Lecture 22:
Parallel Deep Neural Networks

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2018
Training/evaluating deep neural networks
Technique leading to many high-profile AI advances in recent years

Speech recognition/natural language processing

Image interpretation and understanding

a baseball player swinging a bat at a ball a boy is playing with a baseball bat
What is a deep neural network?

A basic unit:
Unit with \( n \) inputs described by \( n+1 \) parameters (weights + bias)

\[
\begin{align*}
\text{Input:} & \quad x_0, x_1, x_2, x_3 \\
\text{Unit ("neuron")} & \quad w_0, w_1, w_2, w_3, b \\
\text{output} & \quad f \left( \sum_i x_i w_i + b \right)
\end{align*}
\]

Example \( f \): rectified linear unit (ReLU)
\[
f(x) = \max(0, x)
\]

Basic computational interpretation:
It’s just a circuit!

Biological inspiration:
unit output corresponds loosely to activation of neuron

Machine learning interpretation:
binary classifier: interpret output as the probability of one class
\[
f(x) = \frac{1}{1 + e^{-x}}
\]
Two Distinct Issues with Deep Networks

- **Evaluation**
  - often takes milliseconds

- **Training**
  - often takes hours, days, weeks
What is a deep neural network? topology

This network has: 4 inputs, 1 output, 7 hidden units
“Deep” = at least one hidden layer
Hidden layer 1: 3 units x (4 weights + 1 bias) = 15 parameters
Hidden layer 2: 4 units x (3 weights + 1 bias) = 16 parameters

Note fully-connected topology in this example
What is a deep neural network? topology

- Fully connected layer
- Sparsely (locally) connected
Recall image convolution (3x3 conv)

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];
float weights[] = {1.0/9, 1.0/9, 1.0/9,
                  1.0/9, 1.0/9, 1.0/9,
                  1.0/9, 1.0/9, 1.0/9};

for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
        output[j*WIDTH + i] = tmp;
    }
}
```

Convolutional layer: locally connected AND all units in layer share the same parameters (same weights + same bias):
(note: network diagram only shows links due to one iteration of ii loop)
Strided 3x3 convolution

int WIDTH = 1024;
int HEIGHT = 1024;
int STRIDE = 2;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[(WIDTH/STRIDE) * (HEIGHT/STRIDE)];

float weights[] = {1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9};

for (int j=0; j<HEIGHT; j+=STRIDE) {
    for (int i=0; i<WIDTH; i+=STRIDE) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            for (int ii=0; ii<3; ii++) {
                tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
            }
        output[(j/STRIDE)*WIDTH + (i/STRIDE)] = tmp;
    }
}

Inputs

Convolutional layer with stride 2
What does convolution using these filter weights do? 

\[
\begin{bmatrix}
0.075 & 0.124 & 0.075 \\
0.124 & 0.204 & 0.124 \\
0.075 & 0.124 & 0.075
\end{bmatrix}
\]

“Gaussian Blur”
What does convolution with these filters do?

-1  0  1
-2  0  2
-1  0  1

Extracts horizontal gradients

-1  -2  -1
0   0   0
1   2   1

Extracts vertical gradients
Gradient detection filters

Horizontal gradients

Vertical gradients

Note: you can think of a filter as a “detector” of a pattern, and the magnitude of a pixel in the output image as the “response” of the filter to the region surrounding each pixel in the input image.
Applying many filters to an image at once

Input: image (single channel): $W \times H$

3x3 spatial convolutions on image $3x3 \times \text{num\_filters}$ weights

Output: filter responses $W \times H \times \text{num\_filters}$

Each filter described by unique set of weights (responds to different image phenomena)

Filter responses
Applying many filters to an image at once

Input RGB image (W x H x 3)

96 11x11x3 filters
(operate on RGB)

96 responses (normalized)
Adding additional layers

Input: image (single channel) $W \times H$

3x3 spatial convolutions
$3x3 \times \text{num}_\text{filters}$ weights

Output: filter responses
$W \times H \times \text{num}_\text{filters}$

Each filter described by unique set of weights (responds to different image phenomena)

Filter responses

ReLU

post ReLU
$W \times H \times \text{num}_\text{filters}$

post pool
$W/2 \times H/2 \times \text{num}_\text{filters}$

Note data reduction as a result of pooling

Conv

Pool
(max response in 2x2 region)
Modern object detection networks

Sequences of cont + reLU + (optional) pool layers

AlexNet [Krizhevsky12]: 5 convolutional layers + 3 fully connected

VGG-16 [Simonyan15]: 13 convolutional layers

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Input/Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>224 x 224 RGB</td>
</tr>
<tr>
<td>Conv/ReLU</td>
<td>3x3 x 128 x 256</td>
</tr>
<tr>
<td>Maxpool</td>
<td>256</td>
</tr>
<tr>
<td>Conv/ReLU</td>
<td>3x3 x 512 x 512</td>
</tr>
<tr>
<td>Maxpool</td>
<td>4096</td>
</tr>
<tr>
<td>Soft-max</td>
<td>1000</td>
</tr>
</tbody>
</table>

[VGG illustration credit: Yang et al.]
Why deep?

Left: what pixels trigger the response
Right: images that generate strongest response for filters at each layer

Layer 1

Layer 2

Layer 3

[image credit: Zeiler 14]
Why deep?

[Image credit: Zeiler 14]
Efficiently implementing convolution layers
Direct implementation of conv layer

```c
float input[INPUT_HEIGHT][INPUT_WIDTH][INPUT_DEPTH];
float output[INPUT_HEIGHT][INPUT_WIDTH][LAYER_NUM_FILTERS];
float layer_weights[LAYER_CONVY, LAYER_CONVX, INPUT_DEPTH];

// assumes convolution stride is 1
for (int img=0; img<IMAGE_BATCH_SIZE; img++)
    for (int j=0; j<INPUT_HEIGHT; j++)
        for (int i=0; i<INPUT_WIDTH; i++)
            for (int f=0; f<LAYER_NUM_FILTERS; f++) {
                output[j][i][f] = 0.f;
                for (int kk=0; kk<INPUT_DEPTH; kk++) // sum over filter responses of input channels
                    for (int jj=0; jj<LAYER_CONVY; jj++) // spatial convolution
                        for (int ii=0; ii<LAYER_CONVX; ii++) // spatial convolution
                            output[j][i][f] += layer_weights[f][jj][ii][kk] * input[j+jj][i+ii][kk];
            }
```

Seven loops with significant input data reuse: reuse of filter weights (during convolution), and reuse of input values (across different filters)

But must roll your own highly optimized implementation of a complicated loop nest.
Dense matrix multiplication

```c
float A[M][K];
float B[K][N];
float C[M][N];

// compute C += A * B
#pragma omp parallel for
for (int j=0; j<M; j++)
  for (int i=0; i<N; i++)
    for (int k=0; k<K; k++)
      C[j][i] += A[j][k] * B[k][i];
```

What is the problem with this implementation?

Low arithmetic intensity (does not exploit temporal locality in access to A and B)
float A[M][K];
float B[K][N];
float C[M][N];

// compute C += A * B
#pragma omp parallel for
for (int jblock=0; jblock<M; jblock+=BLOCKSIZE_J)
    for (int iblock=0; iblock<N; iblock+=BLOCKSIZE_I)
        for (int kblock=0; kblock<K; kblock+=BLOCKSIZE_K)
            for (int j=0; j<BLOCKSIZE_J; j++)
                for (int i=0; i<BLOCKSIZE_I; i++)
                    for (int k=0; k<BLOCKSIZE_K; k++)
                        C[jblock+j][iblock+i] += A[jblock+j][kblock+k] * B[kblock+k][iblock+i];

Idea: compute partial result for block of C while required blocks of A and B remain in cache (Assumes BLOCKSIZE chosen to allow block of A, B, and C to remain resident)

Self check: do you want as big a BLOCKSIZE as possible? Why?
Convolution as matrix-vector product

Construct matrix from elements of input image

<table>
<thead>
<tr>
<th>$X_{00}$</th>
<th>$X_{01}$</th>
<th>$X_{02}$</th>
<th>$X_{03}$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_{10}$</td>
<td>$X_{11}$</td>
<td>$X_{12}$</td>
<td>$X_{13}$</td>
<td>...</td>
</tr>
<tr>
<td>$X_{20}$</td>
<td>$X_{21}$</td>
<td>$X_{22}$</td>
<td>$X_{23}$</td>
<td>...</td>
</tr>
<tr>
<td>$X_{30}$</td>
<td>$X_{31}$</td>
<td>$X_{32}$</td>
<td>$X_{33}$</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

$3 \times 3 = 9$

$O(N)$ storage overhead for filter with $N$ elements
Must construct input data matrix

Note: 0-pad matrix
3x3 convolution as matrix-vector product

Construct matrix from elements of input image

Note: 0-pad matrix

O(N) storage overhead for filter with N elements
Must construct input data matrix
Multiple convolutions as matrix-matrix mult

\[
\begin{bmatrix}
X_{00} & X_{01} & X_{02} & X_{03} & \ldots \\
X_{10} & X_{11} & X_{12} & X_{13} & \ldots \\
X_{20} & X_{21} & X_{22} & X_{23} & \ldots \\
X_{30} & X_{31} & X_{32} & X_{33} & \ldots \\
\vdots & \vdots & \vdots & \vdots & \ddots \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
W_0 & W_1 & W_2 & \ldots & W_N \\
W_{10} & W_{11} & W_{12} & \ldots & W_{1N} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
W_{80} & W_{81} & W_{82} & \ldots & W_{8N} \\
\end{bmatrix}
\]

num filters
Multiple convolutions on multiple input channels

For each filter, sum responses over input channels

Equivalent to $(3 \times 3 \times \text{num\_channels})$ convolution on $(W \times H \times \text{num\_channels})$ input data
# VGG memory footprint

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

<table>
<thead>
<tr>
<th>Layer Description</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>224x224x3</td>
<td>150K</td>
</tr>
<tr>
<td>conv: (3x3x3) x 64</td>
<td>6.5 KB</td>
<td>224x224x64</td>
<td>12.3 MB</td>
</tr>
<tr>
<td>conv: (3x3x64) x 64</td>
<td>144 KB</td>
<td>224x224x64</td>
<td>12.3 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>—</td>
<td>112x112x64</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>conv: (3x3x64) x 128</td>
<td>228 KB</td>
<td>112x112x128</td>
<td>6.2 MB</td>
</tr>
<tr>
<td>conv: (3x3x128) x 128</td>
<td>576 KB</td>
<td>112x112x128</td>
<td>6.2 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>—</td>
<td>56x56x128</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>conv: (3x3x128) x 256</td>
<td>1.1 MB</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>conv: (3x3x256) x 256</td>
<td>2.3 MB</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>conv: (3x3x256) x 256</td>
<td>2.3 MB</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>—</td>
<td>28x28x256</td>
<td>766 KB</td>
</tr>
<tr>
<td>conv: (3x3x256) x 512</td>
<td>4.5 MB</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>—</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>9 MB</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>maxpool</td>
<td>—</td>
<td>7x7x512</td>
<td>98 KB</td>
</tr>
<tr>
<td>fully-connected 4096</td>
<td>392 MB</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>fully-connected 4096</td>
<td>64 MB</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>fully-connected 1000</td>
<td>15.6 MB</td>
<td>1000</td>
<td>4 KB</td>
</tr>
<tr>
<td>soft-max</td>
<td></td>
<td>1000</td>
<td>4 KB</td>
</tr>
</tbody>
</table>
Reducing network footprint

- Large storage cost for model parameters
  - AlexNet model: ~200 MB
  - VGG-16 model: ~500 MB
  - This doesn’t even account for intermediates during evaluation

- Footprint: cumbersome to store, download, etc.
  - 500 MB app downloads make users unhappy!

- Consider energy cost of 1B parameter network
  - Running on input stream at 20 Hz
  - 640 pJ per 32-bit DRAM access
  - \((20 \times 1B \times 640pJ) = 12.8W\) for DRAM access
    (more than power budget of any modern smartphone)
Compressing a network

Step 1: prune low-weight links (iteratively retrain network, then prune)
- Over 90% of weights can be removed without significant loss of accuracy
- Store weight matrices in compressed sparse row (CSR) format

<table>
<thead>
<tr>
<th>Indices</th>
<th>1</th>
<th>4</th>
<th>9</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1.8</td>
<td>0.5</td>
<td>2.1</td>
<td>...</td>
</tr>
</tbody>
</table>

Step 2: weight sharing: make surviving connects share a small set of weights
- Cluster weights via k-means clustering (irregular (“learned”) quantization)
- Compress weights by only storing cluster index (\(\log(k)\) bits)
- Retrain network to improve quality of cluster centroids

Step 3: Huffman encode quantized weights and CSR indices

Equation: 
\[
\text{Compression Rate} = \frac{nb}{n\log_2(k)} + kb
\]

Example:
- Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).
## VGG-16 compression

Substantial savings due to combination of pruning, quantization, Huffman encoding

<table>
<thead>
<tr>
<th>Layer</th>
<th>#Weights</th>
<th>Weights% (P)</th>
<th>Weights bits (P+Q)</th>
<th>Weight bits (P+Q+H)</th>
<th>Index bits (P+Q)</th>
<th>Index bits (P+Q+H)</th>
<th>Compress rate (P+Q)</th>
<th>Compress rate (P+Q+H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1_1</td>
<td>2K</td>
<td>58%</td>
<td>8</td>
<td>6.8</td>
<td>5</td>
<td>1.7</td>
<td>40.0%</td>
<td>29.97%</td>
</tr>
<tr>
<td>conv1_2</td>
<td>37K</td>
<td>22%</td>
<td>8</td>
<td>6.5</td>
<td>5</td>
<td>2.6</td>
<td>9.8%</td>
<td>6.99%</td>
</tr>
<tr>
<td>conv2_1</td>
<td>74K</td>
<td>34%</td>
<td>8</td>
<td>5.6</td>
<td>5</td>
<td>2.4</td>
<td>14.3%</td>
<td>8.91%</td>
</tr>
<tr>
<td>conv2_2</td>
<td>148K</td>
<td>36%</td>
<td>8</td>
<td>5.9</td>
<td>5</td>
<td>2.3</td>
<td>14.7%</td>
<td>9.31%</td>
</tr>
<tr>
<td>conv3_1</td>
<td>295K</td>
<td>53%</td>
<td>8</td>
<td>4.8</td>
<td>5</td>
<td>1.8</td>
<td>21.7%</td>
<td>11.15%</td>
</tr>
<tr>
<td>conv3_2</td>
<td>590K</td>
<td>24%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.9</td>
<td>9.7%</td>
<td>5.67%</td>
</tr>
<tr>
<td>conv3_3</td>
<td>590K</td>
<td>42%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.2</td>
<td>17.0%</td>
<td>8.96%</td>
</tr>
<tr>
<td>conv4_1</td>
<td>1M</td>
<td>32%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.6</td>
<td>13.1%</td>
<td>7.29%</td>
</tr>
<tr>
<td>conv4_2</td>
<td>2M</td>
<td>27%</td>
<td>8</td>
<td>4.2</td>
<td>5</td>
<td>2.9</td>
<td>10.9%</td>
<td>5.93%</td>
</tr>
<tr>
<td>conv5_1</td>
<td>2M</td>
<td>34%</td>
<td>8</td>
<td>4.4</td>
<td>5</td>
<td>2.5</td>
<td>14.0%</td>
<td>7.47%</td>
</tr>
<tr>
<td>conv5_2</td>
<td>2M</td>
<td>35%</td>
<td>8</td>
<td>4.7</td>
<td>5</td>
<td>2.5</td>
<td>14.3%</td>
<td>8.00%</td>
</tr>
<tr>
<td>conv5_3</td>
<td>2M</td>
<td>29%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.7</td>
<td>11.7%</td>
<td>6.52%</td>
</tr>
<tr>
<td>fc6</td>
<td>103M</td>
<td>4%</td>
<td>5</td>
<td>3.6</td>
<td>5</td>
<td>3.5</td>
<td>1.6%</td>
<td>1.10%</td>
</tr>
<tr>
<td>fc7</td>
<td>17M</td>
<td>4%</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4.3</td>
<td>1.5%</td>
<td>1.25%</td>
</tr>
<tr>
<td>fc8</td>
<td>4M</td>
<td>23%</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3.4</td>
<td>7.1%</td>
<td>5.24%</td>
</tr>
<tr>
<td>Total</td>
<td>138M</td>
<td>7.5%</td>
<td>6.4</td>
<td>4.1</td>
<td>5</td>
<td>3.1</td>
<td>3.2% (31×)</td>
<td>2.05% (49×)</td>
</tr>
</tbody>
</table>

P = connection pruning (prune low weight connections)
Q = quantize surviving weights (using shared weights)
H = Huffman encode

### ImageNet Image Classification Performance

<table>
<thead>
<tr>
<th></th>
<th>Top-1 Error</th>
<th>Top-5 Error</th>
<th>Model size</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-16 Ref</td>
<td>31.50%</td>
<td>11.32%</td>
<td>552 MB</td>
</tr>
<tr>
<td>VGG-16 Compressed</td>
<td>31.17%</td>
<td>10.91%</td>
<td>11.3 MB</td>
</tr>
</tbody>
</table>

CMU 15-418/618, Spring 2018
Deep neural networks on GPUs

Today, best performing DNN implementations target GPUs

- High arithmetic intensity computations (computational characteristics similar to dense matrix-matrix multiplication)
- Benefit from flop-rich architectures
- Highly-optimized library of kernels exist for GPUs (cuDNN)
  - Most CPU-based implementations use basic matrix-multiplication-based formulation (good implementations could run faster!)
Emerging architectures for deep learning?

- NVIDIA Pascal (upcoming GPU)
  - Adds double-throughput 16-bit floating point ops
  - Feature that is already common on mobile GPUs

- Google TensorFlow Processing Unit
  - Hardware accelerator for array computations
  - Used in Google data centers

- Intel Xeon Phi (Knights Landing)
  - FLOP-rich 72-core x86 processor

- FPGAs, ASICs?
  - Microsoft including within data centers
  - Not new: FPGA solutions have been explored for years
Programming frameworks for deep learning

- Heavyweight processing (low-level kernels) carried out by target-optimized libraries (NVIDIA cuDNN, Intel MKL)

- Popular frameworks use these kernel libraries
  - Caffe, Torch, Theano, TensorFlow, MxNet

- DNN application development = constructing novel network topologies
  - Programming by constructing networks
  - Significant interest in new ways to express network construction
Summary: efficiently evaluating convnets

- **Computational structure**
  - Convlayers: high arithmetic intensity, significant portion of cost of evaluating a network
  - Similar data access patterns to dense-matrix multiplication (exploiting temporal reuse is key)
  - But straight reduction to matrix-matrix multiplication is often sub-optimal
  - Work-efficient techniques for convolutional layers (FFT-based, Wingrad convolutions)

- **Large numbers of parameters: significant interest in reducing size of networks for both training and evaluation**
  - Pruning: remove least important network links
  - Quantization: low-precision parameter representations often suffice

- **Many ongoing studies of specialized hardware architectures for efficient evaluation**
  - Future CPUs/GPUs, ASICs, FPGs, …
  - Specialization will be important to achieving “always on” applications
Two Distinct Issues with Deep Networks

- Evaluation
  - often takes milliseconds

- Training
  - often takes hours, days, weeks
Training a network is the process of learning the value of network parameters so that output of the network provides the desired result for a task.

- [Krizhevsky12] task = object classification
  - input 224 x 224 x 3 RGB image
  - output probability of 1000 ImageNet object classes: “dog”, “cat”, etc.
  - ~ 60M weights
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 from last time: 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.

<table>
<thead>
<tr>
<th>Easy</th>
<th>Mean</th>
<th>Boring</th>
<th>Nerdy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.04</td>
<td>0.18</td>
<td>0.27</td>
<td>0.51</td>
</tr>
</tbody>
</table>
Where did the parameters come from?
Training data (ground truth answers)
Professor classification network

New image of Bryant
(not in training set)

Ground truth
(what the answer should be)

Network output

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

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

**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

**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 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 a 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($f$, loss_func, params, $x_i$)
    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 to compute $df/dp$ for a complex 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 = $df/d\text{node}$

Recall: $\frac{df}{dx} = \frac{df}{dg} \frac{dg}{dx}$

1. $g(x, y) = x + y$
   - $\frac{dg}{dx} = 1$, $\frac{dg}{dy} = 1$

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

3. $g(x, y) = xy$
   - $\frac{dg}{dx} = y$, $\frac{dg}{dy} = x$
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 (\( x_i \)'s) and output of this unit must be retained in order to compute weight gradients for this unit during backprop.
Backpropagation: matrix form

\[ y = Xw \]

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

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 $x_i$
  
  Note: must retain all layer outputs + output gradients (needed to compute weight gradients during backpropagation)

- 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: $w_i_{\text{new}} = w_i_{\text{old}} - \text{step\_size} \times \text{grad}_i$
## VGG Memory Footprint

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

<table>
<thead>
<tr>
<th>Layer Description</th>
<th>Input Size</th>
<th>Weights Memory</th>
<th>Output Size (per Image)</th>
<th>Memory (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>392 MB</td>
<td>—</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>fully-connected 4096</td>
<td>64 MB</td>
<td>—</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>fully-connected 1000</td>
<td>15.6 MB</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>
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
Deep network 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 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 network across nodes to keep network + 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:
\[
\text{grad} += \text{evaluate_loss_gradient}(f, \text{loss_func}, \text{params}, x_i)
\]
\[
\text{params} += -\text{grad} \times \text{step_size};
\]

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:
\[
// \text{just like single node training}
\]
\[
\text{grad} += \text{evaluate_loss_gradient}(f, \text{loss_func}, \text{params}, x_i)
\]
\[
\text{barrier();}
\]
\[
\text{sum reduce gradients, communicate results to all nodes}
\]
\[
\text{barrier();}
\]
\[
\text{update copy of parameter values}
\]
Challenges of computing at cluster scale

- Slow communication between nodes
  - Clusters do not feature high-performance interconnects 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!
Exploiting SGD Characteristics

- **Convergent computation**
  - Update ordering does not matter
  - OK to have small errors in weight updates

- **How used**
  - Within machine: Don’t synchronize weight updates across threads
  - Between machines:
    - OK to do some computations using stale data
    - Ordering of updates not critical
    - Incomplete or redundant coverage of data set acceptable
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”
Parameter server design

Parameter Server [Li OSDI14]
Google's DistBelief [Dean NIPS12]
Microsoft’s Project Adam [Chilimbi OSDI14]

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 (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

- Separate set of machines to maintain DNN parameters
- Highly fault tolerant (so that worker nodes need not reliable)
- Accept updates from workers asynchronously
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

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)

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
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)

- Open question: how big of networks are needed?
  - >90% of connections could be removed without significant impact on quality of network
  - 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!)