10-425/625: Introduction to Convex Optimization **(Fall 2023)**

Lecture 24: Online Gradient Descent / Parallel SGD

Instructor:[1] *Matt Gormley* *December 4, 2023*

24.1 Parallel SGD

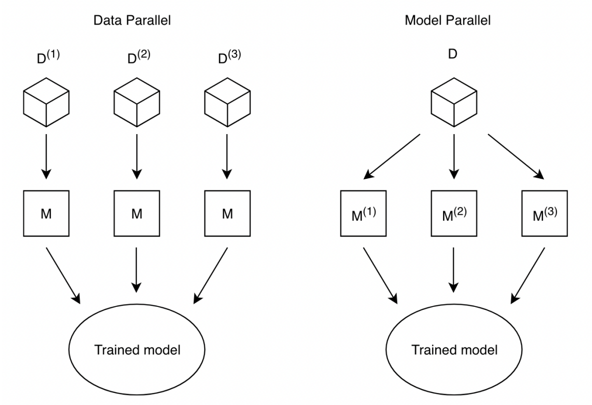
For many applications, stochastic gradient descent (SGD) is used to minimize empirical risk over a *very* large training dataset. In these cases, parallelization can aid greatly in reducing the wall clock time to convergence. However, parallelization is not without costs such as communication overhead, long wait times for straggling workers, redundant storage of training data. We begin by considering the most straightforward applications of parallelism.

Recall from a previous lecture

Data Parallelism vs. Model Parallelism Here we focus on *data parallelism* in which the data is distributed.

Modern deep learning architectures are often too large to fit on a single GPU. This setting requires *model parallelism*, in which the model is divided across GPUs.

[1]



Comparison of data and model parallelism from Verbraeken et al. (2021).

24.1.1 Distrbuted Synchonous Mini-Batch SGD

Suppose our problem is of the form $\min_x \sum_{i=1}^n f_i(x)$. If the computation cost of computing $f_i(x)$ and $\nabla f_i(x)$ is substantially more than that of sending x between machines and computing the update $x^{(t+1)} = x^{(t)} - \eta \nabla f_t(x)$, then we can achieve trivial parallelism by dividing each mini-batch across m machines.

Algorithm 1 DistributedSyncSGD(m)

```
1: Choose initial point $x^0 \in \mathbb{R}^n$
2: for $t = 1, 2, \ldots, T$ do
3:     On the head node: Send $x^{(t)}$ to each worker node $k$
4:     for $k = 1, \ldots, m$ in parallel do
5:         On worker node $k$:
6:         Sample minibatch $I_k^{(t)} \subseteq \{1, \ldots, n\}$ of size $b$
7:         Compute $g_k^{(t)} = \sum_{i \in I_k^{(t)}} \nabla f_i(x^{(t)})$
8:         Send $g_k^{(t)}$ to head node
9:     On the head node: $x^{t+1} = x^t - \eta_t \sum_{k=1}^m g_k^{(t)}$
```

Although the algorithm is exact (i.e. the parallel version and the local version do identical computation), various problems can arise.

- If one worker node is much slower than the rest, then the computation at each iteration will be bottlenecked by the speed of the slowest node.
- If the time to send gradients $g_k^{(t)}$ and iterates $x^{(t)}$ is comparable to the

gradient computation, then the base cost of gradient computation goes up substantially.

- If the dataset is too large to fit on one node, then we will have to substantially increase communication costs by sending data to nodes on demand.

24.1.2 Distributed Asynchronous SGD

A common alternative to distributed *synchronous* SGD is to allow a single parameter server to asynchronously receive *and apply* gradient updates. This comes at a great cost however: the asynchronous algorithm is no longer faithful to SGD, which is an inherently serial algorithm.

Algorithm 2 AsyncSGD-ParameterServer

```
1: Choose initial point $x^0 \in \mathbb{R}^n$
2: for $t = 1, 2, \dots, T$ do
3:     Wait to receive $g_k^{(t)}$ from the next worker
4:     Immediately apply update $x^{t+1} = x^t - \eta_t g_k^{(t)}$
```

Algorithm 3 AsyncSGD-Worker-k

```
1: for $s = 1, 2, 3, \dots$ do
2:     Request $x^{(s)}$ from parameter server
3:     Sample minibatch $I_k^{(s)} \subseteq \{1, \dots, n\}$ of size $b$
4:     Compute $g_k^{(s)} = \sum_{i \in I_k^{(s)}} \nabla f_i(x^{(s)})$
5:     Send $g_k^{(s)}$ to parameter server
```

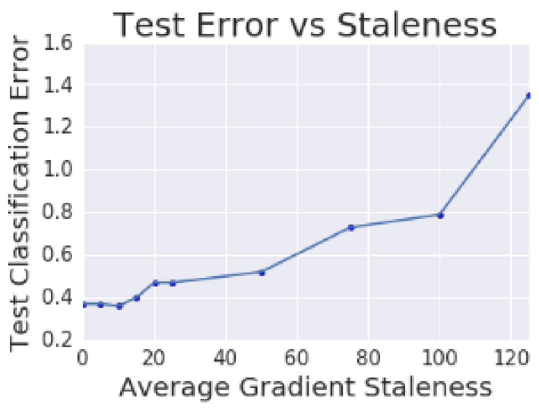
This algorithm solves the problem of straggler nodes that take a long time to send their $g_k^{(t)}$ because the parameter server no longer waits for them all to complete.

The problem of stale parameters However, the gradient updates are not valid because they are usually computed from *stale* iterates $x^{(t)}$. Workers regularly compute their gradients on outdated parameters: for example, a worker might receive parameters $x^{(12)}$ and begin computing its stochastic gradient. Meanwhile the parameters might update five times to $x^{(17)}$. By the time the worker that received parameters $x^{(12)}$ sends its gradient, they will have been based on *stale* parameters.

Definition 24.1 (Staleness)**.** *The staleness of a gradient in an asynchronous distributed learner is the number of updates that were made to the parameters (i.e. iterations) between the iteration the parameters were read to compute the gradient, and the update on the parameter server made with that gradient.*

Stale parameters and test accuracy Chen et al. (2017) systematically evaluated the effect to stale gradients when training a convolutional neural network (CNN) model for MINIST handwritten digit classification. Rather than working in a real-world distributed environment, they simulated staleness so that they could artificially vary the amount of staleness.

Systematic study of the effect of staleness on test accuracy from Chen et al. (2017).



They also found that training became increasingly unstable with more than 15 iterations of staleness and required random restarts on divergence, lower learning rates, and gradual increase of staleness.

24.1.3 Sync-SGD

Distributed synchronous SGD completely avoids the problem of staleness, but can be slowed down by stragglers in the worker pool. Sync-SGD addresses this problem by adding extra workers to the pool (Chen et al., 2017). To obtain a batch size of $m \times b$, Sync-SGD spawns $m + e$ workers, with each one accumulating gradients for b training examples. The parameter server then accepts only the gradients from the *first* m workers, discarding the e

stragglers. In this way, the workers are effectively wasting $e/(m+e)$ of their work, but there is a lower chance the parameter server will be stuck waiting.

Algorithm 4 SyncSGD-ParameterServer

```
1: Choose initial point $x^0 \in \mathbb{R}^n$
2: for $t = 1, 2, \ldots, T$ do
3:     Initialize $\mathcal{G}^{(t)} \leftarrow \{\}$
4:     for $l = 1, \ldots, m$ do
5:         Wait to receive $g_k^{(t)}$ from the next fastest worker $k$
6:         Accumulate $\mathcal{G}^{(t)} \leftarrow \mathcal{G}^{(t)} \cup \{g_k^{(t)}\}$
7:     Apply update $x^{t+1} = x^t - \eta_t \sum_{g \in \mathcal{G}} g$
```

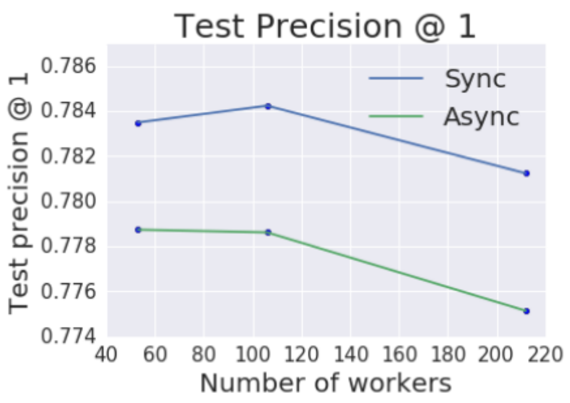
The Sync-SGD worker is identical to the normal asychronous/synchronous settings:

Algorithm 5 SyncSGD-Worker-k

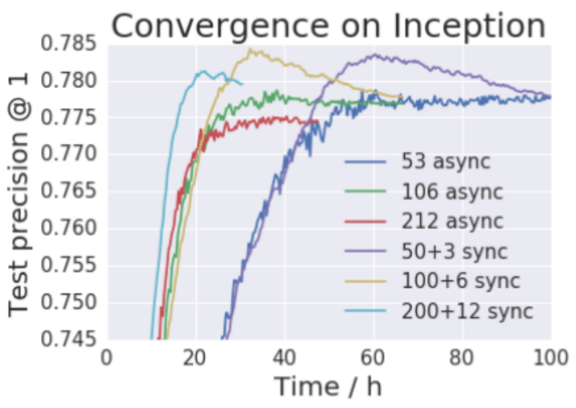
```
1: for $s = 1, 2, 3, \ldots$ do
2:     Request $x^{(s)}$ from parameter server
3:     Sample minibatch $I_k^{(s)} \subseteq \{1, \ldots, n\}$ of size $b$
4:     Compute $g_k^{(s)} = \sum_{i \in I_k^{(s)}} \nabla f_i(x^{(s)})$
5:     Send $g_k^{(s)}$ to parameter server
```

Chen et al. (2017) evaluated the algorithm in a real-world distributed environment to train the Inception model on ImageNet.

The test precision of the synchronous learner is consistently higher than that of the asynchronous learner (from Chen et al. (2017)).



Convergence on test precision of Sync-SGD is also faster than its asynchronous counterpart (from Chen et al. (2017)).



Sync-SGD is not a true stochastic gradient method Because this approach takes the gradients of only the m fastest workers out of the $m + e$ total, the mini-batch that is sampled in the end is not a true uniform distribution over training examples. Instead, it will be skewed towards inclusion of those training examples whose gradients can be computed quickly.

(Could we guarantee that the expected gradient is the true gradient somehow? Consider a setting in which we know that every gradient computation will take an identical number of FLOPS, and the only variance about worker completion time comes from the workers themselves. In this case, the expected gradient should equal the true gradient.)