Data Parallel Deep Learning

Huihuo Zheng

Argonne Leadership Computing Facility
August 7, 2020

huihuo.zheng@anl.gov

www.anl.gov
The need for distributed training on HPC

“Since 2012, the amount of compute used in the largest AI training runs has been increasing exponentially with a 3.5 month doubling time (by comparison, Moore’s Law had an 18 month doubling period).”

https://openai.com/blog/ai-and-compute/

Tal Ben-Nun and Torsten Hoefler, arXiv:1802.09941
## Distributed deep learning for ResNet-50

The table below shows the training time and top-1 validation accuracy with ResNet-50 on ImageNet:

<table>
<thead>
<tr>
<th></th>
<th>Batch Size</th>
<th>Processor</th>
<th>DL Library</th>
<th>Time</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>He et al. [1]</td>
<td>256</td>
<td>Tesla P100 × 8</td>
<td>Caffe</td>
<td>29 hours</td>
<td>75.3 %</td>
</tr>
<tr>
<td>Goyal et al. [2]</td>
<td>8,192</td>
<td>Tesla P100 × 256</td>
<td>Caffe2</td>
<td>1 hour</td>
<td>76.3 %</td>
</tr>
<tr>
<td>Smith et al. [3]</td>
<td>8,192 → 16,384</td>
<td>full TPU Pod</td>
<td>TensorFlow</td>
<td>30 mins</td>
<td>76.1 %</td>
</tr>
<tr>
<td>Akiba et al. [4]</td>
<td>32,768</td>
<td>Tesla P100 × 1,024</td>
<td>Chainer</td>
<td>15 mins</td>
<td>74.9 %</td>
</tr>
<tr>
<td>Jia et al. [5]</td>
<td>65,536</td>
<td>Tesla P40 × 2,048</td>
<td>TensorFlow</td>
<td>6.6 mins</td>
<td>75.8 %</td>
</tr>
<tr>
<td>Ying et al. [6]</td>
<td>65,536</td>
<td>TPU v3 × 1,024</td>
<td>TensorFlow</td>
<td>1.8 mins</td>
<td>75.2 %</td>
</tr>
<tr>
<td>Mikami et al. [7]</td>
<td>55,296</td>
<td>Tesla V100 × 3,456</td>
<td>NNL</td>
<td>2.0 mins</td>
<td>75.29 %</td>
</tr>
<tr>
<td>Yamazaki et al.</td>
<td>81,920</td>
<td><strong>Tesla V100 × 2,048</strong></td>
<td><strong>MXNet</strong></td>
<td><strong>1.2 mins</strong></td>
<td><strong>75.08 %</strong></td>
</tr>
</tbody>
</table>

Quoted from Masafumi Yamazaki, arXiv:1903.12650
The need for distributed training on HPC

- Increase of model complexity leads to dramatic increase of the amount of computation;
- Increase of the size of dataset makes sequentially scanning the whole dataset increasingly impossible;
- The increase in computational power has been mostly coming (and will continue to come) from parallel computing;
- Coupling of deep learning to traditional HPC simulations might require distributed training and inference.

Examples of scientific large scale deep learning

- Thorsten Kurth, Exascale Deep Learning for Climate Analytics, arXiv:1810.01993 (Gordon Bell Prize)
- W. Dong et al, Scaling Distributed Training of Flood-Filling Networks on HPC Infrastructure for Brain Mapping, arXiv:1905.06236
Outline

• Different parallelisms for distributed training
• Mini-batch stochastic gradient descent
• Data parallel training using Horovod
• Hands-on examples
  • https://github.com/argonne-lcf/ATPESC_MachineLearning
Parallelization schemes for distributed deep learning

Worker 4
Worker 3
Worker 2
Worker 1

Model parallelism

import torch
import torch.nn as nn
import torch.optim as optim

class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = torch.nn.Linear(10, 10).to('cuda:0')
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 5).to('cuda:1')

    def forward(self, x):
        x = self.relu(self.net1(x.to('cuda:0')))
        return self.net2(x.to('cuda:1'))

model = ToyModel()
loss_fn = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)

optimizer.zero_grad()
outputs = model(torch.randn(20, 10))
labels = torch.randn(20, 5).to('cuda:1')
loss_fn(outputs, labels).backward()
optimizer.step()

PyTorch multiple GPU model parallelism within a node

https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html
Parallelization schemes for distributed deep learning

### Pipeline parallelization

- Partition model layers into multiple groups (stages) and place them on a set of inter-connected devices.
- Each input batch is further divided into multiple micro-batches, which are scheduled to run over multiple devices in a pipelined manner.

### Pipeline libraries:

- **GPipe**: arXiv:1811.06965
- **Pipe-torch**:
  - DOI: 10.1109/CBD.2019.00020
- **PipeDream**: arXiv:1806.03377
- **DAPPLE**: arXiv:2007.01045
- **PyTorch Distributed RPC Frameworks**:
  - [https://pytorch.org/tutorials/intermediate/dist_pipeline_parallel_tutorial.html](https://pytorch.org/tutorials/intermediate/dist_pipeline_parallel_tutorial.html)
Parallelization schemes for distributed deep learning

  
  https://keras.io/guides/distributed_training/

- PyTorch Distributed Training
  (torch.nn.parallel.DistributedDataParallel)
  https://leimao.github.io/blog/PyTorch-Distributed-Training/

- Horovod – Distributed training framework for TensorFlow, Keras, PyTorch, and Apache MxNet
Mini-batch stochastic gradient descent

Minimizing the loss: \[ L(w) = \frac{1}{|X|} \sum_{x \in X} l(x, w). \]

**Stochastic Gradient Descent**

1. for \( t = 0 \) to \( T \) do
2. \( z \leftarrow \text{Random element from } S \)
3. \( g \leftarrow \nabla \ell(w^{(t)}, z) \)
4. \( w^{(t+1)} \leftarrow w^{(t)} + u(g, w^{(0,\ldots,t)}, t) \)
5. end for

**Mini-batch Gradient Descent**

\[ w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in B} \nabla l(x, w_t) \]

Learning rate Mini-batch

Batch gradient descent Mini-batch gradient Descent
Stochastic gradient descent
Minibatch stochastic gradient descent

\[ w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in B} \nabla l(x, w_t) \]

How to choose the minibatch size \( n \)?

Minibatch Size Effect on Accuracy and Performance
Tal Ben-Nun and Torsten Hoefler, arXiv:1802.09941

Validation error for different minibatch size for Resnet50
Large mini-batch size training tends to be trapped at local minimum with lower testing accuracy (generalize worse).

<table>
<thead>
<tr>
<th>Name</th>
<th>Training Accuracy</th>
<th>Testing Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SB</td>
<td>LB</td>
</tr>
<tr>
<td>$F_1$</td>
<td>99.66% ± 0.05%</td>
<td>99.92% ± 0.01%</td>
</tr>
<tr>
<td>$F_2$</td>
<td>99.99% ± 0.03%</td>
<td>98.35% ± 2.08%</td>
</tr>
<tr>
<td>$C_1$</td>
<td>99.89% ± 0.02%</td>
<td>99.66% ± 0.2%</td>
</tr>
<tr>
<td>$C_2$</td>
<td>99.99% ± 0.04%</td>
<td>99.99% ± 0.01%</td>
</tr>
<tr>
<td>$C_3$</td>
<td>99.56% ± 0.44%</td>
<td>99.88% ± 0.30%</td>
</tr>
<tr>
<td>$C_4$</td>
<td>99.10% ± 1.23%</td>
<td>99.57% ± 1.84%</td>
</tr>
</tbody>
</table>

"... large-batch ... converge to sharp minimizers of the training function ... In contrast, small-batch methods converge to flat minimizers"

Performance of small-batch (SB) and large-batch (LB) variants of ADAM on the 6 networks

Keskar et al, arXiv:1609.04836
Challenges with large mini-batch training

Predicted critical maximum batch size beyond which the model does not perform well.

Data parallel training

Single worker --> N worker
- Mini-batch size increases by N times so that aggregate throughput increases linearly.
- Learning rate should increase proportionally (warmup steps with smaller learning rate might be needed)

- Gradients are aggregated over all the workers through MPI_Allreduce

Time-to-solution decreases as number of steps reduces.

Data parallel training with Horovod

https://eng.uber.com/horovod/
Data parallel training with Horovod

How to change a series code into a data parallel code:

• Import Horovod modules and initialize horovod
• Wrap optimizer in hvd.DistributedOptimizer & scale the learning rate by number of workers
• Broadcast the weights from worker 0 to all the workers
• Worker 0 saves the check point files
• Data loading:
  • Option 1. All the workers scan through the whole dataset in a random way, and decrease the number of steps per epoch by N.
  • Option 2. Divide the dataset and each worker only scans through a subset of dataset.

https://eng.uber.com/horovod/
import tensorflow as tf
import horovod.tensorflow as hvd
layers = tf.contrib.layers
learn = tf.contrib.learn
def main():
    # Horovod: initialize Horovod.
    hvd.init()
    # Download and load MNIST dataset.
    mnist = learn.datasets.mnist.read_data_sets('MNIST-data-%d' % hvd.rank())
    # Horovod: adjust learning rate based on number of GPUs.
    opt = tf.train.RMSPropOptimizer(0.001 * hvd.size())
    # Horovod: add Horovod Distributed Optimizer
    opt = hvd.DistributedOptimizer(opt)
    hooks = [
        hvd.BroadcastGlobalVariablesHook(0),
        tf.train.StopAtStepHook(last_step=20000 // hvd.size()),
        tf.train.LoggingTensorHook(tensors={'step': global_step, 'loss': loss},
                                    every_n_iter=10),
    ]
    checkpoint_dir = './checkpoints' if hvd.rank() == 0 else None
    with tf.train.MonitoredTrainingSession(checkpoint_dir=checkpoint_dir, hooks=hooks, config=config) as mon_sess

More examples can be found in https://github.com/uber/horovod/blob/master/examples/
import tensorflow as tf
import horovod.tensorflow as hvd

hvd.init()

# Horovod: adjust learning rate based on number of GPUs.
opt = tf.optimizers.Adam(0.001* hvd.size())

def training_step(images, labels, first_batch):
    with tf.GradientTape() as tape:
        probs = mnist_model(images, training=True)
        loss_value = loss(labels, probs)

    # Horovod: add Horovod Distributed GradientTape.
    tape = hvd.DistributedGradientTape(tape)
    grads = tape.gradient(loss_value, mnist_model.trainable_variables)
    opt.apply_gradients(zip(grads, mnist_model.trainable_variables))

    if first_batch:
        hvd.broadcast_variables(mnist_model.variables, root_rank=0)
        hvd.broadcast_variables(opt.variables(), root_rank=0)

    return loss_value

for batch, (images, labels) in enumerate(dataset.take(10000 // hvd.size())):
    loss_value = training_step(images, labels, batch == 0)
    if hvd.rank() == 0 and batch % 10 == 0:
        checkpoint.save(checkpoint_dir)
import torch.nn as nn
import horovod.torch as hvd
hvd.init()
train_dataset = datasets.MNIST('datasets', train=True, download=True,
                               transform=transforms.Compose([transforms.ToTensor(),
                                                             transforms.Normalize((0.1307,), (0.3081,))]))
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset, num_replicas=hvd.size(), rank=hvd.rank())
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=args.batch_size, sampler=train_sampler, **kwargs)

# Horovod: broadcast parameters.
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
# Horovod: scale learning rate by the number of GPUs.
optimizer = optim.SGD(model.parameters(), lr=args.lr * hvd.size(),
                      momentum=args.momentum)
# Horovod: wrap optimizer with DistributedOptimizer.
optimizer = hvd.DistributedOptimizer(
    optimizer, named_parameters=model.named_parameters())

More examples can be found in https://github.com/uber/horovod/blob/master/examples/
import keras
import tensorflow as tf
import horovod.keras as hvd

# Horovod: initialize Horovod.
hvd.init()

# Horovod: adjust learning rate based on number of GPUs.
opt = keras.optimizers.Adadelta(1.0 * hvd.size())

# Horovod: add Horovod Distributed Optimizer.
opt = hvd.DistributedOptimizer(opt)
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=opt,
              metrics=['accuracy'])

callbacks = [
    # Horovod: broadcast initial variable states from rank 0 to all other processes.
    hvd.callbacks.BroadcastGlobalVariablesCallback(0),
]

# Horovod: save checkpoints only on worker 0 to prevent other workers from corrupting them.
if hvd.rank() == 0:
    callbacks.append(keras.callbacks.ModelCheckpoint('./checkpoint-{epoch}.h5'))
model.fit(x_train, y_train, batch_size=batch_size,
           callbacks=callbacks,
           epochs=epochs, steps_per_epochs=num_samples//batch_size//hvd.size(),
           verbose=1, validation_data=(x_test, y_test))

More examples can be found in https://github.com/uber/horovod/blob/master/examples/
Scaling TensorFlow using Data parallelism on Theta @ ALCF: fixing local batch size = 512

AlexNet

ResNet-50

Inception V3
I/O and data management in distributed deep learning

Streaming I/O provided by frameworks

• TensorFlow Data Pipeline
• PyTorch Data Loader
• Keras DataGenerator

Some suggestions for large scale training

• Organize your dataset in a reasonable way (file per sample shall be avoided if the file is too small; share file performs poorly in some file system, e.g., Lustre)
• Parallel IO might be needed at large scale
• Shuffling in the memory instead of in I/O
• Taking advantage of the node-local storage on a system, for example, SSD @ Theta, Burst buffer @ Summit

We have developed I/O profiling library, VaniDL for analyzing DL I/O on HPC. Contact us if you want to know more.
Hands on session

I. Running on Google's Colaboratory Platform
https://github.com/argonne-lcf/ATPESC_MachineLearning/
DataParallelDeepLearning/google_collab.ipynb

II. Running on Theta
Thank you!

huihuo.zheng@anl.gov