Lecture 11:
Optimizing Inference via Approximation

Visual Computing Systems
CMU 15-769, Fall 2016
Take-home midterm


- Example forms of questions:
  - Short answer questions
    - Design exercise: choose between processor A and processor B for a particular task and state reason
  - Read a paper (similar to ones we’ve read in class) and offer an analysis of it (or answer a few questions about it)
Assignment 2 coming out next week

- We have finished our “basics of deep learning” prep, now time to get hands on

- Assignment 2: implement a mini-deep learning framework in any programming language you wish
  - Released Tuesday 10/18, due Friday 11/4
  - We will give you basic stater code in C and in Halide
  - You will implement basic layers needed to train a mini-network
  - We will have a “fun” class race on fastest forward evaluation and backward evaluation
Course projects

- Project proposal is due 10/27
  - Short proposal document (2-3 pages max)
  - I will likely accept any project that “embodies the themes of the course” and passes the “will the project answer a question that a student in the class not know the answer to” test
  - For Ph.D. students, this means the idea is ambitious enough to be the seed for a research project

- Project presentations are due during our final exam slot: 12/16
- May work in teams of 2 (in rare circumstances I’ll okay a team of 3)
- I’ll post a list of rough project ideas on the web over the weekend
Three ideas for accelerating inference

- **Sparsification**
  - Remove unnecessary parts of DNN

- **Quantization**
  - Represent values that remain more coarsely

- **Exploiting temporal redundancy (for video)**
  - Skip evaluating parts of a DNN that do not change significantly with time
Why efficient inference?
Last time (and last class’ reading)

- Expert knowledge used to manually design more compact and more efficient (to train and to evaluate) network topologies
Finding a good network topology

- Common practice in modern network design: brute-force parameter sweep

- Train multiple versions of a topology in parallel
  - Vary number of layers, width fully-connected layers
  - Vary number of filters in a cone layer

- Pick the smallest (most efficient) topology that yields adequate performance (accuracy) on task at hand
Reminder: energy cost of data access

Significant fraction of energy expended moving data to processor ALUs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy [pJ]</th>
<th>Relative Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bit int ADD</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>32 bit float ADD</td>
<td>0.9</td>
<td>9</td>
</tr>
<tr>
<td>32 bit Register File</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>32 bit int MULT</td>
<td>3.1</td>
<td>31</td>
</tr>
<tr>
<td>32 bit float MULT</td>
<td>3.7</td>
<td>37</td>
</tr>
<tr>
<td>32 bit SRAM Cache</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>32 bit DRAM Memory</td>
<td>640</td>
<td>6400</td>
</tr>
</tbody>
</table>

Estimates for 45nm process
[Source: Mark Horowitz]

Recall: AlexNet has over 68m weights (>260MB if 4 bytes/weight)
Executing at 30fps, that’s 1.3 Watts just to read the weights
"Pruning" (sparsifying) a network

If weight is near zero, then corresponding input has little impact on output of neuron.

\[ f \left( \sum_{i} x_i w_i + b \right) \]

\[ f(x) = \max(0, x) \]
"Pruning" (sparsifying) a network

Idea: prune connections with near zero weight

Remove entire units if all connections are pruned.

Why not look for large $dL/dw_i$?

More principled to look at second derivatives $d^2L/dw_i dw_j$, but costly...
Iterative pruning

- Step 1: train network (as normal)
- Step 2: remove connections with weight less than threshold (threshold may be chosen based on std-dev of weights)
  - The network is now sparse
- Step 3: retrain network using surviving connections
  - Retraining (fine-tuning) is initialized with previously learned weights
- Repeat steps 2 and 3 as necessary
  - Since a connection is gone forever once pruned, best to prune conservatively each step, use multiple iterations to remove connections
  - L2 regularization during training

Iterative process learns both topology (what connections are needed) and the weights on those connections.

Incur increased training time to accelerate inference.
(recall design of networks like Inception sought to accelerate training as well)
Results of pruning

With promising results on AlexNet, we also looked at a larger, more recent network, VGG-16. The trade-off curve between accuracy and number of parameters is shown in Figure 5. The more convolutional layers are pruned, the more the network needs to be retrained to compensate for the loss of accuracy. The VGG-16 results are, like those for AlexNet, very promising. The network as a whole has been reduced to 7.5% of its original size. It's interesting to see that we have the “free lunch” of reducing computation by 21%.

### 4.3 VGG-16 on ImageNet

<table>
<thead>
<tr>
<th>Layer</th>
<th>Weights</th>
<th>FLOP</th>
<th>Act%</th>
<th>Weights%</th>
<th>FLOP%</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>35K</td>
<td>211M</td>
<td>88%</td>
<td>84%</td>
<td>84%</td>
</tr>
<tr>
<td>conv2</td>
<td>307K</td>
<td>448M</td>
<td>52%</td>
<td>38%</td>
<td>33%</td>
</tr>
<tr>
<td>conv3</td>
<td>885K</td>
<td>299M</td>
<td>37%</td>
<td>35%</td>
<td>18%</td>
</tr>
<tr>
<td>conv4</td>
<td>663K</td>
<td>224M</td>
<td>40%</td>
<td>37%</td>
<td>14%</td>
</tr>
<tr>
<td>conv5</td>
<td>442K</td>
<td>150M</td>
<td>34%</td>
<td>37%</td>
<td>14%</td>
</tr>
<tr>
<td>fc1</td>
<td>38M</td>
<td>75M</td>
<td>36%</td>
<td>9%</td>
<td>3%</td>
</tr>
<tr>
<td>fc2</td>
<td>17M</td>
<td>34M</td>
<td>40%</td>
<td>9%</td>
<td>3%</td>
</tr>
<tr>
<td>fc3</td>
<td>4M</td>
<td>8M</td>
<td>100%</td>
<td>25%</td>
<td>10%</td>
</tr>
<tr>
<td>Total</td>
<td>61M</td>
<td>1.5B</td>
<td>54%</td>
<td>11%</td>
<td>30%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layer</th>
<th>Weights</th>
<th>FLOP</th>
<th>Act%</th>
<th>Weights%</th>
<th>FLOP%</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1_1</td>
<td>2K</td>
<td>0.2B</td>
<td>53%</td>
<td>58%</td>
<td>58%</td>
</tr>
<tr>
<td>conv1_2</td>
<td>37K</td>
<td>3.7B</td>
<td>89%</td>
<td>22%</td>
<td>12%</td>
</tr>
<tr>
<td>conv2_1</td>
<td>74K</td>
<td>1.8B</td>
<td>80%</td>
<td>34%</td>
<td>30%</td>
</tr>
<tr>
<td>conv2_2</td>
<td>148K</td>
<td>3.7B</td>
<td>81%</td>
<td>36%</td>
<td>29%</td>
</tr>
<tr>
<td>conv3_1</td>
<td>295K</td>
<td>1.8B</td>
<td>68%</td>
<td>53%</td>
<td>43%</td>
</tr>
<tr>
<td>conv3_2</td>
<td>590K</td>
<td>3.7B</td>
<td>70%</td>
<td>24%</td>
<td>16%</td>
</tr>
<tr>
<td>conv3_3</td>
<td>590K</td>
<td>3.7B</td>
<td>64%</td>
<td>42%</td>
<td>29%</td>
</tr>
<tr>
<td>conv4_1</td>
<td>1M</td>
<td>1.8B</td>
<td>51%</td>
<td>32%</td>
<td>21%</td>
</tr>
<tr>
<td>conv4_2</td>
<td>2M</td>
<td>3.7B</td>
<td>45%</td>
<td>27%</td>
<td>14%</td>
</tr>
<tr>
<td>conv4_3</td>
<td>2M</td>
<td>3.7B</td>
<td>34%</td>
<td>34%</td>
<td>15%</td>
</tr>
<tr>
<td>conv5_1</td>
<td>2M</td>
<td>925M</td>
<td>32%</td>
<td>35%</td>
<td>12%</td>
</tr>
<tr>
<td>conv5_2</td>
<td>2M</td>
<td>925M</td>
<td>29%</td>
<td>29%</td>
<td>9%</td>
</tr>
<tr>
<td>conv5_3</td>
<td>2M</td>
<td>925M</td>
<td>19%</td>
<td>36%</td>
<td>11%</td>
</tr>
<tr>
<td>fc6</td>
<td>103M</td>
<td>206M</td>
<td>38%</td>
<td>4%</td>
<td>1%</td>
</tr>
<tr>
<td>fc7</td>
<td>17M</td>
<td>34M</td>
<td>42%</td>
<td>4%</td>
<td>2%</td>
</tr>
<tr>
<td>fc8</td>
<td>4M</td>
<td>8M</td>
<td>100%</td>
<td>23%</td>
<td>9%</td>
</tr>
<tr>
<td>total</td>
<td>138M</td>
<td>30.9B</td>
<td>64%</td>
<td>7.5%</td>
<td>21%</td>
</tr>
</tbody>
</table>

Recall similar numbers from SqueezeNet paper: 33% weights survive (recall SqueezeNet is fully convolutional... see convlayers above)
Efficiently storing the surviving connections

Store surviving weights compactly 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>

Reduce storage over head of indices by delta encoding them to fit in 8 bits

<table>
<thead>
<tr>
<th>Indices</th>
<th>1</th>
<th>3</th>
<th>6</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>
Efficiently storing the surviving connections

Weight sharing: make surviving connections share a small set of weights
- Cluster weights via k-means clustering
- Compress weights by only storing index of assigned cluster (lg(k) bits)
- Retraining only modifies the cluster centroids (so that back-prop is constrained to not induce more unique weight values)

![Diagram showing weight sharing and centroids fine-tuning](image)

Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).
### Pruning, quantization, huffman-encoding of VGG-16

#### Table 4: Compression statistics for AlexNet. P: pruning, Q: quantization, H: Huffman coding.

<table>
<thead>
<tr>
<th>Layer</th>
<th>#Weights</th>
<th>Weights% (P)</th>
<th>Weight 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</td>
<td>35k</td>
<td>84%</td>
<td>8</td>
<td>6.3</td>
<td>4</td>
<td>1.2</td>
<td>32.6%</td>
<td>20.53%</td>
</tr>
<tr>
<td>conv2</td>
<td>307k</td>
<td>38%</td>
<td>8</td>
<td>5.5</td>
<td>4</td>
<td>2.3</td>
<td>14.5%</td>
<td>9.43%</td>
</tr>
<tr>
<td>conv3</td>
<td>885k</td>
<td>35%</td>
<td>8</td>
<td>5.1</td>
<td>4</td>
<td>2.6</td>
<td>13.1%</td>
<td>8.44%</td>
</tr>
<tr>
<td>conv4</td>
<td>663k</td>
<td>37%</td>
<td>8</td>
<td>5.2</td>
<td>4</td>
<td>2.5</td>
<td>14.1%</td>
<td>9.11%</td>
</tr>
<tr>
<td>conv5</td>
<td>442k</td>
<td>37%</td>
<td>8</td>
<td>5.6</td>
<td>4</td>
<td>2.5</td>
<td>14.0%</td>
<td>9.43%</td>
</tr>
<tr>
<td>fc6</td>
<td>38M</td>
<td>9%</td>
<td>5</td>
<td>3.9</td>
<td>4</td>
<td>3.2</td>
<td>3.0%</td>
<td>2.39%</td>
</tr>
<tr>
<td>fc7</td>
<td>17M</td>
<td>9%</td>
<td>5</td>
<td>3.6</td>
<td>4</td>
<td>3.7</td>
<td>3.0%</td>
<td>2.46%</td>
</tr>
<tr>
<td>fc8</td>
<td>4M</td>
<td>25%</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3.2</td>
<td>7.3%</td>
<td>5.85%</td>
</tr>
<tr>
<td>Total</td>
<td>61M</td>
<td>11%</td>
<td>5.4</td>
<td>4</td>
<td>3.2</td>
<td>3.7</td>
<td>3.2% (31×)</td>
<td>2.05% (49×)</td>
</tr>
</tbody>
</table>

---


<table>
<thead>
<tr>
<th>Layer</th>
<th>#Weights</th>
<th>Weights% (P)</th>
<th>Weight 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>12K</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>conv4.3</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>conv5.1</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>conv5.2</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>conv5.3</td>
<td>2m</td>
<td>36%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.3</td>
<td>14.8%</td>
<td>7.79%</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% (13×)</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>
Problem

- Techniques discussed above significantly reduce network size, but turn a “HW-friendly” dense workload into a sparse one
  - Many heavily-optimized CPU/GPU libraries for dense conv/fc layers
  - Sparse matrix-vector product significantly less efficient on modern processors

- Also incur new costs during inference time for decompression
  - Huffman decode
  - Process relative index offsets
  - Lookup weight from weight id
Sparse, weight-sharing fully-connected layer

\[ b_i = ReLU \left( \sum_{j=0}^{n-1} W_{ij} a_j \right) \]

Fully-connected layer:
Matrix-vector multiplication of activation vector \( a \) against weight matrix \( W \)

\[ b_i = ReLU \left( \sum_{j \in X_i \cap Y} S[I_{ij}] a_j \right) \]

Sparse, weight-sharing representation:
\( I_{ij} = \) index for weight \( W_{ij} \)
\( S[\cdot] = \) table of shared weight values
\( X_i = \) list of non-zero indices in row \( i \)
\( Y = \) list of non-zero indices in \( a \)

Note: activations are sparse due to ReLU
Efficient inference engine (EIE) ASIC

Custom hardware for decode and evaluate sparse, compressed DNNs

Hardware represents weight matrix in compressed sparse column (CSC) format to exploit sparsity in activations:

```
for each nonzero a_j in a:
    for each nonzero M_ij in column M_j:
        b_i += M_ij * a_j
```

More detailed version:

```
int16* a_values;
PTR* M_j_start;  // column j
int4* M_j_values;
int4* M_j_indices;
int16* lookup;  // lookup table for
                 // cluster values

for j=0 to length(a):
    if (a[j] == 0) continue;  // scan to nonzero
    col_values = M_j_values[M_j_start[j]];
    col_indices = M_j_indices[M_j_start[j]];
    col_nonzeros = M_j_start[j+1]-M_j_start[j];
    for i=0, i_count=0 to col_nonzeros:
        i += col_indices[i_count]
        b[i] += lookup[M_j_values[i]] * a_values[j_count]
```

* Keep in mind there’s a unique lookup table for each chunk of matrix values
Parallelization of sparse-matrix-vector product

Stride rows of matrix across processing elements

Output activations strided across processing elements

Weights stored local to PEs. Must broadcast non-zero \(a_j\)'s to all PEs

Accumulation of each output \(b_i\) is local to PE
EIE unit for quantized sparse/matrix vector product

Tuple representing non-zero activation \((a_j, j)\) arrives and is enqueued
The compressed DNN model is produced as described. The uncompressed DNN model is obtained of models: uncompressed DNN model and the compressed Tegra K1.

Components [26], [27] to report the AP+DRAM power for meter, then assumed so we measured the total power consumption with a power-cuBLAS GEMV for the original dense model and cuS-192 CUDA cores as our mobile GPU baseline. We used cuSPARSE CSRMV kernel, which is optimized for sparse the original dense layer. For the compressed sparse layer, the benchmark, we used cuBLAS GEMV to implement using a state-of-the-art GPU for deep learning as our baseline GPU.

Different off-the-shelf computing units: CPU, GPU and mobile.

Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

Sources of energy savings:
- Compression allows all weights to be stored in SRAM (few DRAM loads)
- Low-precision 16-bit fixed-point math (5x more efficient than 32-bit fixed math)
- Skip math on inputs activations that are zero (65% less math)

Warning: these are not end-to-end: just fully connected layers! And recall most of the compute is in the convo layers!

CPU: Core i7 5930k (6 cores)
GPU: GTX Titan X
mGPU: Tegra K1

CMU 15-769, Fall 2016
Thoughts

- EIE paper highlights performance on fully connected layers (see graph above)
  - Final layers of networks like AlexNet, VGG...
  - Common in recurrent network topologies like LSTMs

- But many state-of-the-art image processing networks have moved to fully convolutional solutions
  - Recall Inception, SqueezeNet, etc.
A fully convolutional network for image segmentation

Setting the pooling layers is the most straightforward way to obtain fusing even lower layers. Improvement visible e.g. in Figure emphasizes large-scale correctness, and also in terms of the minishing returns, both with respect to the IU metric which output. At this point our fusion improvements have met di-
a minor additional improvement to 62.7 mean IU, and find pool4 pool3
the quality of the output. insignificant performance improvement without improving learning rate without adding the skip, which resulted in an this fusion with learning only from the

provement in the fine structure of the output. We compared idation set by 3.0 mean IU to 62.4. Figure

The learning rate is decreased by a factor of 100.

initialized so that the net starts with unmodified predictions. the parameters of the last, coarser net, which we now call FCN-16s. FCN-16s is learned end-to-end, initialized with
tions are upsampled back to the image. We call this net linear interpolation, but allow the parameters to be learned

as described in Section

insignificant performance improvement without improving

and

⇥.conv7 ⇥.conv4 ⇥.conv5 ⇥.conv6-7

FCN-32s FCN-16s FCN-8s Ground truth

32x upsampled prediction (FCN-32s)
16x upsampled prediction (FCN-16s)
8x upsampled prediction (FCN-8s)
2x conv7 pool4 pool3
2x pool4 conv7
4x conv7
Temporal stability of deep features

Observation:
Deeper features feature more temporal stability

(more semantic information changes less rapidly in a scene)
Clockwork network: reuse deeper layer outputs in subsequent frames

Figure 3: The clockwork FCN with its stages and corresponding clocks.

A schematic of our clockwork FCN is shown in Figure 3. There are several choice points in defining a clockwork architecture. We define a novel, generalized clockwork framework, which can purposely schedule deeper layers more slowly than shallower layers. We form our modules by grouping the layers of a convnet to span the feature hierarchy. Our networks persists both state and output across time steps. The clockwork recurrent network of [4], designed for long-term dependency modeling of time series, is an instance of our more general scheme for clockwork computation. The differences in architecture and outputs over time between clockwork recurrence and our clockwork are shown in Figure 4.

While different, these nets can be expressed by generalized clockwork equations

\[ y(t) = f_T(C(t)) + f_H(y(t-1)) + C(t)I \]

\[ y(t) = f_O(C(t))f_H(y(t)) \]

Evaluate lower (early) layers each frame
Optionally combine (fresh) output of lower layers with output of higher layers from previous frames.
Today: three types of optimizations

- **Static, manual:** human construction of new, more efficient topologies (e.g., Inception, SqueezeNet)

- **Static, automatic analysis driven:** (e.g., deep compression) analyze contents of network to determine how to prune topology or quantize weights

- **Dynamic:** analyze network activations during inference to determine when subsequent work can be elided (e.g., clockwork network)

Note: EIE hardware also exploited dynamic sparsity in activations (e.g., due to ReLUs), but this was not an approximation technique like the ones above

- **Custom specialized hardware** to handle irregularity introduced by these optimizations