MPCA SGD - A method for distributed training of deep learning models on Spark

Matthias Langer, Ashley Hall, Zhen He and Wenny Rahayu
La Trobe University, VIC 3086, Australia
Email: {m.langer,awhall,z.he,w.rahayu}@ltu.edu.au

Abstract—Many distributed deep learning systems have been published over the past few years, often accompanied by impressive performance claims. In practice these figures are often achieved in high performance computing (HPC) environments with fast InfiniBand network connections. For average deep learning practitioners this is usually an unrealistic scenario, since they cannot afford access to these facilities. Simple re-implementations of algorithms such as EASGD [1] for standard Ethernet environments often fail to replicate the scalability and performance of the original works [2]. In this paper, we explore this particular problem domain and present MPCA SGD, a method for distributed training of deep neural networks that is specifically designed to run in low-budget environments. MPCA SGD tries to make the best possible use of available resources, and can operate well if network bandwidth is constrained. Furthermore, MPCA SGD runs on top of the popular Apache Spark [3] framework. Thus, it can easily be deployed in existing data centers and office environments where Spark is already used. When training large deep learning models in a gigabit Ethernet cluster, MPCA SGD achieves significantly faster convergence rates than many popular alternatives. For example, MPCA SGD can train ResNet-152 [4] up to 2.8x faster than state-of-the-art systems like MXNet [5], up to 1.7x faster than bulk-synchronous systems like SparkNet [6] and up to 1.5x faster than decentral asynchronous systems like EASGD [1].

Index Terms—deep learning, distributed computing, machine learning, neural networks, spark, stochastic gradient descent

1 Introduction

Many remarkable results in areas such as computer vision, speech recognition or natural language processing were achieved by training deep neural networks using GPUs. However, training complex models on large datasets using a single GPU can take days or even weeks.

Distributed deep learning systems can speed up training considerably. Most existing state-of-the-art distributed deep learning systems rely on a *centralized* approach, where the worker nodes compute gradients using local replicas of the model and submit them to an optimizer that runs in the parameter server [7], [8]. This approach requires very fast networking hardware, since the updated model is re-downloaded by the workers after every optimization step.

Decentralized methods, such as Elastic Averaging SGD (EASGD [1]) take the opposite approach, where the same model is improved in isolation using local optimizers in each worker. The resulting models are asynchronously merged. This significantly reduces the communication demand per worker. However, the parameter server has to cater to each worker independently. Thus, the communication demand on the parameter server still increases linearly with the cluster size, which can limit scalability if network bandwidth is constrained.

High-Bandwidth, low-latency networking hardware is able to overcome most network throughput related limitations. In fact, InfiniBand and high-bandwidth Ethernet are frequently used to demonstrate the performance of distributed deep learning systems [1], [5], [9]. However, the costs for the required infrastructure is still comparatively high. In bandwidth constrained environments, most of these systems perform significantly worse (Section 7). In contrast, our work specifically targets commodity clusters with low single digit gigabit Ethernet bandwidths. Note that the capabilities of affordable GPUs for deep learning increase at a significantly faster pace than that of networking hardware. Thus, developing solutions that target scenarios, where the network bandwidth is limited, is becoming increasingly important.

Simultaneously, Apache Spark [3] has recently emerged as one of the major standardized platforms for Big Data analytics, because its embedded MapReduce-inspired processing with Resilient Distributed Datasets (RDDs) excels in cluster environments composed of low-budget commodity hardware. However, Spark's most powerful communication primitives *broadcast* [10] and *treeReduce* [9], confine purely Spark-based deep learning implementations to operate synchronously [6], [11]. Thus, most Spark-based deep learning systems achieve asynchronous execution by bypassing Spark's highly optimized networking code [12] or bypass its execution engine entirely [2], [5], [7].

The results we present in Section 7 show that decentralized training methods outperform centralized systems in low-bandwidth high-latency environments like gigabit Ethernet. Thus, we will concentrate on these methods. We will refer to the typical way decentralized optimization of machine learning models is implemented in Spark as the *synchronized approach*. As visualized in Fig. 1, the synchronized approach sequentially alternates between training (map) and synchronizing parameters (reduce/broadcast) in order to realize collaborative model training. The synchronized approach, such as that taken by SparkNet [6], is often considered inferior to asynchronous systems [1], [2], [13], due to two main drawbacks. First, the synchronized

approach cannot overlap computation and communication and, second, faster nodes need to wait for stragglers when merging parameters.

In this paper we propose *Multi-Phase Coordinated Asynchronous SGD (MPCA SGD)*. MPCA SGD adopts features from asynchronous EASGD [1], but still uses the highly optimized communication primitives of Spark to maximize the effective use of the available network bandwidth. This contrasts with most existing deep learning implementations in Spark, which effectively bypass Spark's execution model [2], [11], [12]. MPCA SGD allows communication and computation to overlap in order to make the best possible use of the available GPUs and takes various measures to mitigate network delays and minimize wait times to improve the collaboration between the workers. Hence, it inherits most advantages of the synchronous approach and EASGD while avoiding their drawbacks.

In MPCA SGD, the driver-program continuously instigates the broadcasting and reduction of model parameters. Model optimization is conducted by the workers at the maximum pace of the local hardware while parameter updates are being exchanged. Thus, communication is coordinated while computation is conducted asynchronously. Fig. 2 illustrates the relationship between the driver and worker models in this scenario. Note how the worker models continue to evolve during parameter exchange. Thus, the shared joint model state has to be fit-back into the worker models instead of simply replacing them like in the synchronous approach (Fig. 1). To minimize the staleness of updates and improve concurrency, we split the model parameters into multiple shards and then broadcast one shard while reducing another. Updates for model shards are pro-actively distributed, such that they are available at each cluster node when the local model should be updated. As shown in Fig. 3, all machines are permanently sending and receiving at the same time. Because each worker gets dynamically assigned certain roles during broadcast and reduction by Spark, the available network resources are utilized at maximum efficiency. To minimize delays, individual workers can inject updates into the reduction process irrespective of their optimization progress. However, MPCA SGD will not be slowed down by stragglers since it does not wait for them, but instead compensates for their slow progress.

There are three main challenges in applying MPCA SGD to train deep learning models on Spark. First, how to select and merge individual shards of the joint model (aggregated model parameters from previous reductions) into the local state of each worker? Second, since the model is split into shards that are exchanged separately, while all workers continue optimizing the entire model asynchronously, each worker may progress at a different pace for different portions of the model. In addition, the driver's state is always stale with respect to the workers. We have to find a balance between allowing the workers to *exploit* each other's contributions and giving them the ability to *explore* the parameter space [1]. Furthermore, we have to limit the influence of stragglers on the joint model state. Third, how to implement coordinated asynchronous execution within the Apache Spark framework, while preserving Spark's highly optimized networking and scheduling code?

We address the first challenge by dividing our overall communication process into distinct stages and enforce a

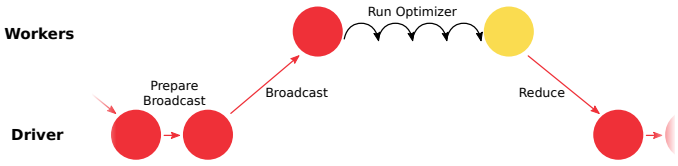


Fig. 1. Classic synchronous approach for decentrally training deep learning models (●) using MapReduce. After broadcasting each worker starts optimizing its local model replica and independently computes updated model parameters (●) that are then aggregated during reduction to form the starting state for the next optimization cycle. Note how communication and computation phases are interleaved.

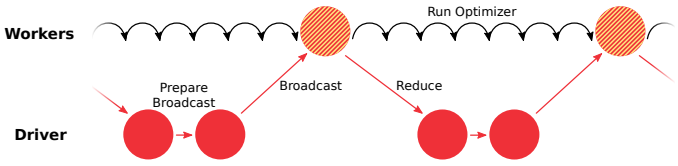


Fig. 2. Coordinated asynchronous model training. Parameter exchange and model optimization occur simultaneously. Workers continue to modify their local model replicas while the driver aggregates them. Using this aggregate, the driver determines parameter updates that generalize well across workers and shares them through broadcasting. Simply replacing the local model would erase any optimization progress made in the meantime. Thus, the broadcasted joint model state has to be integrated by combining both models (Section 4.4).

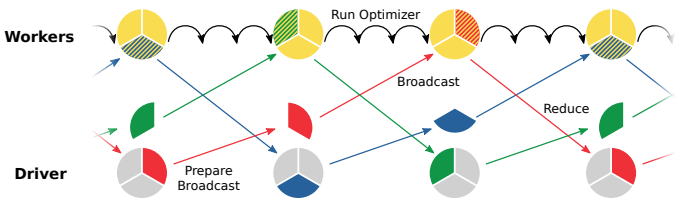


Fig. 3. Multi-Phase coordinated asynchronous model training. The model is divided into multiple shards. The workers continuously improve all shards of their local model replica. To maximize the utilization of the available network I/O bandwidth and minimize the staleness of parameter updates, the driver cyclically triggers the aggregation of individual model shards, such that at any point in time a portion of the model parameter updates is aggregated, prepared for sharing and shared with the workers.

rigid scheduling policy that allocates each model shard to a different stage. Furthermore, we ensure that each stage has finished execution before advancing to the next stage (Section 4.2). For the second challenge, we enforce limits on the asynchronous operation to protect the local models in case of unintended delays and constrain the local optimizers in the workers using a penalization term. Guiding co-adapting optimizers through euclidean distance based penalization has already been extensively researched and works well in practice [14], [15], [16]. We extend these ideas by varying key parameters throughout training to encourage the workers to explore the parameter space, but also control the degree of divergence from the overall optimization trajectory and accommodate slow workers (Sections 4.4 and 6.3). Unlike Zhang et. al. [1], we cannot afford to hold back further model improvement to minimize the staleness in updates due to our expected long communication delays. Instead, we extrapolate the overall optimization trajectory to approximate potential future model states (Section 4.3). To mitigate the influence of stragglers on the joint model we

use a simple, yet efficient weighting mechanism during the reduction phase (Section 4.6). To address the third challenge, we implement a message based scheduling mechanism on top of Spark. This scheduler allows RDDs to execute code in a Spark compliant way after their normal execution scope ends (Section 5).

Our key contributions include:

1) Contradictory to popular belief, we show that distributed training of deep learning models can be accomplished efficiently in low-bandwidth high-latency environments using only the MapReduce styled execution model of Spark with our MPCA SGD approach.
2) To make MPCA SGD work efficiently, we introduce novel techniques for updating parameters asynchronously on Spark, while simultaneously exchanging parameters in both directions.
3) MPCA SGD significantly outperforms MXNet [5] (state-of-the-art centralized asynchronous system) and EASGD [1] (state-of-the-art decentralized asynchronous system) in a standard gigabit Ethernet environment.

For popular image classification problems, MPCA SGD can produce high quality models in less time. Using a cluster of 8 machines connected only through gigabit Ethernet, we can reach an average prediction error of less than 10% on the CIFAR-10 dataset for a ResNet-110 [4] model in under 30 minutes, which is 6.9x faster than a single GPU, 4.8x faster than the state-of-the-art deep learning system MXNet, 30% faster than the synchronous approach and 18% faster than EASGD (Section 7.1). On large scale problems such as ImageNet-1k, MPCA SGD outperforms the synchronous approach by up to 2.1x and EASGD by up to 1.4x for a VGG-A [17] model and MXNet by up to 2.8x for a ResNet-152 [4] model (Section 7.2).

This document is organized as follows. In Section 2, we take a brief look at existing distributed deep learning systems and establish a rough categorization. In Section 3, we analyze the decentral synchronous and asynchronous training methods. Based on this foundation, we then introduce the idea of multi-phase coordinated asynchronous (MPCA) execution in Section 4.2 and refine it throughout subsequent sections to address the first two of the aforementioned challenges. Then we describe how MPCA can be implemented in Spark to address the third challenge (Section 5). In Section 6, we discuss our experimental setup and follow up by presenting benchmark results in Section 7. Finally, we conclude and propose directions for further research in Section 8.

2 Related Works

There has been a lot of existing work on distributed deep learning. In this section, we will describe the key existing distributed deep learning systems. We separate our discussion between non-Spark-based and Spark-based systems.

2.1 Non-Spark systems

Most of the recent works in distributed deep learning originate from the impressive results that were achieved by Le et. al. [18] in large scale image classification. Their system *DistBelief* [8] divides the cluster into parameter servers that maintain shards of the joint model and workers that compute and submit gradients to them. Using these gradients, the parameter servers then apply momentum SGD to update their shard of the joint model. Before the workers can continue, they need to download the updated copy of the joint model. The core ideas of DistBelief have been re-implemented and extended many times.

Parameter Server [19], [20] added fault tolerance through redundancy and methods to identify and limit the influence of slow workers.

Project Adam [21] generalized this idea by organizing the parameter servers in a Paxos-cluster [22] and minimizes network transmission costs by moving parts of the gradient computation into the parameter server for certain layers.

Petuum [23] introduced the idea of imposing staleness constraints to limit asynchrony between workers in order to improve the overall convergence speed.

TensorFlow [7] can be considered as the successor to DistBelief. Among other things, it adds automatic computation graph optimization, which makes distributed model parallelism much more practical. Abadi et. al. [7] also developed a cost model that covers most transactions and constraints of the cluster nodes. Thus, allowing them to determine individual task placements by solving an optimization problem.

MXNet [5] also adds automatic computation graph optimization with special emphasis on improved efficiency through memory reuse. Furthermore, MXNet features a hierarchical parameter server architecture, where intermediate nodes can act as proxy servers to others.

FireCaffe [9] runs instances of the Caffe [24] deep learning framework in parallel on multiple machines connected through a HPC fabric. To minimize communication delays, FireCaffe implements a custom MapReduce-inspired communication protocol.

EASGD [1] decentralizes training by running separate optimizers directly in each worker. The workers independently exchange updates with a parameter server every τ computation cycles. Each worker continues computing while others exchange parameter updates with the parameter server. To limit divergence, workers and the parameter server are penalized based on their euclidean distance.

GoSGD [13] implements the EASGD parameter exchange algorithm, but abandons the concept of having an explicit parameter server. Instead, the workers are organized in a peer-to-peer mesh. Communication partners for parameter exchange are selected individually every τ cycles through a randomized gossip algorithm [25].

2.2 Spark-based systems

SparkNet [6] was inspired by FireCaffe [9], but is targeted at low-bandwidth network environments. It implements synchronous decentralized training. Thus, each worker runs a separate optimizer in isolation for τ steps. The resulting models are then reduced through averaging. Before the next computation cycle begins, this average model is broadcasted to all workers and replaces their local models.

DeepSpark [2] is an attempt to implement EASGD in a commodity hardware environment on top of Spark. Due to the vastly differing execution models, it uses a custom communication protocol that effectively bypasses Spark

during training. Spark is only used to distribute and start the actual EASGD program. However, even with tweaks such as automatically adapting penalty coefficients, their approach suffers significantly from poor networking bandwidth.

CaffeOnSpark [12] implements a purely data-parallel optimizer that works similar to DistBelief on top of Spark. Instead, of having a dedicated parameter server role, each worker also operates as a parameter server for a portion of the model. Since this requires direct communication between workers, they bypass the Spark execution model using the MPI-allReduce communication primitive in conjunction with RDMA (Remote Direct Memory Access).

BigDL [11] implements an optimizer similar to CaffeOnSpark [12]. However, instead of bypassing Spark using MPI, they exchange parameters between workers via the Spark block manager. BigDL uses a very efficient and largely Spark-compliant way to exchange parameters. However, it does so by strictly separating the gradient computation and parameter exchange phases, which limits this approach to a synchronous mode of operation.

To better understand these works, we discriminate between centralized and decentralized systems, which refers to whether model training is performed on a central node or decentrally in the workers. Except SparkNet [6], EASGD [1] and derivative works [2], [13], all aforementioned systems use a centralized approach, where the optimizer is run in the parameter server. Thus, network communication is required at each improvement step to make workers aware of changes. In contrast, decentralized systems allow workers to optimize their local model representation directly without communicating with other nodes. However, they may be forced to conduct long isolated learning phases that increase with the cluster size, which may result in sub-optimal convergence behavior [6]. We can also distinguish whether gradient computations occur synchronous or asynchronous across workers (i.e. whether a worker has to wait for all other workers after it has processed a certain amount of batches). Furthermore, different phases of training (parameter exchange, model-updating, etc.) can either be scheduled simultaneously for all worker nodes or not. In coordinated systems, a driver-node instigates all workers to start training phases at the same time, while workers in uncoordinated systems may follow separate independent schedules. All the systems we discussed operate either in a coordinated synchronous [6], [9], [11], [12], [19], or uncoordinated asynchronous [1], [2], [5], [7], [8], [13], [21], [23] manner. Therefore, they either have to trade off between computation and communication or are potentially susceptible to network bandwidth limitations because the workers exchange parameter updates independently with the parameter server, thus precluding the use of efficient group communication methods [9], [10]. In contrast, our approach uses coordinated asynchronous execution, meaning that it suffers from none of these drawbacks.

3 Analysis of Existing Approaches for Decentral Distributed Training

In this section, we briefly analyze existing approaches for distributed training of deep learning models in detail. In particular, we will analyze the two major existing works that take measures to decouple computation and communication by allowing the workers to independently optimize their local models (decentral training). This analysis motivates the introduction of our proposed MPCA SGD method.

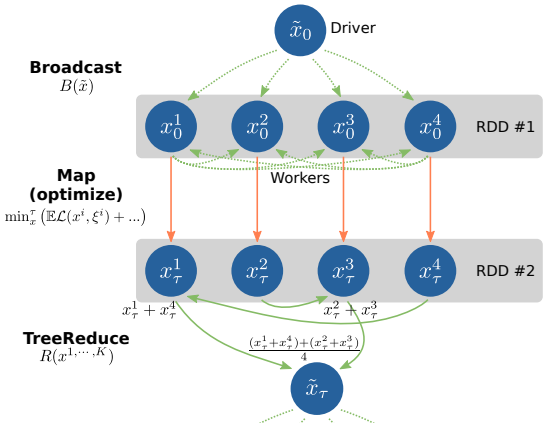


Fig. 4. Synchronous data-parallel training in SparkNet.

3.1 Coordinated synchronous approach (SparkNet)

3.1.1 Working principle

The idea to utilize the synchronous MapReduce communication pattern for data-parallel distributed deep learning was published first by Iandola et. al. [9]. Their system FireCaffe takes a centralized approach where the optimizer is executed in the driver node. Like in a single GPU implementation, this optimizer aims to minimize the expectation for a stochastic loss-function and a regularizer, that both depend on the model parameters (x), through iteratively executing stochastic gradient descent (SGD) using random batches (ξ), drawn from a training distribution (Equation 1).

$$\min_x \left(\mathbb{E}\left[\mathcal{L}(x, \xi)\right] + \lambda \|x\|^2\right) \tag{1}$$

However, the computation of the gradients for each batch is distributed among the workers. During each optimization cycle, the driver sends the current model to the workers (broadcast). Then, each worker computes gradients for a separate portion of the current batch (map). The resulting gradients are aggregated in the driver (reduce), which applies the optimization step.

FireCaffe quickly bottlenecks on the communication costs if bandwidth is constrained [6]. In contrast, SparkNet [6] tries to avoid this problem by taking a decentralized approach. We illustrate this method in Fig. 4. Each worker (i) improves its local copy of the model (x^i) using a separate optimizer for a specified number of iterations (τ) in isolation (map). Once τ iterations have elapsed, the new model parameters are captured in a RDD and averaged ($R(x^{1,\cdots,K})$) in the driver using a binary reduction tree (→). This average model ($\tilde{x}$) is then transmitted back to all workers through broadcasting ($B(\tilde{x})$) and becomes the starting point of the next optimization cycle. Spark broadcasts [10] are conducted by forming a BitTorrent swarm [26] using all nodes where the driver acts as the initial seeder (→).

3.1.2 Influence of τ

The parameter τ denotes the number of optimization iterations between merging model parameters. Moritz et. al. [6]

TABLE 1
Performance figures for training various popular deep neural network architectures for image classification in our test setup (Section 6.1).

	Model	Size *(MiB)*	Communication *(s, GbE, 8 executers⋆)*	Batch Size	Computation *(s/batch, 1 GPU, Adam [27])*	$\frac{\text{Communication}}{\text{Computation}}$	Suggested $\tau^{\star\star}$
CIFAR-10	ResNet-110	7	1.1	64	0.095	11.6	58
ImageNet	VGG-A	491	51.1	50	0.474	107.8	539
	ResNet-34	83	9.2	128	0.582	15.7	79
	ResNet-152	223	23.6	32	0.809	29.2	146

⋆ Measured using $\tau = 0$. Note that broadcast and treeReduce overlap slightly, since workers that complete broadcasting earlier may also enter the reduction phase sooner. Includes times required for data serialization, GPU to CPU and CPU to GPU memory transfers.
⋆⋆ Assuming a target communication to computation ratio of 1:5 [6].

showed that the best convergence rate (assuming no communication costs) can be achieved if τ is small, such that the workers spend only a few iterations *exploring* the parameter space in isolation (map) before sharing their results, which allows them to *exploit* each other's findings (reduce & broadcast). However, in real-world scenarios communication does not occur instantaneously. Thus, τ is actually a measure for controlling how much time should be spent on improving the local models (computation) versus synchronizing across machines (communication). In order to make good use of the GPUs τ has to be increased. Moritz et. al. [6] suggest choosing τ such that the communication to computation ratio is 1:5 (i.e. the average GPU utilization is 83.3%).

To understand what values of τ this translates to in our gigabit Ethernet test-setup (Section 6.1), we implemented various popular deep learning models and present the respective communication and computation times in Table 1. As can be seen, the period of isolated learning has to be hundreds of iterations in some cases if we follow the 1:5 rule. Empirical measurements by Moritz et. al. [6] show that, irrespective of the cluster size, such long phases of isolated learning yield only slow convergence, due to the workers drifting towards different local minima. This leads to contradicting parameter adjustments that erase each other during reduction and setback the optimization progress. Thus, in the synchronous approach, any choice of τ represents a dilemma of finding a balance between two conflicting goals.

3.2 Uncoordinated asynchronous approach (EASGD)

3.2.1 Working principle

Like SparkNet, Elastic Averaging SGD (EASGD) [1] also takes the decentralized approach, where each worker improves its local model in isolation for τ iterations. However, unlike in SparkNet the workers operate asynchronously at their own pace. Once τ iterations have elapsed on a worker, it independently contacts the parameter server to exchange updates. While one worker exchanges parameters, others update their local models. This removes the need for global synchronization, but also prevents the usage of efficient synchronous collaborative communication primitives, such as broadcast and reduce.

Note that the model parameters cannot simply be replaced in EASGD, since workers contact the parameter server without coordination. Instead, EASGD merges models in a way that allows the workers to explore the parameter space, but also prevents them from deviating too far from the joint model state maintained by the parameter server. This is achieved by penalizing the local models and the parameter server model based on $\frac{\alpha}{2}$ times their squared euclidean distance or L^2-norm ($\frac{\alpha}{2}\|x^i - \tilde{x}\|^2$). Every τ iterations, each worker downloads the current joint model state ($\tilde{x}_t$) from the parameter server and adjusts its current local model (x_t^i) by linearly interpolating between them at a rate of α (Equation 2). Small α give the workers more freedom to *explore* the parameter space, while large α force them to stay close to each other and thus, make them *exploit* each other's findings. To establish elastic symmetry, which Zhang et. al. [1] consider to be crucial for the stability of the algorithm, the inverse update has to be applied to the joint model in the parameter server (Equation 3).

$$x_{t+1}^i = x_t^i - \alpha(x_t^i - \tilde{x}_t) \tag{2}$$

$$\tilde{x}_{t+1} = \tilde{x}_t + \alpha(x_t^i - \tilde{x}_t) \tag{3}$$

3.2.2 Influence of τ

Like in the synchronous approach, τ directly controls the amount of communication between each worker and the parameter server by extending the isolated learning period. Furthermore, τ and α constrain how far individual optimizers can explore the parameter space. To realize the parameter exchange between a worker and the parameter server required by Equations 2 and 3, workers in EASGD stop the optimization every τ iterations and download $\tilde{x}_t$. Then, they compute the update $\alpha(x_t^i - \tilde{x}_t)$, apply it to their local model and send it back to the driver. The latter can be done asynchronously while the local optimizer runs, since there are no side-effects on the local model. Note that since the initiative lies with the workers, all communication is peer-to-peer. Hence, assuming all workers are equipped with the same hardware, the optimal minimum bound for τ is equivalent to the number of optimizer iterations it takes to download $\tilde{x}_t$ and update the local model, times the number of workers (K). Thus, τ has to be scaled linearly with the number of workers. However, due to minor timing differences, some workers will eventually attempt to contact the parameter server simultaneously, which may cause transmission delays. In practice, τ must therefore be increased to add idle time between transmissions. This allows the parameter server to recover from minor disturbances, thus ensuring that subsequent parameter exchange requests from other workers can be processed promptly. However, long phases of isolated learning (large τ) detrimentally affect model convergence [28] similar to SparkNet (Section 3.1.2).

4 Multi-Phase Coordinated Asynchronous SGD (MPCA SGD)

The coordinated synchronous approach (Section 3.1) is compatible with Spark, but hinges heavily on choosing the right

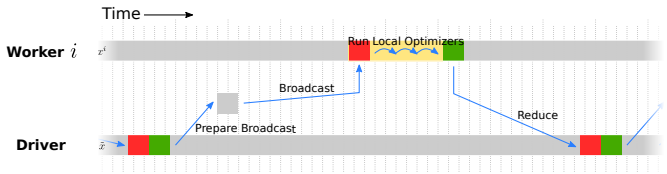


Fig. 5. Synchronous communication pattern (SparkNet). The horizontal bars represent model shards that are either stored in the driver or the workers. Colors indicate actions as follows: ■ means idle (no update), ■ indicates that parameters are transferred from the GPU to the JVM (e.g. if parameters should be reduced by Spark via the network), while ■ stands for parameter uploads from the JVM to the GPU (i.e. the respective model shard is updated). Phases where the workers update their model shards by training using local optimizers are shaded ■. Notice how model training is halted during communication phases.

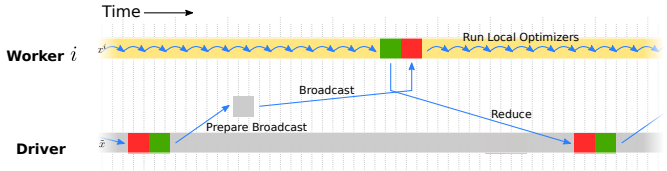


Fig. 6. Single phase coordinated asynchronous communication pattern. The cluster is either exclusively broadcasting or reducing. Because the network bandwidth is poorly utilized, isolated learning phases are very long. However, note how model training is only briefly interrupted in comparison to the significant stalling of the synchronous approach (Fig. 5).

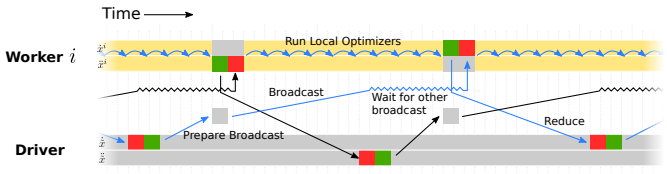


Fig. 7. 2-phase asynchronous communication pattern with simultaneous broadcast and reduction. The model is split into two shards, indicated by the two horizontal bars for both, the driver and the workers. The parameters belonging to both shards are simultaneously trained by the workers (■), but the repeatedly invoked broadcasts and reductions alternate their focus. Thereby, they establish two distinct parameter exchange cycles. One for either model shard. Unlike in the single phase asynchronous approach (Fig. 6), broadcast and reduce are executed simultaneously, which doubles the available bandwidth and reduces the duration of isolated learning phases.

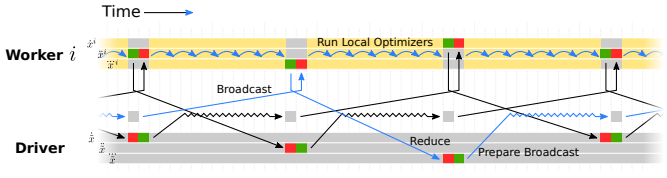


Fig. 8. 3-phase asynchronous communication pattern with simultaneous broadcast, reduction and staging of the next broadcast. Operates like the 2-phase asynchronous communication pattern (Fig. 7), except that the model is split into three shards with distinct parameter exchange cycles. This further improves network utilization since the broadcast-variable for the next model shard is prepared upfront. Thus, waiting periods that delay further reductions as shown in Fig. 7 are eliminated. The next broadcast/reduction-cycle can begin immediately.

τ, which in turn depends on the network speed and the cluster size. Furthermore, each reduction cycle represents a global synchronization barrier that requires all workers to reach a certain state. EASGD (Section 3.2) does not require synchronization barriers but depends on worker initiated peer-to-peer communication, which is not compatible with the Spark MapReduce-style execution and also less efficient, since the communication demand with the parameter server increases linearly with each additional worker. Thus, τ must be scaled accordingly, which may require long phases of isolated learning.

In this section we present our algorithm MPCA SGD, which combines the advantages of the synchronous and the asynchronous approach on top of Spark, but avoids their disadvantages. Spark restricts us to master-initiated communication patterns. Hence, we retain the general idea from SparkNet of having communication phases that are instigated by RDD-commands from the driver. But we let each worker continue to improve its local model asynchronously throughout these communication phases. To combine models we adopt the L^2-norm based penalization approach from EASGD, which offers an intuitive way to integrate multiple co-adapting models with each other. Thus, our approach completely decouples the computation and communication aspects, thereby effectively eliminating most bottlenecks and waiting periods of existing approaches.

4.1 Overall execution behavior

After connecting to the Spark-cluster, we create a special RDD that we call the AgentRDD and commit one Spark execution thread per GPU to it. We explain this RDD-type in more detail in Section 5. Once the AgentRDD has been fully initialized, RDDs inheriting from it may hand-off map-tasks to the Spark threads bound by the AgentRDD. This allows completing reduction tasks without stopping computation upon leaving the execution scope of a RDD partition. Furthermore, we use concurrent programming to minimize network transfer times by allowing broadcasts and reductions to be executed simultaneously. Thus, workers may asynchronously integrate the latest broadcasted state and continue optimizing their local models through the AgentRDD, while model updates are reduced.

Like in SparkNet, our driver program maintains and broadcasts the joint model ($\tilde{x}$). The local models (x^i) are stored in model state RDDs that can be reduced to form a new joint model. However, instead of stopping the optimizers while the local models are being merged, the optimization code is executed indefinitely via the AgentRDD. We call this method *coordinated asynchronous* execution because the local optimizers operate asynchronously, while all parameter exchanges are coordinated by the driver node.

4.2 Multi-Phase parameter exchange

In contrast to the synchronous approach (Fig. 5), the coordinated asynchronous method allows performing communication and computation in parallel (Fig. 6). However, each communication phase still consists of a broadcast followed by a reduction. New model parameters can only be broadcasted after a complete reduction has occurred. Hence, there is a synchronization barrier between both operations.

While Spark utilizes the network bandwidth of multiple cluster nodes to accelerate broadcasts and reductions, there is always a dominant communication direction (from driver to workers or vice versa). In a full-duplex environment like Ethernet, this typically means that a great fraction of the

available network bandwidth is unused if broadcasts and reductions are executed sequentially. We can reduce the time required for parameter exchange round-trips with the driver by using these resources. This will allow us to have shorter isolated learning phases, which leads to less stale parameter updates and better overall control during model training.

To allow broadcast and reduce to be used simultaneously, we slice the model into multiple shards ($\dot{x}$, $\ddot{x}$, etc.) of approximately equal size and alternate the communication phases between them. In the simplest case, we split the model parameters in half and let them alternate between the broadcast and reduction phases. Thus, while the first half of the model is being broadcasted, the second half is being reduced and vice versa (Fig. 7). In practice, the Spark scheduler copes very well with this communication pattern. It adjusts the broadcasts and construction of reduction trees to make the best use of the momentarily available network resources. If the model shards are well balanced, the average communication delay drops by up to 30% with two alternating communication phases.

As shown in Fig. 6 and Fig. 7, broadcasting cannot start immediately after a reduction has been completed because a preparation step is necessary. This preparation step consists of computing and applying the update to the joint model (Section 4.5), extrapolating the joint model (Section 4.3) and constructing the broadcast variable, which involves serializing the model parameters, splitting them into fixed size chunks and registering these chunks with the Spark block manager [10]. While none of these operations is very time-consuming, the cumulative delay often equates to multiple back-propagations in the workers. To avoid such delays, we extend the previously discussed 2-phase asynchronous communication pattern by splitting the model into three shards of preferably the same size and regard the preparation of the next broadcast variable as another phase in our communication cycle. Analogously to the 2-phase pattern, the focus of the three communication phases alternates cyclically between the three model shards. Fig. 8 illustrates this 3-phase asynchronous communication pattern. Aside from the split-seconds during which the focus is switched between shards, the driver and by extension most workers are permanently sending and receiving model parameters. Depending on the individual model shard sizes and Spark broadcast settings, the 3-phase asynchronous communication pattern can utilize up to 80% of the driver's network I/O bandwidth. Simultaneously, the Spark communication primitives broadcast and tree-reduce ensure that network resources across the cluster are dynamically assigned and utilized to maximize throughput.

Note that, although it is possible to have more than three model shards, there are only three atomic operations that must be applied sequentially (broadcast, reduce and preparing the next broadcast). Splitting the model into more shards will decrease their size. Thus, broadcasting and reducing them becomes faster. However, each Spark operation incurs a small overhead and each additional shard edge induces irritations during model optimization due to misalignment issues (Section 4.4). Thus, having four or more shards typically results in lower network utilization and decreased overall training performance.

In Fig. 9, we present an overview of the resource usage over time when training a large deep learning model with

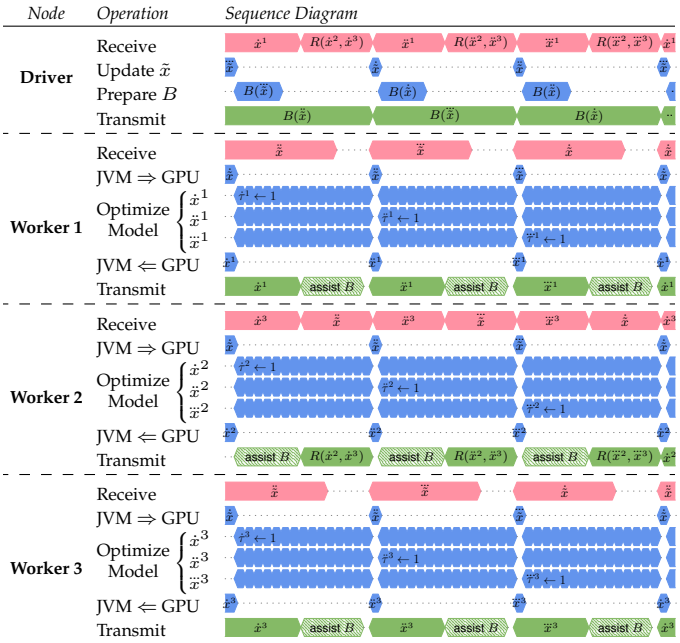


Fig. 9. Communication and computation resource usage over time in MPCA SGD with 3-phase asynchronous processing (cf. Fig. 8). The driver repeatedly broadcasts ($B(\ldots)$) shards from the joint model ($\tilde{x}, \dddot{\tilde{x}}, \ddot{\tilde{x}}$), instigates reductions of shards from the local models ($\dot{x}^i, \ddot{x}^i, \dddot{x}^i$) and prepares the next broadcast variable simultaneously. Since there are three workers in this example, the reduction tree depth is $\lceil \log_2(3) \rceil$. Thus, one worker can send its model shard directly to the driver, while another has to send its corresponding shard to an intermediate aggregator. Once the intermediate aggregator has fully received the shard, it is combined ($R(\ldots)$) with the local representation of that shard and sent to the driver. We rely on the Spark scheduling engine to provide a near-optimal role assignment for each transmission. In reality, all network communication processes are more dynamic than shown here. However, for the sake of clarity we let worker 2 permanently act as intermediate aggregator. Note that while worker 2 receives a model shard it will utilize its unused transmission bandwidth to assist the current broadcasting efforts of the driver. At the same time the workers optimize all three model shards simultaneously. However, since each shard has its own communication cycle, which can sometimes be longer or shorter depending on transmission delays, we maintain an iteration counter ($\dot{\tau}, \ddot{\tau}, \dddot{\tau}$) for each shard on each worker. Unlike SparkNet and EASGD, which require the user to preconfigure τ (Section 3), we measure this value at runtime. Further computations can be scaled using this value (Sections 4.5 and 4.6), which allows MPCA SGD to adapt to changes in the available network bandwidth.

MPCA SGD. Note how the workers collaborate with the driver to accelerate exchanging parameters and how communication (▮/▮) and computation (▮) phases overlap. Also note how all GPUs continuously improve their respective local models, except for the brief moments when parameter updates have to be up- or downloaded.

4.3 Mitigating the staleness of updates

Once the driver has updated the joint model, it has to broadcast the new state to make it visible to all workers. Because the workers continue improving their models while this happens, their local state will have progressed by the time when the updated joint model has been received. If workers use stale updates for penalizing their local models (e.g. using the EASGD method) some progress that has been made in the meantime could be reversed. However, note that updates to the joint model always depend on

aggregates over all local models. Thus, the development of the joint model is very stable and each additional worker improves the stability [28]. We can use this to our advantage to improve updates shared by the driver.

$$\tilde{v}_t = \delta\tilde{v}_{t-1} + (1-\delta)(\tilde{x}_t - \tilde{x}_{t-1}) \quad (4)$$

$$\tilde{x}^*_{t+\gamma} = \tilde{x}_t + \gamma\tilde{v}_t \quad (5)$$

Instead of broadcasting the actual state of the joint model ($\tilde{x}_t$), we track the optimization trajectory ($\tilde{v}_t$) by observing the recent changes to the joint model (Equation 4) and extrapolate $\tilde{x}_t$ along this trajectory (Equation 5) to approximate the model parameters that we expect to see during a later reduction ($\tilde{x}^*_{t+\gamma}$). We implement $\tilde{v}_t$ as a simple moving average, where δ determines the rate at which changes of the joint model state are being adopted into $\tilde{v}_t$. By default, we use $\delta = 0.8$. Note that $\tilde{v}_t$ only depends on actual responses from the workers and not on previous incarnations of $\tilde{x}^*$, since that would be very unstable. The delay and smoothing induced by δ acts as a buffer against sporadic events that may cause the communication speed to suddenly drop or spike. Since $\tilde{v}_t$ shrinks and increases automatically depending on the network speed, the amount by which the joint state is projected into the future correlates with the average number of update steps that the local optimizers apply during a communication cycle. γ controls how far (in terms of $\tilde{v}_t$) the joint model state should be projected into the future. $\gamma = 1$ equals to looking ahead one cycle, while $\gamma = 0$ disables this feature. In practice such simple linear extrapolations are of course inaccurate. Especially, if we train from random values, the initial changes are large and noisy. Thus, we typically start training using a low value for γ that we ramp up as the model stabilizes [29].

Fig. 10a shows the influence of this linear extrapolation method on model training. We trained a ResNet-110 [4] twice in a bandwidth constrained environment with Adam [27] using MPCA SGD on the CIFAR-10 dataset using 8 workers. The first run was conducted without linear extrapolation ($\gamma = 0$) and the second run with $\delta = 0.8$ and $\gamma = 0.7$. We determine the quality of the models using online cross validation. As can be seen, the convergence speed is initially lower when extrapolation is enabled. This is because we deliberately did not ramp up γ in this experiment. However, the extrapolated variant catches up and eventually surpasses its counterpart without extrapolation.

4.4 Worker model update

The optimizers in MPCA SGD adjust their local models during parameter exchange. If we would simply replace the local model once an update has been received, we would effectively discard the progress made in the meantime. However, MPCA SGD splits the model into shards, which are reduced and shared separately. If we increase the number of shards, such that there are significantly more shards than communication phases, we could freeze the parts of the model that are currently being exchanged via the network and focus only on improving the remaining shards. However, in practice this does not work well for two reasons. First, several computation steps for the frozen portion of the model still have to be executed, which wastes resources. Second, suddenly replacing a model shard confuses optimizers with momentum based step size control like NAG [30] or Adam [27]. This is not surprising, since the new parameters may have different properties. After replacing a model shard, the inputs from and outputs to adjacent model shards will be misaligned (Fig. 10b), which can massively increase fluctuations in signals that are propagated through them. Note that this misalignment increases with the length of isolated learning.

To avoid both problems, we let the workers continuously update the entire model. Instead of replacing, we implement an L^2-norm based penalization mechanism similar to EASGD (Section 3.2). However, EASGD and dependent works only penalize their local models after new parameters have been downloaded [1]. In EASGD, this interval is fixed and depends on τ. Thus, communication and penalization have to be aligned around this parameter, which can lead to conflicts (Section 3.2.2). In contrast, parameter sharing in MPCA SGD is conducted at the maximum pace at which the driver can collect and distribute updated model shards. The time between two consecutive updates of the same shard may vary depending on many factors, including influences from other processes. Therefore, τ is determined automatically at runtime (Section 4.6).

However, note that the optimal value for the model penalization-factor (α) still depends on the length of isolated training [28]. Since the round-trip periods vary, α has to be scaled differently for each worker and shard during each update cycle. While it would be possible to scale α dynamically, we found it difficult to configure such a mechanism, since it further complicates the relationship between α and the network bandwidth and GPU performance of each worker. Furthermore, applying uneven penalties across the model could introduce undesired biases. However, instead of aligning the penalization code with the communication cycle [1], [13], we can also perform this step while updating the local models in the optimizers. The benefits of penalizing the local models directly in the optimizer loop are twofold. First, this causes many comparatively small adjustments, which results in a smoother overall behavior. Notice that as long as α is small, the effects of the above mentioned misalignment problem should be dampened as well (Fig. 10b). Second, this method completely decouples computation and communication and, thus, disentangles the related hyper parameters. Regardless of the available network bandwidth, penalization solely scales depending on the number of updates to the local model (i.e. GPU performance) and the distance to the projected joint model state (Section 4.3). Thus, in MPCA SGD we may choose α irrespective of most communication related factors.

We propose two ways to penalize the local models (x^i) based on their euclidean distance [14], [15], [16] from the projected joint model ($\tilde{x}^*_{t+\gamma}$; see Section 4.3) within the optimizer loop: Intuitively, it is possible to implement the penalization as an L^2-regularizer (Equation 6). An alternative method is to directly modify the local models before taking an optimization step (Equation 7).

$$\min_{x^i} \left(\mathbb{E}\mathcal{L}(x^i, \xi^i) + \lambda\|x^i\|^2 + \frac{\alpha}{2}\|x^i - \tilde{x}^*_{t+\gamma}\|^2 \right) \quad (6)$$

$$\min_{x^i} \left(\mathbb{E}\mathcal{L}(x^i - \alpha(x^i - \tilde{x}^*_{t+\gamma}), \xi^i) + \lambda\|x^i - \alpha(x^i - \tilde{x}^*_{t+\gamma})\|^2 \right) \quad (7)$$

Both methods break the potentially large correction step that EASGD performs down into many small steps. The first method (Equation 6) modifies the shape of the loss-

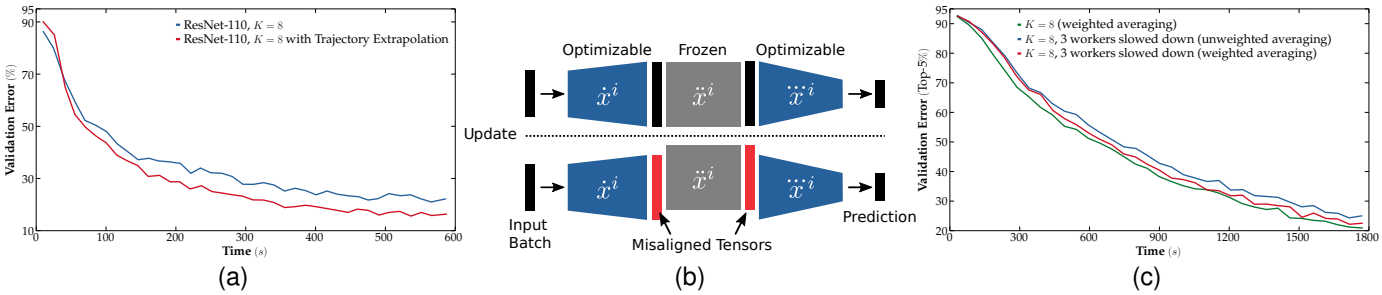


Fig. 10. (a) Influence of linearly extrapolating the joint model on the prediction error; (b) Tensor misalignment upon separately updating a model shard; (c) Influence of stragglers on the prediction error if reduction is done via unweighted averaging, compared to weighted averaging using τ^i.

function. Thus, penalization is applied through slightly tilting the gradients during back-propagation. The second method (Equation 7) gently shifts the model towards the projected joint model state. This operation is transparent to the remaining optimization code. In practice, both methods can work well with MPCA SGD. However, we have found that the first method is significantly more sensitive to the choice of α. Thus, we will focus the remainder of the discussion on the second method.

The penalization $(x^i - \alpha(x^i - \tilde{x}^*_{t+\gamma}))$ in Equation 7 is executed just before computing gradients. Thus, like in the Nesterov Accelerated Gradient (NAG) algorithm, we first displace the model and then allow the optimizer to devise a correction step [30]. This can also be understood as having small force that increases with the distance to the joint model state. Only if successive parameter updates remain larger than this force, the model will have enough momentum to escape the pull of the projected joint model state. Parameter updates in diffuse directions will be largely ignored. As a result, only parameter adjustments that generalize well across many batches are kept and submitted to the driver. The synchronous reduction of results from all workers ensures that only parameter adjustments, where the majority of the workers are in consensus, have a major influence on the next joint model state. We found that this two-stage distillation process is critical to MPCA SGD's success, since it significantly reduces the variance in the joint model and quickly produces models that generalize well.

4.5 Driver model update

In SparkNet [6] the local models are optimized in isolation for τ iterations and then aggregated through arithmetic averaging ($R(x^{1,\cdots,K}) = \frac{1}{K}\sum_{i=1}^{K} x^i$). The aggregate replaces the joint model ($\tilde{x}_t$). For analytical purposes, Zhang et. al. [1] propose a synchronous variant of EASGD with a similar execution pattern. They suggest that blending the aggregated local models and the previous state of the joint model ($\tilde{x}_{t-1}$) together can improve stability through time (Equation 8). The parameter β denotes the rate at which the joint model is being replaced. Note that setting $\beta = K\alpha$ establishes elastic symmetry between the workers and the driver similar to asynchronous EASGD (Section 3.2).

$$\tilde{x}_t = (1-\beta)\tilde{x}_{t-1} + \beta R(x^{1,\cdots,K}) \tag{8}$$

Since MPCA SGD can be expected to exhibit a much more dynamic runtime behavior due to not having a clear distinction between the reduction and model improvement phases, we cannot take an analytic approach. Instead, our method for choosing β is practically motivated. Our intuition is as follows: If we start training from a random initialization, the driver model does not contain any useful information. Hence, it is in the most sub-optimal state with respect to the optimization objective. There is no need to hold onto that state. The optimizers in the workers, on the other hand, immediately start programming information into their local models. Thus, we start training with $\beta = 1$ to absorb this information into the joint model as fast as possible. Then we subsequently ramp down β to stabilize the overall convergence trajectory. However, β must not become too small [28]. According to our experience, having a target value of $\beta = 0.9$ represents a good compromise between stability and model improvement in clusters with 8 workers or less, which is in alignment with findings by Zhang et. al. [1].

4.6 Mitigating performance deviations among workers

Workers in MPCA SGD operate at their own pace. Hence, they may apply different amounts of updates to their local models in the same amount of time. Sporadic performance deviations between workers should not be an issue, since our entire approach is based on merging slightly differently developing models to find better generalizations. However, if individual workers exhibit a systematically different runtime behavior, for instance because they are equipped with different hardware or use contested resources, they will influence the entire cluster if the joint model state is derived from the arithmetic average of the worker models. To mitigate this effect, we follow our intuition from Section 4.5 and assume that workers that can utilize more resources have also more thoroughly explored the parameter space. Thus, their local models (x^i) have absorbed more information than others. By weighting results, we can scale the influence of each worker on the next joint model state accordingly. This allows using workers equipped with different hardware for the same optimization task, which is a problem that has - to our knowledge - not been comprehensively addressed by other works in this field, although heterogeneous cluster setups are common nowadays. We implement scaling by measuring the number of optimization steps (τ^i) that each worker completes between reduction cycles for the same model shard and weight their contribution to the aggregate model proportionally (Equation 9).

$$R(x^{1,\cdots,K}) = \frac{1}{\sum_{i=1}^{K}\tau^i}\left(\sum_{i=1}^{K}\tau^i x^i\right) \tag{9}$$

In Fig. 10c, we demonstrate the influence of both averaging methods in a situation where resources on some cluster nodes are contested, which is a very likely scenario in a multi-tenant system. Each time we trained a ResNet-34 to classify images using Adam on a 100-class subset of the ImageNet dataset and determine the model quality using online cross validation. The run without contested resources represents the upper bound. In the two remaining test runs 3 workers also process another workload. On these nodes, only 50% of the nominal capacity of the GPUs is available to the deep learning task. Note how the configuration with weighted averaging retains a higher convergence rate than that with unweighted averaging.

5 Implementing MPCA SGD on Spark

We will now discuss how MPCA SGD can be implemented in Spark. The coordinated asynchronous execution we described in Section 4 differs fundamentally from the bulk-synchronous iterative MapReduce processing that is normally used to program Spark. Spark jobs are supposed to have no side-effects after they complete. However, to implement MPCA SGD, we have to conduct operations and transformations during the communication phase. We will address this issue in Section 5.1. In Section 5.2 we describe how we realize the simultaneous broadcasting and reduction of model shards. Thereafter, we will discuss strategies to recover from node failures in Section 5.3.

5.1 Allowing asynchronous task execution in Spark

We introduce the agent concept to enable continuous updating while retaining the ability to apply MapReduce transformations to RDDs. Agents are implemented using a special type of RDD that we call *AgentRDD*, but represent a more coarse grained long lived scheduling primitive than ordinary RDD partitions. Each partition of an AgentRDD contains exactly one agent. Typically, we allocate one agent for each GPU, but the number of agents and their assignment to compute-resources can be regulated independently in each worker if desired. A custom partitioning scheme ensures that each agent forms a persistent bond with a specific GPU on a specific worker. Through placement hints and inhibiting task relocation, we enforce that the number and execution locations for partitions of AgentRDDs and dependent RDDs correlate. Partitions derived from an AgentRDD can embed *Contexts* that allow pairing with the corresponding agent. Using these contexts, it is possible to hand-off workloads to the agent for delayed execution or request results of previously handed off tasks. To achieve this, agents need to be active. However, the creation of persistent background-threads interferes with Spark's resource management. We solve this problem by ensuring during initialization that we only bind executors if 2 CPU cores can be exclusively allocated per agent/GPU. This allows us to simultaneously run Spark jobs in the context of the AgentRDD and dependent RDDs without queuing.

Fig. 11 illustrates how agents and RDDs interact during model training. The first AgentRDD allocates the data structures for agents and fires up their event-loops through the *Control API*. This will allocate one Spark worker thread permanently to each agent, thus marking the related Spark executor as *healthy* & *active*, which prevents the resource

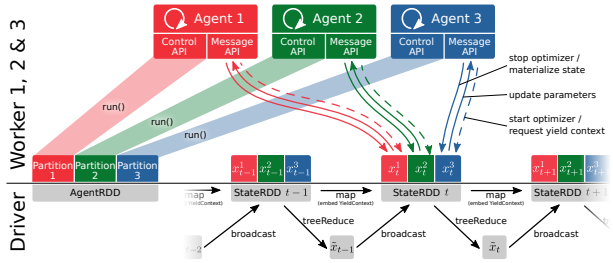


Fig. 11. Working principle of agents during model training using a cluster with 3 workers. We imply single-phase communication (Fig. 6). The AgentRDD borrows its threads to the agents, which in turn service the Message APIs and allow the state RDDs to continue model optimization between reductions. The driver repeatedly instigates MapReduce jobs and uses the reduction result to update the joint model. All updated parameters are broadcasted to the workers during the following cycle. The map-portion of each MapReduce-job forges a new RDD each time. At the beginning of the map-function the workers locate their respective agent through YieldContexts, which are embedded in the current state RDD. Using the Message API, they signal the agent to interrupt optimization and download the current model state. Then they submit updates and re-engage the optimizer. This will return a new YieldContext that can be embedded in the next model state RDD.

management from reclaiming it. Agents operate internally using a simple event-loop that listens to a message queue. Through embedded contexts, dependent RDDs can locate and access the *Message API* of their corresponding agents to signal them or submit new tasks. For long running tasks, such as optimizing a model, agents immediately return a *YieldContext* after the task has been accepted. This context, is best understood as a proxy for the future state of the model after the optimizer is interrupted next. The YieldContext is constructed such that it will stop the optimizer and materialize the current model state if it is accessed for any reason (including serialization). This allows embedding the YieldContext in a RDD to capture the next model state without actually knowing what this model state will be. Thus, the content of the next RDD is not determined at creation. But should any Spark operation attempt to access the YieldContext, it will materialize the model state and become immutable. Thus, subsequent queries against this RDD will yield the same results. Should the next RDD get destroyed for any reason, the finalization code of the YieldContext will automatically signal the agent to stop the optimizer at the next opportunity.

Our method of integrating coordinated asynchronous SGD into Spark is non-conventional but achieves its intended purpose while obeying the constraints imposed by the Spark system. We effectively decouple the model updating phase (map) from the global synchronization barrier that is enforced by reductions, by handing off the model optimization to the Spark worker threads bound by the AgentRDD and allowing the model state to be materialized just in time when it is required. As a result, we gain the ability to continue computation while data is transmitted or received via the network. Our tests show that this method is durable enough, such that multiple models can be trained concurrently on the same Spark-cluster for weeks alongside other Spark applications.

5.2 Allowing simultaneous broadcast and reduction

To realize multi-phase asynchronous execution as discussed in Section 4.2, we need to be able to broadcast and re-

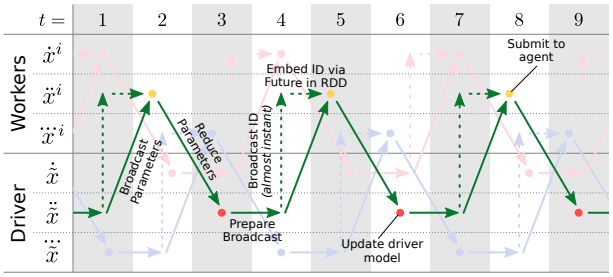


Fig. 12. Data flow of model updates in MPCA SGD to allow executing broadcast and reduce in parallel. The highlighted path represents the parameter exchange cycle for the second model shard ($\ddot{x}$).

duce different model shards simultaneously. Notice that we generate new RDDs as a side-effect of our continuous invocations of map (Fig. 11). Furthermore, note that Spark broadcasts are receiver selective, meaning that downloading the backing variable has to be initiated from the workers. To avoid blocking, we hand-off the receiving of broadcasted model parameters to a background-task by wrapping the broadcast variable into a *Future* and placing it along with the YieldContext in the next model state RDD. When we return during the next parameter exchange phase, we extract this value from the RDD and submit it to the optimizer before re-engaging it.

Fig. 12 depicts how ongoing parallel parameter broadcasting is realized under these constraints in MPCA SGD. During each communication cycle only the ID of a previously prepared broadcast variable is transmitted as we schedule the next reduction (⇡). Once received by the workers, they embed this newly created broadcast variable in a Future and store it in the next model state RDD (⇢). The Future then triggers the downloading and initialization of the variable. Simultaneously, the update received by the previous Future is extracted and submitted to the backing agent (●).

5.3 Recovery from node failures

Spark applications traditionally achieve fault tolerance by either reapplying the transformations of lost partitions from lineage information stored in the RDD, or through restoring previous states by recalling a checkpoint (snapshot of an RDD stored in HDFS). However, in our case each new model state RDD depends on its predecessor and the shard from the joint model broadcasted at the time. To minimize delays, each worker is interrupted ad-hoc once it has received updated parameters. To allow recomputing lost RDD partitions from linage, we have to record the interruption time of each worker and keep the related broadcast variables around. Thus, maintaining long lineages can quickly consume vast amounts of memory due to the cumulative size of the broadcasted shards. Furthermore, it should be noted that repeating the actual transformations (i.e. the model training) is computationally expensive. However, frequently checkpointing the model state RDD to truncate the linage graph is often impractical due to the amount of storage space required (one model copy per worker). This problem is further amplified by the fact that HDFS achieves robustness through replication across the cluster, which reduces the available bandwidth for training.

Therefore, we discourage using the traditional Spark methods to achieve fail-over processing when training with MPCA SGD. Instead, we can recover from failures as follows. There are two scenarios to consider: 1) the driver stays alive, but a worker dies; and 2) many workers or the driver-node have failed. For scenario 1, we simply instantiate a new AgentRDD and initiate agent discovery. This will rediscover the agents that are still active (i.e. optimizing the model) and bind them to the new AgentRDD. Henceforth, we continue training using the salvaged agents. Thus, if a single worker fails, the new AgentRDD has simply one partition less and model state RDDs derived from it also have one partition less. The overall training speed is slightly reduced. To recover from scenario 2, the driver may backup the current state of the joint model. In our implementation, automatic backups in the driver can be realized by adding the function *DumpModel* to the list of optimization objectives. This objective writes the current model state to a persistent storage location (e.g. RAID-drive, network-share, HDFS, etc.). Note that objectives are evaluated asynchronously. Thus, with a multi-core CPU occasional backups of the joint model do not significantly slow down training. Using this backup it is possible to restart the optimization process using a different machine as the driver, or even using a different cluster. The cluster can have a different number of GPUs, etc. Any changes made after the most recent backup are lost, but the amount of wasted resources can easily be limited by adjusting the backup-frequency. Note that upon restarting training all workers are initialized to the restored joint model state. Thus, at first they explore the parameter space at the same location and will propose similar updates, which results in a reduced convergence performance. However, random processes during training (e.g. different training samples, dropout patterns, etc.) quickly cause the workers to diverge from each other again and explore adjacent decent trajectories.

6 Experiment Setup

6.1 Environment

We run all experiments in a dedicated cluster that is managed using Cloudera CDH 5. Our test setup consists of 8 Spark executors equipped with 2 GHz 8-core Xeon CPUs, 64 GB RAM, NVIDIA TitanX GPUs and an Ethernet bandwidth of 1 Gb/s. The number of nodes utilized (K) will be stated explicitly for each experiment.

6.2 Datasets

CIFAR-10 contains 32x32 pixel RGB images that are evenly distributed across 10 classes. The dataset is split into a training set with 50,000 images and a test set consisting of 10,000 images. During all experiments we withheld 5,000 images (500 per class) from the training set that we used for cross validation purposes. All images were augmented by subtracting the per-channel mean of the training set [17] and applying horizontal flipping at random [31]. The training set was further expanded by padding 4 black pixels on all sides of each image and cropping random 32x32 patches [4].

ImageNet-1k consists of one unlabeled (100k) and two labeled (1.3M and 50k) collections of JPEG images that depict objects from 1000 classes. We use the smaller labeled set

for testing and the larger set for training, but withhold 25k images (25 images per class) for cross validation purposes. Since the compressed size of the labeled images is 143 GiB, we will stream them from a HDFS during training. All images were scaled such that their shorter edge measures 256 pixels. We also subtracted the per-channel mean of the training set and further augment the datasets during training and testing by randomly flipping the images horizontally and cropping random 224x224 patches [17], [31].

6.3 Implemented systems

In this section we describe the different distributed deep learning systems that we have included in our experimental study.

The synchronous approach applies synchronous decentralized training as discussed in Section 3.1. Our implementation uses Spark broadcasts and binary reduction trees to speedup communication. Unless stated differently, we configure the length of isolated training (τ) according to Table 1, such that the average GPU utilization is 83.3% as suggested by Moritz et. al. [6].

EASGD [1] applies asynchronous decentralized training as discussed in Section 3.2. As suggested by Zhang et. al. [1], we establish elastic symmetry during all experiments by choosing $\alpha = \frac{0.9}{K}$. To minimize communication delays due to overlapping parameter exchange requests from the workers, we select τ approximately 30% higher than the theoretically optimal value.

Apache MXNet [5] is a state-of-the-art centralized distributed deep learning system that supports synchronous and asynchronous execution. For each experiment, we will only report numbers for whichever mode worked better and indicate this explicitly. To maximize the utilization of the available network bandwidth, we let each of our K workers take over parameter server functions for $\frac{1}{K}$ of the model. MXNet and our deep learning package *Inferno* use the same cuDNN [32] version to schedule processing on the GPU. Like other deep learning systems that rely on cuDNN, both systems exhibit approximately the same single GPU performance [33].

MPCA SGD is our multi-phase coordinated asynchronous execution method as discussed in Section 4. This approach does not require balancing computation and communication manually. Since we penalize the local models every time an optimizer updates the model, penalties (Section 4.4) stack automatically depending on the individual performance of each worker. The countermeasures we take to mitigate staleness (Section 4.3) further improve the effectiveness of this training method. To maximize the convergence performance, we have to balance exploration and exploitation (Section 3.2) by choosing α (Equation 7). Like in EASGD, a sub-optimal value for α results in reduced convergence performance [28]. But in contrast to Kim et. al. [2], we found that complex fine-tuned parameter schedules for α are not necessary in MPCA SGD. Usually, a broad corridor of values for α exists that works reasonably well. To find this range of values, we suggest performing a grid search over α using a significantly reduced training dataset, while keeping all other parameters the same. We will discuss specific configuration choices separately for each experiment.

However, note that one exception exists where establishing a special schedule for α can significantly speedup convergence. If we begin training from a random initialization, the entropy is maximized. Each worker will explore different parts of the parameter space depending on the sequence of samples it chooses. At this point in time we cannot know which decent trajectory will work well. Thus, we suggest to always begin training with $\alpha = 0$. This allows the local optimizers to examine the loss function independently and come up with better model parameters. After a couple of iterations, the joint model contains a more informed state that reflects adjustments that a majority of the workers found useful. If we would continue like this, the workers would eventually diverge. Thus, next we force all workers to quickly adopt coherent decent trajectories by setting α to a high value (e.g. 0.5). Then, we subsequently decay this value until we arrive at the first value of our actual parameter schedule.

6.4 Implemented models

ResNet-110 [4] is a model for classifying images from the CIFAR-10 dataset. It is based on the concept of having shortcut connections in parallel with convolutional layers to learn residual functions. ResNet-110 makes heavy use of batch normalization, which allows stacking residual blocks very deep. However, its size is only 6.7 MiB. Thus, it can be distributed and reduced quickly in our test environment (Table 1). We therefore expect all implemented deep learning systems to perform similar when training this model.

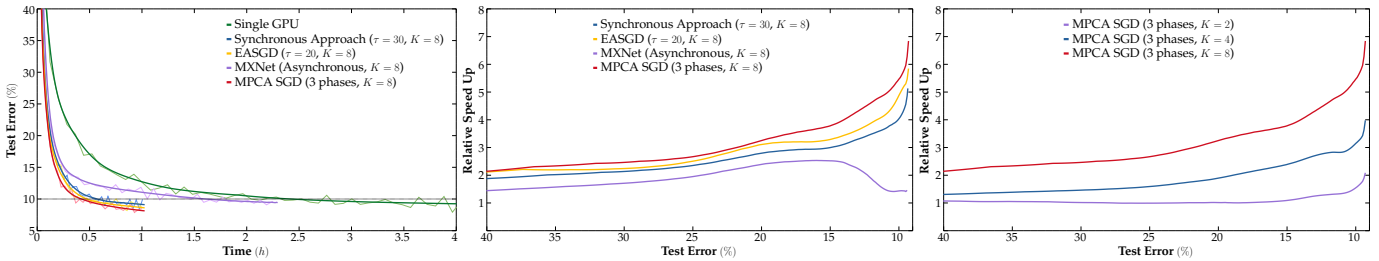
ResNet-152 [4] is a 152-layer ResNet-variant designed for the ImageNet dataset. It requires a lot of memory during back-propagation. This tightly limits the maximum mini-batch size that can be used. Therefore, we expect that this model will benefit significantly from distributed training.

VGG-A [17] is the smallest variant of a popular modular network design to classify and localize objects in images. Due to its model size of approximately 500 MiB, a full parameter exchange for a VGG-A via Ethernet can take many seconds (Table 1). Thus, we expect that centralized and synchronous deep learning systems will perform significantly worse than MPCA SGD and EASGD for this model.

6.5 Initialization, optimizer and hyper-parameters

We initialize all random seeds identically at the beginning of each experiment. For implementations based on our platform this will result in exactly the same initialization of the model parameters. However, the runtime behavior of EASGD, MPCA SGD and some cuDNN operations is not deterministic [1], [32]. Thus, the training performance may still vary slightly across training runs. Due to its fundamentally different inner workings, the initial state of the model when training with MXNet will be different. However, in both cases all parameter values are sampled from a Gaussian distribution with the same properties and then scaled as suggested by He et. al. [34].

Classic optimizers like momentum SGD require complex learning-rate schedules to operate well. However, the optimal learning-rate schedule might be different for each distributed configuration. To allow comparisons, we will use the Adam [27] optimizer, since it is fast, adaptable and well-known for its robust hyper-parameters. Furthermore,



(a) Wall Time vs. Prediction Error (b) Prediction Error vs. Rel. SpeedUp (c) Scale-out behavior

Fig. 13. Training performance for ResNet-110 on the CIFAR-10 dataset using different deep learning implementations.

we will only set modest optimization goals, that we can comfortably reach using a constant learning rate.

6.6 Evaluation metric

For image classification, the major metric of interest is the correctness of the predictions of the classifier. Hence, for all our experiments, we will report the average class prediction error on the test set. For each experiment, we will set a goal for the prediction error and stop training shortly after reaching that goal. We will use the wall time it takes to reach a certain average prediction error as the measure for performance comparisons. Hence, if a deep learning system can reach the same average prediction error in half the time, we regard it as twice as fast.

Instead of reaching state-of-the-art accuracies our objective is to compare the rate of convergence between different distributed deep learning systems. To reach state-of-the-art accuracy would require hand tuning hyper parameters for each model and distributed system [35], which will make the results less comparable. Therefore, we followed established best practices when choosing hyper parameters and tried to keep as many variables (e.g. system environment and hyper parameters) as possible the same within and across experiments. Furthermore, we chose optimization goals such that they represent a decent challenge but can also be reached reliably regardless of factors such as the initialization seed of the model. In particular our target errors are 10% and 40% top-1 class prediction error for CIFAR 10 and ImageNet-1k respectively. We note that many papers that analyze the algorithms we compare against target similar or lower accuracies (e.g. [1], [6], [28]).

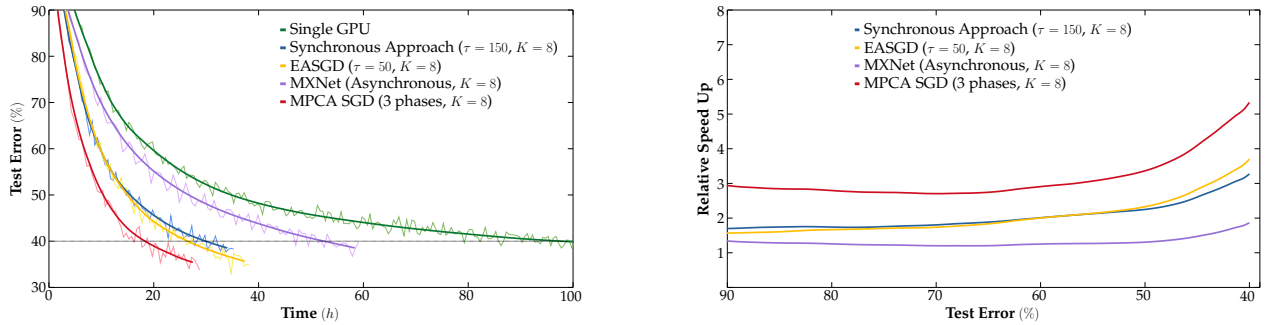
7 Main Experimental Results

7.1 CIFAR-10

We chose fitting a ResNet-110 model for this task. Our tests with single machine implementations indicate that we can comfortably reach a test error of under 10% using Adam with a learning rate of 0.001. We also found that using L2 regularization ($\lambda = 0.0001$) stabilizes training and improves reproducibility. Using grid search, we determined that a batch size of 64 allows us to reach the target average prediction error the quickest. Fig. 13a depicts the best out of 5 training runs with different deep learning systems.

Since ResNet-110 is a comparatively small model, we can choose a small value for τ and still achieve a high GPU utilization when using the synchronous approach. In all experiments, the synchronous approach converged significantly faster than its single GPU counterpart. However, over time the communication delays add up. With respect to our considerations from Section 6.3, the minimal permissible value for τ in EASGD is 7 for our test-setup. However, due to overheads in our implementation $\tau = 20$ (97% average GPU utilization) works best, but is still slightly slower than MPCA SGD. MXNet performs rather poorly, especially in the later stages of the optimization. This can partially be explained with conflicting gradients calculated by the workers due to the staleness of their local models [23]. Note that learning algorithms that employ methods to mitigate this staleness exist [36], [37]. However, in our tests plain Adam always outperformed such methods. The major problem in our environment was that MXNet is network I/O bound. This is not surprising, since executing the back-propagation algorithm can be performed in about 90 ms. But even if all 8 workers share the parameter server role, $\frac{7}{8}$ of the gradients (5.9 MiB) still have to be pushed to remote locations and the respective updated parameters have to be fetched remotely as well. Assuming perfect scheduling and no wait times, the remote parameter exchange procedure takes at least 104 ms using gigabit Ethernet. This mismatch will be a reoccurring problem in all further experiments as well. Using larger batches decreases the variance in the gradients - albeit at an exponentially decaying rate [38] - and can improve GPU utilization in MXNet. However, we found that increasing the batch size does not significantly change the outcome in terms of convergence speed.

MPCA SGD is not affected by any of these considerations and converges slightly faster than the other methods. Especially, during later optimization stages MPCA SGD manages to reduce the prediction error faster than all other methods. During this experiment, MPCA SGD continuously improves the entire model in the workers, reaching a near 100% utilization of the computation hardware. By breaking the model into 3 shards, alternating between the communication stages for each of these shards and projecting the joint model along the overall optimization trajectory ($\gamma = 0.7$, $\delta = 0.8$), we ensure that the local representation of the joint model in all workers is the same and as current as possible. To merge the models in the driver, we ramp β (Equation 8) down using a constant factor from 1.0 to 0.9 throughout the first 20 parameter exchange cycles. For integrating updates into the local models of the workers (Equation 7), we implement the early schedule described in Section 6.3 and set $\alpha = 0.05$ for the remainder of the training. Note that penalization is applied per iteration in MPCA SGD and not



(a) Wall Time vs. Prediction Error

(b) Prediction Error vs. Relative SpeedUp

Fig. 14. Comparison of convergence performance for a ResNet-152 on the full ImageNet-1k dataset.

per parameter exchange. Due to the fast computation speed of the GPUs, penalization is applied 13-17 times between two consecutive updates of the same shard. Thus, we penalize our models stronger than EASGD, but converge slightly faster. When we tried higher penalization coefficients for EASGD training slowed down. Our measurements suggest, that MPCA SGD's feature of projecting the joint model state is responsible for this advantage.

Fig. 13b shows the relative speedup over our single GPU implementation, when training using distributed systems with 8 GPUs. During the entire experiment, MPCA SGD convergences faster than all alternative approaches. Eventually, MPCA SGD reaches a stunning 6.9x speedup.

To demonstrate the scale-out performance of MPCA SGD, we repeated this experiment using different cluster sizes (Fig. 13c). Unfortunately, we do not have the resources to test MPCA SGD on larger clusters. However, as can be seen, the impact of adding resources diminishes only slowly. This gives us confidence that MPCA SGD may have further scaling potential.

7.2 ImageNet-1k

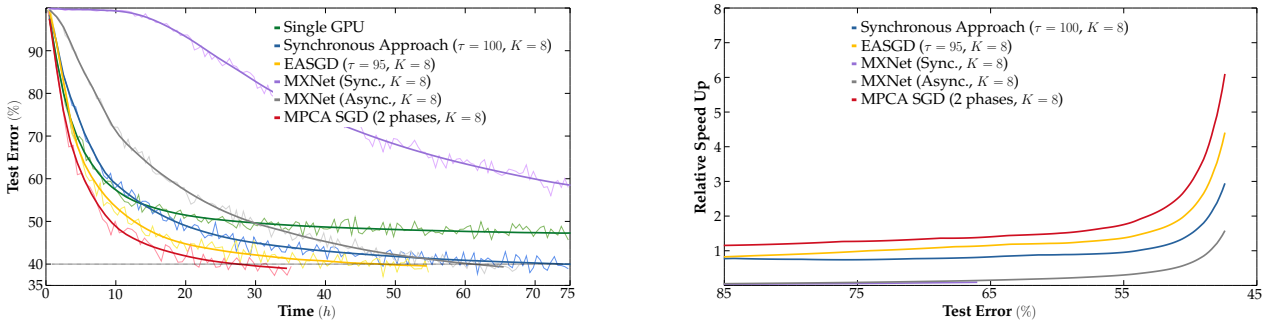
ResNet-152 and VGG-A require vast amounts of memory during back-propagation. Hence, we restrict ourselves to using a batch size of 32 for ResNet-152 and a batch size of 50 for VGG-A. Using a single GPU, both networks can be optimized well with Adam if we set both the learning rate and L2 regularization factor to 0.0001. Due to our limited resources, we stop training once the average prediction error drops below 40%, which is already a quite challenging task. However, as we have seen in the CIFAR-10 experiment (Section 7.1), MPCA SGD also holds up well during later training stages.

7.2.1 ResNet-152

Fig. 14a depicts the training progress over time for ResNet-152 and Fig. 14b shows the corresponding relative speedups in comparison with a single GPU implementation. The single GPU implementation takes just a bit less than 100 hours to arrive at a test error of 40%. As expected, all distributed implementations perform faster. However, the synchronous approach suffers severely, due to the long periods of isolated learning required to achieve a decent GPU utilization [6]. For EASGD, we can choose a significantly smaller value for τ. $\tau = 50$ yields an excellent average utilization of our computation hardware of approximately 95%. As can be seen, EASGD starts out slower than the synchronous approach, but eventually surpasses it as it becomes more difficult to make further progress. Because of the greater mismatch between computation and communication, MXNet suffers even more severely from the limited network bandwidth situation than in the CIFAR-10 experiment and converges only marginally faster than the single GPU implementation. Yet again, this demonstrates the severe limitations that systems with central optimizers face in limited bandwidth environments. MPCA SGD trains the same model significantly faster than all other methods throughout the entire experiment. To achieve this result, we parameterized MPCA SGD like in our CIFAR-10 experiment, which resulted in a near 100% GPU utilization. However, EASGD also makes good use of the computation hardware. So this cannot explain this significant performance deviation. However, note how MPCA SGD already performs faster early on (Fig. 14b). Our tests suggest that this advantage can be attributed to the relatively frequent sharing (i.e. exploiting) of parameter updates in combination with the synchronous nature of our algorithm that limits the influence of individual decisions by workers that are not shared by others. Our mechanism to project future joint model states seems to boost the local optimizers and helps them to quickly traverse descent paths that generalize well.

7.2.2 VGG-A

VGG-A is more than two times larger than ResNet-152, but computes about twice as fast. Thus, we expect to see the most extreme numbers in this experiment. Fig. 15a shows the training progress over time. For the single GPU implementation, further improvement of the model diminishes rapidly as the training progresses and it becomes more difficult to find parameter adjustments that generalize well. Since it was foreseeable that the single GPU implementation would require hundreds of hours to reach an average prediction error of 40%, we gave up training after 75 hours. For the synchronous approach, we attempted choosing $\tau = 500$ to retain a communication to computation ratio of 1:5 (Table 1). However, this did not work well. Depending on the initialization seed, the model did either not converge, or improved only at a very slow pace. With some trial and error, we found that $\tau = 100$ works reliably. Of course this is wasteful, because the GPUs sit idle for approximately $\frac{1}{2}$ of the training time. Note how the synchronous approach improves even slower than the single GPU implementation for a couple of hours. However, having more resources for exploring the parameter space eventually helps as the op-



(a) Wall Time vs. Prediction Error

(b) Prediction Error vs. Relative SpeedUp

Fig. 15. Comparison of convergence performance for a VGG-A on the full ImageNet-1k dataset.

TABLE 2
Average training time for various deep learning models across several training runs using different random initializations.

Model	Test Error	Single GPU	Sync. Approach	EASGD	MXNet (async.)	MPCA SGD
ResNet-110 *(avg. of 5 runs)*	50%	179 s	**103 s**	112 s	165 s	109 s
	30%	457 s	**252 s**	260 s	322 s	255 s
	10%	10429 s	2559 s	2070 s	7361 s	**1841 s**
ResNet-152 *(avg. of 4 runs)*	60%	20.9 h	11.5 h	12.3 h	18.7 h	**8.9 h**
	50%	38.4 h	16.9 h	18.0 h	32.7 h	**13.7 h**
	40%	103.1 h	30.9 h	27.2 h	55.4 h	**22.1 h**
VGG-A *(avg. of 4 runs)*	60%	10.2 h	11.2 h	9.4 h	⋆19.2 h	**6.7 h**
	50%	26.7 h	24.4 h	14.1 h	⋆36.0 h	**10.5 h**
	40%	-	78.4 h	53.4 h	⋆62.5 h	**32.0 h**

⋆ This model was trained without L2 regularization (Section 7.2.2)

timization difficulty increases. In contrast, EASGD achieves a decent GPU utilization of 90% with $\tau = 95$. According to our tests, this value approximately represents the minimum boundary for our test-setup, such that workers do not significantly stall each other when exchanging parameters with the parameter server. MXNet was not able to improve beyond random guessing if asynchronous updating and L2 regularization were simultaneously enabled. However, using L2 regularization together with synchronous updating worked flawlessly. Synchronous updating requires a full parameter exchange between all workers and the parameter servers after each model update. The large model size in combination with the frequent parameter synchronizations lowers the overall training speed significantly. The average throughput per worker dropped to less than 10 samples per second, which is less than the capacity of the single GPU implementation. Thus, although MXNet had 8 times more resources available in this experiment, the single GPU implementation outperforms it. We gave up synchronous training after 75 hours because we could not reach the target test error. Training using asynchronous model updates worked only when we disabled L2 regularization. Asynchronous MXNet reached the target test error after 61.2 hours. However, we advice caution when interpreting these results, since the overall convergence behavior can change considerably without regularization.

MPCA SGD is most efficient if the model can be broken into three shards (Section 4.2). Unfortunately, our deep learning system only supports splitting the model parameters along layer boundaries. This makes it impossible to have evenly sized shards for VGG-A, since the majority of parameters belong to the first fully connected layer. Therefore, we decided to split the model only into two shards and use 2-phase coordinated asynchronous processing (Fig. 7). The first shard contains only the parameters associated with the first fully connected layer, and the second shard covers all remaining parameters. While this leads to sub-optimal network I/O efficiency, MPCA SGD still performs noticeably better than the synchronous approach, EASGD and MXNet throughout the entire experiment. We can reach an average test error of 40% after approximately 27.5 hours. The slope at this time is still more steep than that of the synchronous approach, which requires more than 70 hours, and EASGD, which requires 47.5 hours, to reach the same accuracy. We used the same hyper parameters like in the ResNet-152 experiment (Section 7.2.1), except for α, which we reduced to 0.005. We determined this value through a grid search using 10% of the dataset.

In Fig. 15b, we report the corresponding relative speedups over the single GPU implementation. Note how the advantage of MPCA SGD over other implementations increases consistently throughout the entire training run. Using 8 machines, we achieve an average prediction error of 47% six times faster than the single GPU implementation.

7.3 Overall Results

To confirm our findings, we repeated all experiments several times using different random initializations. Of course, some variance between different training runs can be expected. However, especially during the ImageNet experiments, we observed very similar convergence trajectories after a couple of hours of training. The worst attempt typically deviated less than 15% from the average. Except when training VGG-A with MXNet (Section 7.2.2), we had no issues where the model did not converge at all.

In Table 2, we present the average times required to reach a certain prediction error on the test-set with different deep learning systems. The synchronous approach typically converges fast early on. In the ResNet-110 experiment it even beats all other methods. However, this advantage is marginal and vanishes quickly because the synchronous approach suffers from three problems. 1) the communication delays require choosing large τ (Section 3.1.2); 2) the communication delays add up over time and; 3) as it becomes more difficult to make further progress, the direct replacement of model parameters induces non-continuous changes, which can distract advanced optimizers (Section 4.4). EASGD shows a more consistent behavior but eventually always converged slower than MPCA SGD. We believe that this can also be attributed to the dilemma

that we have to choose high values for τ in our low-bandwidth environment to retain a high GPU utilization (Section 3.2.2). MXNet suffers from its frequent communication needs. But it also showed a comparatively inconsistent behavior throughout our experiments. A look at the network utilization of the workers reveals that this can at least be attributed partially to the fact that sometimes the workers align themselves nicely when communicating with the parameter server and sometimes they do not, which causes subsequent operations to be delayed. Our proposed method MPCA SGD adapts to the network bandwidth and takes measures to mitigate delays and deviations. During all experiments, it maintained a convergence rate that was either on par with or higher than that of other methods.

8 Conclusion

In this paper we presented a technique that combines the advantages of synchronous and asynchronous decentralized SGD-based model training. Like asynchronous systems, MPCA SGD overlaps computation and communication, but it can also take advantage of synchronous communication primitives, such as broadcast and reduce. To our knowledge, MPCA SGD is the only system that combines asynchronous decentral optimization of deep learning models on Spark with its decentralized synchronous MapReduce execution engine to realize collaborative training. We would like to emphasize that, except for low-level operations such as matrix multiplications, our entire system runs in the Spark executor JVMs and can be executed directly on top of free off-the-shelf Spark/Hadoop distributions such as Cloudera CDH.

Recently, TensorFlow [7] demonstrated that they can achieve near linear speedups for up to 64 GPUs. However, in contrast to our work these results were achieved using high-bandwidth networking hardware. Our experiments show that MPCA SGD converges faster than existing methods in bandwidth constrained environments, which is becoming increasingly more important as the performance of GPUs continues to increase rapidly relative to that of networking hardware. MPCA SGD achieves this by decoupling computation and communication to make the best use of all available resources. Unlike SparkNet and EASGD, we continuously exchange parameters for different portions of the model. Furthermore, we use L^2-penalization to gently guide the local optimizers and allow them to explore the parameter space, avoid divergence from the joint model and distill parameter adjustments that generalize well. To increase the effectiveness of these measures, MPCA SGD mitigates performance deviations and takes advantage of the synchronous nature of Spark reductions by utilizing the decreased variance in the aggregated model to lower the impact of stale updates.

The experimental results presented in this paper were focused towards solving image classification problems using CNN-based models. This problem domain is frequently used to demonstrate the capabilities of distributed deep learning systems [1], [6], [8] because it involves training large models (Big Parameters) on vast amounts of training samples (Big Data). An interesting area for future work would be to test and tune MPCA SGD for other problem domains and types of deep learning models, such as recurrent neural networks and memory networks. Furthermore, we believe that the simple linear extrapolation we used to project the joint model state is far from being perfect. It might be useful to look into more structured methods, such as those employed to mitigate delays in centralized systems [37]. Considering the implementation, it might be interesting to decentralize certain aspects of the parameter server role [11]. Furthermore, domain specific compression techniques, like the discretization of relative weight changes [39], might be useful to shrink the model size during network transmissions to speedup parameter exchanges.

References

[1] S. Zhang, A. Choromanska, and Y. LeCun, "Deep learning with Elastic Averaging SGD," *Advances in Neural Information Processing Systems*, pp. 685–693, 2015.

[2] H. Kim, J. Park, J. Jang, and S. Yoon, "DeepSpark: Spark-Based Deep Learning Supporting Asynchronous Updates and Caffe Compatibility," *arXiv preprint*, vol. arXiv:1602.08191, 2016.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation*, 2012, pp. 15–28.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. of. the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[5] 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," in *Proceedings of LearningSys*, 2015.

[6] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "SparkNet: Training Deep Networks in Spark," in *Proc. of the Intl. Conference on Learning Representations*, 2016.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 265–283, 2016.

[8] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Mao, M. Razato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1223–1231.

[9] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer, "FireCaffe: near-linear acceleration of deep neural network training on compute clusters," *arXiv preprint*, vol. arXiv:1511.00175, 2015.

[10] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 98–109, 2011.

[11] Y. Wang, X. Zhang, I. Wong, J. Dai, Y. Zhang, and others, "BigDL Programming Guide," Mar. 2017. [Online]. Available: https://github.com/intel-analytics/BigDL/wiki/Programming-Guide

[12] A. Feng, J. Shi, and M. Jain, "CaffeOnSpark Open Sourced for Distributed Deep Learning on Big Data Clusters," Feb. 2016. [Online]. Available: http://yahoohadoop.tumblr.com/post/139916563586/caffeonspark-open-sourced-for-distributed-deep

[13] M. Blot, D. Picard, M. Cord, and N. Thome, "Gossip training for deep learning," *arXiv preprint*, vol. arXiv:1611.09726, 2016.

[14] H. R. Feyzmahdavian, A. Aytekin, and M. Johansson, "An Asynchronous Mini-Batch Algorithm for Regularized Stochastic Optimization," *arXiv preprint*, vol. arXiv:1505.04824, 2015.

[15] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient Mini-Batch Training for Stochastic Optimization," in *Proc. of the 20th ACM Intl. Conference on Knowledge Discovery and Data Mining*, 2014, pp. 661–670.

[16] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization," *Advances in Neural Information Processing Systems*, vol. 28, pp. 2737–2745, 2015.

[17] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint*, vol. arXiv:1409.1556, 2014.

[18] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *Proc. of IEEE Intl. Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 8595–8598.

[19] 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 *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 583–598.

[20] M. Li, D. G. Andersen, A. Smola, and K. Yu, "Communication Efficient Distributed Machine Learning with the Parameter Server," in *Advances in Neural Information Processing Systems*, 2014, pp. 19–27.

[21] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," in *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 571–582.

[22] L. Lamport, "Paxos Made Simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[23] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A New Platform for Distributed Machine Learning on Big Data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proc. of the ACM Intl. Conference on Multimedia*, 2014, pp. 675–678.

[25] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized Gossip Algorithms," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2508–2530, 2006.

[26] B. Cohen, "The BitTorrent protocol specification," vol. f61df4010379f5c40090a9b77b73e57db7045dee, 2008.

[27] D. P. Kingma and J. L. Ba, "Adam: A Method for Stochastic Optimization," in *Proc. of the Intl. Conference on Learning Representations*, 2015.

[28] S. Zhang, "Distributed stochastic optimization for deep learning," Ph.D. dissertation, New York University, 2016. [Online]. Available: http://www.cs.nyu.edu/media/publications/zhang_sixin.pdf

[29] I. Goodfellow, O. Vinyals, and A. Saxe, "Qualitatively Characterizing Neural Network Optimization Problems," in *Proc. of the Intl. Conference on Learning Representations*, 2015.

[30] I. Sutskever, J. Martens, G. Dahl, and G. E. Hinton, "On the importance of initialization and momentum in deep learning," *Journal of Machine Learning Research*, vol. 28, no. 2010, pp. 1139–1147, 2013.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[32] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *arXiv preprint*, vol. arXiv:1410.0759, 2014.

[33] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking State-of-the-Art Deep Learning Software Tools," *arXiv preprint*, vol. arXiv:1608.07249, 2016.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proc. of the IEEE Intl. Conference on Computer Vision*, 2015, pp. 1026–1034.

[35] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Re, and M. Zaharia, "DAWNBench: An End-to-End Deep Learning Benchmark and Competition," *NIPS ML Systems Workshop*, 2017.

[36] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *Proc. of the 34th Intl. Conference on Machine Learning*, vol. 70, 2017, pp. 3368–3376.

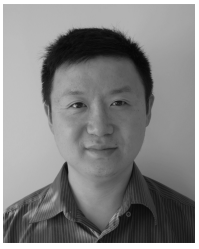
[37] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu, "Asynchronous stochastic gradient descent with delay compensation," in *Proc. of the 34th Intl. Conference on Machine Learning*, vol. 70, 2017, pp. 4120–4129.

[38] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima," *arXiv preprint*, vol. arXiv:1609.04836, 2016.

[39] N. Strom, "Scalable Distributed DNN Training Using Commodity GPU Cloud Computing," in *Proc. of the Annual Conference of the Intl. Speech Communication Association*, 2015, pp. 1488–1492.

Matthias Langer received a B.Sc. degree with honors in computer science at the Naturwissenschaftlich-Technische Akademie, Isny in 2007. Thereafter, he implemented process automation software for steel production and optimized commercial regression models. He is currently a PhD candidate at La Trobe University, Australia. His research interests include large scale deep learning, high performance and distributed computing.

Ashley Hall is a fifth-year computer science and mathematics student, and deep learning researcher at La Trobe University, Melbourne. He plans to continue his education at La Trobe with an honors in computer science, specializing in the field of deep learning. His primary research interests are deep convolutional neural networks for object localization and detection in image and video data.



Zhen He is an associate professor in the department of computer science at La Trobe University. He leads a research group focused on applying deep learning to both image and text domains. His interests include distributed deep learning, human pose estimation, video activity recognition, using deep learning for dialogue systems, and application of deep learning to medical imagining data.

Wenny Rahayu is a professor in computer science and the Head of School of Engineering and Mathematical Sciences at La Trobe University, Australia. The main focus of her research includes heterogeneous data integration, and mobile and distributed databases. In the last 10 years, she has published two authored books, three edited books and 200+ research papers in international journals and conference proceedings, with more than 4000 total citations.