Lecture 19:
Parallel Deep Network Training

Parallel Computer Architecture and Programming
CMU / 清华大学, Summer 2017
How would you describe this professor?

Easy?
Mean?
Boring?
Nerdy?
Professor classification task

Classifies professors as easy, mean, boring, or nerdy based on their appearance.

Input:
image of a professor

Output:
probability of each of four possible labels

\[ f(\text{image}) \]
“professor classifier”

Easy: ??
Mean: ??
Boring: ??
Nerdy: ??
Professor classification network

Classifies professors as easy, mean, boring, or nerdy based on their appearance.

Input: image of a professor

Output: probability of label

Recall: large networks may have 10's-100's of millions of parameters
Professor classification network

- Max-pooling layers follow first, second, and fifth convolutional layers.
- The number of neurons in each layer is given by 253440, 186624, 64896, 64896, 43264, 4096, 4096, 1000.

Easy: 0.26
Mean: 0.08
Boring: 0.14
Nerdy: 0.52
Training data (ground truth answers)
Professor classification network

New image of Kayvon (not in training set)

Ground truth (what the answer should be)

- Easy: 0.0
- Mean: 0.0
- Boring: 0.0
- Nerdy: 1.0

Network output

- Easy: 0.26
- Mean: 0.08
- Boring: 0.14
- Nerdy: 0.52
Error (loss)

Ground truth:
(what the answer should be)

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy</td>
<td>0.0</td>
</tr>
<tr>
<td>Mean</td>
<td>0.0</td>
</tr>
<tr>
<td>Boring</td>
<td>0.0</td>
</tr>
<tr>
<td>Nerdy</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Network output: *

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy</td>
<td>0.26</td>
</tr>
<tr>
<td>Mean</td>
<td>0.08</td>
</tr>
<tr>
<td>Boring</td>
<td>0.14</td>
</tr>
<tr>
<td>Nerdy</td>
<td>0.52</td>
</tr>
</tbody>
</table>

Common example: softmax loss:

\[ L = -log \left( \frac{e^{f_c}}{\sum_j e^{f_j}} \right) \]

* In practice a network using a softmax classifier outputs unnormalized, log probabilities \((f_j)\), but I’m showing a probability distribution above for clarity.
Training

Goal of training: learning good values of network parameters so that the network outputs the correct classification result for any input image

Idea: minimize loss for all the training examples (for which the correct answer is known)

\[ L = \sum_i L_i \]  
(total loss for entire training set is sum of losses \( L_i \) for each training example \( x_i \))

Intuition: if the network gets the answer correct for a wide range of training examples, then hopefully it has learned parameter values that yield the correct answer for future images as well.
Intuition: gradient descent

Say you had a function $f$ that contained hidden parameters $p_1$ and $p_2$: $f(x_i)$

And for some input $x_i$, your training data says the function should output 0.

But for the current values of $p_1$ and $p_2$, it currently outputs 10.

$f(x_i, p_1, p_2) = 10$

And say I also gave you expressions for the derivative of $f$ with respect to $p_1$ and $p_2$ so you could compute their value at $x_i$.

$$\frac{df}{dp_1} = 2 \quad \frac{df}{dp_2} = -5 \quad \nabla f = [2, -5]$$

How might you adjust the values $p_1$ and $p_2$ to reduce the error for this training example?
Basic gradient descent

while (loss too high):
    for each item $x_i$ in training set:
        grad += evaluate_loss_gradient($f$, params, loss_func, $x_i$)

    params += -grad * step_size;

Mini-batch stochastic gradient descent (mini-batch SGD):
choose a random (small) subset of the training examples to use to compute the
gradient in each iteration of the while loop

How do we compute $d\text{Loss}/dp$ for a deep neural network with millions of parameters?
Derivatives using the chain rule

\[ f(x, y, z) = (x + y)z = az \]

Where: \( a = x + y \)

\[
\frac{df}{da} = z \quad \frac{da}{dx} = 1 \quad \frac{da}{dy} = 1
\]

So, by the derivative chain rule:

\[
\frac{df}{dx} = \frac{df}{da} \frac{da}{dx} = z
\]

Red = output of node
Blue = \( \frac{df}{d\text{node}} \)
Backpropagation

**Red = output of node**

**Blue = df/dnode**

**Recall:**
\[
\frac{df}{dx} = \frac{df}{dg} \frac{dg}{dx}
\]

1. \( g(x, y) = x + y \)
   \[
   \frac{dg}{dx} = 1, \quad \frac{dg}{dy} = 1
   \]

2. \( g(x, y) = \max(x, y) \)
   \[
   \frac{dg}{dx} = 1, \text{ if } x > y \\
   0, \text{ otherwise}
   \]

3. \( g(x, y) = xy \)
   \[
   \frac{dg}{dx} = y, \quad \frac{dg}{dy} = x
   \]
Back-propagating through single unit

Recall: behavior of unit:
\[ f(x_0, x_1, x_2, x_3) = \max \left( 0, \sum_i x_i w_i + b \right) \]

Observe: output of prior layer must be retained in order to compute weight gradients for this unit during backprop.
Multiple uses of an input variable

\[ g(x, y) = (x + y) + x \times x = a + b \]
\[
\frac{da}{dx} = 1, \quad \frac{db}{dx} = 2x
\]
\[
\frac{dg}{dx} = \frac{dg}{da} \frac{da}{dx} + \frac{dg}{db} \frac{db}{dx} = 2x + 1
\]

Implication: backpropagation through all units in a convolutional layer adds gradients computed from each unit to the overall gradient for the shared weights

Sum gradients from each use of variable:

Here:
\[
\frac{df}{dx} = \frac{df}{dg} \frac{dg}{dx}
\]
\[
= 10 \frac{dg}{dx}
\]
\[
= 10(2x + 1)
\]
\[
= 10(10 + 1) = 110
\]
Back-propagation: matrix form

\[ y = Xw \]

\[ \frac{dL}{dw} = X^T \frac{dL}{dy} \]

\[ \frac{dy_j}{dw_i} = X_{ji} \]

\[ \frac{dL}{dw_i} = \sum_j \frac{dL}{dy_j} \frac{dy_j}{dw_i} = \sum_j \frac{dL}{dy_j} X_{ji} \]

Therefore:

\[ \frac{dL}{dw} = X^T \frac{dL}{dy} \]
Backpropagation through the entire professor classification network

For each training example $x_i$ in mini-batch:

- Perform forward evaluation to compute loss for $x_i$
- Compute gradient of loss w.r.t. final layer’s outputs
- Backpropagate gradient to compute gradient of loss w.r.t. all network parameters
- Accumulate gradients (over all images in batch)

Update all parameter values: $w_{\text{new}} = w_{\text{old}} - \text{step\_size} \times \text{grad}$
Recall from last class: VGG memory footprint

Calculations assume 32-bit values (image batch size = 1)

<table>
<thead>
<tr>
<th>Layer Description</th>
<th>Output Size (per image)</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>input: 224 x 224 RGB image</td>
<td>224x224x3</td>
<td>150K</td>
</tr>
<tr>
<td>conv: (3x3x3) x 64</td>
<td>224x224x64</td>
<td>12.3 MB</td>
</tr>
<tr>
<td>conv: (3x3x64) x 64</td>
<td>224x224x64</td>
<td>12.3 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>112x112x64</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>conv: (3x3x128) x 128</td>
<td>112x112x128</td>
<td>6.2 MB</td>
</tr>
<tr>
<td>conv: (3x3x128) x 128</td>
<td>112x112x128</td>
<td>6.2 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>56x56x128</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>conv: (3x3x256) x 256</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>conv: (3x3x256) x 256</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>28x28x256</td>
<td>766 KB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>maxpool</td>
<td>7x7x512</td>
<td>98 KB</td>
</tr>
<tr>
<td>fully-connected 4096</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>fully-connected 4096</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>fully-connected 1000</td>
<td>1000</td>
<td>4 KB</td>
</tr>
<tr>
<td>soft-max</td>
<td>1000</td>
<td>4 KB</td>
</tr>
</tbody>
</table>

Many weights in fully-connected players

Storing convolution layer outputs (unit “activations”) can get big in early layers with large input size and many filters

Note: multiply these numbers by N for batch size of N images
Data lifetimes during network evaluation

- Max-pooling layers follow first, second, and fifth convolutional layers
- The number of neurons in each layer is given by 253440, 186624, 64896, 64896, 43264, 4096, 4096, 1000

Weights (read-only) reside in memory

After evaluating layer i, can free outputs from layer i-1
Data lifetimes during training

- Must retain outputs for all layers because they are needed to compute gradients during back-prop
- Parallel back-prop will require storage for per-weight gradients (more about this in a second)
- In practice: may also store per-weight gradient velocity (if using SGD with “momentum”) or step size cache in adaptive step size schemes like Adagrad

\[
\text{vel}_{\text{new}} = \mu \times \text{vel}_{\text{old}} - \text{step\_size} \times \text{grad}
\]
\[
\text{w}_{\text{new}} = \text{w}_{\text{old}} + \text{vel}_{\text{new}}
\]
# VGG memory footprint

Calculations assume 32-bit values (image batch size = 1)

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Input Size</th>
<th>Weights Mem</th>
<th>Output Size (Per Image)</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: 224 x 224 RGB image</td>
<td>—</td>
<td>—</td>
<td>224x224x3</td>
<td>150K</td>
</tr>
<tr>
<td>Conv: (3x3x3) x 64</td>
<td>—</td>
<td>6.5 KB</td>
<td>224x224x64</td>
<td>12.3 MB</td>
</tr>
<tr>
<td>Conv: (3x3x64) x 64</td>
<td>—</td>
<td>144 KB</td>
<td>224x224x64</td>
<td>12.3 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>—</td>
<td>112x112x64</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>Conv: (3x3x64) x 128</td>
<td>—</td>
<td>228 KB</td>
<td>112x112x128</td>
<td>6.2 MB</td>
</tr>
<tr>
<td>Conv: (3x3x128) x 128</td>
<td>—</td>
<td>576 KB</td>
<td>112x112x128</td>
<td>6.2 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>—</td>
<td>56x56x128</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>Conv: (3x3x128) x 256</td>
<td>—</td>
<td>1.1 MB</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>Conv: (3x3x256) x 256</td>
<td>—</td>
<td>2.3 MB</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>Conv: (3x3x256) x 256</td>
<td>—</td>
<td>2.3 MB</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>—</td>
<td>28x28x256</td>
<td>766 KB</td>
</tr>
<tr>
<td>Conv: (3x3x256) x 512</td>
<td>—</td>
<td>4.5 MB</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>—</td>
<td>9 MB</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>—</td>
<td>9 MB</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>—</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>—</td>
<td>9 MB</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>—</td>
<td>9 MB</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>—</td>
<td>9 MB</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>—</td>
<td>7x7x512</td>
<td>98 KB</td>
</tr>
<tr>
<td>Fully-connected 4096</td>
<td>—</td>
<td>—</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>Fully-connected 4096</td>
<td>—</td>
<td>—</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>Fully-connected 1000</td>
<td>—</td>
<td>—</td>
<td>1000</td>
<td>4 KB</td>
</tr>
<tr>
<td>Soft-max</td>
<td>—</td>
<td>—</td>
<td>1000</td>
<td>4 KB</td>
</tr>
</tbody>
</table>

**Note:**
- The calculations assume 32-bit values.
- The output size is per image.
- The memory is calculated as per-image values multiplied by the mini-batch size.
- Unlike forward evaluation, cannot immediately free outputs once consumed by the next level of the network. Must also store per-weight gradients. Many implementations also store gradient "momentum" as well (multiply by 3).
SGD workload

while (loss too high):
    for each item $x_i$ in mini-batch:
        grad += evaluate_loss_gradient($f$, loss_func, $\text{params}$, $x_i$)
        sum reduction

    params += -grad * step_size;
    trivial data-parallel over parameters

At first glance, this loop is sequential (each step of “walking downhill” depends on previous)

Parallel across images

large computation with its own parallelism (but working set may not fit on single machine)
DNN training workload

- **Huge computational expense**
  - Must evaluate the network (forward and backward) for millions of training images
  - Must iterate for many iterations of gradient descent (100’s of thousands)
  - Training modern networks on big datasets takes days

- **Large memory footprint**
  - Must maintain network layer outputs from forward pass
  - Additional memory to store gradients/gradient velocity for each parameter
  - Recall parameters for popular VGG-16 network require ~500 MB of memory (training requires GBs of memory for academic networks)
  - Scaling to larger networks requires partitioning DNN across nodes to keep DNN + intermediates in memory

- **Dependencies /synchronization (not embarrassingly parallel)**
  - Each parameter update step depends on previous
  - Many units contribute to same parameter gradients (fine-scale reduction)
  - Different images in mini batch contribute to same parameter gradients
Data-parallel training (across images)

for each item $x_i$ in mini-batch:
  
  grad += evaluate_loss_gradient(f, loss_func, params, $x_i$)
  
  params += -grad * step_size;

Consider parallelization of the outer for loop across machines in a cluster

partition mini-batch across nodes

for each item $x_i$ in mini-batch assigned to local node:
  
  // just like single node training
  
  grad += evaluate_loss_gradient(f, loss_func, params, $x_i$)
  
  barrier();

sum reduce gradients, communicate results to all nodes

barrier();

update copy of parameter values
Challenges of computing at cluster scale

- Slow communication between nodes
  - Commodity clusters do not feature high-performance interconnects (e.g., infiniband) typical of supercomputers

- Nodes with different performance (even if machines are the same)
  - Workload imbalance at barriers (sync points between nodes)

Modern solution: exploit characteristics of SGD using asynchronous execution!
Parameter server design

Pool of worker nodes

Worker Node 0
Worker Node 1
Worker Node 2
Worker Node 3

Parameter Server

Parameter Server [Li OSDI14]
Google’s DistBelief [Dean NIPS12]
Microsoft’s Project Adam [Chilimbi OSDI14]
Training data partitioned among workers

Pool of worker nodes

Worker Node 0
- training data

Worker Node 1
- training data

Worker Node 2
- training data

Worker Node 3
- training data

Parameter Server
- parameter values ($v_0$)

$X_0 - X_{1000}$
$X_{1000} - X_{2000}$
$X_{2000} - X_{3000}$
$X_{3000} - X_{4000}$
Copy of parameters sent to workers

Pool of worker nodes

Parameter Server

Worker Node 0

Worker Node 1

Worker Node 2

Worker Node 3

training data

local copy of parameters (v0)

training data

local copy of parameters (v0)

training data

local copy of parameters (v0)

training data

local copy of parameters (v0)

params v0

params v0

params v0

params v0
Workers independently compute local “subgradients"
Worker sends subgradient to parameter server

Pool of worker nodes

- Worker Node 0
  - training data
  - local copy of parameters (v0)
  - local subgradients

- Worker Node 1
  - training data
  - local copy of parameters (v0)
  - local subgradients

- Worker Node 2
  - training data
  - local copy of parameters (v0)
  - local subgradients

- Worker Node 3
  - training data
  - local copy of parameters (v0)
  - local subgradients

Parameter Server
- parameter values (v0)
Server updates global parameter values based on subgradient

Worker Node 0
- training data
- local copy of parameters (v0)
- local subgradients

Worker Node 1
- training data
- local copy of parameters (v0)
- local subgradients

Worker Node 2
- training data
- local copy of parameters (v0)
- local subgradients

Worker Node 3
- training data
- local copy of parameters (v0)
- local subgradients

Server
- parameter values (v1)

params += -subgrad * step_size;
Updated parameters sent to worker
Worker proceeds with another gradient computation step

Note:
Node 1 is operating on different set of parameter values than other nodes
Those parameter values were computed without gradient information from the other nodes
Updated parameters sent to worker (again)

Worker Node 0
- training data
- local copy of parameters (v0)
- local subgradients

Worker Node 1
- training data
- local copy of parameters (v1)
- local subgradients

Worker Node 2
- training data
- local copy of parameters (v0)
- local subgradients

Worker Node 3
- training data
- local copy of parameters (v0)
- local subgradients

Parameter Server
- parameter values (v1)

Subgradient
Worker continues with updated parameters

Worker Node 0
- training data
- local copy of parameters (v0)
- local subgradients

Worker Node 1
- training data
- local copy of parameters (v1)
- local subgradients

Worker Node 2
- training data
- local copy of parameters (v0)
- local subgradients

Worker Node 3
- training data
- local copy of parameters (v2)
- local subgradients

Parameter Server
- parameter values (v2)

params v2
Summary: asynchronous parameter update

- **Idea:** avoid global synchronization on all parameter updates between each SGD iteration
  - Design reflects realities of cluster computing:
    - Slow interconnects
    - Unpredictable machine performance

- **Solution:** asynchronous (and partial) subgradient updates

- **Will impact convergence of SGD**
  - Node N working on iteration $i$ may not have parameter values that result the results of the $i-1$ prior SGD iterations
Bottleneck?
What if there is heavy contention for parameter server?

Worker Node 0
- training data
- local copy of parameters (v0)
- local subgradients

Worker Node 1
- training data
- local copy of parameters (v1)
- local subgradients

Worker Node 2
- training data
- local copy of parameters (v0)
- local subgradients

Worker Node 3
- training data
- local copy of parameters (v2)
- local subgradients

Parameter Server
- parameter values (v2)
Shard the parameter server

Partition parameters across servers
Worker sends chunk of subgradients to owning parameter server

Reduces data transmission load on individual servers
(less important: also reduces cost of parameter update)
What if model parameters do not fit on one worker?

Recall high footprint of training large networks (particularly with large mini-batch sizes)
Model parallelism

Partition network parameters across nodes (spatial partitioning to reduce communication)

Reduce internode communication through network design:
- Use small spatial convolutions (1x1 convolutions)
- Reduce/shrink fully-connected layers

Convolutional layers: only need to communicate outputs near spatial partition

Fully-connected layers: all data owned by a node must be communicated to other nodes
Training data-parallel and model-parallel execution

Working on subgradient computation for a single copy of the model

Worker Node 0
- training data
- local copy of parameters (v1): chunk 0
- local subgradients chunk 0

Worker Node 1
- training data
- local copy of parameters (v1): chunk 1
- local subgradients chunk 1

Parameter Server 0
- parameter values (chunk 0)

Worker Node 2
- training data
- local copy of parameters (v0): chunk 0
- local subgradients chunk 0

Worker Node 3
- training data
- local copy of parameters (v0): chunk 1
- local subgradients chunk 1

Parameter Server 1
- parameter values (chunk 1)

Fine-grained communication of layer outputs, subgradients, etc.
Using supercomputers for training?

- Fast interconnects critical for model-parallel training
  - Fine-grained communication of outputs and gradients
- Fast interconnect diminishes need for async training algorithms
  - Avoid randomness in training due to computation schedule (there remains randomness due to SGD algorithm)

OakRidge Titan Supercomputer (Cray low-latency interconnect)
NVIDIA DGX-1: 8 Pascal GPUs connected via high speed NV-Link interconnect
Accelerating data-parallel training

- Use a high-performance Cray Gemini interconnect (Titan supercomputer)
- Use combining tree for accumulating gradients (rather than a single parameter server)
- Use larger batch size (to reduce frequency of communication) and offset by increasing learning rate

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Net</th>
<th>Epochs</th>
<th>Batch size</th>
<th>Initial Learning Rate</th>
<th>Train time</th>
<th>Speedup</th>
<th>Top-1 Accuracy</th>
<th>Top-5 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffe</td>
<td>1 NVIDIA K20</td>
<td>GoogLeNet [41]</td>
<td>64</td>
<td>32</td>
<td>0.01</td>
<td>21 days</td>
<td>1x</td>
<td>68.3%</td>
</tr>
<tr>
<td>FireCaffe (ours)</td>
<td>32 NVIDIA K20s (Titan supercomputer)</td>
<td>GoogLeNet</td>
<td>72</td>
<td>1024</td>
<td>0.08</td>
<td>23.4 hours</td>
<td>20x</td>
<td>68.3%</td>
</tr>
<tr>
<td>FireCaffe (ours)</td>
<td>128 NVIDIA K20s (Titan supercomputer)</td>
<td>GoogLeNet</td>
<td>72</td>
<td>1024</td>
<td>0.08</td>
<td>10.5 hours</td>
<td>47x</td>
<td>68.3%</td>
</tr>
</tbody>
</table>

Dataset: ImageNet 1K

Result: reasonable scalability without asynchronous parameter update: for modern DNNs with fewer weights such as GoogLeNet (due to no fully connected layers)

Also see: Goyal et al. 2017
“Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”
swDNN: DNN library for TaihuLight

https://github.com/THUHPGC/swDNN

Sunway TaihuLight: 40,960 x 256 core processors = 10,485,760 CPU cores
Summary: training large networks in parallel

- Many cluster systems rely on asynchronous update to efficiently use clusters of commodity machines
  - Modification of SGD algorithm to meet constraints of modern parallel systems
  - Effects on convergence are problem dependent and not particularly well understood
  - Efficient use of fast interconnects may provide alternative to these methods (facilitate tightly orchestrated solutions much like supercomputing applications)

- Although modern DNN designs (with fewer weights), large mini batch sizes, and efficient use of high performance interconnects (much like any parallel computing problem) enables scalability without asynchronous execution

- High-performance training of deep networks is an interesting example of constant iteration of algorithm design and parallelization strategy (a key theme of this course! recall the original grid solver example!)