Distributed Training

E4 face-2-face meeting
4 Oct 2019

Jean-Roch Vlimant, with many others
Community Interest

- Exa.TrkX DOE: scaling tracking GNN training at NERSC
  - S. Farrell, P. Calafiura, J. Kowalkowski, ...
  - Allocation on Summit
- HEPGan NESAP: scaling/developing calorimeter GAN
  - B. Nachman, W. Bhimji, S. Vallecorsa, ...
  - Allocation at NERSC
  - Pending hiring a postdoc at NERSC
    - https://inspirehep.net/record/1733162
- BNL effort: study scaling of various training frameworks
  - A. Malik, ...
  - Allocation on summit
- IRIS-HEP:
  - FastML workshop https://indico.cern.ch/event/822126
  - No current aim at supporting in-house development
- ATLAS & CMS ML groups:
  - A. Farbin (ATLAS), JRV (CMS)
  - Interest in training as a service, integration of training in experiment workflow management
- ...

Code Status

- Neural Network Learning and Optimization: NNLO
  https://github.com/vlimant/NNLO
- Lots of development done over the last few months
  - Full checkpointing of training and optimization
  - Model interfacing with python script
  - Streamlined repository
  - Consolidation of options
  - Full logging
  - Better documentation
  - Graph network example (torch)
  - Data adaptor

- Upcoming
  - Option restriction (high prio)
  - Catching worker failure (medium/high prio)
  - TF model adaptor (medium prio)
  - GAN Interface (medium prio)
  - BatchNorm support (low prio)
  - ROOT data format adaptor (low prio)
(Past) Performance

ANN/RNN
Parameter server setup
Linear scaling
Downpour SGD
Worker SGD

GAN
Parameter server setup
Linear scaling
Elastic averaging SGD
Worker rmsprop

https://arxiv.org/abs/1712.05878

https://doi.org/10.1051/epjconf/201921406025
Next Steps

- Summit and/or NERSC
- Scaling plot with GNN
  - With GEM parameter server setting
  - With horovod setting
- Scaling plot with 3D calo GAN (or other example)
  - With GEM parameter server setting
  - With horovod setting
- Hyper-optimization of GNN, GAN
  - Genetic algorithm and bayesian optimization
- CHEP talk
  [https://indico.cern.ch/event/773049/contributions/3474799/](https://indico.cern.ch/event/773049/contributions/3474799/)
- Paper
Future

- Due to some intrinsic limitation of how the package was setup, some bottlenecks will not be reducible (without major rewrite that is)
  → re-write needed
- Figure out involvements from the community
- Capitalize on experience on the topic
- Design a task-based distributed training software
Outline

• Neural network training
• Training workload parallelization
• Hyper-parameters optimization
• Summary and Future work
Motivations

- Large models on large dataset can take days-week to converge on single GPU.
- Simpler models can take as long to converge, on CPU-only hosts.
- Prototyping with model architecture is like testing a new idea for analysis, you want to have the answer “fast”
- Dismissing large model, large dataset because of train time
Deep Learning Training
Artificial Neural Network

- Large number of parameters
- Efficiently adjusted with stochastic gradient descent
- The more parameters, the more data required
- Training to convergence can take minutes to several days, ...
Training Artificial Neural Networks

- ANN and associated loss function have fully analytical formulation and are differentiable with respect to model parameters
- Gradient evaluated over batch of data
  - Too small: very noisy and scattering
  - Too large: information dilution and slow convergence
Distributed Training
Parallelism Overview

➔ Data distribution
Compute the gradients on several batches independently and update the model synchronously or not. **Applicable to large dataset**

➔ Gradient distribution
Compute the gradient of one batch in parallel and update the model with the aggregated gradient. **Applicable to large sample ≡ large event**

➔ Model distribution
Compute the gradient and updates of part of the model separately in chain. **Applicable to large model**
Data Distribution
Data Distribution

1) Compute gradient, send to Master

\[ \nabla Q(\bar{w}) \]

2) Update network weights

\[ \bar{w} \rightarrow \bar{w} - \eta \nabla Q(\bar{w}) \]

3) Send new weights to Worker

https://arxiv.org/abs/1712.05878

- Master node operates as parameter server
- Work nodes compute gradients
- Master handles gradients to update the central model
  - downpour sgd https://tinyurl.com/ycfpwec5
  - Gradient energy matching https://arxiv.org/abs/1805.08469
Basic Layout

- Training master group 0, subrank 0
- Training worker group 0, subrank 1
- Training master group 0, subrank 2
- Training master group 0, subrank $N_w$
Performance with ANN

- Speed up in training recurrent neural networks on Piz Daint CSCS supercomputer
  - Linear speed up with up to ~20 nodes.
  - Needs to compensate for staleness of gradients (see GEM https://arxiv.org/abs/1805.08469)
- Linear scaling on servers with 8 GPUs

https://arxiv.org/abs/1712.05878
Performance with GAN

- Speed up in training generative adversarial networks on Piz Daint CSCS and Titan ORNL supercomputers.
  - Using easgd algorithm with rmsprop.
  - Speed up is not fully efficient. Bottlenecks to be identified.

NVIDIA K20 at Titan, ORNL

NVIDIA P100 on Piz Daint, CSCS
Cray ML Plugin

MPI based. Synchronous SGD. TF1.4
Optimal scaling through a large number of nodes
Observed performance degradation at low energy
Possibly compensate by increasing learning rate
Work in progress

<table>
<thead>
<tr>
<th>Model</th>
<th>GPU System</th>
<th>CPU System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XC40/XC50</td>
<td>XC50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computer nodes</th>
<th>Interconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon E5-2697 v4 @ 2.3GHz (18 cores, 64GB RAM) and NVIDIA Tesla P100 16GB</td>
<td>Aries, Dragonfly network topology</td>
</tr>
<tr>
<td>Two Intel Xeon Platinum 8160 @ 2.1GHz (2 x 24 cores, 192GB RAM)</td>
<td>Aries, Dragonfly network topology</td>
</tr>
</tbody>
</table>

Performance Scaling on GPU
Performance Scaling on CPU

https://sites.google.com/nvidia.com/ai-hpc

Slide S. Vallecorsa
Gradient Distribution
• A logical worker is spawn over multiple processes
• Communicator passed to horovod https://github.com/uber/horovod
• Nvidia NCCL enabled for fast GPU-GPU communication
Intel MKL-DNN

Use keras 2.13 /Tensorflow 1.9 (Intel optimised)
  - AVX512 –FMA-XLA support
  - Intel® MKL-DNN (with 3D convolution support)

Optimised multicore utilisation
  - inter_op_parallellism_threads/intra_op_parallellism threads

Horovod 0.13.4
  - Synchronous SGD approach
  - MPI_AllReduce

Run on TACC Stampede2 cluster:
  - Dual socket Intel Xeon 8160
  - 2x 24 cores per node, 192 GB RAM
  - Intel® Omni-Path Architecture

Test several MPI scheduling configurations
  - 2, 4, 8 processes per nodes.
  - Best machine efficiency with 4 processes/node

Some performance degradation
Mostly in the low energy regions for large batchsize

https://sites.google.com/nvidia.com/ai-hpc
Model Distribution
Intra-Node Model Parallelism

- Perform part of the forward and backward pass on different devices
- Require good device to device communication
- Utilize native tensorflow multi-device manager
- Aiming for machines with multi-gpu per node topology (e.g. summit)

See T. Kurth et al. @ https://pasc18.pasc-conference.org for node to node model parallelism considerations
Hyper-Parameters Optimization
Hyper-Parameters

• Various parameters of the model cannot be learned by gradient descent
  ➢ Learning rate, batch size, number of layers, size of kernels, …

• Tuning to the right architecture is an “art”. Can easily spend a lot of time scanning many directions

• Full parameter scan is resource/time consuming.

  ➔ Hence looking for a way to reach the optimum hyper-parameter set for a provided figure of merit (the loss by default, but any other fom can work)
  ➔ Possible optimization engine (https://github.com/vlimant/mpi_opt)
     ➢ Bayesian optimization with gaussian processes prior
     ➢ Evolutionary algorithm
     ➢ …
K-Folding Cross Validation

- Estimate the performance of multiple model training over different validation part of the training dataset.
- Allows to take into account variance from multiple source (choice of validation set, choice of random initialization, ...).
- Crucial when comparing models performance.
- Training on folds can proceed in parallel.
K-Folding Layout

- H-opt
  - master
  - Rank 0

- Training master
  - group 0, subrank 0

- Training worker
  - group 0, subrank 1
  - group 0, subrank 2
  - group 0, subrank N_w

- Parameter-set group 0
- Parameter-set group 1
- Parameter-set group N_c
Summary & Outlook

- Distributed training is not always necessary (short training time?)
- Many aspects to distributed training to consider
- Several x-factor speedup for ANN, efficient at low number of nodes. Bottleneck on master/node load balance. GEM maintains convergence over nodes.
- Several inefficient x-factors to be gained for GAN training
- Distributed training over CPU facilities is efficient (but not necessarily cost effective)
- Cross validation is a must and can be done in parallel
- Hyper-parameter optimization is almost mandatory, but not fully parallelizable

➢ Interest in the community to have a common software
➢ Imminent publication about distributed training and optimization, seed to a follow up community-wide project.
➢ In-house dev, or use industry provided software?
Putting all Features Together

\[ N_{\text{nodes}} = 1 + N_G \times N_F \times (N_M \times N_W \times N_{\text{GPU}}) \]

- \( N_G \): # of concurrent hyper-parameter set tested
- \( N_F \): # of folds
- \( N_M \): # of masters
- \( N_W \): # of workers per master
- \( N_{\text{GPU}} \): # of nodes per worker (1node=1gpu)

Speed up and optimize models using thousand(s) of GPUs
Sub-master Layout

- Putting workers in several groups
- Aim at spreading communication to the main master
- Need to strike a balance between staleness and update frequency
- One master running the bayesian optimization
- \( N_G \) groups of nodes training on a parameter-set on simultaneously
  - One training master
    - \( N_M \) training sub-masters
    - \( N_W \) training workers
All-reduce Layout

- One master running the bayesian optimization
- $N_G$ groups of nodes training on a parameter-set on simultaneously
  - One training master
  - $N_W$ training worker groups
    - $N_{\text{GPU}}$ used for each worker group (either nodes or gpu)
mpi-skopt Setup

- One master running communication of parameter set
- $N_{SK}$ workers running the bayesian optimization
- $N_G$ groups of nodes training on a parameter-set on simultaneously
  - One training master
  - $N_W$ training workers
Basic Layout

- One master process drives the hyper-parameter optimization
- $N_G$ groups of nodes training on a parameter-set on simultaneously
  - One training master
  - $N_W$ training workers
K-folding Layout

- One master running the optimization. Receiving the average figure of merit over $N_F$ folds of the data
  - $N_G$ groups of nodes training on a parameter-set on simultaneously
  - $N_F$ groups of nodes running one fold each
Bayesian Optimization

- Objective function is approximated as a multivariate gaussian
- Measurements provided one by one to improve knowledge of the objective function
- Next best parameter to test is determined from the acquisition function
- Using the python implementation from https://scikit-optimize.github.io
Evolutionary Algorithm

- Chromosomes are represented by the hyper-parameters
- Initial population taken at random in the parameter space
- Population is stepped through generations
  - Select the 20% fittest solutions
  - Parents of offspring selected by binary tournament based on fitness function
  - Crossover and mutate to breed offspring
- Alternative to bayesian opt. Indications that it works better for large number of parameters and non-smooth objective function

- Chromosome crossover:
  - Let Parent A be more fit than Parent B
  - For each parameter $p$, generate a random number $r$ in $\{0, 1\}$ to find $p_{child}$

  $$p_{child} = (r)(p_{\text{Parent } A} - p_{\text{Parent } B}) + p_{\text{Parent } A}$$

- Non-uniform mutation (Michalewicz):
  - In generation $g$ out of a total $G$ generations, for each parameter $p$ in a child, generate random numbers $r_1, r_2 \in \{0, 1\}$ to define a mutation $m$:

  $$m = \left(1 - r_1 \left(1 - \frac{g}{G}\right)^3\right) \times \begin{cases} (p_{\text{MAX}} - p_{\text{child}}) & \text{IF } r_2 > 0.5 \\ (p_{\text{LOW}} - p_{\text{child}}) & \text{IF } r_2 \leq 0.5 \end{cases}$$

  $$p_{child} = p_{child} + m$$