Decentralized and Distributed Machine Learning Model Training with Actors

Travis Addair
Stanford University
taddair@stanford.edu

Abstract

Training a machine learning model with terabytes to petabytes of data using very deep neural networks doesn't scale well within a single machine. A significant amount of work in recent years has gone into distributing the training of such neural networks across a cluster of machines, by partitioning on both the data and the model itself. The most well-established form of distributed training uses a centralized parameter server to manage the shared state of neural network weights used across all partitions of the data, but this introduces a bottleneck and single-point of failure during training. In this paper, we explore a more experimental form of decentralized training that removes this bottleneck. Finally, we show that by taking advantage of sparse updates to the shared parameter matrix, decentralized training can be tuned to make tradeoffs between training speed and model accuracy.

1. Introduction

Ideally, we would train our machine learning models using as many machines as we have available, but the question then arises: how do we partition the work? There are two general approaches to distributed machine learning: model parallelism and data parallelism. Most research today has gone into data parallelism, as keeping data locality to a subpartition of the network is more efficient than sending every datum to every node.

Naively, we can implement a synchronous system that relies on a shared parameter server that takes in updated parameters from workers and averages them [1]. We can gain higher throughput by relaxing the synchronization constraint; instead of having workers send parameters, we have them send gradients (parameter updates), but this can result in what's known as the stale gradient problem, where by the time an executor finishes updating its parameters, the parameters may have been updated many times by other workers, and their gradient is no longer accurate. Gupta et al. [2] have shown that the average staleness in a naive implementation is proportional to the number of executor nodes.

Google implemented a distributed framework for training neural networks called DistBelief [3], which used a technique called Downpour Stochastic Gradient Descent. DistBelief relied on a shared parameter server that took in updates to the model parameters asynchronously, and did not originally account for the stale gradient problem. More recent implementations ensure that parameters are updated at the end of each minibatch of training ("soft" synchronization [4]).

More recent work [5] in this area has focused on decentralized parameter updating, where instead of having a shared parameter server, nodes pass updates to every other data shard in the cluster. This results in slower convergence at the beginning of training, but much higher throughput.

In this paper we explore both a centralized and decentralized implementation of distributed training. We compare the performance of each method using two metrics: (1) number of parameter updates communicated between nodes, and (2) model accuracy / error on the training dataset. Though centralized training is shown to give the best model accuracy / least error, it also has the greatest communication overhead. Decentralized training can be tuned to greatly reduce the network overhead at the cost of introducing more error in the model.

2. Methods

As mentioned previously, there are two varieties of parallelism when training machine learning models:

data parallelism and model parallelism[1]. With data parallelism, the training dataset is split into separate data shards, and every node in the system has a replica of the model to train using its subset of the data. Using model parallelism, the actual parameters of the network are trained on separate machines. For example, in a neural network, different layers may be trained on separate nodes.

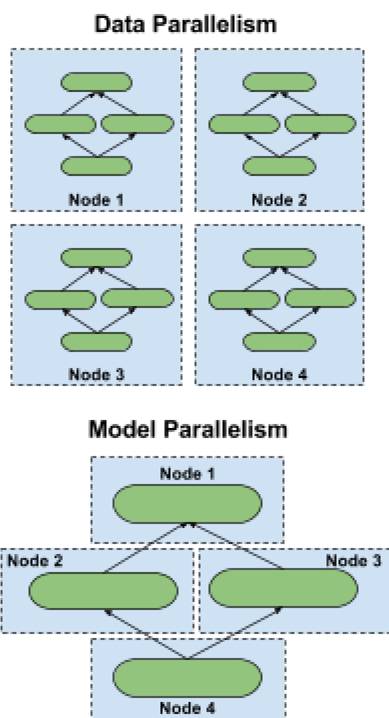

Figure 1. Data parallelism trains the same model on different nodes, where each node is given a subset of the data to train on. Model parallelism trains different partitions of the model on different nodes. Only data parallelism requires managing a shared parameter matrix state.

Our implementations — centralized and decentralized — make use of both data and model parallelism. Specifically, the data is partitioned across different *data shards* that manage training on their own subset of the data. Within each *data shard* we split the model into multiple *layer shards*. Each *layer shard* handles some subset of the total model parameters. In the centralized implementation, the shared parameters (weights) are managed by a central parameter server, that is itself partitioned such that each *parameter shard* holds the weights for a different layer in the model. The *layer shards* then communicate their parameter updates to the *parameter shard* the corresponds to their layer in the model. In contrast, the decentralized implementation removes parameter server and instead has *layer shards* distribute their updates to one another.

2.1. Asynchronous Stochastic Gradient Descent

In a synchronous distributed model training system, the updates that are sent from *layer shards* to update the global state of the model parameters are the parameters themselves. Parameters are updated transactionally, and if any shard attempts to write stale parameters, its write is rejected until it re-reads the new parameter state.

To allow for asynchronous parameter updates, instead of sending individual parameters from the *layer shards*, we instead send gradients, or deltas, giving us parameter updates of the form:

$$W_{i+1} = W_i - \lambda \sum_{j=1}^{N} \Delta W_{i,j}$$

Here, the update being sent from the *layer shard* is the gradient $\Delta W_{i,j}$, and λ is our learning rate, but scaled to account for updates across multiple machines.

2.2. Centralized Training

In centralized training, every *layer shard* writes its gradients to the shared *parameter shard* for its subset of the parameters as backpropagation is performed during the local model training process. At the end of each minibatch of training, the *layer shard* will then read the latest parameters from the *parameter shard* to update to ensure it's up to date for training the next minibatch.

1 https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-of-neural-networks/

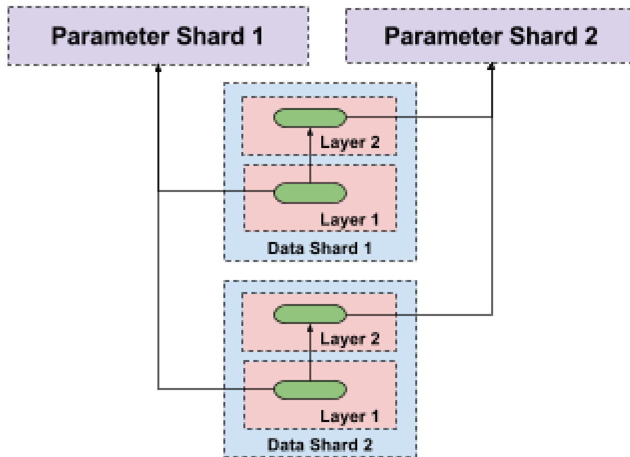

Figure 2. Centralized training. Different partitions of the data communicate their parameter updates asynchronously to a sharded parameter server.

Having a single parameter server act as the source of truth for the parameters within a layer acts as bottleneck during training. All requests go through a single point of failure, so if the parameter server goes down, training is stalled until it can recover.

Additionally, because reads are made infrequently by *layer shards* once for each minibatch, but writes are made frequently for each training example, every read requires sending the entire dense set of parameters from the *parameter shard* to the *layer shard*. Consequently, our ability to compress the response from the parameter server is limited.

2.3. Decentralized Training

In decentralized training, we omit the parameter server and instead have *data shards* communicate updates to every other *data shard* in the network for each training example. The individual *layer shards* send their gradients to their parent *data shard*, which in turn sends a message to every other *data shard* informing it of the new parameter updates, which are applied immediately, as opposed to at the end of the current minibatch.

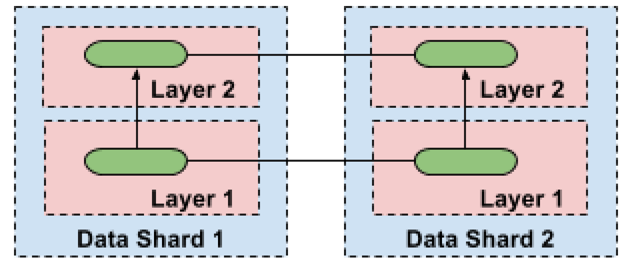

Figure 2. Decentralized training. Updates are delivered directly from one data partition to the other.

Naively, this would significantly increase the amount of network traffic in the system, as we're communicating updates to every *data shard* for every training example. However, we employ an optimization as described by Strom [5] that allows us to significantly reduce this network traffic overhead at the cost of some model accuracy.

We set a threshold parameter τ that tunes the frequency with which updates are sent out to all other *data shards* in the system. As gradients are computed locally, we accumulate them within a local *gradient residual*, and at each update step, we examine every element of the dense *gradient residual* to determine if we wish to emit it to other *data shards*. If any of the parameters in the *gradient residual* have magnitude greater than or equal to τ, we add the parameter to the message and zero out its value in the *gradient residual*.

Using τ in this way serves two purposes. First, it makes our updates sparse, such that instead of transmitting the entire set of parameters at each update step, we only send the parameters that have changed significantly over time. Second, it makes our updates less frequent, as in cases where no parameters exceed the threshold, no message is sent. A further optimization can be made to quantize the message by only sending updates of τ or $-\tau$, as done by Strom [5]. However, due to its effect on performance and limited time available for tuning, we omitted this optimization.

2.4. Actor-Based Concurrency Model

The components described were implemented as

actors using the Akka[2] actor framework written in Scala, based off the work done by Alex Minnaar[3] in implementing Google's DistBelief framework in Akka. All concurrency is message based and asynchronous, with no shared state or synchronization primitives used to communicate between nodes. Individual actors (components) are arranged hierarchically, such that each actor is created and managed by its parent. Actor failures are treated as events to be handled by the actor's parent.

Both centralized and decentralized implementations have a top-level Master actor that manages the runtime. The Master allocates multiple CrossValidator actors to explore different hyper parameters (learning rates, values for τ). Each CrossValidator partitions the input data and creates a sequence of DataShard actors to train on their partition of the data. Each DataShard actor splits the model layers in multiple Layer actors.

In the centralized implementation, a ParameterShard is created for every layer of the model. Layer actors send their updates to the ParameterShard, which updates its global parameter state. When a minibatch completes, the Layer actor will request the latest set of parameters, which will then be sent down to the Layer in a separate message. Once the parameters have been received for all Layers, the DataShard will feed the next minibatch through the network of Layers for training.

In the decentralized implementation, Layer actors send their updates to their parent DataShard with a tag for their layer ID, which then sends the updates to the other DataShards and distributed the update to the appropriate Layer. There is no blocking on minibatches as updates are read continuously as they are received.

3. Results

Our experiments focused on comparing the performance of centralized vs decentralized training, and on understanding the tradeoff between training efficiency and model accuracy in adjusting the τ parameter.

3.1. Setup

A common network architecture and training dataset was used to evaluate each distributed training approach. Our model attempts to learn the exclusive-or (XOR) function described by the following truth table:

X1	X2	Y
0	0	0
1	0	1
0	1	1
1	1	0

The XOR function is commonly used as a "Hello World" baseline for neural networks because the function is not linearly separable (cannot be solved exactly by logistic regression, for example).

Our dataset was synthesized by creating examples for each of these four possibilities, where the input features are given by the X1 and X2 columns, and the label is given by the Y column. We then create 50,000 training examples by randomly sampling from these four possible examples. When sharding the data, we assign each *data shard* 2000 examples, giving us a total of 25 *data shards*.

The model itself consists of three layers, with two input units, two hidden units, and a single output unit (0 or 1). Because each *layer shard* manages the parameters that map from one layer to the next, this gives us a total of 2 *layer shards* per *data shard*, or 50 *layer shards* in total. In the centralized implementation, this gives us 2 *parameter shards*. We train the network in minibatches, but only perform a single epoch, which can limit our ability to converge to the optimal parameter configuration in some cases.

Finally, we use a CrossValidator actor sitting on

[2] https://akka.io/

[3] http://alexminnaar.com/implementing-the-distbelief-deep-neural-network-training-framework-with-akka.html

top of our training procedure to select the best model based on the one that best minimizes the error. We use three different learning rates, giving us three times the number of actors in the system in total.

3.2. Centralized vs Decentralized

In comparing the performance of centralized vs decentralized training, we use the mean squared error of the model for our model accuracy metric, and the number of parameter updates received by each *layer shard* to measure our training speed. Here, the number of updates received is determined by both the total number of parameter update messages and by the size of each message.

Training	Error	Updates
Centralized	0.04	18009
Decentralized (τ =0.1)	0.25	28866
Decentralized (τ =1)	0.29	769

The importance of selecting the right value of τ for the problem is made clear in the table above. The centralized training method clearly outperforms decentralized training on a single epoch over the training data, even when τ is low, and has few updates for low values of τ. However, when τ is better tuned to the dataset and model, we see that we can significantly reduce the number of parameter updates, with only a small increase in the overall error.

3.3. Performance Tradeoffs

From the above experiments, we developed an intuition that tuning τ can give us significant training speed improvements with minimal increases in model error. For this next set of experiments, we ran the decentralized implementation of the model training process with a range of τ values from 0.1 to 2 to see this effect more concretely.

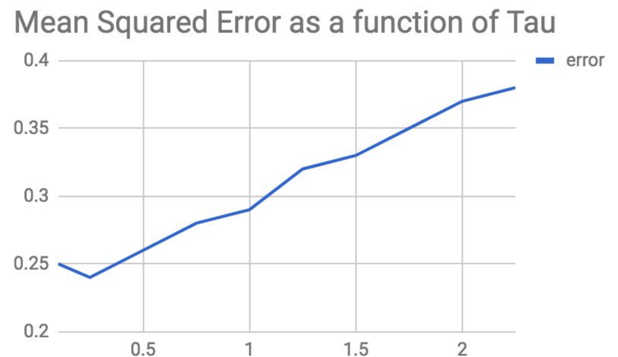

Figure 3. Increasing τ imposes a linear increase in the mean squared error.

Incremental increases in the τ parameter resulted in a linear increase in the mean squared error of the model. The conclusion to be drawn here is that there's no magic value for τ that gives a small increase in model error, that modellers will instead need to tune this value explicitly based on their tolerance for error in the model.

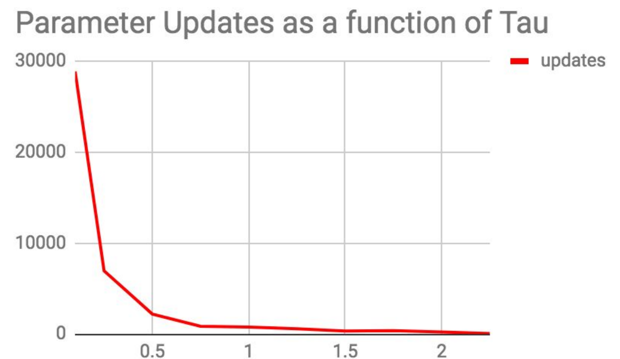

Figure 4. Increasing τ results in an exponential reduction in the number of parameter updates received by *data shards*.

In contrast, increasing τ results in an exponential reduction in the number of parameter updates received by *data shards*, which can translate to a significant reduction in training time.

4. Conclusions

In training a machine learning model as a distributed system, there is a fundamental tradeoff

between training speed and model accuracy. The most accurate results will come from a fully synchronous solution using a shared parameter server, but this bottleneck largely negates the benefits of using a highly distributed training architecture.

Asynchronous centralized training can attain similar accuracy with much higher throughput by using soft synchronization (writes to the parameter server for each example, reads for each minibatch) to solve the stale gradient problem.

Fully asynchronous and decentralized training will net the greatest overall training speed, but at a cost to model accuracy. This cost is configurable based on the setting of τ, which controls the frequency with which updates are sent to other *data shards* in the system. As shown here, the model accuracy drops significantly using decentralized training, but training speed increases at a similar degree. Using higher values of τ reduces model accuracy further, but not as significantly as the additional training speed improvements.

All code for this project can be found on GitHub[4].

References

[1] Hang Su and Haoyu Chen. Experiments on Parallel Training of Deep Neural Network using Model Averaging. arXiv preprint arXiv:1507.01239, 2015.

[2] Suyog Gupta, Wei Zhang, and Josh Milthrope. Model Accuracy and Runtime Tradeoff in Distributed Deep Learning. arXiv preprint arXiv:1509.04210, 2015.

[3] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large Scale Distributed Deep Networks. In Advances in Neural Information Processing Systems, pages 1223–1231, 2012.

[4] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-Aware Async-SGD for Distributed Deep Learning. IJCAI, 2016.

[5] Nikko Strom. Scalable Distributed DNN Training Using Commodity GPU Cloud Computing. In Sixteenth Annual Conference of the International Speech Communication Association, 2015. http://nikkostrom.com/publications/interspeech2015/strom_interspeech2015.pdf.

[4] https://github.com/tgaddair/decentralizedsgd