Towards Scalable Machine Learning

Janis Keuper

itwm.fraunhofer.de/ml

Competence Center High Performance Computing
Fraunhofer ITWM, Kaiserslautern, Germany

Fraunhofer Center Machine Learning
Outline

I  Introduction / Definitions

II  Is Machine Learning a HPC Problem?

III  Case Study: Scaling the Training of Deep Neural Networks

IV  Towards Scalable ML Solutions [current Projects]

V  A look at the (near) future of ML Problems and HPC
Introduction

Machine Learning @CC-HPC

Scalable distributed ML Algorithms
- Distributed Optimization Methods
- Communication Protocols

Distributed DL Frameworks

“Automatic” ML
- DL Meta-Parameter Learning
- DL Topology Learning

HPC-Systems for Scalable ML
- Distributed I/O
- Novel ML Hardware
- Low Cost ML Systems

DL Methods:
- Semi- and Unsupervised DL
- Generative Models
- ND CNNs

Industry Applications
- DL Software optimization for Hardware / Clusters
- DL for Seismic Analysis
- DL Chemical Reaction Prediction
- DL for autonomous driving
## Setting the Stage | Definitions

<table>
<thead>
<tr>
<th>Scalable ML</th>
<th>vs</th>
<th>Large Scale ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Large model size</td>
<td></td>
<td>• Very large data sets</td>
</tr>
<tr>
<td>(implies large data as well)</td>
<td></td>
<td>(online stream)</td>
</tr>
<tr>
<td>• Extreme compute effort</td>
<td></td>
<td>• “normal” model size and compute effort</td>
</tr>
<tr>
<td>• Goals:</td>
<td></td>
<td>(traditional ML methods)</td>
</tr>
<tr>
<td>• Larger models</td>
<td></td>
<td>• Goals:</td>
</tr>
<tr>
<td>• (linear) strong an weak</td>
<td></td>
<td>• Make training feasible</td>
</tr>
<tr>
<td>scaling through (distributed)</td>
<td></td>
<td>• Often online training</td>
</tr>
<tr>
<td>parallelization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ HPC</td>
<td></td>
<td>→ Big Data</td>
</tr>
</tbody>
</table>
Scaling DNNs

Simple strategy in DL
if it does not work: scale it!

Scaling in two dimensions:

1. Add more layers = more matrix mult
   more convolutions

2. Add more units = larger matrix mult
   more convolutions

Don't forget: in both cases

MORE DATA! → more iterations
Scaling DNNs

OUTRAGEOUSLY LARGE NEURAL NETWORKS: THE SPARSELY-GATED MIXTURE-OF-EXPERTS LAYER

Noam Shazeer\textsuperscript{1}, Azalia Mirhoseini\textsuperscript{1,2}, Krzysztof Maziarz\textsuperscript{2}, Andy Davis\textsuperscript{1}, Quoc Le\textsuperscript{1}, Geoffrey Hinton\textsuperscript{1}\textsuperscript{1} and Jeff Dean\textsuperscript{1}

\textsuperscript{1}Google Brain, \{noam,azalia,andydavis,qvl,geoffhinton,jeff\}@google.com
\textsuperscript{2}Jagiellonian University, Cracow, krzysztof.maziarz@student.usz.edu.pl

ABSTRACT

The capacity of a neural network to absorb information is limited by its number of parameters. Conditional computation, where parts of the network are active on a per-example basis, has been proposed in theory as a way of dramatically increasing model capacity without a proportional increase in computation. In practice, however, there are significant algorithmic and performance challenges. In this work, we address these challenges and finally realize the promise of conditional computation, achieving greater than 1000x improvements in model capacity with only minor losses in computational efficiency on modern GPU clusters. We introduce a Sparsely-Gated Mixture-of-Experts layer (MoE), consisting of up to thousands of feed-forward sub-networks. A trainable gating network determines a sparse combination of these experts to use for each example. We apply the MoE to the tasks of language modeling and machine translation, where model capacity is critical for absorbing the vast quantities of knowledge available in the training corpora. We present model architectures in which a MoE with up to 137 billion parameters is applied convolutinally between stacked LSTM layers. On large language modeling and machine translation benchmarks, these models achieve significantly better results than state-of-the-art at lower computational cost.
Is Scalable ML a HPC Problem?

- In terms of compute needed (YES)
- Typical HPC Problem setting: is communication bound = non trivial parallelization (YES)
- I/O bound (New to HPC)
Success in Deep Learning is driven by compute power:

→ # FLOP needed to compute leading model is ~ doubling every 3.5 months!

→ increase since 2012: factor ~300,000!
Impact on HPC (Systems)

- New HPC Systems
  - Like ONCL “Summit”
  - Power 9
  - ~30k NVIDIA Volta GPUs
  - New storage hierarchies
- New Users (=new demands)
- Still limited resources

III Case Study: Training DNNs

I Overview: distributed parallel training of DNNs

II Limits of Scalability

Limitation I: Communication Bounds

Limitation II: Skinny Matrix Multiplication

Limitation III: Data I/O

Based on our paper from SC 16
Deep Neural Networks
In a Nutshell

At an abstract level, DNNs are:

- directed (acyclic) graphs
- **Nodes** are compute entities (=Layers)
- **Edges** define the data flow through the graph

Inference / Training

Forward feed of data through the network
Deep Neural Networks

In a Nutshell

At an abstract level, DNNs are:

- directed (acyclic) graphs
- **Nodes** are compute entities (=Layers)
- **Edges** define the data flow through the graph

Inference / Training

Forward feed of data through the network
Training Deep Neural Networks

The Underlying Optimization Problem

Computed via **Back Propagation** Algorithm:

1. feed forward and compute activation
2. error by layer
3. compute derivative by layer

\[
\delta_i^{(m)} = \frac{\partial}{\partial z_i^{(m)}} \left( \frac{1}{2} \| y - h_{W,b}(x) \|_2^2 \right) = -(y_i - a_i^{(m)}) \cdot f'(z_i^{(m)})
\]

\[
\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}
\]

Minimize Loss-Function via gradient descent *(high dimensional and NON CONVEX!)*

\[
J(W, b) = \left[ \frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{i=1}^{n_l-1} \sum_{i=1}^{s_i} \sum_{j=1}^{s_{i+1}} (W_{ji}^{(l)})^2
\]
Optimization Problem
By Stochastic Gradient Descent (SGD)

1. Initialize weights $W$ at random
2. Take small random subset $X$ (= batch) of the train data
3. Run $X$ through network (forward feed)
4. Compute Loss
5. Compute Gradient
6. Propagate backwards through the network
7. Update $W$

Repeat 2-8 until convergence
Parallelization
Common approaches to parallelize SGD for DL

Parallelization of SGD is very hard: it is an inherently sequential algorithm

1. Start at some state $t$ (point in a billion dimensional space)
2. Introduce $t$ to data batch $d_1$
3. Compute an update (based on the objective function)
4. Apply the update $\rightarrow t+1$

How to gain Speedup?
- Make faster updates
- Make larger updates
Parallelization
Common approaches to parallelize SGD for DL

Internal parallelization = parallel execution of the layer operation

- Mostly **dense matrix multiplication**: standard blas sgemm, MKL, Open-Blas, CuBlas on GPUs
- Task parallelization for special Layers: Cuda-CNN for fast convolutions
Parallelization
Common approaches to parallelize SGD for DL

External: data parallelization over the data batch

1. Split batch and send to workers

Master

Layer 1  Layer 1  Layer 1
Layer 2  Layer 2  Layer 2
...  ...  ...
Layer n-1  Layer n-1  Layer n-1
Layer n  Layer n  Layer n
forward  forward  forward
Parallelization
Common approaches to parallelize SGD for DL

External: data parallelization over the data batch

1. Split batch and send to workers
2. Workers compute forward+backward and send gradients to master
Parallelization
Common approaches to parallelize SGD for DL

External: data parallelization over the data batch

1. Split batch and send to workers
2. Workers compute forward+backward and send gradients to master
3. Master combines gradients and computes updates of W. Sends new W' to workers
Parallelization
Common approaches to parallelize SGD for DL

External: data parallelization over the data batch

1. Split batch and send to workers
2. Workers compute forward+backward and send gradients to master
3. Master combines gradients and computes updates of W. Sends new W' to workers

Speedup
Limitation I
Distributed SGD is heavily Communication Bound

Gradients have the same size as the model

- Model size can be hundreds of MB
- Iteration time (GPU) <1s
Communication Bound
Experimental Evaluation
Solving the Communication Bottleneck
Solutions in Hardware: example NVLink

NVIDIA DGX-1 / HGX-1
- 8x P100
- DGX-1: NVLink between all GPUs

NVLink spec: ~40GB/s (single direction)
NVLink II (Volta): ~75GB/s (single direction)

PCIe v3 16x: ~15GB/s

Diagram DGX-1 by NVIDIA
Solving the Communication Bottleneck
Solutions in Hardware: NVLink Benchmark

IBM Minsky
- 4x P100
- 2x10 core Power8 (160 hw threads)
- NVLink between all components

NVLink spec: ~40GB/s (single direction)
Limitation I
Distributed SGD is heavily Communication Bound

How to solve this in distributed environments?
Limitation II
Mathematical Problems – aka the “Large Batch Size Problem”

Recall:
speedup comes from pure data-parallelism
→ splitting the batch over the workers
“Small Batch Size Problem”

Data Parallelization over the Batch Size

Problem: Batch size decreasing with distributed scaling

Hard Theoretic Limit: $b > 0$

- GoogLeNet: No Scaling beyond 32 Nodes
- AlexNet: Limit at 256 Nodes

External Parallelization hurts the internal (BLAS / cuBlas) parallelization even earlier.

In a nutshell: for skinny matrices there is simply not enough work for efficient internal parallelization over many threads.
“Small Batch Size Problem”
Data Parallelization over the Batch Size

Computing Fully Connected Layers:
Single dense Matrix Multiplication

<table>
<thead>
<tr>
<th>Layer</th>
<th># operations</th>
<th>matrix sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully Connected</td>
<td>1</td>
<td>(b \times I \times I \times O)</td>
</tr>
<tr>
<td>Convolutional</td>
<td>(b)</td>
<td>(C \times I \times I \times Z)</td>
</tr>
<tr>
<td>Softmax</td>
<td>(b)</td>
<td>(I \times 1 \times 1 \times 1)</td>
</tr>
</tbody>
</table>

**Definitions:**
- I: Input size from top layer
- O: Output size of this layer
- b: Local Batch size (train or validation)
- C: Number of filters
- c: Number of input channels (RBG image: \(c = 3\))
- P: Patch size (i.e. pixel)
- k: Kernel size
- Z: Effective size after kernel application.

For convolution \(Z := \left(\sqrt{P} - \lfloor(k/2)\rfloor \right)^2\)

**TABLE III**

SIZE AND NUMBER OF OF THE MATRIX MULTIPLICATIONS (SGEMM) PER FORWARD PASS FOR SELECTED LAYERS.
Experimental Evaluation
Increasing the Batch Size

Solution proposed in literature:
Increase Batch size

But:
Linear speedup against original Problem only if we can reduce the number iterations accordingly

This leads to loss of accuracy

AlexNet on ImageNet
The “large batch” Problem

Central Questions

Why is this happening?

Is it dependent on the Topologie / other parameters?

How can it be solved?

→ large batch size SGD would solve most scalability problems!
The “large batch” Problem
What is causing this effect? [theoretical and not so theoretical explanations]

→ The “bad minimum”

→ gradient variance / coupling of learning rate and batch size

Figure 2: Training and testing accuracy for SB and LB methods as a function of epochs.
The “bad minimum”

Theory: larger batch causes degrease in gradient variance, causing convergence to local minima...

→ empirical evaluation shows high correlation of sharp minima and weak generalization
Limitation II
Mathematical Problems – aka the “Large Batch Size Problem”

Problem not fully understood
No general solutions (yet)!
Do we need novel optimization methods ?!
Limitation III

Data I/O

Hugh amounts of training data need to be streamed to the GPUs

- Usually 50x – 100x the training data set
- Random sampling (!)
- Latency + Bandwidth competing with optimization communication
Distributed I/O

Distributed File Systems are another Bottleneck!

- Network bandwidth is already exceeded by the SGD communication
- Worst possible file access pattern:
  - Access many small files at random

This problem already has effects on local multi-GPU computations

E.g. on DG-X1 or Minsky, single SSD (~0.5 GB/s) to slow to feed >= 4 GPUs
  -> solution: Raid 0 with 4 SSDs
Distributed I/O

Distributed File Systems are another Bottleneck!

Compute time by Layer
AlexNet (GPU + cuDNN)

Results shown for SINGLE node access to a Lustre working directory (HPC Cluster, FDR-Infiniband)

Results shown for SINGLE node Data on local SSD.
IV Towards Scalable Solutions

Distributed Parallel Deep Learning with HPC tools + Mathematics
CaffeGPI:
Distributed Synchronous SGD Parallelization
With asynchronous communication overlay

Better scalability using asynchronous PGAS programming of optimization algorithms with GPI-2.

Direct RDMA access to main and GPU memory instead of message passing

Optimized data-layers for distributed File systems
CC-HPC Current Projects

CaffeGPI: Approaching DNN Communication

CaffeGPI:
Distributed Synchronous SGD Parallelization

- Communication Reduction Tree
- Communication Quantization
- Communication Overlay
- Direct RDMA GPU → GPU
  - Based on GPI
- Optimized distributed data-layer
CaffeGPI:
Open Source

https://github.com/cc-hpc-itwm/CaffeGPI

CaffeGPI is a new, open source tool that speeds up Deep Neural Network training. The tool is based on the Caffe framework and offers new ways to parallelize neural network training.

Using GPI-2 for Distributed Memory Parallelization of the Caffe Toolbox to Speed up Deep Neural Network Training

Martin Kuehn, Janos Kegel and Franzi Josef Pfeifer
Computing Center High Performance Computing
Fraunhofer Institute for Industrial Mathematics
Fraunhofer-Pfitzner 1, D-67663 Kaiserslautern, Germany
May 31, 2017

Abstract

Deep Neural Network (DNN) architectures have improved considerably the accuracy in data classification opening the door for a plethora of new uses in image classification, speech recognition or semantic text understanding. However, the training of DNNs is in very compute intensive tasks. So, the training interest in these architectures created a commensurate demand for compute resources which is further intensified by a race to process sizes of DNNs.

Another important factor to be taken into account is that training DNNs...
Projects: Low Cost Deep Learning Setup: Build on CaffeGPI

Price: <8000 EUR standard components

Specs:
- 32 GB GPU Mem
- 64 GB PGAS Mem
- 2TB BeeGFS for Train Data
- GPU interconnect: PCIe
CaffeGPI: Benchmarks: Single Node

Specs:
4x K80 GPU (PCIe Interconnect)
Cuda 8
CuDNN 5.1

Topology: AlexNet
Batch Size (global) (1024)
Low Cost Deep Learning Setup:
Build on CaffeGPI

Specs: Topology: GoogLeNet, Cuda 8, cuDNN 5.1, CaffeNV 16.4, Batch Size/Node: 64
CaffeGPI: Benchmarks

Specs:
- 1x K80 GPU (per node)
- Cuda 8
- CuDNN 5.1

Topology: GoolgeNet
Batch Size: 64 per node
New Project: Low Cost Deep Learning Cluster

GPU Cluster for Fraunhofer Consortium @ITWM

- **Low cost Hardware**
  - Consumer GPUs
  - Novel AMD architecture
  - Hosting Cost per GPU ~ 1.25 k EUR. Compared to DGX-1 ~ 10k

- **Fast Interconnect and Data I/O**
  - Parallel FS with local NVMe

- **Open Source Multi-User Management**
  - Reservation system
  - Scheduling
  - Custom Containers
  - Web Interface
Low Cost Deep Learning Setup
Currently building: Prototype 16-Node System

CC-HPC - Fraunhofer ITWM, Kaiserslautern
Project Carme
An open source software stack for multi-user GPU clusters

*Carme* (/ˈkɑːrmiː/ KAR-mee; Greek: Κάρμη) is a Jupiter moon, also giving the name for a *Cluster* of Jupiter moons (the carme group).

Or in our case:

an open source frame work to mange resources for multiple users running *Jupyter* notebooks on a *Cluster* of compute nodes.
Project Carme
An open source software stack for multi-user GPU clusters

Common problems in GPU-Cluster operation:

- **Interactive, secure multi user environment**
  - ML and Data Science users want interactive GUI access to compute resources

- **Resource Management**
  - How to assign (GPU) resources to competing users?
    - User management
    - Accounting
    - Job scheduling
    - Resource reservation

- **Data I/O**
  - Get user data to compute nodes (I/O Bottleneck)

- **Maintenance**
  - Meet (fast changing and diverse) software demands of users
Project Carme
An open source software stack for multi-user GPU clusters

Carme core idea:

- **Combine established open source ML and DS tools with HPC back-ends**
  - Use containers
  - (for now) Docker
  - Use Jupyter Notebooks as main web based GUI-Frontend
    - All web front-end (OS independent, no installation on user side needed)
  - Use HPC job management and scheduler
    - SLURM
  - Use HPC data I/O technology
    - ITWM’s BeeGFS
  - Use HPC maintenance and monitoring tools
Project Carme
An open source software stack for multi-user GPU clusters
HP-DLF
High Performance Deep Learning Framework

- Scalable
- Transparent
- generic
- Auto-parallel
- Elastic
- Automatic data flow
- Automatic Hardware selection
- Portable
- Monitoring
- Simulation
- New optimization methods
Our asynchronous optimization algorithm

- Sparse communication for multi model optimization
- Lower demands on the communication bandwidth
- Superior convergence
V A Look a the near Future

For HPC:

- New Hardware Accelerators
- New Interconnects
- New Architectures

From ML

- Models are still growing!
- Learning to learn
Automatic Design
Of Deep Neural Networks

Basically, a graph optimization problem:

- **Select Node Types**
  - And their Meta-Parameters
- **Connect Edges** to define the data flow through the graph

Optimization target: minimize test error

Problems:
- Huge and difficult search space
- Each iteration requires training of a DNN
Automatic Design
Current Approaches: Reinforcement Learning

<table>
<thead>
<tr>
<th>Model</th>
<th>Error rate</th>
<th># params ($\times 10^6$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network in Network [19]</td>
<td>8.81</td>
<td>–</td>
</tr>
<tr>
<td>VGG [27] 1</td>
<td>7.94</td>
<td>15.2</td>
</tr>
<tr>
<td>ResNet [10]</td>
<td>6.61</td>
<td>1.7</td>
</tr>
<tr>
<td>Neural Architecture Search [39]</td>
<td>3.84</td>
<td>32.0</td>
</tr>
<tr>
<td>CGP-CNN (ConvSet)</td>
<td>6.75</td>
<td>1.52</td>
</tr>
<tr>
<td>CGP-CNN (ResSet)</td>
<td>5.98</td>
<td>1.68</td>
</tr>
</tbody>
</table>

Evaluation on CIFAR-10:
Better than “State of the Art” (hand designed) performance
- After ~12500 iterations
- Compute time: ~ 10000 GPU days
Deep Topology Learning

- Genetic Algorithms
- Reinforcement Learning
- Early Stopping
- Graph Embedding
- Meta-Learning
- Pruning

On application size problems
Data basis generation
Discussion

This is your machine learning system?

You pour the data into this big pile of linear algebra, then collect the answers on the other side.

What if the answers are wrong?

Just stir the pile until they start looking right.