Lecture Notes 10: distributed/large-scale systems

Instructor: Ashok Cutkosky

Today there are many problems for which we are fortunate to have a huge abundance of training data. This in turn has helped fuel the rise of extremely large models with millions or billions of parameters. This brings a new computational and systems dimension to designing training algorithms.

1. In order to process the data in a reasonable amount of time, we really need to process it in parallel.
2. In some cases, the model may be so large that even evaluating a single loss $\ell(\mathbf{w}, z)$ is slow, so that we would like to evaluate this in parallel as well.

These two considerations lead to what are called *data parallel* and *model parallel* paradigms for parallel or distributed machine learning. Model parallelism typically refers to the case in which the computation of $\ell(\mathbf{w}, z)$ is distributed over multiple *different computers* that communicate over some network rather than the case of speeding up matrix multiplies in a neural network by using parallel computation on a GPU. Model parallelism is far less common that data parallelism, so we will spend much more time talking about the latter.

1 Data parallelism

Data parallelism is essentially a fancy name for minibatching. At the most basic, we simply want to calculate a large minibatch gradient:

$$\mathbf{g}_t = \frac{1}{B}\sum_{i=1}^{B} \nabla \ell(\mathbf{w}_t, z_{t,i})$$

and then take a gradient descent step:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{g}_t$$

The issue is that B is extremely large so we would like to compute $\mathbf{g}_t$ in parallel. The standard algorithm for this is the following:

Algorithm 1 Parallel Minibatch SGD

for $t = 1 \dots T$ **do**
  for $i = 0 \dots M-1$ **do**
    (Asynchronously) ask processor i to compute $\mathbf{g}_{t,i} = \sum_{j=iB/M}^{(i+1)B/M-1} \nabla \ell(\mathbf{w}_t, z_{t,j})$.
  end for
  Wait for all processors to finish their tasks.
  Set $\mathbf{g}_t = \frac{1}{B}\sum_{i=0}^{M} \mathbf{g}_{t,i}$.
  $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{g}_t$.
end for

In an ideal world, each iteration of this algorithm would take $O(B/M)$ time, as each processor takes $O(B/M)$ time to compute its individual sum. In reality, however, things are somewhat more complicated. One issue is the step "wait for all processors to finish their tasks". This synchronization step can significantly slow down the computation in practice. Further, just launching the different jobs on the different processors can carry some significant overhead in terms of setting up or scheduling tasks on a thread. As a result, typically there is a limit beyond which making M bigger will actually result in slower computation.

Algorithm 2 Hogwild! SGD worker algorithm

i is the worker id.
$\mathbf{w}$ is stored in a *shared memory location* across all workers.
for $t = 1 \dots T$ **do**
  Sample $z_{t,i}$.
  Compute $\mathbf{g}_{t,i} = \nabla \ell(\mathbf{w}, z_{t,i})$.
  for $j = 0 \dots d-1$ **do**
    $\mathbf{w}[j] = \mathbf{w}[j] - \eta \mathbf{g}_{t,i}[j]$.
  end for
end for

In order to mitigate this problem [1] introduced a simple idea, which they call Hogwild!: what if you just don't wait for all the processors to finish. That is, each processor performs the following procedure:

In this algorithm, we've carefully written out the update $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{g}_t$ into the individual operations over the coordinates in order to make clearer what can happen. Specifically, if i_1 and i_2 are two different threads, and $\mathbf{w} \in \mathbb{R}^2$, it is possible for the following order of operations to take place:

1. i_1 performs update $\mathbf{w}[0] = \mathbf{w}[0] - \eta \mathbf{g}_{t,i_1}[0]$.
2. i_2 computes gradient $\mathbf{g}_{t,i_2} = \nabla \ell(\mathbf{w}, z_{t,i_2})$.
3. i_2 finishes its gradient descent update with $\mathbf{w}[1] = \mathbf{w}[1] - \eta \mathbf{g}_{t,i_2}$.

In this setting, the gradient $\mathbf{g}_{t,i_2}$ is computed at a *partially udpated* $\mathbf{w}$, so it is really the gradient at the wrong point! This is what the "wait for all processors to finish their tasks" was supposed to prevent. The basic idea of Hogwild! is to just totally ignore this issue and hope that it does not cause any significant problems. This style of updating is called *lock-free* because we are not "locking" the gradient update in order to let one worker completely finish before allowing another worker to operate on the gradient.

That said, we can show that under some reasonable assumptions, this lock-free update actually isn't so bad. Instead of actual Hogwild!, we'll analyze the following easier-to-analyze algorithm:

Algorithm 3 Simpler Hogwild! SGD worker algorithm

i is the worker id.
$\mathbf{w}$ is stored in a *shared memory location* across all workers.
for $t = 1 \dots T$ **do**
  Sample $z_{t,i}$.
  Compute $\mathbf{g}_{t,i} = \nabla \ell(\mathbf{w}, z_{t,i})$.
  Choose an index $j_{t,i}$ uniformly at random from $0, \dots, d-1$:
  $\mathbf{w}[j_{t,i}] = \mathbf{w}[j_{t,i}] - d\eta \mathbf{g}_{t,i}[j_{t,i}]$.
end for

We also need the following terminologies:

- Define $\mathbf{w}_t$ to be the value of $\mathbf{w}$ after t different updates have been performed, regardless of which worker performs the update (if updates occur at the same time, break ties arbitrarily). Note that this index t is different from the worker's internal index t.
- Define $k(t)$ to be the index such that the tth update to $\mathbf{w}$ uses a gradient evaluated at $\mathbf{w}_{k(t)}$.
- Define $p(t)$ to be the thread index from 0 to $M-1$ such that $\mathbf{w}_{t+1}$ generated by thread $p(t)$.
- Define $j(t)$ to be the coordinate that was updated to produce $\mathbf{w}_{t+1}$.

The definition of $k(t)$ implicitly assumes that the computation of $\nabla \ell(\mathbf{w}, z_{t,i})$ really does take place at a single value of $\mathbf{w}_t$. That is, if $\mathbf{w}$ is updated in the middle of computing $\nabla \ell(\mathbf{w}, z_{t,i})$, this should either not effect the computation,

or result in the gradient being the gradient at the updated value. An example of a loss ℓ that would have such a property is if ℓ depends on $\mathbf{w}$ only through some inner-product, as in a linear regression problem:

$$\ell(\mathbf{w}, z_{t,i}) = f(\langle \mathbf{x}_{t,i}, \mathbf{x} \rangle, \mathbf{y}_{t,i})$$

for some function f. In this case, when we compute the inner product $\langle \mathbf{x}_{t,i}, \mathbf{x} \rangle$, we look at each coordinate of $\mathbf{w}$ only once. This issues is basically the only reason we are making the "random coordinate" modification to the optimizer, as it makes the analysis much simpler.

Note that in this section we are completely ignoring the issues of *data races*, which could potentially be a serious problem. To mitigate this, all of the operations on the coordinates of $\mathbf{w}$ need to be carefully wrapped in atomic primitives.

Theorem 1. *Suppose that $\mathcal{L}$ is H-smooth and that $\|\nabla \ell(\mathbf{w}, z)\| \leq G$ for all $\mathbf{w}$ and z. Suppose further $t - k(t) \leq \tau$ for all t for some τ and define $\Delta = \mathcal{L}(\mathbf{w}_1) - \mathcal{L}(\mathbf{w}_\star)$. Then:*

$$\sum_{t=1}^{T} \mathbb{E}[\|\mathcal{L}(\mathbf{w}_t)\|^2] \leq \frac{\Delta}{\eta} + \eta T \frac{2dGH\tau + HG^2}{2}$$

In particular, with $\eta = \frac{\sqrt{2\Delta}}{\sqrt{T(2GH\tau + HG^2}}$, we have:

$$\frac{1}{T} \sum_{t=1}^{T} \mathbb{E}[\|\mathcal{L}(\mathbf{w}_t)\|^2] \leq 2 \frac{\sqrt{\Delta(dGH\tau + HG^2/2)}}{\sqrt{T}}$$

Note that this scales in a reasonable way with τ: τ is measuring the degree to which not synchronizing the workers results in a different operation than doing the correct synchronization. Thus, it makes sense that the performance degrades as τ increases. Further, if $\tau = O(T)$, then we should expect essentially no guarantee, which is exactly what the result shows.

Proof. Since $\mathcal{L}$ is H-smooth and $\|\nabla \ell(\mathbf{w}, z)\|_\infty \leq G$, we have that after for any t, t':

$$\begin{aligned} \|\mathbf{w}_t - \mathbf{w}_{t'}\| &\leq \eta dG|t - t'| \\ \|\nabla \mathcal{L}(\mathbf{w}_t) - \nabla \mathcal{L}(\mathbf{w}_{t'})\| &\leq \eta dGH|t - t'| \end{aligned}$$

Further, we have:

$$\mathbf{w}_{t+1}[j(t)] = \mathbf{w}_t[j(t)] - \eta d \nabla \ell(\mathbf{w}_{k(t)}, z_{k(t),p(t)})[j(t)]$$

Therefore:

$$\begin{aligned} \mathbf{w}_{t+1} - \mathbf{w}_t &= -\eta d \nabla \ell(\mathbf{w}_{k(t)}, z_{k(t),p(t)})[j(t)] \\ &= -\eta(d \nabla \ell(\mathbf{w}_{k(t)}, z_{k(t),p(t)})[j(t)] - \nabla \mathcal{L}(\mathbf{w}_{k(t)})) + \eta(\nabla \mathcal{L}(\mathbf{w}_t) - \nabla \mathcal{L}(\mathbf{w}_{k(t)})) - \eta \nabla \mathcal{L}(\mathbf{w}_t) \end{aligned}$$

Now, notice that $\mathbb{E}[d \nabla \ell(\mathbf{w}_{k(t)}, z_{k(t),p(t)})[j(t)] - \nabla \mathcal{L}(\mathbf{w}_{k(t)})] = 0$ and $\|\nabla \mathcal{L}(\mathbf{w}_t) - \nabla \mathcal{L}(\mathbf{w}_{k(t)})\| \leq \eta dGH|t - k(t)| \leq \eta dGH\tau$. Therefore:

$$\begin{aligned} \mathcal{L}(\mathbf{w}_{t+1}) &\leq \mathcal{L}(\mathbf{w}_t) + \langle \nabla \mathcal{L}(\mathbf{w}_t), \mathbf{w}_{t+1} - \mathbf{w}_t \rangle + \frac{H}{2} \|\mathbf{w}_{t+1} - \mathbf{w}_t\|^2 \\ \mathbb{E}[\mathcal{L}(\mathbf{w}_{t+1})] &\leq \mathbb{E}[\mathcal{L}(\mathbf{w}_t)] - \eta \, \mathbb{E}[\|\mathcal{L}(\mathbf{w}_t)\|^2] + \eta^2 dGH\tau + \frac{H\eta^2 G^2}{2} \\ \sum_{t=1}^{T} \mathbb{E}[\|\mathcal{L}(\mathbf{w}_t)\|^2] &\leq \frac{\Delta}{\eta} + \eta T \frac{2dGH\tau + HG^2}{2} \end{aligned}$$

□

1.1 Parameter Server architecture

The Hogwild! paradigm is designed for a single computer running multiple threads in parallel using a shared memory. However, this is unlikely to scale to a huge number of threads - typically even a very high-quality computer may only have around O(100) cores. To go further, we need to network together multiple computers. In principle, this allows us to increase the parallelism arbitrarily highly, but there is a significant hurdle: now the different machines must communicate over a network rather than simply using a shared RAM. As a result, the communication overhead can quickly become extremely onerous. This leads to significant new implementation challenges. The standard method to deal with this is the *parameter server* architecture. Please see the following excellent source (which is co-written by some of the original proposers of the parameter server) for details on how this would work `https://d2l.ai/chapter_computational-performance/parameterserver.html`

References

[1] Feng Niu et al. "HOGWILD! a lock-free approach to parallelizing stochastic gradient descent". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. 2011, pp. 693–701.