Lecture 9:

Training Deep Neural Networks (in parallel)

Visual Computing Systems
CMU 15-769, Fall 2016
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

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

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: 10’s-100’s of millions of parameters
Professor classification network

Our model

- 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)

Nerdy

Nerdy
Professor classification network

New image of Kayvon (not in training set)

- 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

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

Ground truth (what the answer should be)

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

**Network output:** *

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

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

**Common example: softmax loss:**

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

\[ L = \sum_i L_i \]  \hspace{1cm} \text{(total loss for entire training set is sum of losses } L_i \text{ 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 compute gradient in each
iteration of the while loop

How do we compute dLoss/dp for a deep neural network with millions of parameters?
Derivatives using the chain rule

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

Where: \( a = x + y \)

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

So, by the derivative chain rule:

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

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

Red = output of node
Blue = df/dnode

Recall: \( \frac{df}{dx} = \frac{df}{dg} \frac{dg}{dx} \)

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

**Example 2:**
\[ g(x, y) = \max(x, y) \]
\[ \frac{dg}{dx} = \begin{cases} 1, & \text{if } x > y \\ 0, & \text{otherwise} \end{cases} \]

**Example 3:**
\[ g(x, y) = xy \]
\[ \frac{dg}{dx} = y, \quad \frac{dg}{dy} = x \]
Backpropagating through single unit

Recall: behavior of unit:

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

let \( y = 10 \), if upper input to max is > 0

\( 0 \), otherwise

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

Let's consider a function $g(x, y) = (x + y) + x \times x = a + b$

Differentiating with respect to $x$:

$$\frac{da}{dx} = 1, \quad \frac{db}{dx} = 2x$$

Now, differentiate $g$ with respect to $x$:

$$\frac{dg}{dx} = \frac{dg}{da} \frac{da}{dx} + \frac{dg}{db} \frac{db}{dx} = 2x + 1$$

**Sum gradients from each use of variable:**

Here:

$$\frac{df}{dx} = \frac{df}{dg} \frac{dg}{dx} = 10 \frac{dg}{dx} = 10(2x + 1) = 110$$

**Implication:** Backpropagation through all units in a convolutional layer adds gradients computed from each unit to the overall gradient for the shared weights.
Backpropagation: matrix form

\[ y = Xw \]

\[ \frac{dL}{dw} \]

\[ \frac{dL}{dy} \]

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

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

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

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: $w_{new} = w_{old} - \text{step_size} \times \text{grad}$
Recall from last class: VGG memory footprint

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

<table>
<thead>
<tr>
<th>Layer Description</th>
<th>Weights Memory (512)</th>
<th>Output Size (per image)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: 224 x 224 RGB image</td>
<td>—</td>
<td>224x224x3 150K</td>
</tr>
<tr>
<td>Conv: (3x3x3) x 64</td>
<td>6.5 KB</td>
<td>224x224x64 12.3 MB</td>
</tr>
<tr>
<td>Conv: (3x3x64) x 64</td>
<td>144 KB</td>
<td>224x224x64 12.3 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>112x112x64 3.1 MB</td>
</tr>
<tr>
<td>Conv: (3x3x64) x 128</td>
<td>228 KB</td>
<td>112x112x128 6.2 MB</td>
</tr>
<tr>
<td>Conv: (3x3x128) x 128</td>
<td>576 KB</td>
<td>112x112x128 6.2 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>56x56x128 1.5 MB</td>
</tr>
<tr>
<td>Conv: (3x3x128) x 256</td>
<td>1.1 MB</td>
<td>56x56x256 3.1 MB</td>
</tr>
<tr>
<td>Conv: (3x3x256) x 256</td>
<td>2.3 MB</td>
<td>56x56x256 3.1 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>28x28x256 766 KB</td>
</tr>
<tr>
<td>Conv: (3x3x256) x 512</td>
<td>4.5 MB</td>
<td>28x28x512 1.5 MB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>28x28x512 1.5 MB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>28x28x512 1.5 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>14x14x512 383 KB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>14x14x512 383 KB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>14x14x512 383 KB</td>
</tr>
<tr>
<td>Maxpool</td>
<td>—</td>
<td>7x7x512 98 KB</td>
</tr>
<tr>
<td>Fully-connected 4096</td>
<td>392 MB</td>
<td>4096 16 KB</td>
</tr>
<tr>
<td>Fully-connected 4096</td>
<td>64 MB</td>
<td>4096 16 KB</td>
</tr>
<tr>
<td>Fully-connected 1000</td>
<td>15.6 MB</td>
<td>1000 4 KB</td>
</tr>
<tr>
<td>Soft-max</td>
<td>—</td>
<td>1000 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

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 cache in Adagrad

\[
\text{vel\_new} = \mu \times \text{vel\_old} - \text{step\_size} \times \text{grad}
\]

\[
\text{w\_new} = \text{w\_old} + \text{vel\_new}
\]
# VGG memory footprint

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

<table>
<thead>
<tr>
<th>Layer Description</th>
<th>Weights Mem (per image)</th>
<th>Output Size (mem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: 224 x 224 RGB image</td>
<td></td>
<td>224x224x3 150K</td>
</tr>
<tr>
<td>Conv: (3x3x3) x 64</td>
<td>6.5 KB</td>
<td>224x224x64 12.3 MB</td>
</tr>
<tr>
<td>Conv: (3x3x64) x 64</td>
<td>144 KB</td>
<td>224x224x64 12.3 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>112x112x64 3.1 MB</td>
</tr>
<tr>
<td>Conv: (3x3x64) x 128</td>
<td>228 KB</td>
<td>112x112x128 6.2 MB</td>
</tr>
<tr>
<td>Conv: (3x3x128) x 128</td>
<td>576 KB</td>
<td>112x112x128 6.2 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>56x56x128 1.5 MB</td>
</tr>
<tr>
<td>Conv: (3x3x128) x 256</td>
<td>1.1 MB</td>
<td>56x56x256 3.1 MB</td>
</tr>
<tr>
<td>Conv: (3x3x256) x 256</td>
<td>2.3 MB</td>
<td>56x56x256 3.1 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>28x28x256 766 KB</td>
</tr>
<tr>
<td>Conv: (3x3x256) x 512</td>
<td>4.5 MB</td>
<td>28x28x512 1.5 MB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>28x28x512 1.5 MB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>28x28x512 1.5 MB</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>14x14x512 383 KB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>14x14x512 383 KB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>14x14x512 383 KB</td>
</tr>
<tr>
<td>Conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>14x14x512 383 KB</td>
</tr>
<tr>
<td>Maxpool</td>
<td></td>
<td>7x7x512 98 KB</td>
</tr>
<tr>
<td>Fully-connected 4096</td>
<td>392 MB</td>
<td>4096 16 KB</td>
</tr>
<tr>
<td>Fully-connected 4096</td>
<td>64 MB</td>
<td>4096 16 KB</td>
</tr>
<tr>
<td>Fully-connected 1000</td>
<td>15.6 MB</td>
<td>1000 4 KB</td>
</tr>
</tbody>
</table>

* inputs/outputs get multiplied by mini-batch size

* Unlike forward evaluation: 1. cannot immediately free outputs once consumed by next level of network

* Must also store per-weight gradients

* Many implementations also store gradient "momentum" as well (multiply by 3)

CMU 15-769, Fall 2016
SGD workload

while (loss too high):

  for each item \( x_i \) in mini-batch:
    grad += evaluate_loss_gradient(f, loss_func, params, x_i)

  params += -grad * step_size;

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)

trivial data-parallel over parameters
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 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

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

Pool of worker nodes

Parameter Server

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

Pool of worker nodes

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

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

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

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

Parameter Server
- parameter values (v0)

params v0

CMU 15-769, Fall 2016
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

```
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)

- Training data
- Local copy of parameters ($v_0$)
- Local subgradients

Worker Node 0

- Training data
- Local copy of parameters ($v_1$)
- Local subgradients

Worker Node 1

- Training data
- Local copy of parameters ($v_0$)
- Local subgradients

Worker Node 2

- Training data
- Local copy of parameters ($v_0$)
- Local subgradients

Worker Node 3

Parameter Server

Parameter values ($v_1$)

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 (v0)
- 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)

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 0
- parameter values (chunk 0)

Parameter Server 1
- parameter values (chunk 1)

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
Training data-parallel and model-parallel execution

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

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 0
- parameter values (chunk 0)

Parameter Server 1
- parameter values (chunk 1)

Working on subgradient computation for a single copy of the model

Find-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

NVIDIA DGX-1: 8 Pascal GPUs connected via high speed NV-Link interconnect
Accelerating data-parallel training has culminated in the FireCaffe distributed DNN training system. Learning these and other compelling applications led us to focus on the problem of accelerating DNN training, and our work to market more rapidly. Finally, there are a number of compelling use-cases for real-time DNN training, such as robot self-sizes in a tractable amount of time. Accelerating DNN training also enables product teams to bring DNN-based products to market more rapidly. So far in this section, we have discussed strategies for compressing or quantizing data to communicate in distributed DNN training. There has also been a series of studies on applying dimensionality reduction to DNNs once they have been trained.

Reducing communication among servers is to discard (and not communicate) gradients whose numerical values fall below a certain threshold. Backpropagation, albeit with a drop in the accuracy of the trained model. Recently, a number of techniques have been developed to accelerate convolution on GPUs. Unlike CPUs, NVIDIA GPUs couple with current and future GPU hardware and convolution libraries for further speedups. Amazon presented such a thresholding strategy in a recent paper on scaling up DNN training for speech recognition. Amazon presented such a thresholding strategy in a recent paper on scaling up DNN training for speech recognition. Amazon presented such a thresholding strategy in a recent paper on scaling up DNN training for speech recognition. Amazon presented such a thresholding strategy in a recent paper on scaling up DNN training for speech recognition. Amazon presented such a thresholding strategy in a recent paper on scaling up DNN training for speech recognition. Amazon presented such a thresholding strategy in a recent paper on scaling up DNN training for speech recognition. Amazon presented such a thresholding strategy in a recent paper on scaling up DNN training for speech recognition.

<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>GoogLeNet</td>
<td>64</td>
<td>32</td>
<td>0.01</td>
<td>21 days</td>
<td>1x</td>
<td>68.3%</td>
<td>88.7%</td>
</tr>
<tr>
<td>FireCaffe (ours)</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>
<td>88.7%</td>
</tr>
<tr>
<td>FireCaffe (ours)</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>
<td>88.7%</td>
</tr>
</tbody>
</table>

Dataset: ImageNet 1K

Result: reasonably scalability without asynchronous parameter update: for modern DNNs with fewer weights (due to no fully connected layers) such as GoogLeNet.
Parallelizing mini-batch on one machine

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 cores

Good: completely independent computations (until gradient reduction)
Bad: complete duplication of parameter gradient state (100’s MB per core)
Asynchronous update on one node

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

Cores update shared set of gradients.
Skip taking locks / synchronizing across cores: perform “approximate reduction”

Project Adam [Chilimbi OSDI14]
Summary: training large networks in parallel

- Most systems rely on asynchronous update to efficiently use clusters of commodity machines
  - Modification of SGD algorithm to meet constraints of modern parallel systems
  - Open question: effects on convergence are problem dependent and not particularly well understood
  - Tighter integration / faster interconnects may provide alternative to these methods (facilitate tightly orchestrated solutions much like supercomputing applications)

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