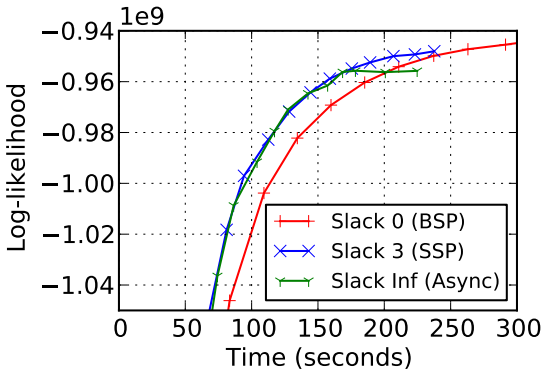



Figure 15: LDA on NYTimes dataset with 8 CPU machines. SSP (with a slack of 3 clocks) has the fastest convergence.

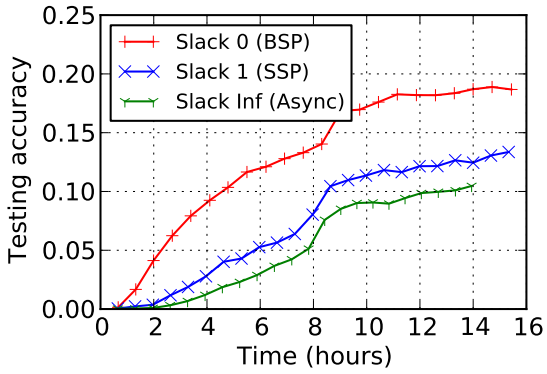



Figure 16: DNN on ImageNet dataset using 8 GPU machines. BSP has the fastest convergence.

more data staleness hurts more than it helps (Figure 16). The model converges faster using BSP.

The best consistency model and data staleness bound are problem-specific and none of the previous work gives clues about how to decide the best setting.

## 5.2 Proposed approach: adaptive online tuning

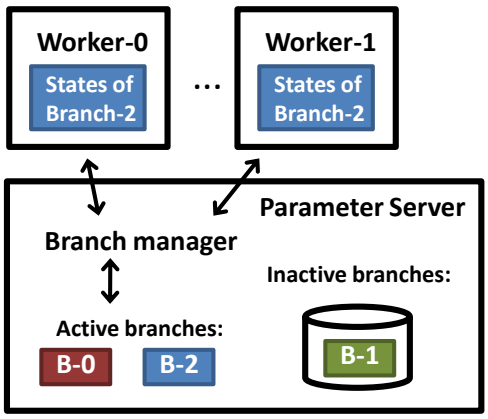



Figure 17: System architecture. The parameter server keeps multiple *branches* of training state (parameter data, local data, and training hyperparameters). The branches are managed by a *branch scheduler* module. In this example, `branch-2` is active and currently being scheduled; `branch-0` is active and will be scheduled the next iteration; `branch-1` is inactive and the states are kept in disk.

We hope to address the problems described above by adding support for adaptive training hyperparameter tuning to the parameter server. In particular, we will design a parameter server system that is able to adaptively tune the training hyperparameters *online* during the training, so that we can train a model to convergence in much less time and with less human effort.

The approach we take is to have the parameter server actively try and compare different training hyperparameters during the training, and focus resources on the best one to further train the model. One way of doing that is to adaptively *fork* the model training procedure into multiple *branches*, each with a

different set of training hyperparameters.

As is shown in Figure 17, the parameter server maintains multiple branches, each consisting of one set of model parameter data, worker-local intermediate data, and training hyperparameters. The parameter server will schedule the workers to work on a specific branch by servicing their data requests with the states (parameter data, local datam and training hyperparameters) of this branch. The *branch manager* module inside the parameter server will also make decisions to fork new branches and kill branches that do not show evidence of outperforming other branches. The killed branches become *inactive* and their states are stored in the disk, so that they can be activated again.

This design allows the parameter server to adaptively tune the training procedure online. Figure 14 shows an example of adaptively tuning the learning rate for DNNs. The parameter server forks multiple branches during the training to try different learning rates, and focuses resources on the best one to further train the model. Compared to manually tuning the learning rate, this adaptive training design saves human effort and makes the model converge to the same accuracy in much less time.

## 5.3 Research questions to be explored

In this proposed future work, there are many research questions to be explored, and we categorize them into two classes: adaptive training techniques and system designs for efficiently supporting them.

### 5.3.1 Adaptive training design

Our proposed work will adaptively tune the training hyperparameters by forking multiple model branches, and there are many research questions to be answered about this.

- **Branching decisions.** When shall we fork a new branch, and when shall we kill a branch?
- **Hyperparameter choice for new branches.** When we fork a new branch, how do we choose the training hyperparameters for that branch?
- **Branch scheduling.** When we have multiple branches, do we schedule them in a round-robin manner or some other mechanisms?
- **Comparing branches.** How do we compare the progress of the branches? When we try to kill a branch, do we always kill the branch with worst accuracy?

### 5.3.2 System challenges and optimization opportunities

When the parameter server keeps multiple model branches and context switch among them, there are also challenges (optimization opportunities) associated with the parameter server design.

- **Efficient context switching.** The parameter server needs to efficiently switch between model branches.
- **Consistent context switching decisions.** The parameter data is sharded across multiple server, so these server shards need to make consistent context switching decisions and always serve the parameter data of the same branch.
- **Switching between consistency models.** We consider the data staleness bound as one of the training hyperparameters, and different consistency models can be used at different iterations. The parameter server needs to be able to efficiently switch between consistency models.
- **Managing the local data associated with each branch.** Each model branch consists of worker-local data as well as shared parameter data. When we switch to a new branch, the local data should also be changed accordingly.

- **"Free slack" from multi-branching.** When we have multiple branches, the workers will be able to work on one branch for a clock and then switch to another branch. This means, even using BSP, the workers don't need to wait for each other to finish the previous clock before starting their next clock, giving us extra slack for free.

### 5.4 Additional related work

Recently, people have started to explore techniques and systems for automatic model selection for machine learning tasks [29, 32, 17]. In particular, TUPAQ [29] is a component in the MLbase system [18] that adaptively finds and trains the best model for a given ML task. TUPAQ focuses on prediction tasks with support vector machines (SVM) and logistic regression models, and it tries to search for the best hyperparameters for these models that yield the best prediction accuracy for a given task. Its model search techniques are based on an assumption that one can easily train a model to convergence, because of its use of relatively simple models. Our proposed future work, however, focus on speeding up the training of a single selected model, so they are orthogonal problems and can be complementary. Moreover, TUPAQ uses a constant hyperparameter choice for the training of each model, while we believe the hyperparameter should be tuned during the training. Some of the techniques that TUPAQ uses might apply in our system, such as batched training of multiple models, preemptive pruning of bad models, and sophisticated hyperparameter searching algorithms.

There are also some work in the machine learning community on hyperparameter optimization. For example, Snoek et al. [28] provides a practical solution for tuning the hyperparameters with Bayesian optimization. They evaluate the choice of each set of hyperparameter values by training the model to convergence and checking the validation error, and use Bayesian optimization to decide the next hyperparameters to try. Similar to the model selection work, their hyperparameter optimization techniques also focuses on finding the best model among all possible models, instead of training a single model efficiently. But we can borrow their idea of using Bayesian optimization to decide the next set of hyperparameter values to use for newly forked training branches.

## 6 Thesis timeline

- **March-April 2016:**

  More related work study and motivating experiments on the adaptive training work.

  Preparing the conference presentation of the GeePS work.

- **May-July 2016:**

  Prototype implementation of the adaptive training system.

- **August-October 2016:**

  More experiments on adaptive training.

  Paper submission of the adaptive training work.

- **November-December 2016:**

  Continued improvement of the adaptive training system.

  Continued exploration on adaptive training, looking for more use cases with more extensive experiments.

- **January-May 2017:**

  Dissertation writing and defense.

  Job search.

# References


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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12, 2011.

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

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

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

[13] 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.

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

[15] 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.

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

[17] B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, 2014.

[18] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

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

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

[21] 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.

[22] 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.

[23] Y. Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[24] 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.

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

[26] T. Schaul, S. Zhang, and Y. LeCun. No more pesky learning rates. *arXiv preprint arXiv:1206.1106*, 2012.

[27] A. Senior, G. Heigold, M. Ranzato, and K. Yang. An empirical study of learning rates in deep neural networks for speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[28] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[29] 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.

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

[31] 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.

[32] M. Vartak, P. Ortiz, K. Siegel, H. Subramanyam, S. Madden, and M. Zaharia. Supporting fast iteration in model building. In *Workshop on Machine Learning Systems at Neural Information Processing Systems*, 2015.

[33] 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.

[34] 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.

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

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

[37] 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.