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
{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 an extremely simple parallelization scheme for stochastic gradient descent, SparkNet scales well with the cluster size and tolerates high-latency communication. Furthermore, it is easy to deploy and use with no parameter tuning, and it is compatible with existing Caffe models. We report results on the ImageNet Large Scale Visual Recognition Challenge.

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 (often asynchronous SGD) [3, 1]. This paradigm uses a parameter server [9, 5], which holds the latest model parameters in memory and serves them to the workers 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 [2], a number of frameworks for distributed computing have emerged to make it easier to write distributed programs that leverage the resources of a cluster [14, 6, 11]. 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 [2, 14] 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. An extended version of this paper appears in Moritz et al. [10].

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

Listing 1: **SparkNet API**

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.

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 broadcasts model parameters only once every minute or so. 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. 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 implementations 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.

Some work has been done to train deep networks in batch-processing frameworks [4, 3, 12]. However, these approaches are based on parallelizing the gradient computation over individual minibatches (or over the full dataset) and are thus extremely communication intensive. Furthermore, the benefit of increasing the size of the minibatch in SGD diminishes rapidly with the minibatch size.

In the bandwidth-limited setting, Zinkevich et al. [16] analyze a simple algorithm for convex optimization that is easily implemented in the MapReduce framework and can tolerate high-latency communication between machines. Zhang and Jordan [15] propose a scheme for parallelizing stochastic optimization algorithms along with a Spark implementation.

2 Implementation

Here we describe our implementation of SparkNet. SparkNet builds on Apache Spark [14] and the Caffe deep learning library [7]. In addition, we use Java Native Access for accessing Caffe data and weights natively from Scala and the ScalaBuff implementation of Google Protocol Buffers to allow 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 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 multidimensional tensor library. We describe the `NDArray` API in Listing 4. 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

```
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.mapPartitions(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(broadcastParams.value))
  weights = nets.map(net => {
    net.step(50)
    net.getWeights()}).mean() // a mean of WeightCollection objects
}
```

Listing 3: **Distributed training example**

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

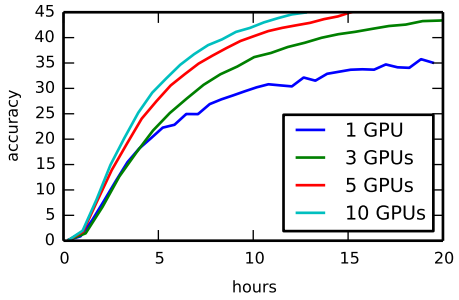
To perform well in bandwidth-limited environments, we recommend a parallelization scheme for SGD that requires minimal communication. This approach is not specific to SGD. Indeed, SparkNet works out of the box with any Caffe solver.

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, we use 500 training iterations, which takes about 17 minutes.

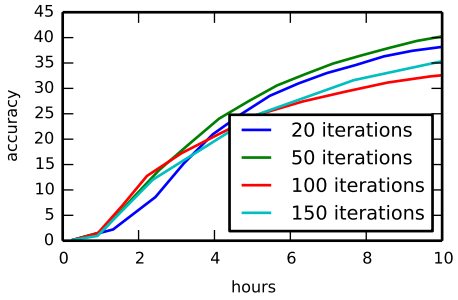
The standard approach to parallelizing each gradient computation requires broadcasting and collecting model parameters (hundreds of megabytes per worker and gigabytes in total) after every SGD

```
class NDArray {
  def NDArray(data: Array[Float], shape: Array[Int]): NDArray
  def subArray(offsets: Array[Int], shape: Array[Int]): NDArray
  def slice(dim: Int, index: Int): NDArray
  def set(indices: Array[Int], value: Float)
  def get(indices: Array[Int]): Float
}
```

Listing 4: **NDArray API**



(a) 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.



(b) This figure shows the dependence of the parallelization scheme described in Section 2.1 on τ. Each experiment was run with $N = 5$ 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.

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

To explore the scaling behavior of our algorithm and implementation, we train the default Caffe model of AlexNet [8] on the ImageNet Large Scale Visual Recognition Challenge [13]. We run SparkNet with $N = 3$, 5, and 10 GPUs and plot the results in Figure 1a. For comparison, we also run Caffe on the same cluster with a single GPU and no communication overhead to obtain the $N = 1$ 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 1b. 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. All experiments were run on EC2 using a cluster of five g2.8xlarge nodes. The single GPU experiment in Figure 1a was trained on a single GPU node with no communication overhead.

References

[1] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 571–582, 2014.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[4] P. Farber and K. Asanovic. Parallel neural network training on Multi-Spert. In *Algorithms and Architectures for Parallel Processing, 1997. ICAPP 97., 1997 3rd International Conference on*, pages 659–666. IEEE, 1997.

[5] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2013.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, 2007.

[7] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[9] 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 *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 583–598, 2014.

[10] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. SparkNet: Training deep networks in Spark. *arXiv preprint arXiv:1511.06051*, 2015.

[11] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[12] C. Noel, J. Shi, and A. Feng. Large scale distributed deep learning on Hadoop clusters, 2015. URL `http://yahoohadoop.tumblr.com/post/129872361846/large-scale-distributed-deep-learning-on-hadoop`.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, pages 1–42, 2015.

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[15] Y. Zhang and M. I. Jordan. Splash: User-friendly programming interface for parallelizing stochastic algorithms. *arXiv preprint arXiv:1506.07552*, 2015.

[16] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 2595–2603, 2010.