SPARKNET: TRAINING DEEP NETWORKS IN SPARK

Philipp Moritz[*], **Robert Nishihara**[*], **Ion Stoica**, **Michael I. Jordan**
Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720, USA
{pcmoritz,rkn,istoica,jordan}@eecs.berkeley.edu

ABSTRACT

Training deep networks is a time-consuming process, with networks for object recognition often requiring multiple days to train. For this reason, leveraging the resources of a cluster to speed up training is an important area of work. However, widely-popular batch-processing computational frameworks like MapReduce and Spark were not designed to support the asynchronous and communication-intensive workloads of existing distributed deep learning systems. We introduce SparkNet, a framework for training deep networks in Spark. Our implementation includes a convenient interface for reading data from Spark RDDs, a Scala interface to the Caffe deep learning framework, and a lightweight multi-dimensional tensor library. Using a simple parallelization scheme for stochastic gradient descent, SparkNet scales well with the cluster size and tolerates very high-latency communication. Furthermore, it is easy to deploy and use with no parameter tuning, and it is compatible with existing Caffe models. We quantify the dependence of the speedup obtained by SparkNet on the number of machines, the communication frequency, and the cluster's communication overhead, and we benchmark our system's performance on the ImageNet dataset.

1 INTRODUCTION

Deep learning has advanced the state of the art in a number of application domains. Many of the recent advances involve fitting large models (often several hundreds megabytes) to larger datasets (often hundreds of gigabytes). Given the scale of these optimization problems, training can be time-consuming, often requiring multiple days on a single GPU using stochastic gradient descent (SGD). For this reason, much effort has been devoted to leveraging the computational resources of a cluster to speed up the training of deep networks (and more generally to perform distributed optimization).

Many attempts to speed up the training of deep networks rely on asynchronous, lock-free optimization (Dean et al., 2012; Chilimbi et al., 2014). This paradigm uses the parameter server model (Li et al., 2014; Ho et al., 2013), in which one or more master nodes hold the latest model parameters in memory and serve them to worker nodes upon request. The nodes then compute gradients with respect to these parameters on a minibatch drawn from the local data shard. These gradients are shipped back to the server, which updates the model parameters.

At the same time, batch-processing frameworks enjoy widespread usage and have been gaining in popularity. Beginning with MapReduce (Dean & Ghemawat, 2008), a number of frameworks for distributed computing have emerged to make it easier to write distributed programs that leverage the resources of a cluster (Zaharia et al., 2010; Isard et al., 2007; Murray et al., 2013). These frameworks have greatly simplified many large-scale data analytics tasks. However, state-of-the-art deep learning systems rely on custom implementations to facilitate their asynchronous, communication-intensive workloads. One reason is that popular batch-processing frameworks (Dean & Ghemawat, 2008; Zaharia et al., 2010) are not designed to support the workloads of existing deep learning systems. SparkNet implements a scalable, distributed algorithm for training deep networks that lends itself to batch computational frameworks such as MapReduce and Spark and works well out-of-the-box in bandwidth-limited environments.

[*] Both authors contributed equally.

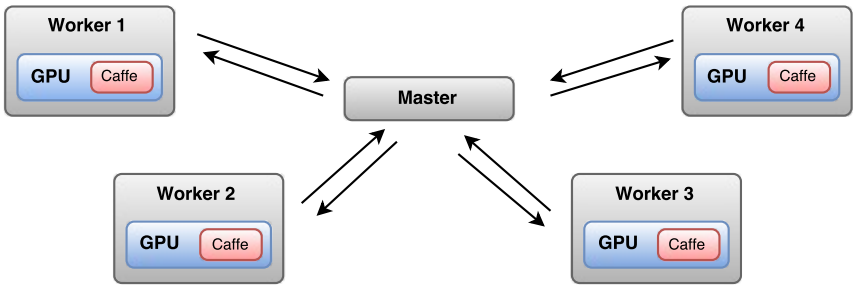


Figure 1: This figure depicts the SparkNet architecture.

The benefits of integrating model training with existing batch frameworks are numerous. Much of the difficulty of applying machine learning has to do with obtaining, cleaning, and processing data as well as deploying models and serving predictions. For this reason, it is convenient to integrate model training with the existing data-processing pipelines that have been engineered in today's distributed computational environments. Furthermore, this approach allows data to be kept in memory from start to finish, whereas a segmented approach requires writing to disk between operations. If a user wishes to train a deep network on the output of a SQL query or on the output of a graph computation and to feed the resulting predictions into a distributed visualization tool, this can be done conveniently within a single computational framework.

We emphasize that the hardware requirements of our approach are minimal. Whereas many approaches to the distributed training of deep networks involve heavy communication (often communicating multiple gradient vectors for every minibatch), our approach gracefully handles the bandwidth-limited setting while also taking advantage of clusters with low-latency communication. For this reason, we can easily deploy our algorithm on clusters that are not optimized for communication. Our implementation works well out-of-the box on a five-node EC2 cluster in which broadcasting and collecting model parameters (several hundred megabytes per worker) takes on the order of 20 seconds, and performing a single minibatch gradient computation requires about 0.5 seconds (for AlexNet). We achieve this by providing a simple algorithm for parallelizing SGD that involves minimal communication and lends itself to straightforward implementation in batch computational frameworks. Our goal is not to outperform custom computational frameworks but rather to propose a system that can be easily implemented in popular batch frameworks and that performs nearly as well as what can be accomplished with specialized frameworks.

2 IMPLEMENTATION

Here we describe our implementation of SparkNet. SparkNet builds on Apache Spark (Zaharia et al., 2010) and the Caffe deep learning library (Jia et al., 2014). In addition, we use Java Native Access

```
class Net {
  def Net(netParams: NetParams): Net
  def setTrainingData(data: Iterator[(NDArray,Int)])
  def setValidationData(data: Iterator[(NDArray,Int)])
  def train(numSteps: Int)
  def test(numSteps: Int): Float
  def setWeights(weights: WeightCollection)
  def getWeights(): WeightCollection
}
```

Listing 1: SparkNet API

```
val netParams = NetParams(
  RDDLayer("data", shape=List(batchsize, 1, 28, 28)),
  RDDLayer("label", shape=List(batchsize, 1)),
  ConvLayer("conv1", List("data"), kernel=(5,5), numFilters=20),
  PoolLayer("pool1", List("conv1"), pool=Max, kernel=(2,2), stride=(2,2)),
  ConvLayer("conv2", List("pool1"), kernel=(5,5), numFilters=50),
  PoolLayer("pool2", List("conv2"), pool=Max, kernel=(2,2), stride=(2,2)),
  LinearLayer("ip1", List("pool2"), numOutputs=500),
  ActivationLayer("relu1", List("ip1"), activation=ReLU),
  LinearLayer("ip2", List("relu1"), numOutputs=10),
  SoftmaxWithLoss("loss", List("ip2", "label"))
)
```

Listing 2: Example network specification in SparkNet

```
var trainData = loadData(...)
var trainData = preprocess(trainData).cache()
var nets = trainData.foreachPartition(data => {
  var net = Net(netParams)
  net.setTrainingData(data)
  net)
var weights = initialWeights(...)
for (i <- 1 to 1000) {
  var broadcastWeights = broadcast(weights)
  nets.map(net => net.setWeights(broadcastWeights.value))
  weights = nets.map(net => {
    net.train(50)
    net.getWeights()}).mean() // an average of WeightCollection objects
}
```

Listing 3: Distributed training example

for accessing Caffe data and weights natively from Scala, and we use the ScalaBuff implementation of Google Protocol Buffers to allow the dynamic construction of Caffe networks at runtime.

The Net class wraps Caffe and exposes a simple API containing the methods shown in Listing 1. The `NetParams` type specifies a network architecture, and the `WeightCollection` type is a map from layer names to lists of weights. It allows the manipulation of network components and the storage of weights and outputs for individual layers. To facilitate manipulation of data and weights without copying memory from Caffe, we implement the `NDArray` class, which is a lightweight multi-dimensional tensor library. One benefit of building on Caffe is that any existing Caffe model definition or solver file is automatically compatible with SparkNet. There is a large community developing Caffe models and extensions, and these can easily be used in SparkNet. By building on top of Spark, we inherit the advantages of modern batch computational frameworks. These include the high-throughput loading and preprocessing of data and the ability to keep data in memory between operations. In Listing 2, we give an example of how network architectures can be specified in SparkNet. In addition, model specifications or weights can be loaded directly from Caffe files. An example sketch of code that uses our API to perform distributed training is given in Listing 3.

2.1 PARALLELIZING SGD

To perform well in bandwidth-limited environments, we recommend a parallelization scheme for SGD that requires minimal communication. This approach is not specific to SGD. Indeed, SparkNet works out of the box with any Caffe solver.

The parallelization scheme is described in Listing 3. Spark consists of a single master node and a number of worker nodes. The data is split among the Spark workers. In every iteration, the Spark master broadcasts the model parameters to each worker. Each worker then runs SGD on the model with its subset of data for a fixed number of iterations τ (we use $\tau = 50$ in Listing 3) or for a fixed length of time, after which the resulting model parameters on each worker are sent to the master and averaged to form the new model parameters. We recommend initializing the network by running SGD for a small number of iterations on the master. In our experiments in Section 3.2, we use 500 training iterations, which takes about 17 minutes.

The standard approach to parallelizing each gradient computation requires broadcasting and collecting model parameters (hundreds of megabytes per worker and gigabytes in total) after every SGD update, which occurs tens of thousands of times during training. On our EC2 cluster, each broadcast and collection takes about twenty seconds, putting a bound on the speedup that can be expected using this approach without better hardware or without partitioning models across machines. Our approach broadcasts and collects the parameters a factor of τ times less for the same number of iterations. In our experiments, we set $\tau = 50$, but other values seem to work about as well.

3 EXPERIMENTS

In Section 3.2, we will benchmark the performance of SparkNet and measure the speedup that our system obtains relative to training on a single node. However, the outcomes of those experiments depend on a number of different factors. In addition to τ (the number of iterations between synchronizations) and K (the number of machines in our cluster), they depend on the communication overhead in our cluster S. In Section 3.1, we find it instructive to measure the speedup *in the idealized case of zero communication overhead* ($S = 0$). This idealized model gives us an upper bound on the maximum speedup that we could hope to obtain in a real-world cluster, and it allows us to build a model for the speedup as a function of S (the overhead is easily measured in practice).

3.1 THEORETICAL CONSIDERATIONS

Before benchmarking our system, we determine the maximum possible speedup that could be obtained in principle in a cluster with no communication overhead. We determine the dependence of this speedup on the parameters τ (the number of iterations between synchronizations) and K (the number of machines in our cluster).

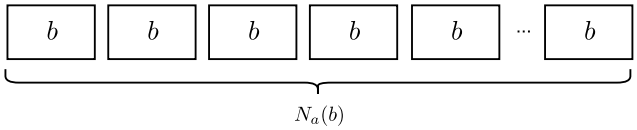
3.1.1 LIMITATIONS OF NAIVE PARALLELIZATION

To begin with, we consider the theoretical limitations of a naive parallelism scheme which parallelizes SGD by distributing each minibatch computation over multiple machines (see Figure 2b). Let $N_a(b)$ be the number of serial iterations of SGD required to obtain an accuracy of a when training with a batch size of b (when we say accuracy, we are referring to test accuracy). Suppose that computing the gradient over a batch of size b requires $C(b)$ units of time. Then the running time required to achieve an accuracy of a with serial training is

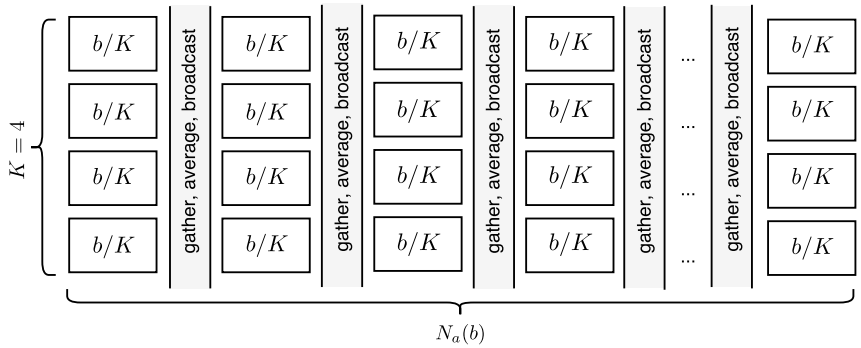
$$N_a(b)C(b). \tag{1}$$

A naive parallelization scheme attempts to distribute the computation at each iteration by dividing each minibatch between the K machines, computing the gradients separately, and aggregating the results on one node. Under this scheme, the cost of the computation done on a single node in a single iteration is $C(b/K)$ and satisfies $C(b/K) \geq C(b)/K$ (the cost is sublinear in the batch size). In a system with no communication overhead and no overhead for summing the gradients, this approach could in principle achieve an accuracy of a in time $N_a(b)C(b)/K$. This represents a linear speedup in the number of machines (for values of K up to the batch size b).

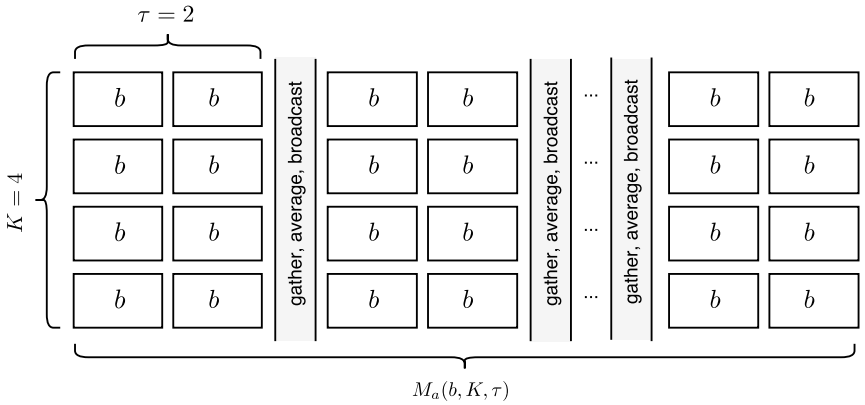
In practice, there are several important considerations. First, for the approximation $C(b/K) \approx C(b)/K$ to hold, K must be much smaller than b, limiting the number of machines we can use to effectively parallelize the minibatch computation. One might imagine circumventing this limitation by using a larger batch size b. Unfortunately, the benefit of using larger batches is relatively modest. As the batch size b increases, $N_a(b)$ does not decrease enough to justify the use of a very large value of b.



(a) This figure depicts a serial run of SGD. Each block corresponds to a single SGD update with batch size b. The quantity $N_a(b)$ is the number of iterations required to achieve an accuracy of a.



(b) This figure depicts a parallel run of SGD on $K = 4$ machines under a naive parallelization scheme. At each iteration, each batch of size b is divided among the K machines, the gradients over the subsets are computed separately on each machine, the updates are aggregated, and the new model is broadcast to the workers. Algorithmically, this approach is exactly equivalent to the serial run of SGD in Figure 2a and so the number of iterations required to achieve an accuracy of a is the same value $N_a(b)$.



(c) This figure depicts a parallel run of SGD on $K = 4$ machines under SparkNet's parallelization scheme. At each step, each machine runs SGD with batch size b for τ iterations, after which the models are aggregated, averaged, and broadcast to the workers. The quantity $M_a(b, K, \tau)$ is the number of rounds (of τ iterations) required to obtain an accuracy of a. The total number of parallel iterations of SGD under SparkNet's parallelization scheme required to obtain an accuracy of a is then $\tau M_a(b, K, \tau)$.

Figure 2: Computational models for different parallelization schemes.

Furthermore, the benefits of this approach depend greatly on the degree of communication overhead. If aggregating the gradients and broadcasting the model parameters requires S units of time, then the time required by this approach is at least $C(b)/K + S$ per iteration and $N_a(b)(C(b)/K + S)$ to achieve an accuracy of a. Therefore, the maximum achievable speedup is $C(b)/(C(b)/K + S) \leq C(b)/S$. We may expect S to increase modestly as K increases, but we suppress this effect here.

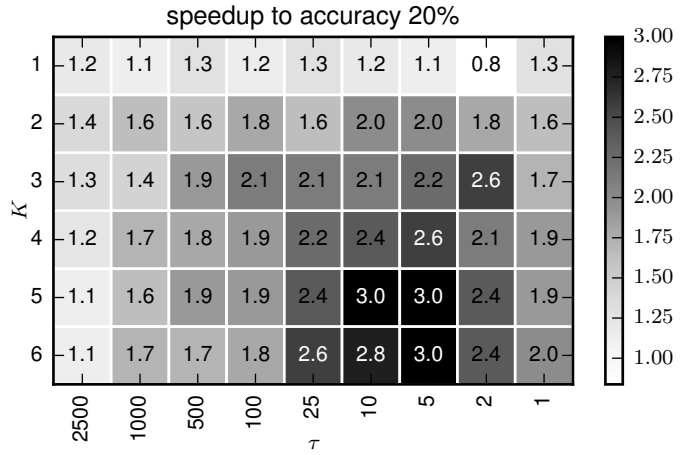


Figure 3: This figure shows the speedup $\tau M_a(b, \tau, K)/N_a(b)$ given by SparkNet's parallelization scheme relative to training on a single machine to obtain an accuracy of $a = 20\%$. Each grid square corresponds to a different choice of K and τ. We show the speedup in the zero communication overhead setting. This experiment uses a modified version of AlexNet on a subset of ImageNet (100 classes each with approximately 1000 images). Note that these numbers are dataset specific. Nevertheless, the trends they capture are of interest.

3.1.2 Limitations of SparkNet Parallelization

The performance of the naive parallelization scheme is easily understood because its behavior is equivalent to that of the serial algorithm. In contrast, SparkNet uses a parallelization scheme that is not equivalent to serial SGD (described in Section 2.1), and so its analysis is more complex.

SparkNet's parallelization scheme proceeds in rounds (see Figure 2c). In each round, each machine runs SGD for τ iterations with batch size b. Between rounds, the models on the workers are gathered together on the master, averaged, and broadcast to the workers.

We use $M_a(b, K, \tau)$ to denote the number of rounds required to achieve an accuracy of a. The number of parallel iterations of SGD under SparkNet's parallelization scheme required to achieve an accuracy of a is then $\tau M_a(b, K, \tau)$, and the wallclock time is

$$(\tau C(b) + S)M_a(b, K, \tau), \tag{2}$$

where S is the time required to gather and broadcast model parameters.

To measure the sensitivity of SparkNet's parallelization scheme to the parameters τ and K, we consider a grid of values of K and τ. For each pair of parameters, we run SparkNet using a modified version of AlexNet on a subset of ImageNet (the first 100 classes each with approximately 1000 data points) for a total of 20000 parallel iterations. For each of these training runs, we compute the ratio $\tau M_a(b, K, \tau)/N_a(b)$. This is the speedup achieved relative to training on a single machine when $S = 0$. In Figure 3, we plot a heatmap of the speedup given by the SparkNet parallelization scheme under different values of τ and K.

Figure 3 exhibits several trends. The top row of the heatmap corresponds to the case $K = 1$, where we use only one worker. Since we do not have multiple workers to synchronize when $K = 1$, the number of iterations τ between synchronizations does not matter, so all of the squares in the top row of the grid should behave similarly and should exhibit a speedup factor of 1 (up to randomness in the optimization). The rightmost column of each heatmap corresponds to the case $\tau = 1$, where we synchronize after every iteration of SGD. This is equivalent to running serial SGD with a batch size of Kb, where b is the batchsize on each worker (in these experiments we use $b = 100$). In this column, the speedup should increase sublinearly with K. We note that it is slightly surprising that the speedup does not increase monotonically from left to right as τ decreases. Intuitively, we might expect more synchronization to be strictly better (recall we are disregarding the overhead due to synchronization). However, our experiments suggest that modest delays between synchronizations can be beneficial.

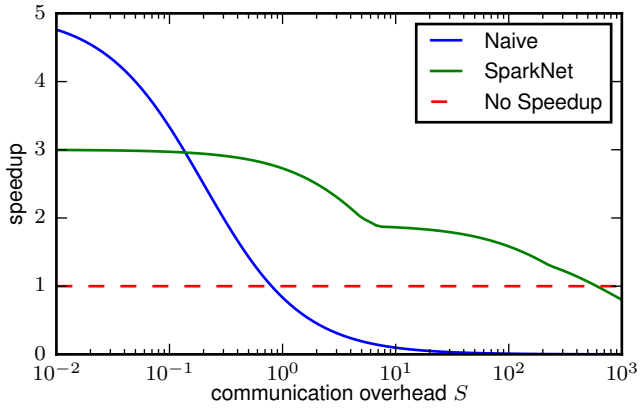


Figure 4: This figure shows the speedups obtained by the naive parallelization scheme and by SparkNet as a function of the cluster's communication overhead (normalized so that $C(b) = 1$). We consider $K = 5$. The data for this plot applies to training a modified version of AlexNet on a subset of ImageNet (approximately 1000 images for each of the first 100 classes). The speedup obtained by the naive parallelization scheme is $C(b)/(C(b)/K + S)$. The speedup obtained by SparkNet is $N_a(b)C(b)/[(\tau C(b) + S)M_a(b, K, \tau)]$ for a specific value of τ. The numerator is the time required by serial SGD to achieve an accuracy of a, and the denominator is the time required by SparkNet to achieve the same accuracy (see Equation 1 and Equation 2). For the optimal value of τ, the speedup is $\max_\tau N_a(b)C(b)/[(\tau C(b) + S)M_a(b, K, \tau)]$. To plot the SparkNet speedup curve, we maximize over the set of values $\tau \in \{1, 2, 5, 10, 25, 100, 500, 1000, 2500\}$ and use the values $M_a(b, K, \tau)$ and $N_a(b)$ from the experiments in the fifth row of Figure 3. In our experiments, we have $S \approx 20s$ and $C(b) \approx 0.5s$.

This experiment capture the speedup that we can expect from the SparkNet parallelization scheme in the case of zero communication overhead (the numbers are dataset specific, but the trends are of interest). Having measured these numbers, it is straightforward to compute the speedup that we can expect as a function of the communication overhead.

In Figure 4, we plot the speedup expected both from naive parallelization and from SparkNet on a five-node cluster as a function of S (normalized so that $C(b) = 1$). As expected, naive parallelization gives a maximum speedup of 5x (on a five-node cluster) when there is zero communication overhead (note that our plot does not go all the way to $S = 0$), and it gives no speedup when the communication overhead is comparable to or greater than the cost of a minibatch computation. In contrast, SparkNet gives a relatively consistent speedup even when the communication overhead is 100x the cost of a minibatch computation.

The speedup given by the naive parallelization scheme can be computed exactly and is given by $C(b)/(C(b)/K+S)$. This formula is essentially Amdahl's law. Note that when $S \geq C(b)$, the naive parallelization scheme is slower than the computation on a single machine. The speedup obtained by SparkNet is $N_a(b)C(b)/[(\tau C(b) + S)M_a(b, K, \tau)]$ for a specific value of τ. The numerator is the time required by serial SGD to achieve an accuracy of a from Equation 1, and the denominator is the time required by SparkNet to achieve the same accuracy from Equation 2. Choosing the optimal value of τ gives us a speedup of $\max_\tau N_a(b)C(b)/[(\tau C(b)+S)M_a(b, K, \tau)]$. In practice, choosing τ is not a difficult problem. The ratio $N_a(b)/(\tau M_a(b, K, \tau))$ (the speedup when $S = 0$) degrades slowly as τ increases, so it suffices to choose τ to be a small multiple of S (say $5S$) so that the algorithm spends only a fraction of its time in communication.

When plotting the SparkNet speedup in Figure 4, we do not maximize over all positive integer values of τ but rather over the set $\tau \in \{1, 2, 5, 10, 25, 100, 500, 1000, 2500\}$, and we use the values of $N_a(b)$ and $M_a(b, K, \tau)$ corresponding to the fifth row of Figure 3. Including more values of τ would only increase the SparkNet speedup. The distributed training of deep networks is typically thought of as a communication-intensive procedure. However, Figure 4 demonstrates the value of SparkNet's parallelization scheme even in the most bandwidth-limited settings.

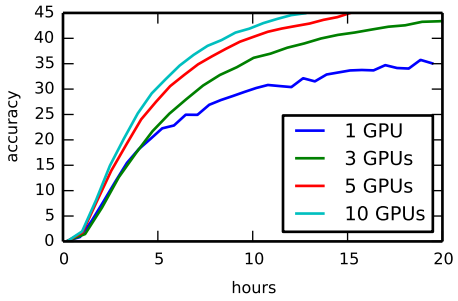


Figure 5: This shows the scaling of SparkNet with 3, 5, and 10 GPUs and $\tau = 50$. The 1 GPU plot was obtained by running Caffe with no communication, whereas the other experiments communicate parameters between machines incurring an overhead of about 20 seconds per synchronization.

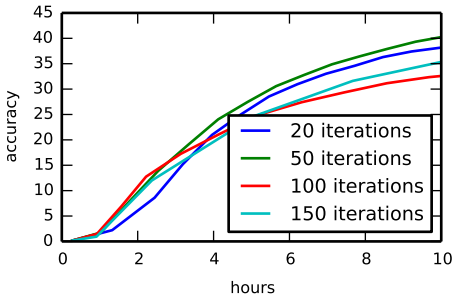


Figure 6: This figure shows the dependence of the parallelization scheme described in Section 2.1 on τ. Each experiment was run with $K = 5$ workers. This figure shows that there is no need to collect and broadcast the model more frequently than every 50 iterations in our bandwidth-limited cluster.

The naive parallelization scheme may appear to be a straw man. However, it is a frequently-used approach to parallelizing SGD (Noel et al., 2015), especially when asynchronous updates are not an option (as in computational frameworks like MapReduce and Spark).

3.2 Training Benchmarks

To explore the scaling behavior of our algorithm and implementation, we perform experiments on EC2 using a cluster of five g2.8xlarge nodes. Each node has four NVIDIA GRID GPUs and 60GB memory. We train the default Caffe model of AlexNet (Krizhevsky et al., 2012) on the ImageNet dataset (Russakovsky et al., 2015). We run SparkNet with $K = 3$, 5, and 10 and plot the results in Figure 5. For comparison, we also run Caffe on the same cluster with a single GPU and no communication overhead to obtain the $K = 1$ plot. To measure the speedup, we compare the wall-clock time required to obtain an accuracy of 45%. With 1 GPU and no communication overhead, this takes 55.6 hours. With 3, 5, and 10 GPUs, SparkNet takes 22.9, 14.5, and 12.8 hours, giving speedups of 2.4x, 3.8x, and 4.4x.

Furthermore, we explore the dependence of the parallelization scheme described in Section 2.1 on the parameter τ which determines the number of iterations of SGD that each worker does before synchronizing with the other workers. These results are shown in Figure 6. Note that in the presence of stragglers, it suffices to replace the fixed number of iterations τ with a fixed length of time, but in our experimental setup, the timing was sufficiently consistent and stragglers did not arise. The single GPU experiment in Figure 5 was trained on a single GPU node with no communication overhead.

4 Related Work

Much work has been done to build distributed frameworks for training deep networks. Coates et al. (2013) build a model-parallel system for training deep networks on a GPU cluster using MPI over Infiniband. Dean et al. (2012) build DistBelief, a distributed system capable of training deep networks on thousands of machines using stochastic and batch optimization procedures. In particular, they highlight asynchronous SGD and batch L-BFGS. Distbelief exploits both data parallelism and model parallelism. Chilimbi et al. (2014) build Project Adam, a system for training deep networks on hundreds of machines using asynchronous SGD. Li et al. (2014); Ho et al. (2013) build parameter servers to exploit model and data parallelism, and though their systems are better suited to sparse gradient updates, they could very well be applied to the distributed training of deep networks. More recently, Abadi et al. (2015) build TensorFlow, a sophisticated system for training deep networks and more generally for specifying computation graphs and performing automatic differentiation.

These custom systems have numerous advantages including high performance, fine-grained control over scheduling and task placement, and the ability to take advantage of low-latency communication between machines. On the other hand, due to their nature as custom systems, they lack the benefits of tight integration with general-purpose computational frameworks such as Spark. For some of these systems, preprocessing is done separately by a MapReduce style framework, and data is written to disk between segments of the pipeline. With SparkNet, preprocessing and training are both done in Spark.

Training a machine learning model such as a deep network is often one step of many in real-world data analytics pipelines (Sparks et al., 2015). Obtaining, cleaning, and preprocessing the data are often expensive operations, as is transferring data between systems. Training data for a machine learning model may be derived from a streaming source, from a SQL query, or from a graph computation. A user wishing to train a deep network in a custom system on the output of a SQL query would need a separate SQL engine. In SparkNet, training a deep network on the output of a SQL query, or a graph computation, or a streaming data source is straightforward due to its general purpose nature and its support for SQL, graph computations, and data streams (Armbrust et al., 2015; Gonzalez et al., 2014; Zaharia et al., 2013).

Some attempts have been made to train deep networks in general-purpose computational frameworks, however, existing work typically hinges on extremely low-latency intra-cluster communication. Noel et al. (2015) train deep networks in Spark on top of YARN using SGD and leverage cluster resources to parallelize the computation of the gradient over each minibatch. To achieve competitive performance, they use remote direct memory accesses over Infiniband to exchange model parameters quickly between GPUs. In contrast, SparkNet tolerates low-bandwidth intra-cluster communication and works out of the box on Amazon EC2.

A separate line of work addresses speeding up the training of deep networks using single-machine parallelism. For example, Caffe con Troll (Abuzaid et al., 2015) modifies Caffe to leverage both CPU and GPU resources within a single node. These approaches are compatible with SparkNet and the two can be used in conjunction.

Many popular computational frameworks provide support for training machine learning models (Meng et al., 2015) such as linear models and matrix factorization models. However, due to the demanding communication requirements and the larger scale of many deep learning problems, these libraries have not been extended to include deep networks.

Various authors have studied the theory of averaging separate runs of SGD. In the bandwidth-limited setting, Zinkevich et al. (2010) analyze a simple algorithm for convex optimization that is easily implemented in the MapReduce framework and can tolerate high-latency communication between machines. Zhang & Jordan (2015) propose a scheme for parallelizing stochastic optimization algorithms along with a Spark implementation.

5 Discussion

We have described an approach to distributing the training of deep networks in communication-limited environments that lends itself to an implementation in batch computational frameworks like MapReduce and Spark. We provide SparkNet, an easy-to-use deep learning implementation for Spark that is based on Caffe and enables the easy parallelization of existing Caffe models with minimal modification. As machine learning increasingly depends on larger and larger datasets, integration with a fast and general engine for big data processing such as Spark allows researchers and practitioners to draw from a rich ecosystem of tools to develop and deploy their models. They can build models that incorporate features from a variety of data sources like images on a distributed file system, results from a SQL query or graph database query, or streaming data sources.

Using a smaller version of the ImageNet benchmark we quantify the speedup achieved by SparkNet as a function of the size of the cluster, the communication frequency, and the cluster's communication overhead. We demonstrate that our approach is effective even in highly bandwidth-limited settings. On the full ImageNet benchmark we showed that our system achieves a sizable speedup over a single node experiment even with few GPUs.

The code for SparkNet is available at `https://github.com/amplab/SparkNet`. We invite contributions and hope that the project will help bring a diverse set of deep learning applications to the Spark community.

ACKNOWLEDGMENTS

We would like to thank Tomer Kaftan, Evan Sparks, and Shivaram Venkataraman for valuable advice. This research is supported in part by NSF grant number DGE-1106400. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Blue Goji, Bosch, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Fujitsu, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

REFERENCES

Abadi, Martín, Agarwal, Ashish, Barham, Paul, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Abuzaid, Firas, Hadjis, Stefan, Zhang, Ce, and Ré, Christopher. Caffe con Troll: Shallow ideas to speed up deep learning. *arXiv preprint arXiv:1504.04343*, 2015.

Armbrust, Michael, Xin, Reynold S, Lian, Cheng, Huai, Yin, Liu, Davies, Bradley, Joseph K, Meng, Xiangrui, Kaftan, Tomer, Franklin, Michael J, Ghodsi, Ali, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394. ACM, 2015.

Chilimbi, Trishul, Suzue, Yutaka, Apacible, Johnson, and Kalyanaraman, Karthik. Project Adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pp. 571–582, 2014.

Coates, Adam, Huval, Brody, Wang, Tao, Wu, David, Catanzaro, Bryan, and Andrew, Ng. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 1337–1345, 2013.

Dean, Jeffrey and Ghemawat, Sanjay. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Dean, Jeffrey, Corrado, Greg, Monga, Rajat, Chen, Kai, Devin, Matthieu, Mao, Mark, Ranzato, Marc'Aurelio, Senior, Andrew, Tucker, Paul, Yang, Ke, Le, Quoc V., and Ng, Andrew Y. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

Gonzalez, Joseph E, Xin, Reynold S, Dave, Ankur, Crankshaw, Daniel, Franklin, Michael J, and Stoica, Ion. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of OSDI*, pp. 599–613, 2014.

Ho, Qirong, Cipar, James, Cui, Henggang, Lee, Seunghak, Kim, Jin Kyu, Gibbons, Phillip B, Gibson, Garth A, Ganger, Greg, and Xing, Eric P. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2013.

Isard, Michael, Budiu, Mihai, Yu, Yuan, Birrell, Andrew, and Fetterly, Dennis. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pp. 59–72, 2007.

Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, and Darrell, Trevor. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pp. 675–678. ACM, 2014.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

Li, Mu, Andersen, David G, Park, Jun Woo, Smola, Alexander J, Ahmed, Amr, Josifovski, Vanja, Long, James, Shekita, Eugene J, and Su, Bor-Yiing. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pp. 583–598, 2014.

Meng, Xiangrui, Bradley, Joseph, Yavuz, Burak, Sparks, Evan, Venkataraman, Shivaram, Liu, Davies, Freeman, Jeremy, Tsai, DB, Amde, Manish, Owen, Sean, et al. MLlib: Machine learning in Apache Spark. *arXiv preprint arXiv:1505.06807*, 2015.

Murray, Derek G, McSherry, Frank, Isaacs, Rebecca, Isard, Michael, Barham, Paul, and Abadi, Martín. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455. ACM, 2013.

Noel, Cyprien, Shi, Jun, and Feng, Andy. Large scale distributed deep learning on Hadoop clusters, 2015. URL `http://yahoohadoop.tumblr.com/post/129872361846/large-scale-distributed-deep-learning-on-hadoop`.

Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., and Fei-Fei, Li. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, pp. 1–42, 2015.

Sparks, Evan R., Venkataraman, Shivaram, Kaftan, Tomer, Franklin, Michael, and Recht, Benjamin. KeystoneML: End-to-end machine learning pipelines at scale. 2015.

Zaharia, Matei, Chowdhury, Mosharaf, Franklin, Michael J, Shenker, Scott, and Stoica, Ion. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, pp. 10, 2010.

Zaharia, Matei, Das, Tathagata, Li, Haoyuan, Hunter, Timothy, Shenker, Scott, and Stoica, Ion. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 423–438. ACM, 2013.

Zhang, Yuchen and Jordan, Michael I. Splash: User-friendly programming interface for parallelizing stochastic algorithms. *arXiv preprint arXiv:1506.07552*, 2015.

Zinkevich, Martin, Weimer, Markus, Li, Lihong, and Smola, Alex J. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pp. 2595–2603, 2010.