Distributed Deep Learning: Leveraging Heterogeneity and Data-Parallelism

Jæmin Choi

University of Illinois Urbana-Champaign

May 1, 2019
A Quick Introduction to Deep Learning

- Feed data into model (neural network)
- Model has many (deep) layers of neurons
- Model learns from existing data (i.e. training) and outputs predictions for new data (i.e. inference)
- Applications
  - Image classification
  - Natural language processing (NLP)
  - Autonomous driving

Mini-batch Training

- Feeding training samples one by one is the most accurate but slow.
- Feed data in small **batches** to speed up tensor operations.
  - Too big batches usually hurt convergence.
  - Typical batch sizes: 128, 256.
- Going through the entire dataset once is called an **epoch**.
- Training process (repeat for all batches & epochs):
  1. Load batch into memory
  2. **Forward pass**
  3. Compare model output with labels, compute loss function
  4. **Backward propagation**
  5. Update model based on the gradients.
Why Distributed Deep Learning?

- Training with single device/node is too slow
- Model is too big to fit in memory
Distributed Deep Learning

- Perform training in distributed memory
- Approaches: data-parallel vs. model-parallel
- **Data-parallel**
  - Most widely used approach
  - Partition the dataset/batch between **workers**
  - Each worker has a copy of model
  - Usually 1 worker per device
  - E.g. 4 GPUs & batch size 128 $\rightarrow$ batch size 32 per worker
  - For each partitioned batch, train individually
    $\rightarrow$ **aggregate gradients** (e.g. all-reduce)
- Model-parallel: partition the model, *not* data
Synchronize or Not is the Question

- **Synchronous SGD**
  - Workers synchronize after training every (partitioned) batch
  - Usually using all-reduce
  - Has *straggler problem*

- **Asynchronous SGD**
  - Allow workers to proceed without waiting for gradient updates from other workers
  - Problem of *stale weights*
Heterogeneous Training

- All DL frameworks use either CPU or GPU, not both
- GPUs are favored over CPUs due to tensor computation speed
- But why not use both together?
  - On cloud environments, GPUs will be more cost-effective
  - On HPC environments, CPUs just sit idle
- Can also be used with GPUs of varying compute capabilities
- **Goal**: Perform distributed & heterogeneous training
- **Main challenges**
  - Reconcile training speed difference
  - Gradient aggregation between workers
Batch Partitioning

Total batch size must be kept the same

Give smaller batch partition to a slower worker (usually CPU), so that training speeds match between workers

Need **weighted gradient aggregation** to prevent bias
  - More weight to bigger partition
Heterogeneous All-reduce

- All-reduce among all CPU and GPU workers
  - Synchronous SGD

**Strategy 1**
1. Move GPU gradients to host memory
2. Add all gradients in host memory using OpenMP
3. MPI all-reduce
4. Move gradients back to GPU

**Strategy 2**
1. Move CPU gradients to GPU
2. NCCL all-reduce
3. Move gradients back to CPU
Heterogeneous All-reduce

2-node Performance

- PSC Bridges: 1 CPU worker (2 sockets), 2 GPU workers (2 GPUs)
- Default is strategy 2, much faster at larger data sizes
Applying Heterogeneous Training

- Framework is in place (using PyTorch)

- Which applications are suitable?
  - Image classification
    - Uses CNNs
    - GPU has much better performance
  
  - NLP
    - Uses RNNs & LSTMs
    - CPU has comparable performance

- Machine translation with Google’s Transformer model
  - Link to Google’s blog

- Image captioning with a pre-trained CNN (ResNet-152) as encoder and LSTM as decoder
  - Link to PyTorch tutorial
Problem: Variability in Batch Processing

- Implemented heterogeneous & distributed training, works correctly
- But significantly slower than homogeneous training (using only GPUs), why?
  - A lot of idle time before all-reduce
  - Although batch was partitioned to have matching training times on CPU and GPU on average,
  - Actual times differ significantly
  - Average time: 1.9 s

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU 1</th>
<th>GPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch 1</td>
<td>1.9 s</td>
<td>2.5 s</td>
<td>2.5 s</td>
</tr>
<tr>
<td>Batch 2</td>
<td>2.3 s</td>
<td>1.4 s</td>
<td>1.4 s</td>
</tr>
<tr>
<td>Batch 3</td>
<td>1.5 s</td>
<td>1.8 s</td>
<td>1.8 s</td>
</tr>
</tbody>
</table>

* Total batch size: 128, CPU: 32, GPU: 96
Ongoing Work

- Find out what is causing the variability
  - See if same issue occurs with other frameworks
  - Currently trying out MXNet, only 10% variability with MNIST
- Apply asynchronous SGD training
- Performance evaluation
Thank You