Accelerating Model Training from a System Perspective

Ziji Shi

School of Computer Science and Engineering,
Nanyang Technological University

November 29, 2019

Self-Introduction

I am a Research Assistant affiliated to MICL. I graduated from NTU with BEng. in Computer Science. In the past, I have been:

- Large-Scale ML Intern, Apple, Summer 2019
- Algorithm Intern, SenseTime, Fall 2018 - Spring 2019
- Research Assistant, NUS, Summer 2018

I am ex-captain of NTU High-Performance Computing (HPC) team. Visit our web page: www.ntuhpc.org

Outline

Motivation

Large scale machine learning is moving to the distributed setting because of growing size of datasets/models, and modern learning paradigms like Federated learning.

Paradigms of Distributed Training

Model Parallelism

Split models by layers or sub-models and distribute among di erent workers.

In each step, workers use the same batch of data.

Data Parallelism

Split data by random sampling and distribute among workers.

In each step, workers have identical copies of model and di erent data in each step.

Since model parallelism usually has a higher requirement on bandwidth, data parallelism is the mainstream implementation of distributed training.

Data Parallelism

Parameter Server:
separates model serving and computation on di erent nodes. Parameter server is in charge of maintaining an updated copy of model.

Figure: Parameter Server

Allreduce:
uses allreduce protocol to concurrently exchange gradients from other workers. Model is updated locally.

Figure: Allreduce

Data Parallelism

Parameter Server

Figure: SyncSGD - Parameter Server

PS: store the model's weights

Worker:

i. Fetch model weights
ii. Compute the gradient update
iii. Push the gradient back to PS

Data Parallelism

Allreduce

Figure: SyncSGD - Allreduce

1. Compute gradients using a mini-batch on each GPU
2. Compute the mean of the gradients all(reduce) via inter-GPU communication
3. Update the model

Comparison between PS and Allreduce

Test	PS	Allreduce
Pros	More exible (sync. or async.)	E cient
Cons	The PS/worker ratio matters, communication and computation trade-o	Single point of failure, impeded by slowest worker

Table: Comparison between PS and Allreduce

Distributed Optimization Algorithms

Sync SGD

Workers get identical models after each step

E ectively increasing batch size on single worker, thus guaranteeing convergence

Aync SGD

Split data by sampling and distribute among workers

Workers have identical copies of model and di erent data in each step

May have "staleness" problem[a]

---

[a]Dai, Wei, et al. "Toward Understanding Distributed Machine Learning." arXiv prepri

Compilation Strategy

Building with newer dependencies usually yields better performance. For example,

- Horovod
  - NCCL 2.2
  - OpenMPI
- PyTorch[1]
  - CUDA 9.0 or higher
  - cuDNN 7 or higher

[1] https://github.com/pytorch/pytorch#from-source

Compilation Strategy

NCCL

NCCL stands for Nvidia Collective Communication Library. It can e ciently broadcast and aggregate data across di erent GPUs. It schedules a cooperating kernel on each GPU that knows how to best utilize the underlying hardware topology.

Figure: NCCL performance on Tensor ow VGG16. Benchmarked wit
16 V100 GPUs.

Compilation Strategy

CUDA+cuDNN

Figure: CUDA and cuDNN performance on Tensor ow VGG16. Benchmarked with 16 V100 GPUs.

Mixed-Precision Training[2]

FP16: half-precision oating points (16 bits)

FP32: standard 32-bit oating points

Decrease the required amount of memory : Lowering the required memory enables training of larger models or training with larger mini-batches.

Shorten the training or inference time : Execution time can be sensitive to memory or arithmetic bandwidth.

Possibly under ow or over ow

[2] https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html

Mixed-Precision Training

Benchmark

Figure: Mixed-Precision Performance.

Mixed-Precision Training

when to use

Figure: Histogram of activation gradient magnitudes throughout FP32 training of Multibox SSD network. The x-axis is logarithmic, except for the zero entry. For example, 66.8% of values were 0, 4% had magnitude in the $(2^{32}; 2^{30})$ range.

Mixed-Precision Training

when to use

Figure: Histogram of same network, with both x- and y-axes in log scale. When converted to FP16, 31% of these values become zeros leaving only 5.3% as non-zeros which for this network lead to divergence.

Mixed-Precision Training

solution to under ow and over ow

Problems:

- **under ow** updates (weight gradients multiplied by the learning rate) become too small (2^{-24}) to be represented in FP16
- **over ow** if ratio of the weight value to the weight update is very large, it could become zero when addition operation right-shifts it to align the binary point with the weight

Solutions:[3]

- Scale the small losses by shifting n bits (e ectively the same as multiplying by 2^n).
- Keep a FP32 copy of weight parameters

[3] Micikevicius, Paulius, et al. "Mixed precision training." arXiv preprint arXiv:1710.03740 (2017).

Complexity of Allreduce Implementations

n: amount of data to transfer, K: # of nodes

Implementation	Amount of Comm. Per Node	Communication Freq.
Star	2n	2K
Tree	$2n \log_2 K = K$	$2 \log_2 K$
Butter y	$n \log_2 K = K$	$\log_2 K$
Ring	$n = K$	$2(K \quad 1)$

Table: Comparison between allreduce implementations.[4]

[4] Tie-Yan Liu, Wei Chen, Taifeng Wang, and Fei Gao, Distributed Machine Learning, Theories, Algorithms, and Systems, China Machine Press, 2018

Ring-Allreduce

Phase 1

Figure: Scatter-reduce

Ring-Allreduce

Phase 1

Phase 2

Figure: Scatter-reduce

Figure: All-gather

Ring-Allreduce

Phase 1

Phase 2

Figure: Scatter-reduce

Figure: All-gather

The amount of data transferred per process is $2(K-1)n/K$, which means it scales almost linearly w.r.t. K.

Horovod

Horovod is a distributed training framework for TensorFlow, Keras, PyTorch, and MXNet. It implements Ring-Allreduce with NCCL, and is easy to use.

Figure: Horovod Performance[5]

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

Horovod

Tensor Fusion

Tensor Fusion works by attempting to combine all the tensors that are ready to be reduced at given moment of time into one reduction operation.
You can set{fusion-threshold-mbin horovodrunto adjust the threshold bu er size. Default value is 64 MB, however, you can set to 0 if you wish to disable Tensor Fusion.

Horovod

with PyTorch

To use horovod on PyTorch, the main steps are:

1. Initializing Horovod
2. Pinning GPU to Horovod local rank
3. Adding Horovod distributed optimizer
4. Broadcasting parameters to initialize model

See example here.

Which PyTorch Backend to Use?

- Use the NCCL backend for distributed GPU training
- Use the Gloo backend for distributed CPU training.
- GPU hosts with In niBand interconnect

On our cluster, since both In niBand and IPoIB driver are installed, it's recommended to build PyTorch with NCCL and CUDA.[6]

[6] Use NCCL, since it's the only backend that currently supports In niBand and GPUDirect.

Tuning Hyper-parameters

Common practices

Use largest possible batch size

Warm-up training[7]

Explore learning rate schedule (exponential decay, cosine decay...)

[7] Goyal, Priya, et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour." arXiv preprint arXiv:1706.02677 (2017).

Rule of Thumb

Use the whole machine whenever possible.

Avoid communication when possible.

Use parallel/pipelined data loader[8]

Be aware of Non-Uniform Memory Access (NUMA)

- additional communication cost to transfer data from one CPU to another CPU
- use GPUs a liated to the same CPU can avoid the cost
- set CUDA-visible device or process-pinning

[8] https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

Rule of Thumb

Use the whole machine whenever possible.

Avoid communication when possible.

- Use parallel/pipelined data loader[8]
- Be aware of Non-Uniform Memory Access (NUMA)
  - additional communication cost to transfer data from one CPU to another CPU
  - use GPUs a liated to the same CPU can avoid the cost
  - set CUDA-visible device or process-pinning

[8] https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

Rule of Thumb

Use the whole machine whenever possible.

Avoid communication when possible.

Use parallel/pipelined data loader [8]

Be aware of Non-Uniform Memory Access (NUMA)

- additional communication cost to transfer data from one CPU to another CPU
- use GPUs affiliated to the same CPU can avoid the cost
- set CUDA-visible device or process-pinning

[8] https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

Rule of Thumb

Use the whole machine whenever possible.

Avoid communication when possible.

Use parallel/pipelined data loader [8]

Be aware of Non-Uniform Memory Access (NUMA)

- additional communication cost to transfer data from one CPU to another CPU
- use GPUs affiliated to the same CPU can avoid the cost
- set CUDA-visible device or process-pinning

[8] https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html