FUNDAMENTALS OF
SCALING OUT DL TRAINING

Paulius Micikevičius, NVIDIA
HotChips 2020, DL Scale Out Tutorial
Larger is Better in DL

• Larger models lead to higher task accuracies
  • Language models: in the past 2 years grew from 340M to 175B parameters
  • Recommender models: largest ones are reaching O(1B) parameters
  • Vision models: deeper and wider Resnets and ResNeXTs

• Larger datasets lead to higher accuracies
  • Recommender data (user behavior): terabytes to petabytes
  • Image data: 1B Instagram dataset, JFT (300M images)

• Challenges:
  • Larger models -> training state no longer fits on a single processor
  • Larger {models, datasets} -> long time to train

• Solution: scale out computing
Outline

• Brief Review of DNN Training
• Data Parallelism
• Model Parallelism
  • Pipeline
  • Intra-layer
• Communication Pattern Review
• Summary
Neural Network Training

• Start with randomly initialized weights
• Iterate through your data a minibatch of training data samples at a time:
  • Forward pass
  • Backward pass
  • Weight update
Simple Example

- Network of 3 linear layers
- Each layer:
  - Input: vector
  - Output: vector
  - Learned parameters (weights): projection matrix
  - Operation:
    - Multiply the input vector with the matrix
    - Apply a point-wise nonlinearity, say, ReLU
Forward Pass

- Network of 3 linear layers
- Each layer:
  - Input: vector
  - Output: vector
  - Learned parameters (weights): projection matrix
  - Operation:
    - Multiply the input vector with the matrix
    - Apply a point-wise nonlinearity, say, ReLU
Forward Pass

- Network of 3 linear layers
- Each layer:
  - Input: vector
  - Output: vector
  - Learned parameters (weights): projection matrix
  - Operation:
    - Multiply the input vector with the matrix
    - Apply a point-wise nonlinearity, say, ReLU
Forward Pass

- Network of 3 linear layers
- Each layer:
  - Input: vector
  - Output: vector
  - Learned parameters (weights): projection matrix
  - Operation:
    - Multiply the input vector with the matrix
    - Apply a point-wise nonlinearity, say, ReLU
Forward Pass: minibatch of 2 inputs

- Matrix-vector multiplies turn into matrix-matrix multiplies
Simplified Example: Forward Pass, batch of 2

- Matrix-vector multiplies turn into matrix-matrix multiplies
Forward Pass: Compute Loss

- **Loss function:**
  - Produces a loss value that indicates how “wrong” the network was
    - Compares the output to the ground truth for each sample
    - Exact function math varies by task, doesn’t matter for our discussion
- **Goal of training: minimize the loss value**
  - Update network weights so the output closely matches ground truth
Backward Pass

- Goal is to compute the updates to the layer weights
- Achieved by “back propagating” the loss through the layers
  - Each layer computes weight gradient, used to update the weights
  - Each layer computes activation gradient, to be backpropagated to preceding layer
Backward Pass

Compute the weight gradient
- \( dW \): weight gradient (to update weights)
- \( dY \): incoming activation gradient
- \( X \): input activations (from fwd pass)

Compute the activation gradient
- \( dX \): output activation gradient
to backpropagate to the preceding layer
Weight Update

- Also known as ‘optimizer step’
  - Optimizer choices: SGD, Adam, Adagrad, ...

- Input:
  - Current network weights
  - Weight gradients (computed during bwd pass)

- Output: updated weights

- Operation:
  - Increment each weight with the corresponding gradient value
  - In practice, operation is more complex:
    - Update internal state with weight gradient, then update weights using internal state
    - Exact math doesn’t matter for our discussion

- Internal state:
  - 1 or 2 “momenta”
  - Each momentum is as big as the weights
    - Usually fp32 in reduced precision (FP16/BF16) training
    - Optimizer may need 2-6x more memory than just the model

\[
W - lr \times dW = W
\]
Weight Update

- Also known as ‘optimizer step’
  - Optimizer choices: SGD, Adam, Adagrad, ...
- Input:
  - Current network weights
  - Weight gradients (computed during bwd pass)
- Output: updated weights
- Operation:
  - Increment each weight with the corresponding gradient value
  - In practice, operation is more complex:
    - Update internal state with weight gradient, then update weights using internal state
    - Exact math doesn’t matter for our discussion
- Internal state:
  - 1 or 2 “momenta”
  - Each momentum is as big as the weights
    - Usually fp32 in reduced precision (FP16/BF16) training
    - Optimizer may need 2-6x more memory than just the model
Summary of Compute Stages per Layer

Forward Pass

\[ W \times X = Y \]

Backward Pass: weight gradients

\[ dY \times X^T = dW \]

Backward Pass: activation gradients

\[ W^T \times dY = dX \]

- Backward compute is \( \sim 2x \) of forward
- Backward pass requires activations computed during the fwd pass
  - \( X \) in the example (produced by a preceding layer)
  - This can be a major fraction of memory required to train, leading to scale-out for the larger models

Example:
R50 training in fp16 at batch size 256:
- requires \( \sim 15 \) GB of memory
- \( \sim 12 \) GB of that is for activations
Parallelism Taxonomy

Parallel Training

Data Parallel

Model Parallel

Intra Layer

Inter Layer
Data Parallel

- Each worker:
  - Has a copy of the entire neural network model
  - Responsible for compute of a portion of data (training minibatch)
- Forward pass:
  - Computes output activations for its portion of minibatch
  - No communication is needed
- Backward pass:
  - Computes activation gradients for its portion of minibatch
  - Computes contribution to the weight gradient based on its portion of minibatch
    - All workers’ contributions must be summed before weight update
- Weight update:
  - Each worker updates its copy of the model with combined gradients
  - Variants: distributed optimizer
Data Parallel: Forward Pass

- No communication needed
  - Own portion of output becomes own portion of input for next layer
- Backward activation-gradient compute is essentially the same

Worker 0:

\[
W \times X = Y
\]

Worker 1:

\[
W \times X = Y
\]

Worker 2:

\[
W \times X = Y
\]

Worker 3:

\[
W \times X = Y
\]
Data Parallel: Backward Pass

- Each worker computes a different weight gradient ($dW$)
  - Based only on its own unique portion of data
- Weight gradients will have to be communicated so that after update each worker has the same exact weights

Worker 0:

```
\begin{array}{ccc}
dY & X^T & dW \\
\end{array}
```

Worker 1:

```
\begin{array}{ccc}
dY & X^T & dW \\
\end{array}
```

Worker 2:

```
\begin{array}{ccc}
dY & X^T & dW \\
\end{array}
```

Worker 3:

```
\begin{array}{ccc}
dY & X^T & dW \\
\end{array}
```
Data Parallel: Communication

- **Allreduce:**
  - Sum all the workers’ gradients
  - Distribute the sum to all the workers

- **After Allreduce each worker has the same “global” gradient**
  - Can execute a weight update on its own model -> all workers will have the same model

- **Any exposed communication is overhead, thus:**
  - Use efficient communication (hw and sw), overlap communication, etc.
Allreduce Implementation Choices

• Each of $N$ workers is responsible for:
  • Summing $1/N$ gradients collected from $(N - 1)$ peers
  • Distributing the sums to the $(N - 1)$ peers

• “Ring” reduction
  • For any topology that contains a 1D torus (ring)
  • Each worker communicates with 2 neighbors
  • $2(N - 1)$ steps, worker sends/receives $1/N$ of all bytes
    • Each step requires a synchronization -> $2(N - 1)$ syncs total

• “One-shot” reduction:
  • For fully-connected topologies (switches)
  • Each worker communicates with $(N - 1)$ neighbors
  • 2 steps, each with $(N - 1)$ substeps
    • One step per synchronization -> 2 syncs total
  • Allows the use of arithmetic in switches (Mellanox SHARP)
    • Reduces memory accesses and math by the worker
Communication Implementation

• Communication libraries take care of complex details
  • Accelerator can have multiple ports
  • Links can be duplex
  • Pipelining is used to hide latencies and syncs

• NCCL: NVIDIA Collective Communication Library

• Examples:
  • NVIDIA DGX-1
    • Each of 8 GPUs has 6 NVLINK ports
    • Each NVLINK port is duplex
    • GPUs are connected via hybrid mesh
    • NCCL uses multiples of 12 rings are used for allreduce
  • NVIDIA DGX-A100
    • Each of 8 GPUs has 12 NVLINK ports
    • Each NVLINK port is duplex (25 GB/s per direction)
    • GPUs are fully-connected through switches
    • NCCL uses multiples of 24 rings or one-shots are used for allreduce
Communication Overlap

- Data Parallel training can overlap compute and communication
  - Allreduce gradients for layer $K$, while computing gradients for layer $(K - 1)$
  - Cannot be hidden completely - last portion of the pipeline is exposed
  - Tradeoff between communication granularity and link bw utilization
    - Made by training framework SW and libraries like Horovod

- Reduction in switches (Mellanox SHARP) helps free up compute resources
  - Allreduce will compete for resources (memory and math bw) with computation
Distributed Optimizer

- At larger scales optimizer (weight update) can start dominating time
  - Each of $N$ workers does $1/N$ of compute for fwd/bwd passes
  - Each of $N$ workers does all the work to update model weights (stays constant with $N$)
- Solution: distributed optimizer
  - Appeared in: MLPerf v0.6 and later, ZERO paper
  - Include weight update as part of allreduce (each worker is responsible for $1/N^{th}$ of the weights)
    1) Collect and sum up the gradients from peers
    2) Update own portion of the weights ($1/N^{th}$ of the work compared to before)
    3) Broadcast own portion of the updated weights to peers
Data Parallel: Challenges

- **Strong scaling (increase the number of workers, keep minibatch size constant)**
  - Certain layers require minimum minibatch sizes to properly operate
    - Example: batch normalization (BN) generally requires 16+ samples
  - Extra communication is needed between workers when worker minibatch is small
    - Reductions within small subsets of workers

- **Weak scaling (increase the number of workers, increase minibatch size)**
  - Training networks with large minibatches requires hyper-parameter adjustment
    - Learning rate schedule, BN decay, ...
    - Example: R50 (SGD up to bs=16K, LARS above 16K, ...)
  - Often increase the amount of work required to reach the same model accuracy
Workload Increase with Batch Size

- Epochs to reach the same model accuracy (from various submissions to MLPerf v0.7)
  - Epoch = 1 processing pass through entire dataset
Model Parallel

Inter-layer Parallel (aka Pipeline Parallel):
A worker is responsible for its portion of the layers

Intra-layer Parallel:
A worker is responsible for its portion of each layer
Pipeline Parallel

Layer 1
Layer 2
Layer 3
Layer 4
Layer 5

Worker 0
Worker 1
Worker 2

Time

Forward  Loss  Backward
Pipeline Parallel

Worker 0
Worker 1
Worker 2

Layer 1 → Layer 2 → Layer 3 → Layer 4 → Layer 5

Forward  Loss  Backward

Time
Pipeline Parallel

Time

Worker 0
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5

Worker 1

Worker 2

Forward
Loss
Backward
Pipeline Parallel

Time

Worker 0
Worker 1
Worker 2

Layer 1
Layer 2
Layer 3
Layer 4
Layer 5

Forward
Loss
Backward

Forward

Loss

Backward

Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Pipeline Parallel

Time

Layer 1
Layer 2
Layer 3
Layer 4
Layer 5

Worker 0
Worker 1
Worker 2

Forward  Loss  Backward
Pipeline Parallel

Worker 0, Worker 1, Worker 2

Layer 1, Layer 2, Layer 3, Layer 4, Layer 5

Time

Forward, Loss, Backward
Pipeline Parallel

Worker 0
Worker 1
Worker 2

Layer 1
Layer 2
Layer 3
Layer 4
Layer 5

Time

Forward
Loss
Backward
Pipeline Parallel

- Idle bubbles:
  - 67%: 12/18 step-slots
- For $N$ workers:
  - $(N - 1)/N$ idle slots
**Pipeline Parallel: Subminibatches**

- **2 subminibatches**
- **2x more steps**
- Each step is $\frac{1}{2}$ compute
- **Idle bubbles: 50%**
- 12/24 steps-slots
Pipeline Parallel: Subminibatches

- **N workers, K subminibatches:**
  - 2(N + K - 1) steps for fwd/bwd
  - Total step-slots: 2N(N + K - 1)
  - Idle step-slots: 2N(N - 1)
  - Fraction of idle slots: (N - 1)/(N + K - 1)

- **As N grows:**
  - K = N → 50% idle slots
  - K = 4N → 20% idle slots
Pipeline Parallel: Interleaved Layers

- **Benefit:** increases the percentage of time each worker is busy
  - Worker 0 is busy for 4 out of 6 fwd pass steps (compared to 2/4 in the previous slide)
- **Downsides:**
  - Increases communication linearly (with the number of interleaved layers per worker)
  - Problematic if skip connections cross workers
Pipeline Parallel: Communication

- A worker communicates with its 2 neighbors
  - 1D mesh topology
  - 1D torus when interleaving layers
- Communication in each step of the fwd and bwd pass
  - Activations in fwd, activation gradients in bwd
- Communication very hard to overlap with computation
Pipeline Parallel: Challenges

• Lots of hard hard to hide communication
• Idle slots reduce scaling efficiency
  • Many subminibatches help with this, but run into the same problems as strong-scaling of data-parallel
• Load balancing workload across workers is difficult
  • Different layers of a network can take different amounts of time
  • Leads to even busy slots for other workers idling for portions of time
Model Parallel: Intra-Layer Parallel

- Partition a given layer’s weights among the workers
- Addresses some of the Pipeline Parallel challenges
  - Idle slots, load imbalance
- Two variants:
  - Row-wise partitioning
  - Column-wise partitioning
Row-wise Partitioning

Each worker:
- Has a portion of weight rows
- All of input activations
- Computes a portion of output activations

Fwd communication:
- Allgather: next layer needs all activations
Column-wise Partitioning

- Each worker:
  - Has a portion of weight columns
  - Has a portion of input activations
  - Computes partial activations

- Fwd communication:
  - Reduce_scatter: next layer needs full activations
Reducing Synchronization By Alternating Partitioning

- Note: no communication is needed
- Worker $i$ produces output, which is its input for the next layer
Reducing Synchronization By Alternating Partitioning

Row-wise partitioning

Worker 0

Worker 1

Worker 2

Layer $K$ fwd

Col partitioning

Layer $(K+1)$ fwd

Communication: Allreduce

Layer $(K+2)$ fwd
Intra-Layer Parallel: Communication

- Row-wise in fwd becomes Col-wise in bwd
- Col-wise in fwd becomes Row-wise in bwd
- **Row-wise:**
  - Fwd: allgather
  - Bwd: reduce_scatter
- **Col-wise:**
  - Fwd: reduce_scatter
  - Bwd: allgather
- **When row- and col- are alternated:**
  - Allreduce every two layers, in fwd and bwd
  - Halves the synchronizations compared to not alternating
Communication Pattern Summary

- **Data Parallel:**
  - `Allreduce` of weights
  - Can be overlapped with computation

- **Pipeline Parallel:**
  - `Point-wise` communication of activations and activation gradients
  - Hard to overlap with computation
  - Hard to load-balance

- **Intra-layer Parallel:**
  - `Allgather`, `Reduce_scatter` of activations and activation gradients
  - `Allreduce` if row-wise and col-wise partitioning is alternated
  - Hard to overlap with computation

- **Hybrid Parallel: some layers data parallel, some layer model-parallel**
  - Common for recommendation networks (model parallel embeddings, data-parallel MLP)
  - `Alltoall` of activations and activation gradients: each pair of workers exchange unique values
    - Most performant on switched or fully connected topologies
  - Hard to overlap with computation
Summary

- Networks and dataset are getting larger to set new state of art results
- Scale-out enables these networks to be trained
- Success requires many optimized components:
  - Hardware:
    - Fast accelerators for DL
    - High-bandwidth, low-latency interconnects
      - Topologies matter (must match communication patterns)
      - Network switches with math capabilities free up DL accelerators to do compute
  - Software:
    - Math libraries (CUDNN, CUBLAS, MKL, ...)
    - Collective communication libraries (NCCL, Horovod, ...)
    - Training frameworks (MxNet, PyTorch, TensoFlow, HugeCTR, ...)
  - Proper choice of parallelism (manual, MeshTensorFlow, Gshard, WSE)
Throughput Improvements, MLPerf v0.5 → v0.6
Largest Improvements were due to Scale-Out SW

Identical machines submitted to v0.5 and v0.6
- Same chips, chip count, interconnect
- Adjusted for epoch differences
  - Due to some rule and hyper-parameter changes

Patterned bars: multi-node
MLPerf Submission Scale in Chips
MLPerf Submission Scale in Chips, Log Scale