Integrated model and data (batch + domain) parallelism for training deep neural networks

Leo Laugier
Master's Student
UC Berkeley
leo_laugier@berkeley.edu

Lysia Li
Master's Student
UC Berkeley
lysia_li@berkeley.edu

Avery Nisbet
Master's Student
UC Berkeley
danisbet@berkeley.edu

Evan Thompson
Master's Student
UC Berkeley
thompe5@berkeley.edu

I. Problem

Our project aim is to implement Model Parallelism alongside Data Parallelism when training Deep Neural Networks. We will use Horovod, a project supported by Uber[1], to develop a distributed layer to allow functionality for these methods of parallelism. Through Horovod[2], we will be implementing these methods of parallelism as a layer on top of Keras and TensorFlow using MPI.

II. Types of Neural Network Parallelism

There are a couple of ways to actually achieve parallel computation when training Neural Networks. Model Parallelism splits the weights of the model across several processes, and each process computes just the changes in those weights [3]. Data Parallelism, instead, computes the gradients of batches in parallel and then applies the gradients together [3].

Let $P \in \mathbb{N}^* \backslash \{1\}$ be the total number of processes, α the network latency, β the inverse bandwidth, $L \in \mathbb{N}$ the number of layers in the neural network and $X_l \in \mathbb{R}^{d_l \times B}$ the input activations for the l^{th} layer W_l with $|W_l|$ parameters.

A. Model Parallelism

Model Parallelism is performed by splitting the weights of a model across processes, and the same data is used for each part of the model. The advantages of this method are immediately apparent when working on a large model. When performing computation on a GPU, all the weights may not fit on the memory of a single GPU. This process can be performed by splitting up the weights of each layer into multiple parts to each be handled by a different process.

A methodology for the computation of model parallelism is presented by Shrivastava *et al.*[4]:

1) Forward Pass
   a) Broadcast the weights to all processes.
   b) Each process computes the subset of the output using its weights.
   c) The output vectors are gathered and stacked.
   d) Repeat this for every layer.
2) Backwards Pass
   a) Broadcast the error vector to all processes.
   b) Each process computes the subset of the weight changes.
   c) Each process computes the contribution to the error vector at the next layer.
   d) The error vector is gathered, and summed.
   e) Repeat this for every layer

By this algorithm, data must be synchronized at every layer. However the amount of communication is small - limited to just the output of a layer, and on back-propagation, the updates for a subset of weights. Gholami *et al.*[3] showed that the communication complexity for model parallelism was:

$$3 \times \sum_{l=0}^{L} (\alpha \times \lceil log(P) \rceil + \beta \times B \times \frac{P}{P-1} \times d_l)$$

B. Data Parallelism

Data Parallelism is performed by computing the training on different minibatches of data at the same time. Each process must have the same copy of the model weights, but performs work on a different portion of the data. This works well for training small models, especially over large datasets.

Each individual process performs the standard stochastic gradient descent over one batch before synchronizing and updating the weights of the model.

By this algorithm, synchronization occurs less often than in model parallelism. However, the communication overhead is large. After the backwards pass, the entire gradient for the model must be passed to each network. Additionally, batch parallelism does not scale infinitely. Decreasing the batch size in a single node below a certain point decreases efficiency[5]. This occurs since the speed of computation is dependent on the tiling size BLAS used and the size of the input data. Gholami *et al.*[3] showed that the communication complexity for data parallelism was:

$$2 \times \sum_{l=0}^{L} (\alpha \times \lceil log(P) \rceil + \beta \times \frac{P}{P-1} \times |W_l|)$$

III. Horovod

Horovod is a framework, created by Uber, to train TensorFlow and Keras neural networks across several compute processes. It has several advantages over TensforFlow's distributed framework. One of the primary benefits of Horovod is that it uses many of TensorFlows optimizations for very

fast computation but leverages MPI[1] to spread computation across many compute processes. Using these optimizations, Horovod is able to achieve very high scaling efficiency on several mainstream benchmarks (Inception V3, ResNet-101, and VGG-16)[1].

IV. Related Work

In a paper by Hegde *et al.*[6], several methods of implementing parallel and distributed deep learning algorithms were discussed. These discussions included pseudo-code algorithms and the pros and cons associated with those design choices. This can help serve as indication as to what direction to go when designing our Horovod based model and data parallelization.

We also found related work from Shrivastava *et al.*[7] that discussed at length their implementation of Model and Data parallelism for Deep Learning using Spark. While this will prove useful, it is slightly tangential to our work but will still nonetheless prove useful as a way of guiding our work.

V. Project plan

Our approach will be to propose to Tensorflow users to be able to train their architecture first with data parallelism, then with model parallelism and eventually with an integrated data and model parallelism. To help us to analyze performance, Horovod Timeline is a tool that will enable us to record the timeline of its processes. We will use distributed memory system methods with MPI to communicate the weights of the model and the training data among multiple processes.

Thus, this project will explore parallelism for training deep neural networks with the following goals:

1) Set up the deep learning programming environment on NERSC's Cori supercomputer by installing our own Tensorflow and Horovod on it.
2) Implement both model parallelism and data parallelism with MPI through Horovod by creating a layer wrapper, to train fully connected neural networks.
3) Evaluate the performance of training a fully connected neural network with our parallelizations in comparison to training the same network with Distributed TensorFlow, and regular, non-distributed TensorFlow. [8].
4) Repeat for other artificial neural networks such as convolutional neural networks.

VI. Desired Results

Ideally, at the end of this project, we will have produced quality additions to Horovod providing the ability to train fully connected networks and convolutional networks using model and data parallelism. Furthermore, we aim to produce graphs showing how we scale the training on multiple processes and how this compares to the theoretical speedup. Therefore, we will be able to analyze which of pure model parallelism, pure data parallelism and integrated model and data parallelism have the best performance for a particular architecture. Lastly, we plan to use *ImageNet LSVRC-2012*[1] as traning dataset. Time permitting, we would evaluate our method on the AlexNet architecture [9] with both convolutional and fully connected layers.

[1]http://www.image-net.org/challenges/LSVRC/2012/

VII. Potential Issues

Horovod is a framework that we are unfamiliar with using. As such, there could be significant challenges fully implementing the model and data parallelism pipelines that we want. However, we have access to people with experience using the framework who will be able to help us should we need it.

We will also need to install Horovod on NERSC Cori. This is a process that we are unfamiliar with but, similar to the previous issue, we have contacted someone who would be able to help us with that process.

References

[1] Uber. (2018). horovod. [online] Available at: https://github.com/uber/horovod [Accessed 27 Mar. 2018].

[2] Sergeev, A. (2017) Horovod - Distributed TensorFlow Made Easy. Retrieved from https://www.slideshare.net/AlexanderSergeev4/horovod-distributed-tensorflow-made-easy

[3] Gholami, A., Azad, A., Jin, P., Keutzer, K. and Buluc, A. (2018). Integrated Model, Batch and Domain Parallelism in Training Neural Networks. [online] Arxiv.org. Available at: https://arxiv.org/abs/1712.04432 [Accessed 27 Mar. 2018].

[4] Shrivastava, D., Chaudhury, S., and Jayadeva (2017). A Data and Model-Parallel, Distributed and Scalable Framework for Training of Deep Networks in Apache Spark. [online] Arxiv.org. Available at: https://arxiv.org/pdf/1708.05840.pdf [Accessed 26 Mar. 2018].

[5] Keuper, J., Pfreundt, F. J. (2016). Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability. [online] Arxiv.org. Available at: https://arxiv.org/pdf/1609.06870.pdf [Accessed 27 Mar. 2018].

[6] Hegde, V. and Usmani, S. (2016). Parallel and Distributed Deep Learning. [online] Stanford. Available at: https://stanford.edu/ rezab/classes/cme323/S16/projects_reports/hedge_usmani.pdf [Accessed 27 Mar. 2018].

[7] Shrivastava, D., Chaudhury, S. and Dr. Jayadeva (2017). A Data and Model-Parallel, Distributed and Scalable Framework for Training of Deep Networks in Apache Spark. [online] arXiv. Available at: https://arxiv.org/pdf/1708.05840.pdf [Accessed 27 Mar. 2018].

[8] Google. (2018) tensorflow. [online] Available at: https://www.tensorflow.org/deploy/distributed

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 10971105, 2012.