Integrated model and data parallelism for training deep neural networks

Leo Laugier
UC Berkeley
leo_laugier@berkeley.edu

Lysia Li
UC Berkeley
lysia_li@berkeley.edu

Daniel Avery Nisbet
UC Berkeley
danisbet@berkeley.edu

Evan Thompson
UC Berkeley
thompe5@berkeley.edu

May 24, 2018

1 Abstract

Neural Networks, used in a myriad of problems from financial modeling to computer vision, are widely considered one of the most powerful machine learning models known. These complicated collections of "neurons" can approximate any function but require significant amounts of time to train. However, this training time can be greatly reduced through the use of parallel computing. Two common types of parallelism within Neural Networks are **batch parallelism** and **model parallelism**. These paradigms allow for large datasets and models to be efficiently computed in a high performance computing environment. We investigate the performance of both batch and model parallelism as well as a combined method.

Using pure batch parallelism, we achieved **5.8x** speedup by running 32 processes with batch size = 100 and model size = 1,024. Using pure model parallelism, we achieved **1.2x** speedup with model size = 1,024. Integrating batch and model parallelism, the performance was **0.99x** compared to the pure batch approach under the experiments setup.

2 Introduction

In recent years, the quantity of available training data has been increasing and network architectures have become increasingly large as task complexity grows. Modern networks such as VGG16 have tens of millions of parameters. Distributed network training procedures are commonly used so that models can be trained in a reasonable amount of time.

Stochastic gradient descent (SGD) is a common optimization procedure used to train deep neural networks. The procedure iteratively selects a sub-sample of the data-set, called a minibatch, computes the output of each network's layer, and propagates the gradient so the weights of the model can be adjusted. In the case of a network built entirely of fully connected layers, stochastic gradient descent can be reduced to a series of matrix multiplications.

One approach to a distributed SGD is to split the training samples in a minibatch between different processes. A smaller amount of training data is computed by each process, at the cost of increased communication time after every minibatch. This technique is called batch parallelism. An alternative approach is to split up each layer between different processes. Only a portion of the output of each layer is computed at each step, requiring less computing power and significantly less memory on each process to store the network parameters, however, communication must be performed after each layer to collect and propagate the layer output.

Selecting which parallelization technique will confer the largest speedup is nontrivial. It is highly dependent on several hyperparameters for training the model such as batch size, layer sizes and depth of the network in addition to the computer's architecture. In this paper, we analyze the performance of both batch and model parallelism under various network and batch sizes. Furthermore, we investigate the performance of a hybrid procedure that both splits the minibatch and the model between different processes.

3 Related Work

As neural networks become more and more widely used, many researchers start to study parallelizing deep neural networks' training process. Researchers have developed systems that implement different types of parallelism with various architectures. In 2014, Chilimbi *et al.*[1] designed and implemented a distributed system called Adam, that uses commodity server machines to train models on visual recognition tasks. Adam implements data parallelism, allowing multiple replicas of the model to be trained in parallel on different partitions of the training data set. With the different data shards, all the workers share a common set of parameters, which sits on a global parameter server. The workers then asynchronously publish model weight updates (gradients) to and receive updated parameters from the global parameter server.

DistBelief is a software framework developed by Dean *et al.*[3] addressing the problem of training a deep network with billions of parameters using tens of thousands of CPU cores. They developed algorithms for asynchronous stochastic gradient descent procedure *Downpour SGD*, and for distributed batch optimization procedures *Sandblaster*. *Downpour SGD* works similarly as before, where there is a centralized parameter server. The optimization algorithms implements an intelligent version of data parallelism. In addition, *DistBelief* also supports model parallelism. This framework was able to train the biggest neural network at the time.

Asynchronous SGD is a variation of the SGD algorithm and changes the behavior of the algorithm. Das *et al.*[4] studied distributed multinode synchronous SGD, that does not alter hyperparameters, or compresses data, or alters algorithmic behavior. They tested on the Cori system and achieved high efficiency.

Combining parallelism from different paradigms and using synchronous SGD, Gholami *et al.*[6] propose a new integrated method incorporating model, batch, and domain parallelism using minibatch SGD. Inspired by the communication-avoiding algorithms in linear algebra, they demonstrated that the lowest communication costs are not achieved with the pure model or pure data parallelism.

4 Definitions and conventions

4.1 Communication cost related definitions

In this section, we define variables and constants that will appear in the communication cost analysis of data and model parallelism. Let $P \in \mathbb{N}^* \backslash \{1\}$ be the total number of processes, n_b the number of nodes for batch parallelism, n_m the number of nodes for model parallelism, *data* the amount of data in a given minibatch, 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.

4.2 Conventions

This section gives some background and formalism to clarify the pseudo-codes in this report.

We denote by a single ∇ the gradient of the loss $\mathcal{L}$ with respect to the network's parameters *i.e.* the weights (∇w) and the biases (∇b).

We use the following naming convention so that the reader visualizes which key variables are shared or unique to some processes, the key variables being $miniBatch$, $previous/next\_y$, $previous/next\_\nabla y$ and ∇:

- $variable_{rank}$ is unique to the process $rank$.
- $variable$ is common to all the processes in the communicator (it is often the result of a collective communication such as AllGather or AllReduce).
- In the description of integrated data and model parallelism, $variable_{parallelismType\_split}$ means that variable is unique to the processes involved in the $parallelismType$ and common to all the processes involved in the other type ($parallelismType \in \{model, data\}$)

4.3 Random generator manipulation

As model parallelism requires that the processors share the same weights and biases, we manipulate the draws by seeding the Numpy random generator. $small\_layers\_init()$ and $layer\_init()$ represent functions that initialize weights so that processes involved in batch parallelism get the same seed for the same layer whereas processes used in model parallelism get different seeds. What is more, two matrices from different layers belonging to the same process must have different seeds.

5 Data Parallelism

Data parallelism consists of splitting up a minibatch of data, distributing it across the processes and performing the forward and backward propagation. Batch parallelism is the particular version of data parallelism we implement, which splits the minibatch by samples, as shown in Figure 1 (Left). 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.

As shown in Algorithm 1 in Appendix, 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 does not occur frequently, just once per batch. However, the communication overhead is large. Therefore, for a system with high bandwidth, this procedure will work well, even with high latency. 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[7]. 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.*[6] showed that the communication complexity for data parallelism was:

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

Note that, when we implemented data parallelism, we chose to perform AllReduce on models parameter gradients, after the back-propagation rather than for each backward operation (*i.e.* for each layer) to minimize communication costs and unnecessary communication for each epoch (Algorithm 1).

6 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. This is advantageous for large models where less cache memory needs to be used on storing the model parameters. Therefore, model parallelism is performed by splitting up the weights of each layer into multiple parts to each be handled by each process, the procedure for which is shown in Figure 1.

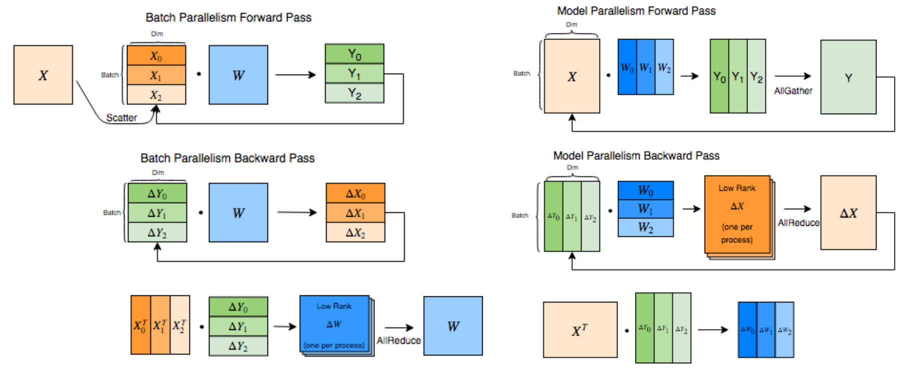
Batch Parallelism Forward Pass
Scatter
Batch Parallelism Backward Pass
AllReduce
Model Parallelism Forward Pass
AllGather
Model Parallelism Backward Pass
Low Rank
(one per process)
AllReduce

Figure 1: Parallelism procedure for one affine layer - Left: Batch Parallelism - Right: Model Parallelism

By this algorithm, outlined in Algorithm 2 in Appendix, 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. Therefore, for a system with low latency, this procedure will work very well, even with low bandwidth. Gholami *et al.*[6] showed that the communication complexity for model parallelism was:

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

7 Combined Data and Model Parallelism

We combine model and batch parallelism by assigning each process a unique portion of the data and model. Shown in Algorithm 3 in Appendix, during the forward pass, each process computes a small portion of the output, which is gathered among the process for each batch split. During the backwards pass, each process produces low rank matrices of the propagated gradient, ΔX, which is reduced among the processes for each data split. Each process also produces low rank matrices of the model's parameter gradients, which must be reduced among the processors for each model split.

As shown in Figure 2, several processes each need the same partition of the minibatch. This suggests that there are two methods which the data can be scattered between the processes. The first would be to send the data that each process needs from process 0. This has a high bandwidth cost, but low latency cost. The alternative would be to first send one copy of each partition of the data to several nodes (first scatter data), then have those nodes send that partition to each process that needs the same data (then broadcast data).

The speed at which it takes to send a message by the first method is: $\alpha +$

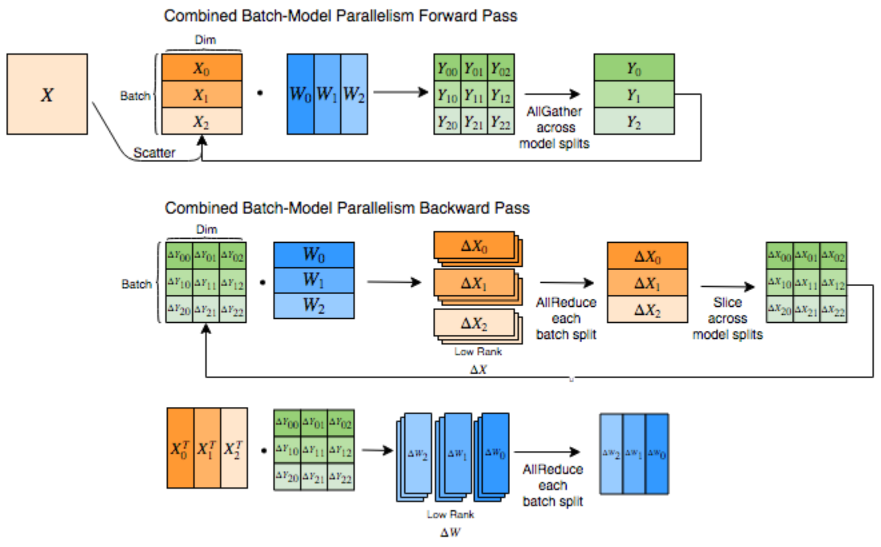
Combined Batch-Model Parallelism Forward Pass
Dim
Batch
Scatter
AllGather across model splits
Combined Batch-Model Parallelism Backward Pass
AllReduce each batch split
Slice across model splits
Low Rank
ΔX
AllReduce each batch split
Low Rank
ΔW

Figure 2: Combined batch-model parallelization procedure for one affine layer

$n_b * n_m * data * \beta$. The speed at which it takes to send a message by the second method is $2\alpha + max(n_b, n_m) * data * \beta$

Setting these two expressions equal and solving with the communication constants on Cori, where bandwidth is 5.6TB/s and latency is $1\mu s$ shows that method one is faster as long as $data > \frac{5.6MB}{n_b * n_m - max(n_b, n_m)}$; that is why we implemented method one (called scatter_broadcast in Algorithm 3)

Communicators

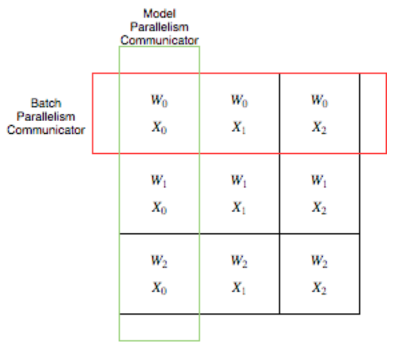
Model Parallelism Communicator
Batch Parallelism Communicator

Figure 3: Communicators define the set of processes across which either the model weights or input data must be split

In order to reuse the collective communication functions we wrote for pure batch and pure model parallelism (AllReduce and AllGather), we built two sets of communicators: the model parallelism related communicators and the batch parallelism related communicators (Figure 3). We set colors with some arithmetic so that each process knows the communicator it belongs to, regarding both parallelism types.

8 Framework: Hardware, Network Architecture and Datasets

The network that we tested on is a **fully connected feed-forward** neural network, shown in Figure 4. The intermediate output then goes through a ReLU before it gets passed on to the next layer. We implemented the neural network from scratch using Python and Numpy, which is our baseline serial program. The parallelization was performed using MPI.

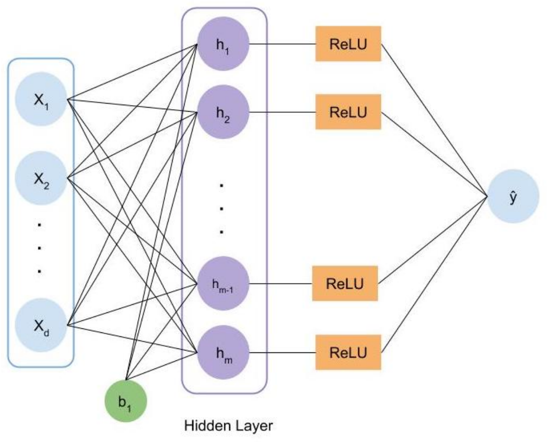

Figure 4: Fully connected neural network

Throughout the experiments, we trained the neural networks in 10 epochs. For different scaling experiments, we varied the batch size, network depth, number of processes, width of the model, and size of the datasets. All experiments were run on Intel Xeon "Haswell" processors using University of California Berkeley's supercomputer Cori.

The medium dataset used is UC Irvine Airfoil Noise dataset [5]. It is a NASA data set obtained from a series of aerodynamic and acoustic tests of airfoil blade sections conducted in an anechoic wind tunnel. We used 4 features to predict the decibel of an airfoil. The large dataset used is UC Irvine Ethylene Dataset [5]. We used 16 features to determine the concentration of ethylene in a methane mixture. The dimensions of the dataset is shown in Table 1 below.

	Medium: Airfoil Noise	Large: Ethylene
Input	$1,503 \times 4$	$700,000 \times 16$
Output	$1,503 \times 1$	$700,000 \times 2$

Table 1: Datasets used in the experiments

9 Results

9.0.1 Batch parallelism

In this phase of the experiments, we first kept the model parameter size fixed and increased the number of processes to perform strong scaling analysis. We ran this procedure for different sizes' model, i.e model size = 32, 64, 128, 512, and 1,024, shown in Fig 5. Each line denotes a model of different size and $P_r = 1$ in all cases. As we increase the number of processes, we start to observe better and better speedup. For the model of size 512, it achieved **5.8x** speedup as the best performance with 32 processes (32×1). With relatively larger models, the amount of computation (i.e gradient calculation) each process has to do increases and in our case, it is still less than the communication each batch needs to do.

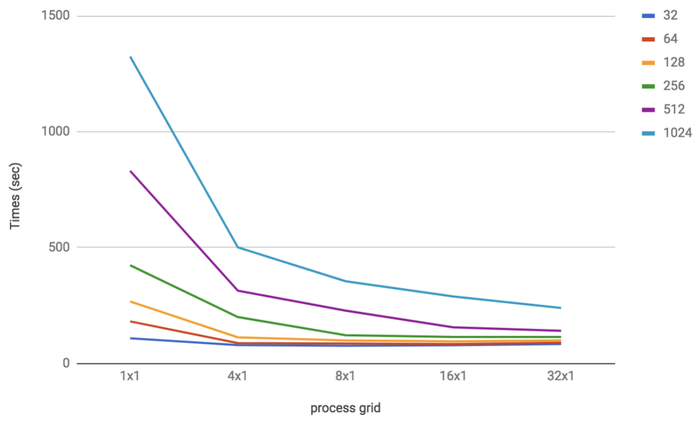

Figure 5: Strong scaling analysis of batch parallel implementations

9.0.2 Model parallelism

For model parallelism, we fixed the model width and increased the number of processes by a power of 2 each time. Similar to batch parallel experiments, we also ran this procedure with model size = 32, 64, 128, 512, and 1,024. The result is shown in Fig 6. With model size $= 1,024$, we were able to achieve **1.2x** speedup in execution time (seconds). As the model size increases from 32 to 1,024, we get more speedup as model parallelism is designed to work well with large models. When using fewer nodes, the communication is costly in model parallelism as it happens at every single layer of the network. The advantages gradually show as we increase the model size. In our experiments, the model size was not large enough to show the significant advantages of model parallelism. In comparison, Shrivastava *et al.* [8] has model size from 4 million to 4 billion.

Increasing the model size will not increase the speedup infinitely. Shrivastava *et al.* [8] also found that there is a saturation point as they kept increasing the parameter size to 4 billion. In our case, since our model size is below the saturation point, we were able to see constant increase in speedup as the model size increases.

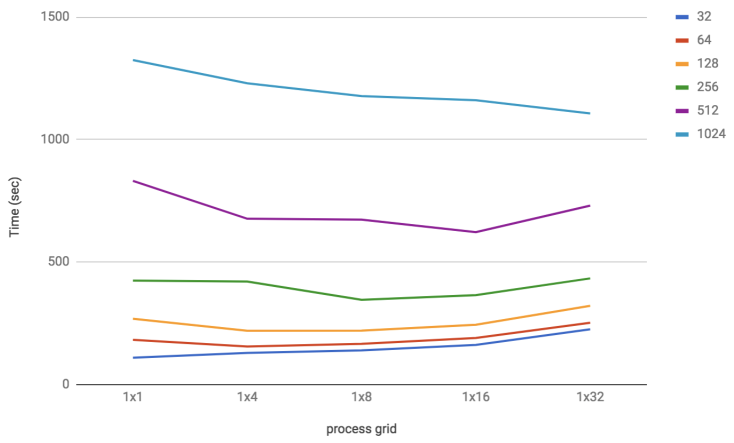

Figure 6: Strong scaling analysis of model parallel approach for different model sizes

9.0.3 Integrated batch and model parallelism

Testing integrated batch and model, we trained the model using P = 4 to P = 32 with a fixed mini-batch size as before, and model size = 512. In each subfigure in Fig. 7 only the process grids vary. With fewer processes, the integrated approach has a suboptimal performance to those of batch alone. The advantage of integrated implementations over the others shows up with more processes. With P = 16 and P = 32, integrated implementation starts to have better performance and approaches the performance of batch parallelism. However, in our experiments' setup, integrated training was very close to but did not surpass batch parallelism training's performance i.e **0.99x**. Theoretically, integrated batch and model parallelism can help reduce the communication overhead of the pure batch parallel case, since only a subset of the weights is replicated each time instead of the entire weights. In our setup, we suspect that the model size was not large enough to have an effect on reducing the communication overhead.

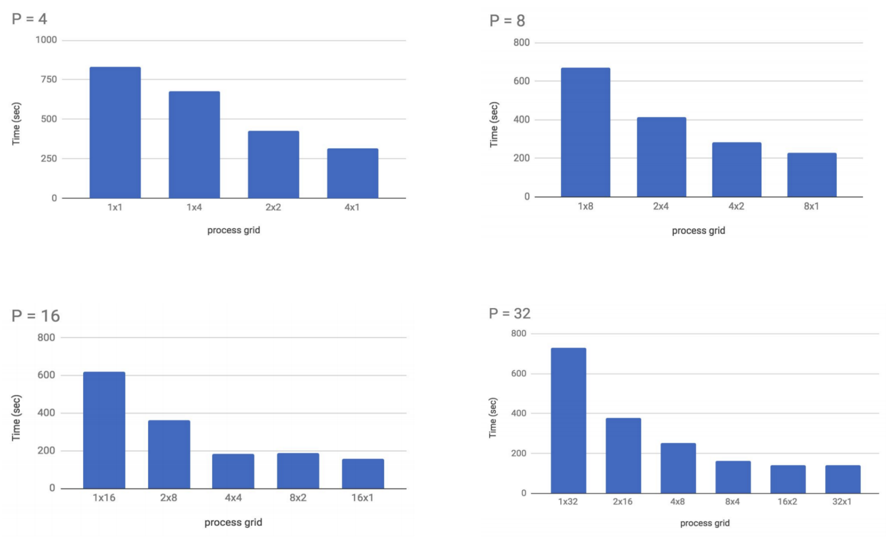

Figure 7: Strong scaling analysis of integrated batch and model for fixed batch size, different processes

10 Conclusion

In this paper, we implemented a fully connected neural network and explored batch, model and integrated parallelism for training the fully connected neural network. For the experiments, we trained the networks using a dataset of size $700,000 \times 10$. Using pure batch parallelism, we achieved **5.8x** speedup by running 32 processes with batch size = 100 and model size = 1,024. Using pure model parallelism, we achieved **1.2x** speedup with model size = 1,024. Integrating batch and model parallelism, the program's execution time was **0.99x** compared to the pure batch approach under the experiments setup.

For future work, based on the current results we have, it will be interesting to explore what is the threshold of model size and data size for the different parallelisms to have significant effects. Based on the thresholds, users can train with the corresponding parallelism based on the data and model size. In this work, we only tested with the basic fully connected network with ReLU as activation. Different network architectures, e.g CNN, RNN, and different activation, e.g Softmax, will have different behavior that is worth studying. In addition to batch and model, there is one more approach – domain parallelism that we did not have enough time to incorporate. Lastly, as a side note, while researchers are studying training in parallel extensively, testing in parallel remains relatively unexplored that could have a significant performance gain in practice.

References

[1] Chilimbi, T. M., Suzue, Y., Apacible, J., Kalyanaraman, K. (2014, October). Project Adam: Building an Efficient and Scalable Deep Learning Training System. In OSDI (Vol. 14, pp. 571-582).

[2] Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., Andrew, N. (2013, February). Deep learning with COTS HPC systems. In International Conference on Machine Learning (pp. 1337-1345).

[3] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., ... Ng, A. Y. (2012). Large scale distributed deep networks. In Advances in neural information processing systems (pp. 1223-1231).

[4] Das, D., Avancha, S., Mudigere, D., Vaidynathan, K., Sridharan, S., Kalamkar, D., ... Dubey, P. (2016). Distributed deep learning using synchronous stochastic gradient descent. arXiv preprint arXiv:1602.06709.

[5] Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[6] 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].

[7] 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].

[8] 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].

11 Appendix

11.1 Batch parallelism implementation

These are the algorithms described in section 5, 6, 7.

Algorithm 1 Batch parallelism

```
procedure TrainBatchParallelism($training\_data, epochs, miniBatchSize, \eta$)
    $\{layers\} \leftarrow$ layers_init(), $\mathcal{L} \leftarrow$ loss_init()
    for $epoch \in \{epochs\}$ do
        if rank = 0 then
            $\{miniBatches\} \leftarrow$ CreateMiniBatches($training\_data, miniBatchSize$)
        for $miniBatch \in \{miniBatches\}$ do
            $miniBatch_{rank} \leftarrow \underline{\text{Scatter}}(miniBatch)$
            $previous\_y_{rank} \leftarrow miniBatch_{rank}$
            for $layer \in \{layers\}$ do
                $next\_y_{rank} \leftarrow$ ForwardPass($previous\_y_{rank}$)
            $previous\_\nabla y_{rank} \leftarrow \text{Compute}\frac{\partial \mathcal{L}}{\partial \hat{y}}(\mathcal{L},\ next\_y_{rank})$
            for $layer \in reverse(\{layers\})$ do
                $\nabla_{layer,rank},\ next\_\nabla y_{rank} \leftarrow$ BackwardPass($previous\_\nabla y_{rank}$)
            $\nabla \leftarrow \underline{\text{AllReduce}}(\nabla_{rank})$
            ApplyGradient($\nabla, \eta$)
```

11.2 Model parallelism implementation

Algorithm 2 Model parallelism

```
procedure TrainModelParallelism($training\_data, epochs, miniBatchSize, \eta$)
  $\{layers\} \leftarrow$ small_layers_init(), $\mathcal{L} \leftarrow$ loss_init()
  for $epoch \in \{epochs\}$ do
    if rank = 0 then
      $\{miniBatches\} \leftarrow$ CreateMiniBatches($training\_data, miniBatchSize$)

    for $miniBatch \in \{miniBatches\}$ do
      $previous\_y \leftarrow miniBatch$
      for $layer \in \{layers\}$ do
        $next\_y_{rank} \leftarrow$ ForwardPass($previous\_y$)
        $next\_y \leftarrow$ AllGather($next\_y_{rank}$)

      $previous\_\nabla y \leftarrow$ Compute$\frac{\partial \mathcal{L}}{\partial \hat{y}}$($\mathcal{L}$, $next\_y$)
      for $layer \in reverse(\{layers\})$ do
        $previous\_\nabla y_{rank} \leftarrow$slice($previous\_\nabla y$)
        $\nabla_{layer,rank}$, $next\_\nabla y_{rank} \leftarrow$ BackwardPass($previous\_\nabla y_{rank}$)
        $next\_\nabla y \leftarrow$ AllReduce($next\_\nabla y_{rank}$)
        ApplyGradient($\nabla_{layer,rank}, \eta$)
```

11.3 Integrated batch and model parallelism implementation

11.4 Github Repo

Here is the link to our code: `github.com/emt13/Model-And-Data-Parallelism-In-Neural-Networks`

Algorithm 3 Integrated batch and model parallelism

```
procedure TrainIntegratedParallelism($training\_data, epochs, miniBatchSize, \eta$)
  $\{layers\} \leftarrow$ small_layers_init(), $\mathcal{L} \leftarrow$ loss_init()
  for $epoch \in \{epochs\}$ do
    if rank $= 0$ then
      $\{miniBatches\} \leftarrow$ CreateMiniBatches($training\_data, miniBatchSize$)

    for $miniBatch \in \{miniBatches\}$ do
      $miniBatch_{batch\_split} \leftarrow$ scatter_broadcast($miniBatch$)
      $previous\_y_{batch\_split} \leftarrow miniBatch_{batch\_split}$
      for $layer \in \{layers\}$ do
        $next\_y_{rank} \leftarrow$ ForwardPass($previous\_y_{batch\_split}$)
        $y_{batch\_split} \leftarrow$ AllGather($next\_y_{rank}$)

      $previous\_\nabla y_{batch\_split} \leftarrow$ Compute$\frac{\partial \mathcal{L}}{\partial \hat{y}}$($\mathcal{L}$, $next\_y_{batch\_split}$)
      for $layer \in reverse(\{layers\})$ do
        $previous\_\nabla y_{rank} \leftarrow$slice($previous\_\nabla y_{batch\_split}$)
        $\nabla_{layer,rank},\ next\_\nabla y_{rank} \leftarrow$ BackwardPass($previous\_\nabla y_{batch\_split}$)
        $\nabla_{layer,model\_split} \leftarrow$ AllReduce($\nabla_{layer,rank}$)
        $next\_\nabla y_{batch\_split} \leftarrow$ AllReduce($next\_\nabla y_{rank}$)
        ApplyGradient($\nabla_{layer,model\_split}, \eta$)
```