Benchmarking data parallel distributed training of deep learning models on PyTorch and TensorFlow

Roshan Ramkeesoon
University of Washington
Data Science
roshanr@uw.edu

ABSTRACT

We vary the dataset size, model size, batch size, and number of GPUs and train using PyTorch DataParallel and TensorFlow MirroredStrategy. We observe TensorFlow has higher processing rates and increased scaleup, but recognize a fairer comparison would be between TensorFlow MirroredStrategy and PyTorch DistributedDataParallel. We observe that GPUs improve performance when models have many parameters and batch size is high.

1. INTRODUCTION

In recent years, deep learning models have become state-of-the-art models for learning from increasing volume and complexity of data. Deep learning models require large volumes of data to extract useful features. This process can require large amounts of time to train -- on the order of weeks. Because of the repetitiveness of some of the computations, distributed computing has presented value in this domain. Distributed deep learning can be divided into two potentially overlapping subcategories: model parallelism and data parallelism.

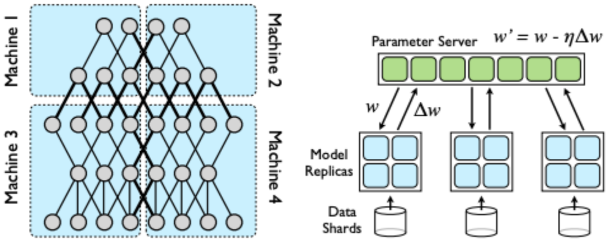


Figure 1. (left) The DistBelief model is an early example of model parallelism. (right) the parameter server method for data parallelism. [1]

1.1 Model Parallelism

In model parallelism, parts of the model sit on different processors. Only the nodes with edges that cross partition boundaries will need to have their state transmitted between machines. If a model is too big to fit on a single GPU, training almost certainly requires some form of model parallelism. A prominent early successful example of data parallelism is Google's DistBelief model [1].

1.2 Data Parallelism

In data parallelism, the training data is divided across processors and each model is replicated on the different machines. Each machine trains its replica on the data contained locally and parameter updates are shared. The method of sharing weight updates is a subject of research. These methods can be performed synchronously or asynchronously, each presenting their own challenges.

In the parameter server method, each processor shares its weight updates with the parameter server. The parameter server is responsible for storing the current global state of the model. Processors write weight updates to the parameter server after training and read the newest weights before training. This method runs into bottleneck issues reading and writing to the parameter server.

A more efficient method is ring all-reduce. This was popularized by Uber Horovod software [4], but is now implemented natively in other packages including TensorFlow. In this method, there is no central parameter server, but the machines form a ring and each machine only listens from one other machine and sends element-wise updates to one other machine.

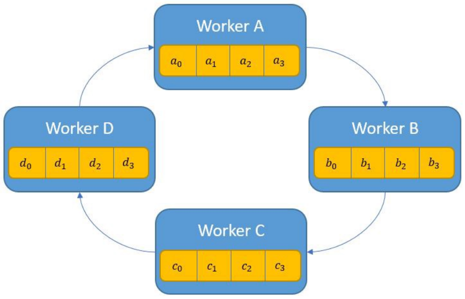


Figure 2. Ring allreduce diagram from Uber Horovod paper. During state transmission phase, elements of the updated states are shared one at a time in a ring formation. [4]

2. EVALUATED SYSTEMS

There are many software packages for performing data parallel distributed deep learning. Two of the most prominent ones are TensorFlow [5] and PyTorch [2], which will be evaluated in this work.

TensorFlow distributed [5] offers a variety of distribution strategies. I evaluate the synchronous MirroredStrategy on the Keras API. In the MirroredStrategy, each GPU receives a portion of the training data as well as a replica of the model. Each GPU trains locally and then communicates variable updates using efficient all-reduce algorithms.

PyTorch offers DataParallel for data parallel training on a single machine with multiple cores. It works similarly to TensorFlow MirroredStrategy where each core contains a replica of the model. However, I observed two key differences. Whereas in TensorFlow MirroredStrategy, each GPU trains on a static shard of data, in PyTorch DataParallel each GPU is served a batch of data on each batch iteration, which may result in increased communication costs. In addition, PyTorch uses a parameter server strategy and does not give options for changing the communication strategy.

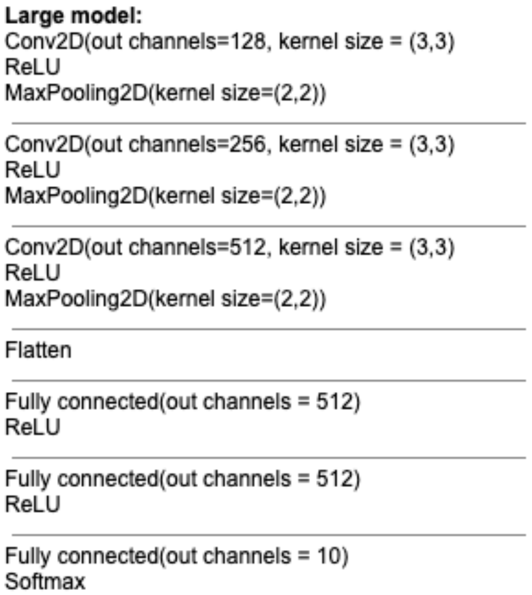
[3] The all-reduce strategy has been seen in prior work to be more efficient than parameter server. [4]

After running my experiments, I later found that PyTorch does have a framework that is expected to be faster than DataParallel called DistributedDataParallel. One optimization is that it shards data before train time instead on each batch iteration. In a future study, it would be appropriate to compare its performance with TensorFlow MirroredStrategy.

3. PROBLEM STATEMENT & METHOD

For my experiments, I trained on the MNIST digits dataset. I varied the number of GPUs, batch size, dataset size, and model size. I recorded the time to train the first epoch and the second epoch. During the first epoch, the model may incur one-time startup costs, for example to shard data. The model will likely have a similar run time during the second epoch and all future epochs because the operations of synchronous data parallel training are deterministic: forward pass, calculate loss, backpropagate error, gradient step, with many resources becoming free between epochs. For this reason, I only record the first and second epoch train time.

I vary the batch size between 128, 256, and 512 images per batch. I use two sizes of model: a smaller model with 402,442 trainable parameters, and a larger model with 2,636,554 trainable parameters. Both use cross entropy loss and adam optimizer with parameters: learning rate=0.001, betas=(0.9, 0.999), eps=1e-07, weight_decay=0, and amsgrad=False.



Large model:
Conv2D(out channels=128, kernel size = (3,3)
ReLU
MaxPooling2D(kernel size=(2,2))

Conv2D(out channels=256, kernel size = (3,3)
ReLU
MaxPooling2D(kernel size=(2,2))

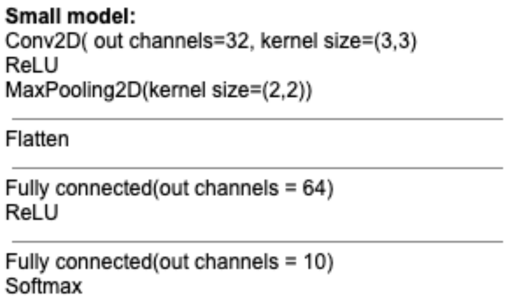
Conv2D(out channels=512, kernel size = (3,3)
ReLU
MaxPooling2D(kernel size=(2,2))

Flatten

Fully connected(out channels = 512)
ReLU

Fully connected(out channels = 512)
ReLU

Fully connected(out channels = 10)
Softmax



Small model:
Conv2D(out channels=32, kernel size=(3,3)
ReLU
MaxPooling2D(kernel size=(2,2))

Flatten

Fully connected(out channels = 64)
ReLU

Fully connected(out channels = 10)
Softmax

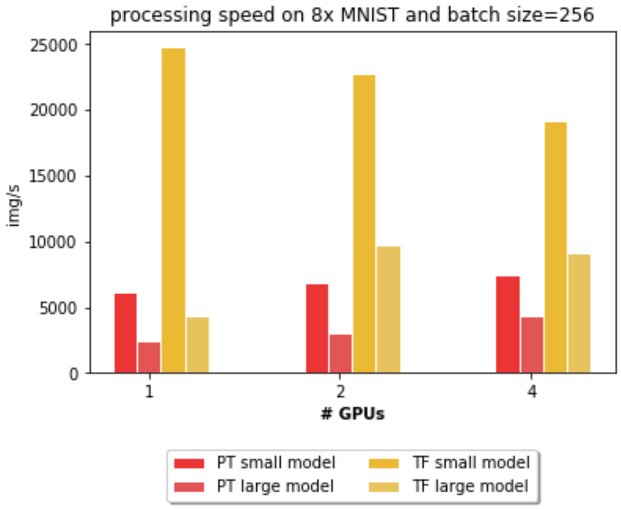
Figure 4. Small and large model architectures for image classification. The small model contains 402,442 trainable parameters and the large model contains 2,636,554 trainable parameters.

The MNIST dataset contains 60k train set images and 10k test set images. When I vary the train set size, I simply concatenate the MNIST dataset to duplicates of itself. I run experiments on 1x MNIST, 4x MNIST, and 8x MNIST which correspond to 60k, 240k, and 480k images.

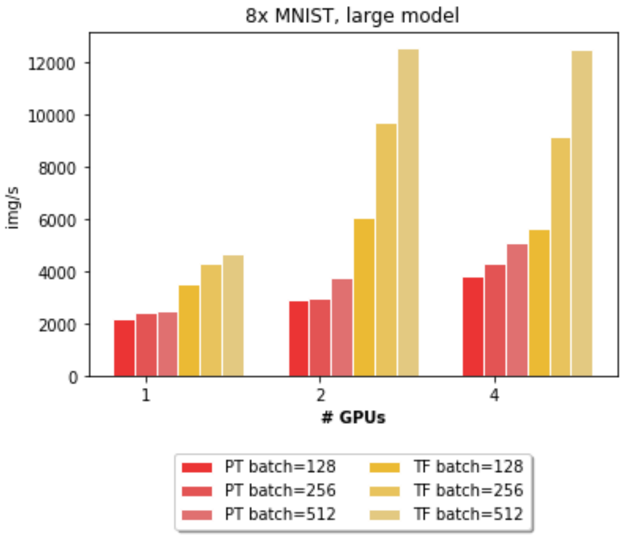
The number of GPUs vary between 1, 2 and 4. All experiments were run on the Google Cloud Platform on machines composed of n1-highmem-2 (2 vCPUs, 13 GB memory) with a variable number of NVIDIA Tesla K80 GPUs. TensorFlow experiments were run on instances with the Google, Intel® optimized Deep Learning Image: TensorFlow 2.0.0. PyTorch experiments were run on instances with Google, Deep Learning Image: PyTorch 1.2.0.

4. RESULTS

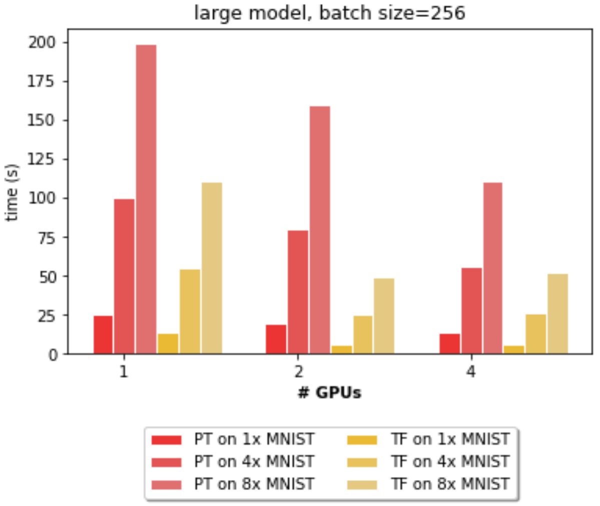
(a)



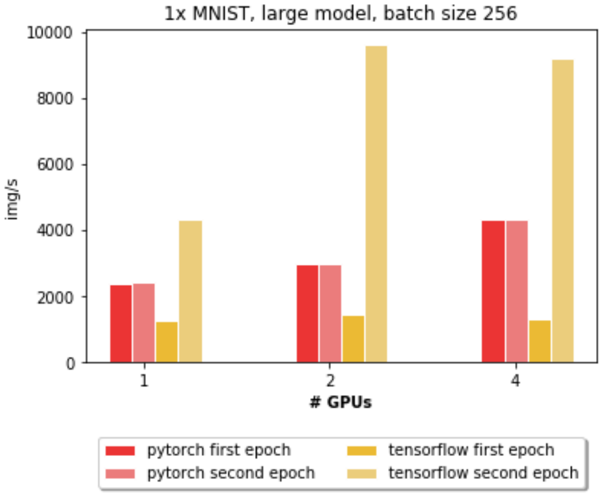
(b)



(c)



(d)



(e)

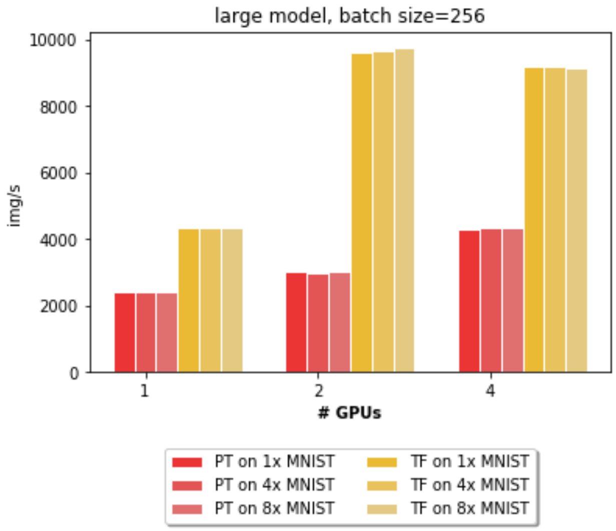


Figure 5. (a) Second epoch processing speed for varied model size. (b) varied batch sizes for large model on 8x MNIST. (c) Second epoch train time with various sizes of dataset. (d) First and second epoch processing rates for large model. (e) Second epoch processing speed for varied dataset size for same model.

In Figure 5a, for TensorFlow on the small model with the 1x MNIST dataset, second epoch processing rate unintuitively decreases as the number of GPUs increase. On PyTorch we see the second epoch processing rate increase with GPU's. The result indicates that for TensorFlow, either the dataset size or model size is so small the overhead from parallelization outweighs the increased speed from parallelization.

We see in Figure 5c that the total second epoch train time decreases with increased GPU's, and that the effect is more pronounced when we use a larger dataset. However we see in Figure 5e that the processing speed remains constant when we only vary the dataset size. This contrasts the results from Figure 5a which show that the processing rate increases with increased GPUs when we increase the model size. This effect appears to become parabolic with optimal performance at 2 GPUs. Perhaps the large model used in this experiment with ~2.6 million parameters is still not large enough to take advantage of data parallelism at 4 GPUs.

In addition we see in Figure 5b that increased batch size increases processing rates for a given number of GPUs but that this effect is more pronounced with increased number of GPUs. Data parallelism works best when the batch size maximizes the memory available on each device. (Batch size is a learning parameter and should not be adjusted without making adjustments to learning rate.) Overall we note that TensorFlow MirroredStrategy outperforms PyTorch DataParallel, and that the different strategies have notable effects on processing speed, especially when comparing first and second epoch training processing speed (Figure 5d), where it appears TensorFlow is slower in the first epoch likely due to data sharding.

5. CONCLUSION

In terms of ease-of-use, it was simpler to implement data parallelism in PyTorch than in TensorFlow. However, we observe that TensorFlow MirroredStrategy has faster processing speeds and scaleup than PyTorch DataParallel. This is likely due in part to its initial slow epoch in which it shards data. However, this may not have been an apples-to-apples comparison and future studies should compare TensorFlow MirroredStrategy to PyTorch DistributedDataParallel.

6. ACKNOWLEDGMENTS

Our thanks to ACM SIGCHI for allowing us to modify templates they had developed. Thanks to all the open source bloggers who provided information on implementing PyTorch and TensorFlows. Thanks to Brandon Haynes and Dan Siciu for all the mentorship.

7. REFERENCES

[1] Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

[2] PyTorch (2019) version 1.2.0. Github repository. https://github.com/pytorch/pytorch

[3] Raschka, Sebastian, and Erick Guan. “How PyTorch's Parallel Raschka, Sebastian, and Erick Guan. “How PyTorch's Parallel Method and Distributed Method Works?” *PyTorch*, Nov. 2018, discuss.pytorch.org/t/how-pytorchs-parallel-method-and-distributed-method-works/30349/16.

[4] Sergeev, Alexander, and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow." *arXiv preprint arXiv:1802.05799* (2018).

[5] TensorFlow (2019) version 2.0.0. Github repository. Code: https://github.com/tensorflow/tensorflow Documentation: https://www.tensorflow.org/guide/distributed_training