Preserving communication bandwidth with a gradient coding scheme

Scott Sievert

December 21, 2017

Abstract

Large–scale machine learning involves the communicaiton of gradients, and large models often saturate the communication bandwidth to communicate gradients. I implement an existing scheme, quantized stochastic gradient descent (QSGD) to reduce the communication bandwidth. This requires a distributed architecture and we choose to implement a parameter server that uses the Message Passing Interface (MPI). I provide results that illustrate this method is performing, and mention some ideas on future work.

1 Introduction

Large–scale machine learning (ML) involves large models and many data. ML tries to train the model to predict the outputs given the input. This requires that the ML model sees many examples to learn the mapping between the input and output.

Distributed ML has become common in recent years. Models can be large (up to 3GB) and require days to train [6]. Thousands or millions of training examples are often required [7]. This exceeds the capacity of one machine, especially the required training time. Using multiple machines for training can provide speed benefits [6].

The core of machine learning involves the minimization of some loss function. This minimization requires the communication of gradients (or derivatives) which point in the minimization direction. The frequent communication is almost always a speed bottleneck, especially if gradient computation happens quickly and especially if the gradients are large. Communication bandwidth is often an issue and I believe deserves attention.

Lossy compression can reduce the bandwidth required to communicate gradients, and can preserve statistical properties that guarantee convergence with certain methods. These methods will reduce the bits required to communicate the gradient. Recent exciting work shows it will accelerate training by a significant margin.

2 Problem statement

We are trying to minimize some loss function ℓ with respect to our model x and n training examples:

$$\widehat{x} = \arg\min_x \ell(x)$$

This minimization requires communication of the gradient $\nabla \ell$, or some approximation of the gradient. This is typically done via stochastic gradient descent (SGD) or a variant [3, 10] because the converegence of SGD and it's variants do not depend on n [4].

Of course, the number of bits required to transmit the gradient effects the communication bandwidth required. Performing some lossy compression of the gradient will reduce the bandwidth required for optimization. If communication is a bottleneck, training will be accelerated, and likely not by an insignificant margin [1, 11, 12].

Then our goal is to minimize the communication required to transmit the gradient. This can be done via the new exciting work on quantized SGD (QSGD) [1]. Implementing QSGD on a distributed system will require

- implementation of the QSGD encoding scheme
- managing communication between processors with a parameter server [8]

The QSGD coding scheme not deterministic: there are random unknown elements. Elements of arrays are included with some probability, and the length of arrays are unknown.

3 Solution

The implementation of my solutions is on GitHub under

- `stsievert/WideResNet-pytorch` which implements the encoding scheme and the surrounding code to glue everything together
- `stsievert/pytorch_ps_mpi` which implements the MPI parameter server to be used with the coding scheme and PyTorch
- `stsievert/ec2-cluster-with-mpi` which launches an EC2 cluster, installs the required packages and enables MPI

And the results are in `759--stsievert/Project/WideResNet-pytorch-out`. All graphs and results in this paper are from the directory `output/2017-12-19`.

We choose PyTorch as the software platform to implement QSGD on. PyTorch has the advantage of a dynamic computational graph, and not a static computational graph (like Tensorflow, MXNet, etc). Dynamic computational graphs are advantageous and allow for easier manipulation of the stochastic coding scheme.

Additionally, the dynamic computation graph enables for explicit declaration of the parameter server (and not the implicit communication found in other packages like Tensorflow, etc). For an overview of different software packages and their features, see the Chainer comparison in their documentation.

3.1 Quantized SGD

Quantized stochastic gradient descent (QSGD) quantizes the gradient from a floating point number x with high precision to a number $Q(x)$ of much lower precision [1].

The theory enabling this quantization says that the quantization is an unbiased estimator and that the variance is bounded by the true variance. The quantification of this for QSGD is

$$\mathbb{E}\left[Q(v)\right] = v$$
$$\mathbb{E}\left[\|Q(v)\|_F^2\right] \leq \sqrt{n}\,\|v\|_F^2$$

when the gradient $v \in \mathbb{R}^n$ and means that QSGD will require at most $\sqrt{n}$ more iterations [1]. The coding scheme that achieves these bounds is

$$Q_i(v) = \|v\|_2 \operatorname{sign}(v_i)\eta_i(v)$$

where $\eta_i(v)$ are independent random variables such that $\eta_i(v) = 1$ with probability $|v_i| \,/\, \|v\|_2$ and 0 otherwise [1]. The main benefit behind this coding scheme can be communicated using $\sqrt{n}\,(\log n + \log 2e) + F$ bits in expectation when F is the number of bits for number (32 for `float`) [1].

3.1.1 Implementation

The implementation can be found on GitHub at qsgd.py.

We only implement the 1-bit encoding scheme for QSGD, though QSGD generalizes further to k-bit encoding schemes. However, the implementation is not straightforward. The only hints they give in Corollary 3.7 is

> This encoding scheme is not entirely obvious ... we can employ a recursive encoding which optimizes code length given that certain values occur less often.

Future work is to utilize this recursive strategy.

3.2 Parameter server

We have P processors in our network, and they need to share some model vector x. This model vector may be too large to fit in the memory of a single machine. This, alongside the straggler effect [2] motivate a parameter server. The parameter server's only job is to serve the model parameters to the workers.

The parameter server communication is a master–worker setup. Every worker initializes the model to the same value, sends gradients and receives new model parameters.

Our parameter server must support sending arbitrary sized objects back and forth because the coding scheme is stochastic. This requires serialization of the objects that represent the coding scheme.

3.2.1 Implementation

We implement our parameter server using MPI with the Python wrapper `mpi4py`. We implement synchronous communications, and wait for all workers to finish computing their gradient before continuing with the next optimization iteration.

Our communication scheme only uses non-blocking communication. We can send arbitrary sized inputs by using the MPI function `Igatherv` (the last `v` is for variable sized inputs).

This enables us to send arbitrary Python objects. However, serialization is still required. We cast the PyTorch Tensors as NumPy arrays because PyTorch lacks fast serialization even though moves the Tensor to the CPU from the GPU (which we want to avoid). NumPy is doing some work to support GPU NumPy arrays[1] which will make this transfer unnecessary.

Connecting the various components for synchronous communications involves performing the communication and computation in parallel, since the gradients are received sequentially (not all at once).

We do this by sending an asynchronous request as soon as the gradient for a particular layer is calculated. Then the worker continues to calculate more gradients while the communication happens in the background. Currently, the encoding process happens synchronously and blocks computation of the gradient. Future work is to resolve this, and more detail is in Section 3.2.2.

For full detail, see the complete code at on GitHub at stsievert/pytorch_ps_mpi.

3.2.2 Future work

We aim to support asynchronous communication to remove the straggler problem. Every worker will

1. compute a gradient and send it to the master
2. check to see if the master is ready to send new model parameters
3. compute a new gradient using the current model parameters regardless if they've been refreshed

regardless of what other workers are doing – there will be no synchrony. Systems have been built to support this communication scheme, and there is some theoretical analysis on the algorithm convergence [9]. This idea is described with in more detail in pytorch_ps_mpi#1.

Additionally, we want to

- use MPI's `Iallgatherv` to send variable length inputs to every other node. This will only be used if the asynchronous communication above is not implemented (and likely depends on what the QSGD authors used to show their results). This will reduce the communication required to send the newly computed parameters to the workers.

[1] See Nathaniel Smith's BIDS talk at `https://www.youtube.com/watch?v=fowHwlpGb34`

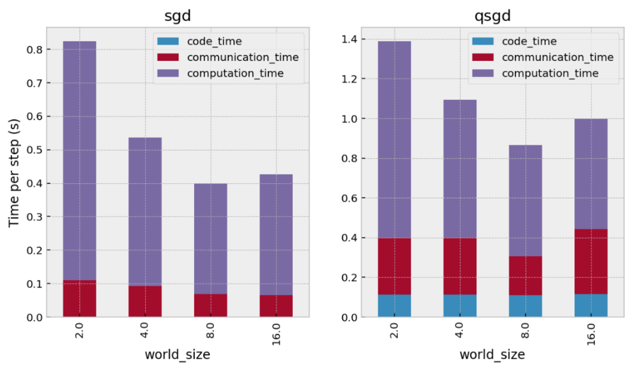

Figure 1: A histogram showing the proportion of time doing different tasks as the number of workers varies (and generated from the results in `WideResNet-pytorch-out/2017-12-19`). Note that the y-axis has different scales, which reflects a QSGD specific bug to be squashed.

- implement the encode function asynchronously. There's no reason to wait for the encoding process to finish before continuing computation. The implementation will probably rely on Dask (and specifically dask.delayed and future.add_done_callback).

We believe an asynchronous encode function would further help reduce the communication costs and make our system more practical.

4 Evaluation

We evaluate our parameter server and model with $\{2, 4, 8, 16\}$ Amazon EC2 p2.xlarge instances. We treat each machine as one worker, because each machine has one GPU. Some complications arose in trying to use a single machine with mutiple GPUs as one system[2].

Using the setup described above we can generate some results. These results will highlight the communication time differences, mirror the results in [1]. This aids the fact that different quantization levels change the optimal step size mentioned in Theorem 6.3 of [5] as the noise variance σ increases.

The results for a different number of workers is shown in Figure 1. I believe the increase in times from 8 to 16 workers is due to the "straggler effect", where the entire cluster is waiting on several slow nodes. I believe implementation of an asynchronous parameter server will resolve this.

[2]Specifically related to PyTorch's slow serialization relative to NumPy. See my question on discuss.pytorch.org for more detail.

5 Conclusion

There is existing work to create new compression schemes with the same properties as QSGD. This is the focus of my research, and these tools aid the design.

References

[1] Dan Alistarh, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Randomized quantization for communication-optimal stochastic gradient descent. *arXiv preprint arXiv:1610.02132*, 2016.

[2] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013.

[3] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[4] Olivier Bousquet and Léon Bottou. The tradeoffs of large scale learning. In *Advances in neural information processing systems*, pages 161–168, 2008.

[5] Sébastien Bubeck et al. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning*, 8(3-4):231–357, 2015.

[6] 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.

[7] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[8] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.

[9] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[10] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, pages 1–30, 2013.

[11] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *arXiv preprint arXiv:1705.07878*, 2017.

[12] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. The zipml framework for training models with end-to-end low precision: The cans, the cannots, and a little bit of deep learning. *arXiv preprint arXiv:1611.05402*, 2016.