DeepSpark: Spark-Based Deep Learning Supporting Asynchronous Updates and Caffe Compatibility

Hanjoo Kim, Jaehong Park, Jaehee Jang, and Sungroh Yoon
Electrical Engineering and Computer Science, Seoul National University, Seoul 08826, Korea
sryoon@snu.ac.kr

ABSTRACT
The increasing complexity of deep neural networks (DNNs) has made it challenging to exploit existing large-scale data processing pipelines for handling massive data and parameters involved in DNN training. Distributed computing platforms and GPGPU-based acceleration provide a mainstream solution to this computational challenge. In this paper, we propose DeepSpark, a distributed and parallel deep learning framework that simultaneously exploits Apache Spark for large-scale distributed data management and Caffe for GPU-based acceleration. DeepSpark directly accepts Caffe input specifications, providing seamless compatibility with existing designs and network structures. To support parallel operations, DeepSpark automatically distributes workloads and parameters to Caffe-running nodes using Spark and iteratively aggregates training results by a novel lock-free asynchronous variant of the popular elastic averaging stochastic gradient descent update scheme, effectively complementing the synchronized processing capabilities of Spark. DeepSpark is an ongoing project, and the current release is available at http://deepspark.snu.ac.kr.

CCS Concepts
• Networks → Cloud computing; • Theory of computation → Distributed computing models; • Computing methodologies → Neural networks;

Keywords
distributed computing; deep learning; asynchronous SGD

1. INTRODUCTION
Deep neural networks (DNNs) continue to push the boundaries of their application territories. For instance, convolutional neural networks (CNNs) have become the de facto standard method for image/object recognition in computer vision [1]. Other types of DNNs have also shown outstanding performance in various machine learning problems including speech recognition [2] and image classification [13].

DNNs deliver a sophisticated modeling capability underpinned by multiple hidden layers (e.g., a recent CNN model called ResNet consists of over 150 layers [14]), which effectively provide intermediate representations of the original input data. Leveraged by this condition, DNNs can better handle complications in machine learning problems than previous techniques. Although having multiple hidden layers allows DNNs to have powerful non-linear modeling capability, training such DNNs generally requires a large volume of data and a huge amount of computation resources for training. This leads to the training time ranging from several hours to days even with general purpose general purpose graphics processing unit (GPGPU) based acceleration [13, 14].

Various approaches have been proposed to improve the efficiency of deep learning training. Highly optimized GPGPU implementations have significantly shortened the time spent in training DNNs, often showing dozens of times speedup [6, 7]. However, acceleration on a single machine has limitations because of the limited resources such as the host machine’s GPU memory or main memory [6]. Scaling out methods in distributed environments have been suggested [3, 10, 11, 12] to overcome the issues of training DNNs on a single machine. Those examples exploit data parallelism and/or model parallelism and can potentially provide scalability on deep learning training.

Seamless integration of DNN training with existing data processing pipelines is also an important practical point. Many real-world datasets used for DNN training (such as raw images or speech signals) need to be converted into a trainable format on deep learning platforms and often require preprocessing to improve robustness [6, 13]. The data for training DNNs are typically huge in scale, thus the preprocessing procedure demand a considerable amount of time and resources and require carefully designed software to process. SparkNet [14] was motivated by this condition and combined deep learning algorithms with existing data analytic pipelines on Apache Spark [15]. Given that Spark was not originally devised for DNN training, it lacks useful (and often inevitable) techniques and optimizations. For instance, Spark underperforms on the tasks that require updating shared parameters in an asynchronous manner, which is the general scheme of distributed deep learning systems [16].

We propose DeepSpark, a new deep learning framework on Spark to address the issues encountered in large-scale
data handling and accelerate DNN training. Specifically, our contributions include the following:

1. **Seamless integration of scalable data management capability with deep learning**: We implemented our deep learning framework interface and the parameter exchanger on Apache Spark, which provides a straightforward but effective data parallelism layer.

   (a) **Enhancing both training and communication using asynchrony**: We implemented an asynchronous stochastic gradient descent (SGD) method for better DNN training in Spark. We implemented an adaptive variant of the elastic averaging SGD (EASGD), which accelerated parameter updates and improved the overall convergence rate.

   (b) **Adopting the handy and qualified Caffe engine**: To integrate the widely used Caffe framework [17] into Spark, we designed a novel Java Caffe wrapper. Using this interfacing component, users can continue to use their existing Caffe models without making numerous adjustments to integrate DeepSpark with their analysis pipeline.

2. **Experimental evaluations that demonstrate the effectiveness of our system scaling and asynchrony for expediting DNN training**: We tested DeepSpark with popular benchmarks including CIFAR-10 [18] and ILSVRC 2012 ImageNet [19] datasets and observed consistent speedups over existing scale-up and scale-out approaches.

3. **Availability**: The proposed DeepSpark library is freely available at http://deepspark.snu.ac.kr.

2. RELATED WORK

2.1 Deep Learning Training

Training DNN is composed of two steps, namely, feed-forward and backpropagation [20]. Feed-forward produces output using previous activations and hidden parameters. The error compared with the target value is then computed at the output classifier and is backpropagated to the previous layer through the entire network. Optimization is executed during the backpropagation step in order to better approximate target distribution. This process is iterated until the end of training. Although the complex neural network model successfully approximates input data distribution, it inherently leads to a large amount of parameters to learn.

\[
\mathcal{L}(w) = \frac{1}{N} \sum_{i=1}^{N} J_w(x^i) + \lambda \text{Reg}(w)
\]

This situation demands a huge amount of time and computational resources, which are two of the major concerns in deep learning research.

2.2 Stochastic Gradient Descent

Gradient descent is the most commonly used optimization method for training DNN practically [21]. Minibatch SGD [22] is the sequential variant of the batch gradient descent in that it only takes the gradient of the randomly sampled data at each update rather than the gradient of the entire data. Given its sequential property, minibatch SGD is more suited to process large datasets compared to its batch version. In deep learning settings, the optimization objective is usually given as follows:
2.3 Distributed Deep Learning

A naïve parallelization SGD can be implemented by splitting batch calculations over multiple worker nodes [24]. The global parameters are initialized and broadcasted from master to each worker, and the workers will then derive gradients from their local data. Considering that the throughput of each node can differ, two strategies can be utilized to parallelize gradient computing: synchronized or non-synchronized.

2.3.1 Asynchronous SGD

Synchronous SGD waits for every worker node to finish its computation and reach the barrier. Once all worker nodes have completed their tasks, the master node collects all the gradients, averages them and applies them to the center parameter. Worker nodes then pull the updated parameters from the master. While synchronous SGD is the most straightforward form of parallelizing SGD, its performance is highly degraded by the slowest worker. It also suffers from network overhead while aggregating and broadcasting network parameters.

Asynchronous SGD has been suggested to resolve the inefficiency caused by the synchronous barrier locking. In the lock-free asynchronous SGD, each worker node independently communicates with the central parameter server without waiting for other nodes to finish. This situation can result in stale gradients, but asynchronous SGD has been theoretically and empirically investigated to converge faster than the SGD on a single machine [9, 26, 27].

2.3.2 Parameter Server

The notion of parameter server is a framework that aims for large-scale machine learning training and inference [8, 10]. It has a master-slave architecture, while data and tasks are distributed over workers, and server nodes manage the global parameters. Communication between nodes is performed asynchronously, and the framework provides fault tolerance and flexible consistency over independent tasks. In previous works, a distributed parameter server has been successfully used in training various machine learning algorithms, such as logistic regression and latent Dirichlet allocation on petabytes of real data.

2.3.3 Reducing Communication Overhead

The distributed training of DNNs consists of two steps: exploration and exploitation [27]. The former is to explore the parameter space to identify the optimal weight parameters and the latter is to update center parameter weights using local worker’s training results and proceed to the next step. In distributed settings, given that the parameter exchanging causes network overhead, a performance tradeoff exists between worker’s exploration and master’s exploitation. SparkNet presented the iteration hyperparameter \( \tau \), which is the number of processed minibatches before the next communication [14]. The distributed training system can benefit from the large value of \( \tau \) under the high-cost communication scenario by reducing communication overhead. However, large \( \tau \) may end up requiring more iterations to attain convergence, which slows down the learning process [14, 27].

Zhang et al. (2015) suggested the EASGD strategy to maximize the benefit of exploring [27]. In EASGD, master and local workers exchange their weight parameters similar to the original SGD methods. When updating parameters,
Algorithm 1: Pseudo-procedure of DeepSpark for \( p \) workers and master node

1: **Input:** communication period \( \tau \), iterations \( t_{\text{max}} \), learning rate \( \eta \), data \( D \) on HDFS
2:   
3: **// Data preprocessing and partitioning**
4:   \[ RIDD = \text{Load\_From\_HDFS}(D) \]
5:   \[ RDP\_\text{processed} = \text{Preprocess}(RIDD) \]
6:   \[ \text{Partition}_D = \text{Repartition}(RDP\_\text{processed}, k) \]
7:   
8: **// Initialize for the \( a \) worker node \( k \)**
9:   \[ D_k = \text{spill\_To\_Local}(\text{Partition}_D), \]
10:   
11: **// a variant of asynchronous EASGD update**
12:   parameter weights \( x_k = x_{\text{master}} \)
13:   for \( i = 0 \) to \( t_{\text{max}} \) do
14:     \[ x_k = x_k - \eta g(x_k^{(i)}; D_k) \]
15:     for every \( \tau \) iteration do
16:       \[ \alpha_i = \text{modify\_Moving\_Rate}(\alpha_0, i) \]
17:       \[ \text{EASGD\_Update}(x_k, x_{\text{master}}, \alpha_i) \]
18:     end for
19:   end for

however, they compute the elastic difference between them and apply the difference on both master and worker weight parameters. To compute the elastic force, moving rate \( \alpha \) is involved. At every communication, each worker and the master node update their parameter weights as follows:

\[
\begin{align*}
    w_{\text{worker}} &= w_{\text{worker}} - \alpha(w_{\text{worker}} - w_{\text{master}}) \\
    w_{\text{master}} &= w_{\text{master}} + \alpha(w_{\text{worker}} - w_{\text{master}})
\end{align*}
\] (4)

This method is different from downpour SGD [9], where gradients of local workers are shipped to the master and updated center parameters are sent back to workers at every update. EASGD shows faster convergence of the training even with large value of \( \tau \), with which downpour SGD shows slow convergence rate or even cannot converge [27].

2.4 Apache Hadoop and Spark

Apache Hadoop YARN [29] and Spark [15] are cluster frameworks that allow large-scale data processing on commodity hardware. In Hadoop, dataset would be split into multiple blocks in Hadoop Distributed File System (HDFS) [30]. HDFS provides the overall control of these blocks and maintains fault tolerance. Hadoop YARN, which is the framework for resource management and job monitoring, is responsible for containers in parallel.

Spark is the cluster computing engine running on YARN, Apache Mesos or EC2. The core of Spark is the in-memory distributed processing using resilient distributed dataset (RDD). RDD is the read-only collection of data partitioned across a set of machines. On RDD, parallel actions are performed to produce actual outcome or transformations can be applied to convert a certain RDD into other type of RDD. For further reuse, RDD can be cached in cluster memory, which prevents unnecessary storage I/O and thus accelerates data processing. Hadoop and Spark are originally designed for batch-synchronized analysis on large data. They are, however, less suited for tasks that requires asynchronous actions on parallel workers [10].

3. **PROPOSED DEEPSPIARK FRAMEWORK**

3.1 Motivations

Apache Spark is an attractive platform for data-processing pipelines such as a database query processing. However, Spark RDD provides limited asynchronous operations between the master and the workers.

To addressing the disadvantages of Spark, we implemented a new asynchronous SGD solver with a custom parameter exchanger on the Spark environment. To improve asynchronous performance, we noticeably improved the EASGD algorithm [27] by considering adaptive parameter updates, thereby delivering faster convergence.

3.2 Structure Overview

DeepSpark consists of three main parts, namely, Apache Spark, a parameter exchanger for asynchronous SGD, and the Caffe software, as shown in Figure 1. Apache Spark manages workers and available resources assigned by a resource manager.

Figure 2 depicts how the Spark workflow progresses for asynchronous SGD, and we provide more detailed descriptions in Section 3.3. Subsequently, we explain the parameter exchanger and asynchronous SGD process exploiting Spark in Sections 3.4–6 and Figure 3. We clarify how to integrate Caffe as our SGD computing engine with Spark using Java Native Access (JNA) interface in Section 3.7. The overall procedure of DeepSpark is summarized in Algorithm 1. We assumed that DeepSpark runs on Hadoop YARN [29] and HDFS [30] in the following explanation of our workflow.

3.3 Distributed Setup for Spark

In this section, we explain DeepSpark’s distributed workflow from the data preparation to asynchronous SGD, which is corresponding to lines 3–9 in Algorithm 1 and from load to spilling phase in Figure 3. Given that DeepSpark is running on top of the Spark framework, it needs to load and transform the raw data in the form of Spark RDD [10].

The first step of DeepSpark training is to create RDD for training and inference. We defined a container class, which stores label information and corresponding data for each data sample. The data specific loader then creates the RDD of this data container class, which is followed by the preprocessing phase. In the preprocessing phase, data containers would be connected to the preprocessing pipeline such as filtering, mapping, transforming, or shuffling. The processed RDD repartitioning is then performed to match the number of partitions to the number of worker executors.

Caffe, the actual computing engine, however, cannot directly access RDD. In DeepSpark, the entire dataset is distributed across all workers, and each worker can cache or convert its own parts of dataset into LMDB file format if the data are relatively larger than the memory size. For a relatively small dataset, the RDD foreachPartition action is executed, where every data partition is loaded in local worker’s memory as a form of Java List. These data then become available by Caffe’s neural network model using the memory data layer in Caffe library. In this case, we should set the batch size, the number of channels, the image width and height based on the Caffe memory data layer specifica-

1. [https://github.com/java-native-access/jna](https://github.com/java-native-access/jna)
2. [http://lmdb.readthedocs.org/en/release/](http://lmdb.readthedocs.org/en/release/)
The other approach to feed the data into Caffe is spilling the dataset on a worker node’s storage. For a large dataset that is difficult to hold in the physical memory, the RDD `foreach` operation is performed, and each data partition is converted to LMDB file format and stored in the temporary local repository of the node it belongs to. Once the LMDB files are created, Caffe automatically computes data dimension and finally completes the neural network model parameter. We used LMDB JNI to manipulate LMDB on Spark.

### 3.4 Asynchronous EASGD Operation

Inherently, Spark does not support step-wise asynchronous operations for asynchronous SGD updates. We adopt the method that exploits Spark RDD operations in overcoming the limitation of Spark. The dummy RDD represented in Figure 2 can mimic the asynchrony.

Once the LMDB local repository for each worker has been prepared, the dummy RDD is created and distributed across every workers. These dummy data have an important role to launch the distributed action (i.e., parallel model training is performed). Although the explicit dependency between spilling and training steps is nonexistent at the code level, each worker node would be guided to launch the training process with spilled dataset by the dummy RDD. This exploits the property of Spark that the Spark scheduler reuses the pre-existing worker node session. The size of the dummy RDD is explicitly set to the number of workers for full parallel operation, and the `foreachPartition` action is executed on this dummy RDD. Inside the `foreachPartition` process, each worker can use the local data repository that has been created in the previous job and starts the training step.

During the training process, the Spark driver program serves as a central parameter exchanger, which performs asynchronous EASGD update. At the initial step, The driver program broadcasts its network address and neural network setup files to all workers. Each worker then creates their own model and starts training using broadcasted data. The hyper parameters for learning neural network are basically identical to those of Caffe. These parameters include learning rate, learning rate policy, momentum, and the number of max iterations.

### 3.5 Adaptive Update Rules

In this section, we explain the two variants of EASGD update rules for fast convergence. For the model parameters in the master node, the moving rate $\alpha$ acts as the learning rate for the sequential SGD process. As the adaptive learning rate policy [31] in the single node SGD process, we expect to improve the converge rate of training result during the training process by adjusting the moving rate adaptively.

In the DeepSpark framework, we also explored the effect of modifying $\alpha$ during training. Equation 5 shows our implementation of decay $\alpha$ with power rate $\rho$ when the update

---

**Figure 4:** Test accuracy versus training time (sec) on CIFAR10 dataset.

**Figure 5:** Speedup versus a single Caffe to achieve the target test accuracy on CIFAR10 dataset. The target accuracy was from 0.71 to 0.75.
count $t$ arrives at each step size $s$.

$$\alpha_t = \alpha \rho^{|t/s|}$$  \hspace{1cm} (5)

As the training proceeds, the elastic force between the master and the workers gradually decreases. We expected that this helps to converge the training with more stability. Optionally, stopping decay $\alpha$ is possible by setting the decay count limit.

Furthermore, we study the effect of the asymmetric moving rate between the workers and the master. This is to ensure that the master’s training convergence is not interfered with the worker’s ill-posed exploration after a specific step. In our implementation, we first start with the same moving rate for both master and workers. The moving rate for the master gradually decreases at the left iterations. Equation (5) shows the asymmetric updates between the master and the workers. The decaying moving rate for the master follows the same process as Equation (5)

$$w_{\text{worker}} = w_{\text{worker}} - \alpha (w_{\text{worker}} - w_{\text{master}})$$
$$w_{\text{master}} = w_{\text{master}} + \alpha (w_{\text{worker}} - w_{\text{master}})$$  \hspace{1cm} (6)

### 3.6 Parameter Exchanger

The parameter exchanger is the DeepSpark implementation of parameter server concepts, which is essential for asynchronous update. In DeepSpark, the application driver node serves as the parameter exchanger to enable worker nodes to update their weights asynchronously. Figure 3 shows the outline of the learning cycle with the parameter exchanger. The driver node starts the parameter exchanger as a separate thread before worker nodes begin training the model.

When multiple weight exchange requests from worker nodes exist, a thread pool is implemented to handle the requests at the same time. For each connection request, the thread pool allocates the pre-created threads that process the weight exchange requests. The size of the thread pool is fixed in the program and we set this up to eight threads because of limited memory and network bandwidth. If the number of requests exceeds the size, the unallocated requests wait in a queue until the preceding requests are completed as shown in Figure 3(a).

Exchange threads asynchronously access to and update the neural net model in the parameter exchanger based on the EASGD algorithm. In Figure 3(b), given that it is a lock-free system, weights can be overwritten by simultaneous updates. Nevertheless, training results accumulate successfully as proven in [32]. After the parameter exchange action, each worker returns to the SGD phase to explore the

![Figure 6: Test accuracy versus a wall-clock time (sec) for different environments.](image-url)

The parameter exchanger is the DeepSpark implementation of parameter server concepts, which is essential for asynchronous update. In DeepSpark, the application driver node serves as the parameter exchanger to enable worker nodes to update their weights asynchronously. Figure 3 shows the outline of the learning cycle with the parameter exchanger. The driver node starts the parameter exchanger as a separate thread before worker nodes begin training the model.

When multiple weight exchange requests from worker nodes exist, a thread pool is implemented to handle the requests at the same time. For each connection request, the thread pool allocates the pre-created threads that process the weight exchange requests. The size of the thread pool is fixed in the program and we set this up to eight threads because of limited memory and network bandwidth. If the number of requests exceeds the size, the unallocated requests wait in a queue until the preceding requests are completed as shown in Figure 3(a).

Exchange threads asynchronously access to and update the neural net model in the parameter exchanger based on the EASGD algorithm. In Figure 3(b), given that it is a lock-free system, weights can be overwritten by simultaneous updates. Nevertheless, training results accumulate successfully as proven in [32]. After the parameter exchange action, each worker returns to the SGD phase to explore the
Figure 7: ImageNet result. GoogLeNet’s three output and test loss are represented. In the bar graph, $O_N^k$ represents the $N$th output with top-$k$ classification. The speedup was measured by comparing our result with the highest accuracy obtained by Caffe running on a single machine.

Figure 8: CIFAR10 training result versus training time (sec) with adaptive moving rate $\alpha$.

3.7 SGD Engine

Each worker node in DeepSpark uses the Caffe library as the GPU accelerated SGD engine. However, Spark application is written in Java, Scala and Python, which cannot use the native Caffe library directly in the source code level. We implemented our code with JNA so that Spark executors can reference the Caffe native library.

To parallelize the Caffe models, the Caffe model parameters should be accessible for reading and writing. The original SGD solver class of Caffe does not provide an interface for that. Thus, we derived a custom solver class from the SGD-Solver. We defined some operations of the derived solver class to perform an atomic iteration action, acquire current trained parameter weights, and modify parameter weights. This custom solver class provides an interface to control the Caffe library for the DeepSpark application. Therefore, current Caffe models can be used in a distributed environment without changing the Caffe network specifications numerous times. Line 16 in Algorithm 1 is corresponding to the SGD engine operation.

4. EXPERIMENTS

4.1 Experimental Setup

We prepared a single-machine environment and a distributed cluster environment. The distributed cluster shown in Fig-
In this section, we set the learning rate $\alpha = 0$ and weight decay as 0.004 and maximum iterations up to 20,000.

To observe the advantages of parallelization in terms of data scalability, we used two datasets: CIFAR10 and ImageNet.

### 4.2 Dataset and Neural Network Models

To observe the advantages of parallelization in terms of data scalability, we used two datasets: CIFAR10 and ImageNet.

#### 4.2.1 CIFAR10

The CIFAR10 dataset contains 60,000 images with $32 \times 32$ size, 3 channels, and 10 classes. It consists of 50,000 images for training and 10,000 images for test. For the CIFAR10 dataset, we trained a CNN model that is a variant of Cuda-convnet. This model consists of three sets, including a convolution layer (5 × 5, stride 1), a pooling layer, and a rectified linear unit activation layer, followed by a fully connected layer and a softmax layer to calculate output (128 feature maps). This model achieved ~75% test accuracy.

#### 4.2.2 ImageNet

The original ILSVRC 2012 ImageNet dataset consists of 1,281,167 color images with 1,000 classes of different image concepts. To reduce the experiment time, we composed a training set from ImageNet that includes 383,690 images of 300 classes and a validation set with 5,000 images that were not used in training. As pre-processing, we unified the size of images ($256 \times 256$), converted them into a readable format for Apache Spark and saved them on HDFS.

To train the ImageNet dataset, we used a Caffe model that is a replication of Google’s GoogLeNet. GoogLeNet is a 22-layer deep network made with the Hebbian principle and multi-scale processing intuition with nine of its inception modules for classification. The replica GoogLeNet Caffe model, however, has several differences: it does not train with the rectified and scale/aspect-ratio data augmentation, and its weight initialization method is Xavier instead of Gaussian.

### 4.3 Experimental Results

#### 4.3.1 Training on CIFAR10

We examined the time that a single Caffe machine and the distributed DeepSpark and SparkNet cluster took to reach target accuracies (.72, .73, .74, .75) for different numbers of nodes and communication period $\tau$. For all experiments in this section, we set the learning rate $\eta = 0.001$, momentum $\delta = 0.9$, weight decay as 0.004 and maximum iterations up to 20,000.

We first compared the DeepSpark cluster with Caffe and the results are shown in Table 1. Based on these results, we confirmed approximately 66% of the experiment cases converged faster than Caffe, excluding the ones that were not able to achieve the target accuracy within maximum iterations. The best speedup case for the .75 test accuracy took 28% less time than Caffe, when eight nodes existed and $\tau = 40$. Figure 4 illustrates test accuracy versus wall-clock time on Caffe (single), DeepSpark and SparkNet clusters. DeepSpark showed a similar converging tendency to Caffe, which was faster than SparkNet with 25% higher accuracies in average. We could not determine whether SparkNet was working properly for large $\tau$ ($\tau \geq 40$).

#### 4.3.2 Training on ImageNet

We observed the training tendencies of two different methods on ImageNet: a single Caffe machine and the distributed DeepSpark. All the experiments were performed in the same configuration, which includes 100,000 maximum iterations, learning rate $\eta = 0.05$, weight decay 0.002, and moving rate $\alpha = 0.1$. Additionally, the configuration included eight nodes and communication period $\tau = 50$ for the distributed experiment. Figure 7 shows the accuracies of three outputs versus training time. DeepSpark converged faster and more accurately than Caffe on a single node. Among the three outputs, output 3 showed the highest accuracy. For the maximum achievements of single node Caffe within the same wall-clock time, DeepSpark showed speed up by 1.62. Finally, DeepSpark attained higher test accuracy by 13% on average than Caffe on a single node at full iterations.

#### 4.3.3 Training with Different Moving Rates

For all experiments in this section, we applied some changes on the moving rate with the same experiment setup for CIFAR10 as the best speedup case of the .75 test accuracy from 4.3.1 that includes eight nodes and communication period $\tau = 40$. We used different fixed moving rates from the

| Acc | $\tau$ = 5 | $\tau$ = 10 | $\tau$ = 20 | $\tau$ = 40 | $\tau$ = 100 |
|-----|-----------|-----------|-----------|-----------|-----------|
| 4 nodes | 1.06 | 1.05 | 1.15 | 1.71 | 1.09 |
| 8 nodes | 1.09 | 0.95 | 1.55 | 1.60 | 1.42 |
| 16 nodes | 1.42 | 1.13 | 0.50 | 1.13 | 1.90 |
| 24 nodes | 1.90 | 1.37 | 1.18 | 0.44 | 0.97 |

| Acc | $\tau$ = 75 | $\tau$ = 10 | $\tau$ = 20 | $\tau$ = 40 | $\tau$ = 100 |
|-----|-----------|-----------|-----------|-----------|-----------|
| 4 nodes | - | - | 1.26 | 1.25 | 0.96 |
| 8 nodes | 0.96 | - | 1.38 | 1.37 | 1.21 |
| 16 nodes | 1.21 | 1.02 | - | 0.96 | 1.19 |
| 24 nodes | 1.19 | 1.34 | 0.77 | - | 1.68 |

Table 1: Speedup of DeepSpark over Caffe (single node) on CIFAR10. The target accuracy ranged from .71 to .75. The blank represents the experiments that failed to reach target accuracy within 20,000 iterations.
5. DISCUSSION

5.1 Speedup Analysis

DeepSpark achieved high speedup compared with the sequential Caffe and the distributed SparkNet by alleviating communication overhead and reducing disk I/O.

Communication overhead played a crucial role in the slowdown. According to SparkNet [14], speedup of parallelization versus a single sequential machine is given as follows:

\[ \text{SpeedUp} = \frac{N_a(b)C(b)}{(\tau C(b) + S)M_a(\tau, b, K)} \]  

where \( N_a(b) \) and \( M_a(\tau, b, K) \) are the required number of iterations to achieve the accuracy level \( \alpha \) for sequential and parallel, \( b \) is the size of the minibatch, \( C(b) \) is time for a batch computation, \( S \) is the communication overhead, and \( K \) represents the number of nodes.

We attempted to relax the communication overhead \( S \) by asynchronous update and succeeded in accelerating DNN training. The time DeepSpark spent to aggregate and broadcast was less than 1/10 of the time SparkNet spent. Taking CIFAR10 training on DeepSpark (8 nodes, \( \tau = 40 \)) for example, the time spent in parameter exchange is \( 2 \times 5 \times \alpha \) of the time spent in minibatch training. The parameter exchanging period can develop as the number of nodes increases, and this affects the overall training time. We observed this condition in Table 1 where the speedup for target accuracy .75 with \( \tau = 40 \) decreased as the number of nodes increased from 8 to 16.

In addition, the adaptive EASGD update rules further relieved the share of \( S \), while suppressing the increase in \( M_a(\tau, b, K) \) [27], and we confirmed this condition from the results in Figure 8.

For the dataset that is too large to hold in memory, the distributed environment was helpful in reducing the disk operation load. The disk I/O overhead transforms Equation 7 into the following equation for a large dataset:

\[ \text{SpeedUp} = \frac{N_a(b)C(b) + L(|D|)M_a(\tau, b, K) + L(|D|)/K)}{(\tau C(b) + S) + L(|D|)/K} \]  

where \( L(|D|) \) implies the disk I/O overhead for the size of the entire dataset \( D \). In the distributed system, the dataset is divided into \( K \) shards, and the shared data can cause the reducing of disk I/O. In the ImageNet result, the disk I/O overhead was estimated approximately an hour for 32 batch size and 100,000 iterations in a single node. The overhead decreased by 42% for eight nodes.

5.2 Moving Rate

Higher test accuracy was acquired from higher moving rate setting for CIFAR10 as shown in Figure 4. By contrast, the models with low moving rate tended to converge more smoothly. As shown in Figure 8, adaptive adjusting moving rate \( \alpha \) was effective to improve convergence. Meanwhile, the result for the ImageNet dataset was different from that of training on CIFAR10. Although we did not present the ImageNet dataset training result for \( \alpha = 0.9 \), it was inferior to that of the lower \( \alpha \).

If we set the initial moving rate \( \alpha_0 \) high, the model parameters of the master node can escape from the local optima near the initial model parameter. As the training progresses, however, the master node should wobble less to converge, and lower elasticity between the master and worker nodes represented by decay \( \alpha \) can be helpful.

Asymmetrically adapted moving rate also showed positive effects on convergence but with less stability. Figure 8 shows that asymmetric \( \alpha \) may cause degradation of the convergence.

5.3 Data Spilling

To implement asynchronous EASGD, the scheduling that Spark provides should be sacrificed. Originally, Spark operates data in memory, which leverages performance up to 100 times faster for certain applications [34] and if the data size is bigger than the memory limit, Spark stores lingering data in disk and schedules operations. To run asynchronous operations that are not suitable for Spark, we mimicked the spilling process of register allocation and spilled data into the local storage.

Time delay caused by disk operations however, did not affect the overall performance of DeepSpark. On eight worker nodes, approximately a few seconds and 8 minutes were spent to spill CIFAR10 and ImageNet dataset, respectively. These delays accounted for only below 2% of the entire running time, which were negligible.

5.4 Further Improvements

DeepSpark can be further improved by reducing the network overhead. Our analysis assumes that the weight parameter serialization during network communication could have caused additional network overhead. In our experiments, we used the basic serialization method on Java for convenience and optimized object creation speed. The basic object serialization process in Java is known to be inferior to many other advanced libraries [35]. By replacing the serialization procedure with more sophisticated libraries, we expect the reduction in object de/serialization time and the size of serialized data. Furthermore, changing network protocols may also be helpful for additional performance boosts. For convenience and reliability, our asynchronous update communication structure was based on TCP sockets. By replacing the current communication structure by the UDP datagram communication [26], additional speedups may become possible. Although the reliability of the UDP datagram is inherently not guaranteed, the use of local (usually reliable) networks and the robustness of the asynchronous EASGD may make it irrelevant to DNN training.
6. CONCLUSION

We have described our new deep learning framework named DeepSpark, which provides seamless integration with existing large-scale data processing pipelines as well as highly accelerated DNN training procedure. DeepSpark is an example of a successful combination of diverse components including Apache Spark, asynchronous parameter updates, and GPGPU-based Caffe framework. Based on our experiments with popular benchmarks, DeepSpark demonstrated its effectiveness by showing faster convergence than the alternative parallelization schemes.

References

[1] K. He, et al. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385, 2015.
[2] J. K. Chorowski, et al. Attention-based models for speech recognition. In NIPS, pages 577–585, 2015.
[3] K. Simonyan et al. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.
[4] C. Szegedy, et al. Going deeper with convolutions. In CVPR, pages 1–9, 2015.
[5] A. Krizhevsky, et al. Imagenet classification with deep convolutional neural networks. In NIPS, pages 1097–1105, 2012.
[6] S. Chetlur, et al. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759, 2014.
[7] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997, 2014.
[8] M. Li, et al. Scaling distributed machine learning with the parameter server. In OSDI, pages 583–598, 2014.
[9] J. Dean, et al. Large scale distributed deep networks. In NIPS, pages 1223–1231, 2012.
[10] Q. Ho, et al. More effective distributed ml via a stale synchronous parallel parameter server. In NIPS, pages 1223–1231, 2013.
[11] E. P. Xing, et al. Petuum: A new platform for distributed machine learning on big data. In SIGKDD, KDD ’15, pages 1335–1344, New York, NY, USA, 2015. ACM.
[12] B. C. Ooi, et al. Singa: A distributed deep learning platform. In Proceedings of the ACM International Conference on Multimedia, pages 685–688. ACM, 2015.
[13] J. T. Geiger, et al. Investigating nnmf speech enhancement for neural network based acoustic models. In INTERSPEECH, pages 2405–2409, 2014.
[14] P. Moritz, et al. Sparknet: Training deep networks in spark. arXiv preprint arXiv:1511.06051, 2015.
[15] M. Zaharia, et al. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, volume 10, page 10, 2010.
[16] M. Zaharia, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, pages 2–2. USENIX Association, 2012.
[17] Y. Jia, et al. Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the ACM International Conference on Multimedia, pages 675–678. ACM, 2014.
[18] A. Krizhevsky et al. Learning multiple layers of features from tiny images, 2009.
[19] O. Russakovsky, et al. ImageNet Large Scale Visual Recognition Challenge. IJCV, 115(3):211–252, 2015.
[20] R. Hecht-Nielsen. Theory of the backpropagation neural network. In IJCNN, pages 593–605. IEEE, 1989.
[21] Y. LeCun, et al. Deep learning. Nature, 521(7553):436–444, 2015.
[22] Y. Anzai. Pattern Recognition & Machine Learning. Elsevier, 2012.
[23] Y. N. Dauphin, et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In NIPS, pages 2933–2941, 2014.
[24] C. H. Teo, et al. A scalable modular convex solver for regularized risk minimization. In SIGKDD, pages 727–736. ACM, 2007.
[25] M. Zinkevich, et al. Parallelized stochastic gradient descent. In NIPS, pages 2595–2603, 2010.
[26] X. Lian, et al. Asynchronous parallel stochastic gradient decent for nonconvex optimization. In NIPS, pages 2719–2727, 2015.
[27] S. Zhang, et al. Deep learning with elastic averaging sgd. In NIPS, pages 685–693, 2015.
[28] D. M. Blei, et al. Latent dirichlet allocation. the Journal of machine Learning research, 3:993–1022, 2003.
[29] V. K. Vavilapalli, et al. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th Annual Conference on Cloud Computing, page 5. ACM, 2013.
[30] K. Shvachko, et al. The hadoop distributed file system. In MSST, pages 1–10. IEEE, 2010.
[31] R. A. Jacobs. Increased rates of convergence through learning rate adaptation. Neural networks, 1(4):295–307, 1988.
[32] B. Recht, et al. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In NIPS, pages 693–701, 2011.
[33] X. Glorot et al. Understanding the difficulty of training deep feedforward neural networks. In AISTATS, pages 249–256, 2010.
[34] R. S. Xin, et al. Shark: Sql and rich analytics at scale. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of data, pages 13–24. ACM, 2013.

[35] A. Sumaray et al. A comparison of data serialization formats for optimal efficiency on a mobile platform. In IMCOM, page 48. ACM, 2012.

[36] J. Kay et al. Profiling and reducing processing overheads in tcp/ip. IEEE/ACM TON, 4(6):817–828, 1996.