Distributed Deep Learning

CentOS Dojo at Oak Ridge National Labs
Bryant Nelson
April 16, 2019

Outline

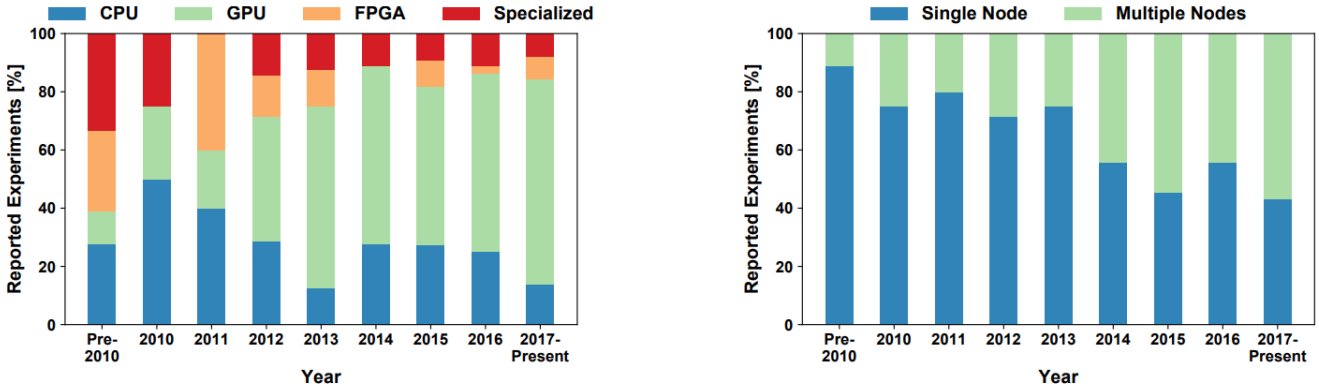
1. Survey of Distributed Deep Learning
   A. Problem Statement
   B. Basic Technology
      - Hyper-Parameter & Architecture Search Parallelism
      - Model Parallelism
      - Data Parallelism
2. Data Parallel Distributed Deep Learning
   A. Description
   B. Tools
      - Tensorflow Distributed Strategy
      - Horovod
      - IBM PowerAI DDL
   C. PowerAI DDL Example
   D. Technical Considerations

SURVEY OF DISTRIBUTED DEEP LEARNING

Survey of Distributed Deep Learning: Problem Statement

- Deep Neural Networks (DNNs) first became relevant with increases in compute power driven largely by the use of GPUs for compute.
- Datasets have increased in size. DNNs have increased in complexity. Both factors have led to increases in memory consumption and computation required to train models.
- With only 4-8 GPUs in a single machine, it has become necessary to distribute training across multiple machines.

Survey of Distributed Deep Learning Basic Technology



(a) Hardware Architectures (b) Training with Single vs. Multiple Nodes

Fig. 3. Parallel Architectures in Deep Learning

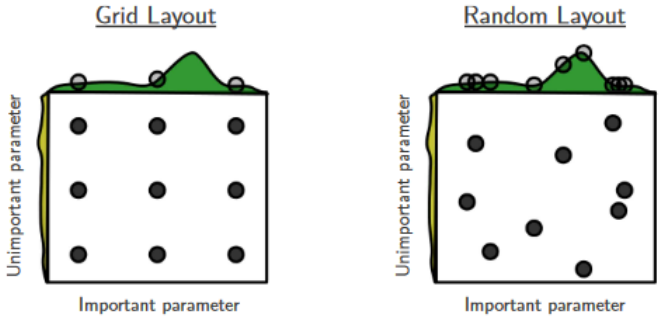
[Ben-Nun et al. 2018] “Fig. 3a shows a summary of the machine architectures used in research papers over the years. We see a clear trend towards GPUs, which dominate the publications beginning from 2013. However, even accelerated nodes are not sufficient for the large computational workload. Fig. 3b illustrates the quickly growing multi-node parallelism in those works”

Survey of Distributed Deep Learning Basic Technology

- Common distribution methodologies:
  - Hyper-parameter search parallelism
    - Different models (architectures/hyperparameters) run in parallel with same data
  - Model Parallelism
    - Model partitioned across multiple machines
  - Data Parallelism
    - Multiple replicas of the model work on different subsets of data

Basic Technology: Hyper-Parameter & Architecture Search Parallelism

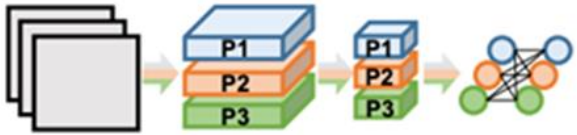
- Different models (different architectures/hyperparameters) run in parallel with same data
- Embarrassingly parallel, though combinations grow exponentially
- Methods of Searching for good values:
  - Random search
  - Grid search: Conduct hyperparameter search at logarithmic scale
  - Probe values known to work well; make educated guesses
  - Spectral methods like Compressed Sensing [Hazan et al. 2018]
  - Hyperband [Li et al. 2017]
- Hyperparameter tuning with Watson Machine Learning Accelerator
- IBM NeuNets automatic network creation.



[Bergstra et al. 2012]

Basic Technology: Model Parallelism - Network Parallelism

- Model Parallelism (aka Network Parallelism)
  - Models are partitioned across multiple devices.
  - Example: DistBelief [Dean et al. 2012]
  - Parallelism within network layers
    - divide neurons of a fully connected layer onto separate devices
    - divide C, H, or W dimensions of a CNN layer onto separate devices
    - copy input minibatch to all devices
    - compute different parts on different devices
      - conserves memory by not putting full network in one place, but adds a communication cost – eg. fully connected layers require all-to-all communication of outputs to next layer
- Not used that much nowadays

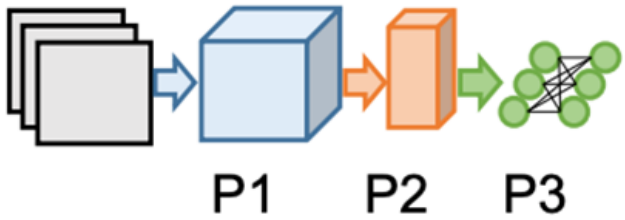


(b) Model Parallelism

[Bergstra et al. 2012]

Basic Technology: Model Parallelism - Layer Pipelining

- Separate NN layers onto separate devices
  - advantages
    - avoids need to store all parameters on all devices (in contrast to network parallelism)
    - fixed number of communication points between nodes (at layer boundaries)
    - weights can be cached since devices always compute the same layers
  - disadvantages
    - inputs have to arrive at a specific rate in order to fully utilize the system
    - latency proportional to the number of devices is incurred
- Google's GPipe [Huang 2019]

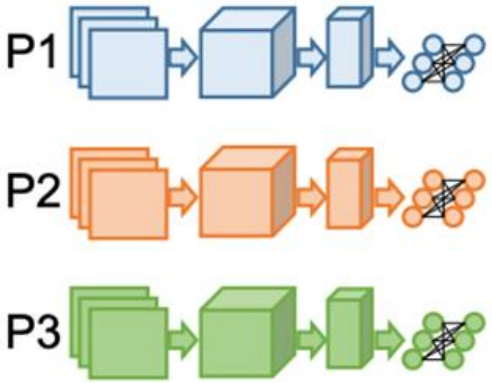


(c) Layer Pipelining

[Bergstra et al. 2012]

Basic Technology: Data Parallelism

- distribute work by dividing samples of a minibatch across multiple devices
- After each minibatch average the gradients on each device to obtain the gradient for the whole minibatch
- all parameters need to be accessible on all devices
- works for most standard layers:
  - activation / convolution / fully connected / pooling



(a) Data Parallelism

[Bergstra et al. 2012]

DATA PARALLEL DISTRIBUTED DEEP LEARNING

Data Parallel Distributed Deep Learning: Description

- Synchronous All-to-All Data-Parallel Distributed GPU Deep Learning
- A process is created for each GPU in the cluster
- Each process contains a complete copy of the model
- Mini-batch is spread across all of the processes
  - Each process uses different input data
- After each iteration, all of the processes sync and average together their gradients, and those averages are used to update the local weights.
- Models on each GPU should always be identical.

Data Parallel Distributed Deep Learning: Tools

- Communication Libraries
  - MPI
  - NCCL
  - IBM PowerAI DDL
- Integrations / Frameworks
  - TensorFlow Distribution Strategies
  - Horovod
  - IBM PowerAI DDL

Data Parallel Distributed Deep Learning: Tools – Communication Libraries

- The following tools are libraries, which provide the communication functions necessary to perform distributed training. Primarily allReduce and broadcast functions.
  - MPI
    - Classic tool for distributed computing.
    - Still commonly used for distributed deep learning.
  - NCCL
    - Nvidia's gpu-to-gpu communication library.
    - Since NCCL2, between-node communication is supported.
  - IBM PowerAI DDL
    - Provides a topology-aware allReduce.
    - Capable of optimally dividing communication across hierarchies of fabrics.
    - Utilizes different communication protocols at different hierarchies.

Data Parallel Distributed Deep Learning: Tools – Integrations / Frameworks

- The following tools are libraries, which provide integrations into deep learning frameworks to enable distributed training using common communication libraries.
  - TensorFlow Distribution Strategies
    - Native Tensorflow distribution methods.
  - Horovod [Sergeev et al. 2018]
    - Provides integration libraries into common frameworks which enable distributed training with common communication libraries, including
  - IBM PowerAI DDL
    - Provides integrations into common frameworks, including a Tensorflow operator that integrates PowerAI DDL with Tensorflow.

Data Parallel Distributed Deep Learning: IBM PowerAI DDL [Cho et al. 2016]

- PowerAI DDL provides:
  - C and Python libraries that provide communication functions.
    - The library utilizes the MPI and NCCL libraries
  - Framework integrations
    - Provides a custom operator for TensorFlow. To use in Tensorflow, only need to 'import ddl'.
    - DDL integration is built into PowerAI's version of Caffe and PyTorch
  - A tool for launching jobs across a cluster called ddlrun, simplifying the launching of distributed jobs.
    - e.g. ddlrun -H server1,server2,server3,server4 python train.py
- DDL's allReduce uses knowledge of the cluster layout to perform reductions between nodes in a certain order
  - DDL attempts to perform reductions between nodes in the order that will cause the lowest communication overhead.
  - It takes into account the fact that not all nodes are connected with the same interface
  - DDL performs best compared to other allreduce libraries when used in a cluster with a non-flat topology.
- Almost linear scaling for 64 machines with 4 GPUs each for ResNet-50. Training time took 50 mins for 90 epochs with a batch of 32 per GPU

Data Parallel Distributed Deep Learning: PowerAI DDL Example

Steps to distribute the training of a tf.keras model:

1. Import the ddl library.
2. Split the training data.
3. Modify hyperparameters.
4. Add callbacks.

We will go through the changes necessary to use DDL to distribute the training of an mnist model in tf.keras.

Original script: https://github.com/keras-team/keras/blob/4f2e65c385d60fa87bb143c6c506cbe428895f44/examples/mnist_cnn.py

Data Parallel Distributed Deep Learning: PowerAI DDL Example

1. Import the ddl library.

This is the only ***necessary*** step to enable distributed training with DDL.

```
| from tensorflow.python import keras as keras
| from tensorflow.python.keras.datasets import mnist
| from tensorflow.python.keras.models import Sequential
| from tensorflow.python.keras.layers import Dense, Dropout, Flatten
| from tensorflow.python.keras.layers import Conv2D, MaxPooling2D
| from tensorflow.python.keras import backend as K
> import ddl
> import numpy as np
```

Data Parallel Distributed Deep Learning: PowerAI DDL Example

2. Split the training data.

- If training works by iterating over all of the data, each process should only iterate over equal sections of the data.
- If training works by grabbing random data, modifications may not be necessary, although it should be verified that a different seed is being used for each process

Data Parallel Distributed Deep Learning: PowerAI DDL Example

2. Split the training data.

```
> # DDL: Save the full test data before splitting for final accuracy check.
> x_test_full = x_test.astype('float32') / 255
> y_test_full = keras.utils.to_categorical(y_test, num_classes)
>
> # DDL: Split the training & testing data.
> x_train = np.array_split(x_train, ddl.size())[ddl.rank()]
> x_test = np.array_split(x_test, ddl.size())[ddl.rank()]
  x_train = x_train.astype('float32')
  x_test = x_test.astype('float32')
  x_train /= 255
  x_test /= 255
  print('x_train shape:', x_train.shape)
  print(x_train.shape[0], 'train samples')
  print(x_test.shape[0], 'test samples')

> # DDL: Split the training & testing data.
> y_train = np.array_split(y_train, ddl.size())[ddl.rank()]
> y_test = np.array_split(y_test, ddl.size())[ddl.rank()]
```

Data Parallel Distributed Deep Learning: PowerAI DDL Example

3. Modify hyperparameters.

- In this example the only hyperparameter we change is the learning rate.
  - We scale the learning rate by the total number of “learners” to offset the effect of the larger global batch size.

```
> # DDL: adjust learning rate based on number of GPUs.
  model.compile(loss=keras.losses.categorical_crossentropy,
|               optimizer=keras.optimizers.Adadelta(lr=1.0 * ddl.size()),
                metrics=['accuracy'])
```

Data Parallel Distributed Deep Learning: PowerAI DDL Example

4. Add callbacks.

- DDL provides two tf.keras callbacks.
  - ddl.DDLCallback() is responsible for synchronizing keras metrics
    - Should always be the first callback in the callbacks list.
  - ddl.DDLGlobalVariablesCallback() is responsible for initializing global variables to the same values across all learners
    - Should always be the last callback in the callbacks list.

```
> callbacks = list()
>
> # DDL: Add the DDL callback.
> callbacks.append(ddl.DDLCallback())
> callbacks.append(ddl.DDLGlobalVariablesCallback())
```

Data Parallel Distributed Deep Learning: PowerAI DDL Example

- Execution
- DDL provides a utility called DDLRUN which is used to launch the learning job on any number of nodes/gpus.

```
(demo) [bnelson@dlw12 ~]$ ddlrun -H dlw03  python \
~/anaconda3/envs/demo/tf_cnn_benchmarks/tf_cnn_benchmarks.py --variable_update=ddl \
--model=resnet50 --num_gpus=1 --batch_size=32
...
----------------------------------------------------------------
total images/sec: 1248.06
----------------------------------------------------------------


(demo) [bnelson@dlw12 ~]$ ddlrun -H dlw04,dlw05,dlw06,dlw07,dlw08,dlw09,dlw10,dlw11,dlw12,dlw13 \
python ~/anaconda3/envs/demo/tf_cnn_benchmarks/tf_cnn_benchmarks.py --variable_update=ddl \
--model=resnet50 --num_gpus=1 --batch_size=32
...
----------------------------------------------------------------
total images/sec: 12043.78
----------------------------------------------------------------
```

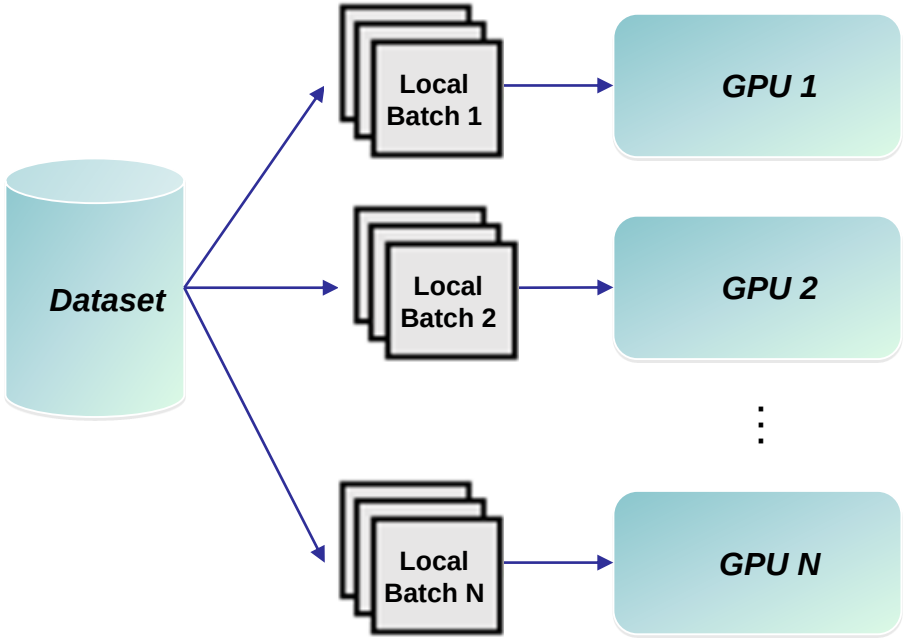
Data Parallel Distributed Deep Learning: Technical Considerations

- Batch Size
- Learning Rate
- Batch Normalization
- On-The-Fly Validation
- Data Pipelining

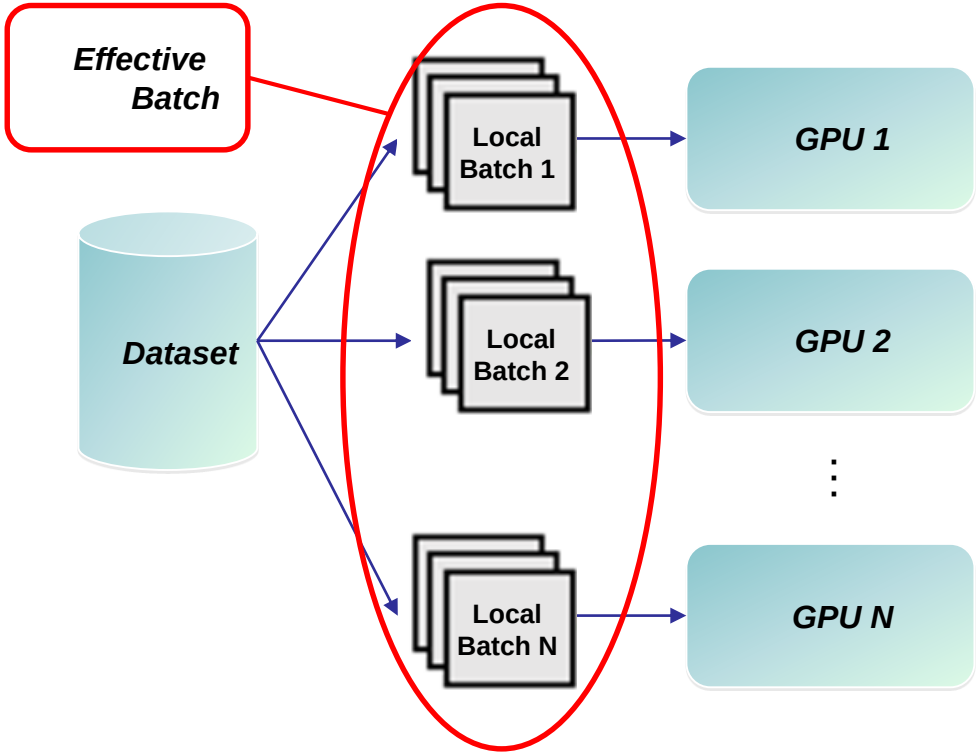
Data Parallel Distributed Deep Learning: Technical Considerations – Batch Size

- The accuracy of a model is often sensitive to changes in the batch size.
- In data parallel distributed training the effective batch size is equal to the local batch size * number of learners.

Data Parallel Distributed Deep Learning: Technical Considerations – Batch Size



Data Parallel Distributed Deep Learning: Technical Considerations – Batch Size



Data Parallel Distributed Deep Learning: Technical Considerations – Batch Size

- The accuracy of a model is often sensitive to changes in the batch size.
- In data parallel distributed training the effective batch size is equal to the local batch size * number of learners.

- With a large number of learners the batch size can quickly become large enough to affect accuracy convergence.
- It is possible to reduce the batch size per learner (at least to one input per batch) but there is a performance trade-off with under-utilized GPUs.

Data Parallel Distributed Deep Learning: Technical Considerations – Learning Rate

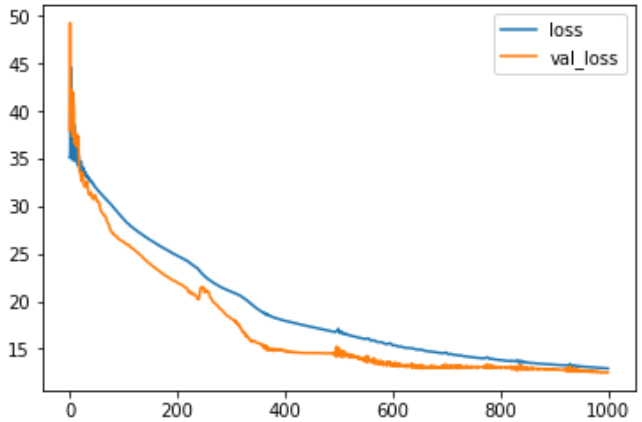
- The learning rate can be used to offset the negative impact of a larger effective batch size.
- Scaling the learning rate up by a factor of the number of GPUs speeds up learning, similar to the way that a smaller mini batch size speeds up learning.
- At some point a larger learning rate negatively impacts learning convergence.

Data Parallel Distributed Deep Learning: Technical Considerations – Batch Normalization

- Data parallel distributed training does not work for batch normalization
  - Batch norm operates on multiple samples(the entire global batch) at the same time.
    - The overhead of fully synchronizing the mean and variance for the global minibatch quickly becomes unfeasible.
  - Alternative 1:
    - Do BN on small subsets (eg. The local minibatch) so these subsets can be normalized locally which increases scaling
    - This is the default behavior in most frameworks.
  - Alternative 2:
    - Instead of BN, use alternate normalization techniques such as weight normalization

Data Parallel Distributed Deep Learning: Technical Considerations – On-The-Fly Validation

- Most frameworks support some method of on-the-fly validation.
- If the validation is not also distributed, it will quickly bottleneck the training.



Data Parallel Distributed Deep Learning: Technical Considerations – Data Pipelining

- On-the-fly data processing can bottleneck distributed training.
- If the pre-processing of the input data was tuned to use the total available CPU processing when training on a single GPU, and that training is distributed to 6 GPUs in the box, then the GPUs will spend time waiting on the CPUs to finish the pre-processing.

References

- [Ben-Nun et al. 2018] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. arXiv:1802.09941v2 https://arxiv.org/pdf/1802.09941.pdf.
- [Bergstra et al. 2012] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. In Journal of Machine Learning Research 13 (2012) 281-305.
- [Hazan et al. 2018] Elad Hazan, Adam Klivans, and Yang Yuan. 2018. Hyperparameter optimization: a spectral approach. In International Conference on Learning Representations (ICLR). https://arxiv.org/abs/1706.00764.
- [Li et al. 2017] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. International Conference onLearning Representations, 2017.
- [Dean et al. 2012] Dean, Jeffrey, Gregory S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang and Andrew Y. Ng. Large Scale Distributed Deep Networks. In NIPS 2012
- [Cho et al. 2017] M. Cho, U. Finkler, S. Kumar, D. Kung, V. Saxena, and D. Sreedhar, “PowerAI DDL”, arXiv preprint arXiv:1708.02188, 2017 https://arxiv.org/abs/1708.02188
- [Huang 2019] Yanping Huang, “Introducing GPipe, an Open Source Library for Efficiently Training Large-scale Neural Network Models”, March 4, 2019 https://ai.googleblog.com/2019/03/introducing-gpipe-open-source-library.html
- [Sergeev et al. 2018] Alexander Sergeev and Mike Del Balso, “Horovod: fast and easy distributed deep learning in TensorFlow”, arXiv:1802.05799, 2018 http://arxiv.org/abs/1802.05799
- https://developer.ibm.com/linuxonpower/2018/09/19/distribute-tensorflow-keras-training-ddl/

Thank You

Bryant Nelson
—
bryant.nelson@ibm.com
http://www.ibm.com