Professor classification network

Ground truth
(what the answer should be)
Easy: 0.0
Mean: 0.0
Boring: 0.0
Nerdy: 1.0

Easy: 0.26
Mean: 0.08
Boring: 0.14
Nerdy: 0.52

Network output
## Error (loss)

<table>
<thead>
<tr>
<th>Ground truth: (what the answer should be)</th>
<th>Network output: *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy: 0.0</td>
<td>Easy: 0.26</td>
</tr>
<tr>
<td>Mean: 0.0</td>
<td>Mean: 0.08</td>
</tr>
<tr>
<td>Boring: 0.0</td>
<td>Boring: 0.14</td>
</tr>
<tr>
<td>Nerdy: 1.0</td>
<td>Nerdy: 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.
DNN 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.)
Gradient descent
Basic gradient descent

while (loss too high):
    for each epoch: // a pass through items in training dataset
        grad = 0
        for each item x_i in training set:
            grad += evaluate_loss_gradient(f, params, loss_func, x_i)
        params += -grad * learning_rate;

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

while (loss too high):
    for each epoch:
        for all mini batches in training set:
            grad = 0;
            for each item x_i in minibatch:
                grad += evaluate_loss_gradient(f, params, loss_func, x_i)
            params += -grad * learning_rate;
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
- In practice: store per-weight gradient velocity (if using SGD with “momentum”) or step size cache in adaptive step size schemes like Adagrad

\[ \text{vel}_\text{new} = \mu \ast \text{vel}_\text{old} - \text{step}_\text{size} \ast \text{grad} \]
\[ \text{w}_\text{new} = \text{w}_\text{old} + \text{vel}_\text{new} \]
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 step)

Parallel across training images

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

trivially data-parallel over parameters
DNN training workload

- Large 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
  - Scaling to large 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
Synchronous data-parallel training (across images)

for each item $x_i$ in mini-batch of size $n$:
\[ \text{grad} += \text{evaluate_loss_gradient}(f, \text{loss_func}, \text{params}, x_i) \]
\[ \text{params} += -\text{grad} \times \text{learning_rate}; \]

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

partition dataset across nodes
for each item $x_i$ in mini-batch assigned to local node:
\[
\text{// just like single node training}
\]
\[ \text{grad} += \text{evaluate_loss_gradient}(f, \text{loss_func}, \text{params}, x_i) \]
\[ \text{barrier();} \]
\[ \text{sum reduce gradients, communicate results to all nodes} \]
\[ \text{barrier();} \]
\[ \text{update copy of parameter values} \]
Synchronous training

- All nodes cooperate to compute gradients for a mini-batch *

- Gradients are summed (across the entire machine)
  - All-to-all communication

- Update model parameters
  - Typically done without wide parallelism (e.g. each machine computes its own update)

- All nodes proceed to work on next mini-batch given new model parameters

* If curious about batch norm in a parallel training setting. In practice each of $k$ nodes works on a set of $n$ images, with batch norm statistics computed independently for each set of $n$ (so the overall mini-batch size is $kn$).
Overlapping communication and computation during back-propagation

(“Wait-free back propagation”)

Back-prop for current mini-batch …

Compute grad for layer i → Compute grad for layer i-1 → Compute grad for layer i-2 → Compute grad for layer i-3 → …

Send/recv grad for layer i → Send/recv grad for layer i-1 → Send/recv grad for layer i-2

time
Challenge of scaling data parallel training out to many nodes

- Slow communication between nodes
  - Synchronous SGD involves all-to-all communication after each mini-batch
  - GPUs in the same system have high communication cost (e.g., compared to communication between cores on the same GPU)
  - Commodity clusters do not feature high-performance interconnects between nodes typical of supercomputers (e.g., infiniband)

[Figure credit: Narayanan et al. SOSP 19]
Model parallelism

- For large models that do not fit on a single node
- Partition model weights across workers
- Naive solution: dependencies result in low utilization
- Notice: communication between nodes is point-to-point (not broadcast)

* In diagram backward pass shown to take 2x longer than forward pass

[Figure credit: Narayanan et al. SOSP 19]
Model parallelism (pipelined)

- Divide mini-batch into sub-mini-batches (shown here as 1-4)
- Pipeline forward/backward pass for one sub-mini-batch with communication of activations/gradients for another

[Figure credit: Narayanan et al. SOSP 19]
Model parallelism (aggressively pipelined)

- Immediately begin backward pass for “1” before forward pass for other mini-batches are complete.
- Problem: first stage of forward pass of “5” uses weights that reflect gradient update from “1”, but during backward pass through same stage, weights reflect gradient updates from “1,2,3,4”.

[Figure credit: Narayanan et al. SOSP 19]
Model parallelism (aggressively pipelined)

- Solution: “weight stashing”: each worker maintains multiple versions of model weights, always uses same weights for forward and backward pass.
Mixing data and pipeline parallelism

- Workers 1 and 2 maintain duplicate copies of stage part of model
- Mini-batch is partitioned across these two workers (data parallel)

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

Worker 1
Worker 2
Worker 3

[Figure credit: Narayanan et al. SOSP 19]
Challenges of scaling out (many nodes)

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

Alternative solution: exploit properties of SGD by 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 values

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

Pool of worker nodes

Parameter values (\(v_0\))

Worker Node 0

Worker Node 1

Worker Node 2

Worker Node 3

training data

\(x_0 - x_{1000}\)

\(x_{1000} - x_{2000}\)

\(x_{2000} - 3000\)

\(x_{3000} - 4000\)
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
Data parallelism: workers independently compute local “sub-gradients” on different pieces of data

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)
Worker sends sub-gradient 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 sub-gradient

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

Notice:
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)
Summary: asynchronous parameter update

- Idea: avoid global synchronization on all parameter updates between each SGD iteration
  - Algorithm 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

Worker Node 1

Worker Node 2

Worker Node 3

Parameter Server

Parameter values (v2)

local subgradients

local copy of parameters (v0)

Training data

local subgradients

local copy of parameters (v1)

Training data

local subgradients

local copy of parameters (v2)

Training data
Shard the parameter server

Partition parameters across servers
Worker sends chunk of sub-gradients 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)
Data-parallel and model-parallel execution

Working on subgradient computation for a single copy of the model

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

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

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

Worker Node 3
- 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)

Fine-grained communication of layer outputs, subgradients, etc.
Asynchronous vs. synchronous debate

- Asynchronous training: significant distributed system complexity incurred to combat bandwidth/latency constraints of modern cluster computing

- High interest in ways to improve the scalability of synchronous training
  - Better hardware
  - Better algorithms for existing hardware
Better hardware: using supercomputers for training

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

OakRidge Titan Supercomputer
(low-latency interconnect used in a number of recent training papers)

NVIDIA DGX-1: 8 GPUs connected via high speed NV-Link interconnect ($150,000 in 2018)
News from 2019…

NVIDIA buys high-performance chipmaker Mellanox for $6.9 billion

It beat Intel in a bid that will boost its server, self-driving and networking segments.

Kevin Krewell  Contributor
Tirias Research  Contributor Group
Enterprise & Cloud

The 2019 semiconductor merger and acquisition season has officially been kicked off with a blockbuster $6.9B deal for networking chipset and technology provider Mellanox. Graphic chip maker NVIDIA made the offer after a number of companies, rumored to include Intel, Microsoft, and Xilinx, had bid on buying the company. NVIDIA CEO Jenson Huang said in an analyst call that Mellanox management had invited him to bid on the company and he was happy to do so. By acquiring long-time data center partner Mellanox, Jensen is doubling down on the high-performance data center market.
Modified algorithmic techniques (again): improving scalability of synchronous training...

- Larger mini-batches increase computation-to-communication ratio: communicate gradients summed over $B$ training inputs

  for each item $x$ in mini-batch on this node:
  
    grad += evaluate_loss_gradient(f, loss_func, params, x)

  barrier();

  sum-reduce gradients across all nodes, communicate results to all nodes

  barrier();

  update copy of local parameter values

- But large mini-batches (if used naively) reduce accuracy of the model trained
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>GoogLeNet [41]</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><strong>47x</strong></td>
<td>68.3%</td>
<td>88.7%</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)
Increasing learning rate with mini-batch size: linear scaling rule

Recall: mini-batch SGD parameter update

\[ w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in B} \nabla l(x, w_t) \]

Consider processing of k mini-batches (k steps of gradient descent)

\[ w_{t+k} = w_t - \eta \frac{1}{n} \sum_{j<k} \sum_{x \in B_j} \nabla l(x, w_{t+j}) \]

Consider processing one mini-batch that is of size kn (one step of gradient descent)

\[ \hat{w}_{t+1} = w_t - \hat{\eta} \frac{1}{kn} \sum_{j<k} \sum_{x \in B_j} \nabla l(x, w_t) \]

Suggests that if \( \nabla l(x, w_t) \approx \nabla l(x, w_{t+j}) \) for \( j < k \) then minibatch SGD with size \( n \) and learning rate \( \eta \) can be approximated by large mini batch SGD with size \( kn \) if the learning rate is also scaled to \( k\eta \)
When does $\nabla l(x, w_t) \approx \nabla l(x, w_{t+j})$ not hold?

1. At beginning of training
   - Suggests starting training with smaller learning rate (learning rate “warmup”)

2. When minibatch size begins to get too large (there is a limit to scaling minibatch size)

---

**Mini-batch size = 256 (orange) vs. 8192 (blue)**

[Figure credit: Goyal et al. 2017]
Gradient compression

- Since overhead of communication (in particular in a data-parallel training configuration) is sending gradients, perhaps some gradients are more important than others
  - Idea: only send sparse gradient updates to reduce communication costs
Gradient compression

- Each node computes gradients for mini-batch, but only sends gradients with magnitude **above a threshold**

- Nodes, locally accumulate gradients below threshold over multiple SGD steps (then send when they exceed threshold)

For all iterations $t$ of SGD:

$$G^k_t = G^k_{t-1} + \eta \frac{1}{Nb} \sum_{k=1}^{N} \sum_{x \in B_k} \nabla f(x; w_t)$$

$N$ nodes, each computing gradients for a mini-batch of $b$ images (across the parallel machine the SGD batch size is $Nb$)

Compress and send ONLY the elements of $G^k_t$ greater than threshold.
(then locally zero out the gradients that were sent.)

After each iteration, SGD on all nodes only uses the sent weights...
Gradient compression is like using a larger mini-batch size for selected weights (lower gradients $\rightarrow$ larger batch size for these weights)

For weights with low gradients...
\[
\nabla f(x, w_t) \approx \nabla f(x, w_{t+\tau})
\]

So $T$ steps of regular SGD (mini-batch $b$ processed by all $N$ nodes) for weight (i):
\[
w_t^{(i)} = w_t^{(i)} - \eta \frac{1}{N b} \sum_{k=1}^{N} \left \{ \sum_{\tau=0}^{T-1} \sum_{x \in B_k, \tau} \nabla^{(i)} f(x, w_{t+\tau}) \right \}
\]

Is well approximated despite not updating weight (i) for $T$ steps: (effectively a $T$ times larger mini-batch size for weight (i))
\[
w_t^{(i)} = w_t^{(i)} - \eta T \frac{1}{N b T} \sum_{k=1}^{N} \left \{ \sum_{\tau=0}^{T-1} \sum_{x \in B_k, \tau} \nabla^{(i)} f(x, w_t) \right \}
\]
Many cool ideas popping up

- Gradient compression
  - Reduce the frequency of gradient update (sparse updates)
  - Apply compression techniques to the gradient data that is sent

- Account for communication latency in SGD momentum calculations
  - Asynchronous execution or sparse gradient updates means SGD continues forward (with potentially stale gradients)
  - SGD with momentum has a similar effect (keep descending in the same direction, don’t directly follow gradient)
  - Idea: reduce momentum proportionally to latency of gradient update
Summary: training large networks in parallel

- Modern DNN designs, large mini-batch sizes, careful learning rate schedules enable scalability without asynchronous execution on commodity clusters.

- Data-parallel training with asynchronous update to efficiently use clusters of commodity machines with low speed interconnect.
  - Modification of SGD algorithm to meet constraints of modern parallel systems.
  - Effects on convergence are problem dependent and not particularly well understood.

- 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!)