Scaling Deep Learning Training

Mustafa Mustafa, Steven Farrell
and Thorsten Kurth
NERSC

Data Analytics Tutorial
CUG19 – Montreal, Canada
Overview

● Foundations of Distributed Training
  ● motivation
  ● training parallelization strategies
  ● large batch training
    ● learning rate scaling
    ● batch size scaling
    ● generalization gap

● Hands-On
  ● dataset introduction
  ● Horovod and CPE ML
  ● Some remarks for improving accuracy
  ● play around…
Why do we need to scale deep learning applications?

- Rapid prototyping/model evaluation
- Problem scale

- Volume of scientific datasets can be large
- Scientific datasets can be complex (multivariate, high dimensional)
Why do we need to scale deep learning applications?

Models get bigger and more compute intensive as they tackle more complex tasks.

“... total amount of compute, in petaflop/s-days, that was used to train selected results ... A petaflop/s-day (pfs-day) = ... \(10^{15}\) neural net operations per second for one day, or a total of about \(10^{20}\) operations.” -- OpenAI Blog
Parallelism strategies

Data Parallelism
Distribute input samples.

Model Parallelism
Distribute network structure (layers).

Layer Pipelining
Partition by layer.

Fig. credit: Ben-Nun and Hoefler arXiv:1802.09941
Data parallelism, synchronous Updates

Gradients are computed locally and summed across nodes. Updates are propagated to all nodes:

- stable convergence
- scaling is not optimal because all nodes have to wait for reduction to complete
- global (effective) batch size grows with number of nodes

Synchronous SGD, decentralized
Data parallelism, asynchronous Updates

Gradients are sent to parameters server. Parameters servers incorporates gradients into model as they arrive and sends back the updated model

- nodes don’t wait (perfect scaling)
- resilient
- stale gradients impact convergence rate (depends on #workers)
- parameter server is a bottleneck
Data parallelism, stale-synchronous Updates (pipelining)

Current gradients are computed and pushed into queue while at the same time, older gradients are popped from the queue and reduced across all the nodes synchronously.

- better scaling than fully synchronous (especially on heterogeneous systems)
- not as extreme as fully asynchronous
- convergence can be negatively impacted if lag (=number of steps between reduced and current gradients) is large
- not resilient but smoothens runtime variability
Large-Batch Training (LBT), synchronous weak scaling

- applies to SGD-type algorithms
  - data batch per node. Model updates are computed independently
  - updates are collectively summed and applied to the local model

Local batch-size = B
Global batch-size = N * B
Stochastic Gradient Descent (SGD)

\[ w_{t+1} \leftarrow w_t - \frac{\eta}{B} \sum_{i=1}^{B} \nabla L(x_i, w_t) \]

\( N \) is total sample size

\( B \) is batch-size

\( \eta \) is learning rate

\( \Delta w \) is the parameter update in one gradient descent step
Linear learning-rate scaling

Upper: 3 SGD steps w. learning-rate = \(\eta\)
Lower: 1 SGD step w. learning-rate = \(3 \times \eta\)

\(\eta \rightarrow N \times \eta\)
Linear learning-rate scaling

\[ w_{t+1} \leftarrow w_t - \frac{\eta}{B} \left( \sum_{i=1}^{B} \nabla L(x_i, w_t) + \sum_{j=1}^{B} \nabla L(x_j, w_{t+1}) \right) \]

\[ w_{t+1} \leftarrow w_t - \frac{\eta_2}{2B} \sum_{i=1}^{2B} \nabla L(x_i, w_t) \]

Where:

\[ \eta_2 = 2 \times \eta \]

Assumption:

\[ \nabla L(x_j, w_{t+1}) \approx L(x_j, w_t) \]

Upper: 3 SGD steps w. learning-rate = \eta

Lower: 1 SGD step w. learning-rate = 3 \times \eta
Sqrt learning-rate scaling

\[ \eta \rightarrow \sqrt{N} \times \eta \]

Motivated by the observation that the variance of the gradient scales with 1/batch-size:

\[ \text{cov}(\Delta w, \Delta w) \approx \frac{\eta^2}{B} \left( \frac{1}{N} \sum_{i=1}^{N} g_i g_i^T \right) \]
Learning-rate scaling

In practice, we see anywhere between sub-sqrt (e.g. You et al. [arXiv:1708.03888]) to linear scaling (e.g. Goyal et al. [arXiv:1706.02677]).

Recent OpenAI ([arXiv:1812.06162]) study has illuminated the dependence of optimal learning-rate on batchsize:

![SVHN (SGD) - Optimal Learning Rate](image)

Fig. McCandlish, Kaplan and Amodei [arXiv:1812.06162]
Challenges with Large Batch Training

- Training with large learning rates is not stable in the initial stages of the training.
  \[ \nabla L(w_{t+1}) \approx \nabla L(w_t) \] 
  assumption breaks when parameters are changing rapidly.
- A generalization gap appears: networks trained with small batches tend to optimize and generalize better.

![Graph showing Top-1 Test Accuracy vs Epochs for AlexNet-BN for ImageNet with batch sizes 512 and 8192.]

<table>
<thead>
<tr>
<th>Batch</th>
<th>Base LR</th>
<th>Accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.02</td>
<td>60.2</td>
</tr>
<tr>
<td>4096</td>
<td>0.16</td>
<td>58.1</td>
</tr>
<tr>
<td>4096</td>
<td>0.18</td>
<td>58.9</td>
</tr>
<tr>
<td>4096</td>
<td>0.21</td>
<td>58.5</td>
</tr>
<tr>
<td>4096</td>
<td>0.30</td>
<td>57.1</td>
</tr>
<tr>
<td>8192</td>
<td>0.23</td>
<td>57.6</td>
</tr>
<tr>
<td>8192</td>
<td>0.30</td>
<td>58.0</td>
</tr>
<tr>
<td>8192</td>
<td>0.32</td>
<td>57.7</td>
</tr>
<tr>
<td>8192</td>
<td>0.41</td>
<td>56.5</td>
</tr>
</tbody>
</table>

AlexNet You et al. [arXiv:1708.03888](https://arxiv.org/abs/1708.03888)
Challenges with Large Batch Training

- Training with large learning rates is not stable in the initial stages of the training. The assumption $\nabla L(w_{t+1}) \approx \nabla L(w_t)$ breaks when parameters are changing rapidly.

- A generalization gap appears: networks trained with small batches tend to optimize and generalize better.

### Scaling generalization gap

<table>
<thead>
<tr>
<th>Batch</th>
<th>Base LR</th>
<th>Accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.02</td>
<td>60.2</td>
</tr>
<tr>
<td>4096</td>
<td>0.16</td>
<td>58.1</td>
</tr>
<tr>
<td>4096</td>
<td>0.18</td>
<td>58.9</td>
</tr>
<tr>
<td>4096</td>
<td>0.21</td>
<td>58.5</td>
</tr>
<tr>
<td>4096</td>
<td>0.30</td>
<td>57.1</td>
</tr>
<tr>
<td>8192</td>
<td>0.23</td>
<td>57.6</td>
</tr>
<tr>
<td>8192</td>
<td>0.30</td>
<td>58.0</td>
</tr>
<tr>
<td>8192</td>
<td>0.32</td>
<td>57.7</td>
</tr>
<tr>
<td>8192</td>
<td>0.41</td>
<td>56.5</td>
</tr>
</tbody>
</table>

Sources:
Explaining the generalization gap?

“... large-batch ... converge to sharp minimizers of the training function ... In contrast, small-batch methods converge to flat minimizers” -- Keskar et al, arXiv:1609.04836

Conceptual sketch of sharp and flat minimas of a loss function

Fig. credit: Keskar et al, arXiv:1609.04836
Explaining the generalization gap?

Loss at the end of training CIFAR-10 (axes are dominant eigenvectors of the Hessian)

Z. Yao et al. arXiv:1802.08241
Explaining the generalization gap?

Hessian top-20 eigenvalues. Larger batchsize converge to points with higher spectrum.

Z. Yao et al. arXiv:1802.08241
ResNet-50 ImageNet in 1 hour

FaceBook scaling result in 2017, batch-size=8k (using 256 GPUs):

- **Linear learning-rate warm-up** over 5 epochs to target rate
- **Linear scaling of learning-rate** \((N \times \eta)\) followed by original decay schedule
- The paper also clarifies subtleties and common pitfalls in distributed training


This scheme breaks down beyond batch-size = 8k for ResNet on ImageNet
Don’t decay the learning-rate, increase batch-size

Smith et al. arXiv:1711.00489 use batch-size scaling to train on ImageNet in 2500 parameter updates. Starting at batch-size 8k and scaling to 80k!

Inception-ResNet-V2 on ImageNet. Multiple runs to illustrate variance.
Adaptive batch-size scaling with 2nd-order information (ABSA)

Z. Yao et al. arXiv:1810.01021 close the generalization gap for a wide range of architectures on image classification tasks, using

- 2nd-order info. (~ loss surface curvature) to adaptively increase the batch-size
- adversarial training to regularize against sharp-minima

ABS and ABSA with ResNet-18 on ImageNet dataset with up to 16k batch-size
ImageNet/ResNet-50 Training in 224 Seconds

Hiroaki Mikami, Hisahiro Soganuma, Pongsakorn U-chupala, Yoshiki Tanaka and Yuichi Kageyama
Sony Corporation

Abstract
Scaling the distributed deep learning to a massive GPU cluster level is challenging due to the instability of the large mini-batch training and the overhead of the gradient synchronization. We address the instability of the large mini-batch training with batch size control. We address the overhead of the gradient synchronization with 2D-Torus all-reduce. Specifically, 2D-Torus all-reduce arranges GPUs in a logical 2D grid and performs a series of collective operation in different orientations. These two techniques are implemented with Neural Network Libraries (NNL)\(^1\). We have successfully trained ImageNet/ResNet-50 in 224 seconds without significant accuracy loss on ABCI\(^2\) cluster.
Limits of batch-size scaling

Recent empirical studies by OpenAI (arXiv:1812.06162) and Google Brain (arXiv:1811.03600) show that:

- A relationship between gradient noise scale and *critical* batch-size holds across many models, algorithms and datasets
- *gradient noise scale predicts maximum useful batch-size*
- More complex datasets/tasks have higher gradient noise, thus can benefit from training with larger batch-sizes
“Increasing parallelism makes it possible to train more complex models in a reasonable amount of time… a Pareto frontier chart is [used] to visualize comparisons between algorithms and scales” -- blog.openai.com/science-of-ai
Summary and outlook

● Distributed training is imperative for larger and more complex models/datasets

● Data parallelism distributes more data among more workers

● Large batch training is unstable and may impact generalization error if hyper-parameters are not tuned well

● Use learning-warm up and linear scaling to scale to modest scales < 10x. No guarantees that it will work for all models

● Batch-size scaling seems to be more robust across many models

● A simple statistic, gradient noise scale, can predict maximum useful batch-size
Let’s get practical
Distributed training hands-on session

We will use ResNet on CIFAR10 to demonstrate implementation and speedup.

- Note that this small dataset doesn’t necessarily require scale
- But it allows us to get some results in the allotted time frame

We will use Keras and Horovod/CPE ML for distributed training

- Easy to use/teach
- Fast (relies on optimized backend, MPI/RDMA)
- Only few code modifications necessary
CIFAR-10

- prepared by University of Toronto
- slightly more complicated than MNIST, but less complex than Imagenet
- 60K, 32x32 color images
- 10 classes (plane, car, bird, car, deer, dog, frog, horse, ship, truck)
- 50K training, 10K test
- [cifar-10 link](cifar-10)
- intuitive, fast training/model development times, good for demonstrating the essentials of distributed training
- good for tutorials
ResNet Topology (34-layer-version)

- convolution layers arranged in blocks
- skip connections combine input and output of block (residual learning)
- FC layer for classification
- ResNet performs well on image classification tasks
Horovod

Enables distributed synchronous data-parallel training with minimal changes to user code

Uses efficient all-reduces from MPI to collectively combine gradients across workers

Such approaches shown to scale better than parameter-server approaches (e.g. distributed TensorFlow with gRPC)

https://eng.uber.com/horovod/
CPE ML (Cray Programming Environment ML Plugin)

Enables distributed synchronous data-parallel training with minimal changes to user code

Uses RDMA operations or reductions

Might perform better than Horovod on large networks and large scales

Advanced training features already implemented: pipelining, warmup, cooldown, etc.
Scaling concepts demonstrated today

Today we will utilize:

- Synchronous data-parallel training (weak scaling) using Horovod/CPE ML
- Learning rate linear warmup
- Linear learning rate scaling, $\eta \rightarrow N \times \eta$, followed by original decay schedule
Ingredients for multi-node training (Horovod)

Initialize Horovod and MPI:

```python
hvd.init()
```

Wrap your optimizer in the Horovod distributed optimizer:

```python
opt = keras.optimizers.SGD(lr=lr*hvd.size(), ...)
opt = hvd.DistributedOptimizer(opt)
```

Construct the variables broadcast callback:

```python
callbacks =
[hvd.callbacks.BroadcastGlobalVariablesCallback(0), ...]
```
Comparison Horovod vs. CPE ML - Initialization

Horovod

```python
# Import and MPI Initialization
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

# import Horovod library
import horovod.keras as hvd

# initialize Horovod
hvd.init()
```

CPE ML

```python
# Import and MPI Initialization
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

# Additional pkgs used to calculate buffer sizes, etc.
import numpy as np
import math
import tensorflow as tf

# import Cray ML library and user defined
# Callbacks, Distributed Optimizer
import ml_comm as mc

from plugin_keras import InitPluginCallback, BroadcastVariablesCallback, DistributedOptimizer

# initialize CPE
mc.init_mpi()
```
# Optimizer and model compile

base_lr = 1.0

# Adjust epochs based on parallel throughput
epochs = int(epochs/hvd.size())

# Non-distributed compile
optimizer = optimizer=keras.optimizers.Adadelta(base_lr)

# Horovod: Add Distributed Optimizer
optimizer = hvd.DistributedOptimizer(optimizer)

# Run the training loop
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=optimizer, metrics=['accuracy'])

---

# Optimizer and model compile

base_lr = 1.0

# Adjust epochs based on parallel throughput
epochs = int(epochs/mc.get_nranks())

# Non-distributed compile
optimizer = optimizer=keras.optimizers.Adadelta(base_lr)

# Cray ML Plugin: Add Distributed Optimizer
optimizer = DistributedOptimizer(optimizer)

# Run the training loop
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=optimizer, metrics=['accuracy'])
CPE ML - DistributedOptimizer Implementation example

- implementation example for DistributedOptimizer
- extracting gradients and reducing them explicitly
- allows to inject/wrap other optimizations such as gradient manipulation (LARS/LARC)
- can be done in Horovod as well
Ingredients for multi-node training (Horovod/CPE ML)

Train model as usual; it should now synchronize at every mini-batch step:

```python
model.fit(..., callbacks=callbacks)
```

Launch your script with MPI

```bash
srun -n ${SLURM_NNODES} … -u python train.py …
```

(we’ll use SLURM and srun instead of mpirun for generic MPI installations)
Running the multi-node training

Refer again to the documentation on the github repo:

https://github.com/NERSC/cug19-da-tutorial

You can try these examples out on your own system

Feel free to try and tweak things and get better performance

- Change optimizer
- Change learning rate, number of warmup epochs, decay schedule
- Change learning rate scaling (e.g., $lr\sqrt{N}$ instead of $lrN$)
Scaling results for ResNet CIFAR10

Training time goes down

Training loss and accuracy are still converging at similar rates
Deep Learning for science excites you?

We are hiring: goo.gl/De4wBU

Thank You