Lecture 18:
Efficiently Evaluating Deep Networks

Parallel Computer Architecture and Programming
CMU / 清华大学, Summer 2017
Today

- High-performance evaluation of deep neural networks (performing “inference”)

- We will focus on the parallelism challenges of training deep networks next time
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

Figure 10: Qualitative results of our method on the MPII, LSP and FLIC datasets respectively. We see that the method is able to handle non-standard poses and resolve ambiguities between symmetric parts for a variety of different relative camera views.

Figure 11: Comparing PCKh-0.5 across various viewpoints in the MPII dataset. Our method is significantly better in all the viewpoints.

FLIC Dataset. We evaluate our method on the FLIC Dataset which consists of 3987 images for training and 1016 images for testing. We report accuracy as per the metric introduced in Sapp et al. for the elbow and wrist joints in Figure 12. Again, we outperform all prior art at PCK@0.2 with 97.59% on elbows and 95.03% on wrists. In higher precision region our advantage is even more significant: 14.8 percentage points on wrists and 12.7 percentage points on elbows at PCK@0.05, and 8.9 percentage points on wrists and 9.3 percentage points on elbows at PCK@0.1.

5. Discussion

Convolutional pose machines provide an end-to-end architecture for tackling structured prediction problems in computer vision without the need for graphical-model style inference. We showed that a sequential architecture composed of convolutional networks is capable of implicitly learning a spatial models for pose by communicating increasingly refined uncertainty-preserving beliefs between stages. Problems with spatial dependencies between variables arise in multiple domains of computer vision such as semantic image labeling, single image depth prediction and object detection and future work will involve extending our architecture to these problems. Our approach achieves state-of-the-art accuracy on all primary benchmarks, however we do observe failure cases mainly when multiple people are in close proximity. Handling multiple people in a single end-to-end architecture is also a challenging problem and an interesting avenue for future work.
Consider the following expression

\[
\max( \max(0, (a \times b) + (c \times d)) + (e \times f) + (g \times h), i \times j )
\]
Consider expressing logic via a lookup table

- Recall example from last week: 6-input, 1 output lookup table (LUT)
  - Think of a LUT6 as a 64 element table capable of expression functions of 6 bit inputs

Example: 6-input AND

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>63</td>
<td>1</td>
</tr>
</tbody>
</table>
What is a deep neural network?

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

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

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

Basic computational interpretation:
It is 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
$$f(x) = \frac{1}{1 + e^{-x}}$$
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
(every hidden unit receives input from all units on prior layer)
What is a deep neural network: topology

- **Fully connected layer**

- **Sparsely (locally) connected layer** (each unit only received inputs from three input nodes)
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 illustration above only shows links for a 1D conv: a.k.a. one iteration of ii loop)
Strided 3x3 convolution

```c
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;
    }
}
```
What does convolution using these filter weights do? 

\[
\begin{bmatrix}
0.111 & 0.111 & 0.111 \\
0.111 & 0.111 & 0.111 \\
0.111 & 0.111 & 0.111 \\
\end{bmatrix}
\]

“Box blur”
What does convolution using these filter weights do?

\[
\begin{bmatrix}
.075 & .124 & .075 \\
.124 & .204 & .124 \\
.075 & .124 & .075 \\
\end{bmatrix}
\]

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

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

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

Extracts horizontal gradients
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 \( \times \) num\_filters weights

Output: filter responses \( W \times H \times \) num\_filters

Each filter described by unique set of 3x3 weights (each filter “responds” to different image phenomena)

Filter response maps (num\_filters of them)
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 x num_fi weights

Output: filter responses \(W \times H \times \text{num}_f\)lter\)

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

Filter responses

post ReLU \(W \times H \times \text{num}_f\)lter\)

ReLU

post pool \(W/2 \times H/2 \times \text{num}_f\)lter\)

(max response in 2x2 region)

Note data reduction as a result of pooling
Example: “AlexNet” object detection network

Sequences of conv + reLU + pool (optional) layers

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

Another example: VGG-16 [Simonyan15]: 13 convolutional layers

input: 224 x 224 RGB
conv/reLU: 3x3x3x64
conv/reLU: 3x3x64x64
maxpool
conv/reLU: 3x3x64x128
conv/reLU: 3x3x128x128
maxpool
conv/reLU: 3x3x128x256
conv/reLU: 3x3x256x256
maxpool
conv/reLU: 3x3x256x512
conv/reLU: 3x3x512x512
maxpool
conv/reLU: 3x3x512x1024
conv/reLU: 3x3x1024x1024
maxpool
conv/reLU: 3x3x1024x2048
conv/reLU: 3x3x2048x2048
maxpool
conv/reLU: 3x3x2048x4096
conv/reLU: 3x3x4096x4096
maxpool
conv/reLU: 3x3x4096x8192
conv/reLU: 3x3x8192x8192
maxpool
fully-connected 4096
fully-connected 4096
soft-max

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

Layer 1

Layer 2

Layer 3

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

[Image credit: Zeiler 14]
Why deep?

[Image credit: Zeiler 14]
More recent image understanding networks

![Image](image-url)

Residual Network.

Figure 3: GoogLeNet network with all the bells and whistles.

More recent image understanding networks

Fig. 3). When the dimensions increase (dotted line shortcuts are usually reported: the top-1 accuracy rate, which compares the ground truth images for testing. Each image is associated with one parameter; (B) The projection shortcut in Eqn. performs identity mapping, with extra zero entries padded horizontally sampled in...
Deep networks learn useful representations

- Simultaneous, multi-scale learning of useful concepts for the task at hand
  - Example on previous slides: subparts detectors emerged in network for object classification

- But wait... how did you learn the values of all the weights?
  - Next lecture!
  - For today, assume the weights are given (today is about evaluating deep networks, not training them)
Efficiently implementing convolution layers
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)**
Blocked dense matrix multiplication

\[
\begin{align*}
\text{float } A[M][K]; \\
\text{float } B[K][N]; \\
\text{float } C[M][N];
\end{align*}
\]

// compute \( C \leftarrow A \times B \)
#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++)
                        \[ C[jblock+j][iblock+i] \leftarrow A[jblock+j][kblock+k] \times B[kblock+k][iblock+i]; \]

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?
Hierarchical blocked matrix mult

Exploit multiple levels of memory hierarchy

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

// compute C += A * B
#pragma omp parallel for
for (int jblock2=0; jblock2<M; jblock2+=L2_BLOCKSIZE_J)
  for (int iblock2=0; iblock2<N; iblock2+=L2_BLOCKSIZE_I)
    for (int kblock2=0; kblock2<K; kblock2+=L2_BLOCKSIZE_K)
      for (int jblock1=0; jblock1<L1_BLOCKSIZE_J; jblock1+=L1_BLOCKSIZE_J)
        for (int iblock1=0; iblock1<L1_BLOCKSIZE_I; iblock1+=L1_BLOCKSIZE_I)
          for (int kblock1=0; kblock1<L1_BLOCKSIZE_K; kblock1+=L1_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++)
                  ...
```

Not shown: final level of “blocking” for register locality…
Blocked dense matrix multiplication (1)

Consider SIMD parallelism within a block

\[
\begin{array}{c}
\text{BLOCKSIZE}_I \\
\text{BLOCKSIZE}_J \\
\text{BLOCKSIZE}_K \\
\end{array}
\]

\[
\text{C} = \text{A} \times \text{B}
\]

... 
for (int j=0; j<BLOCKSIZE_J; j++) {
    for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {
        simd_vec C_accum = vec_load(&C[jblock+j][iblock+i]);
        for (int k=0; k<BLOCKSIZE_K; k++) {
            // C = A*B + C
            simd_vec A_val = splat(&A[jblock+j][kblock+k]); // load a single element in vector register
            simd_muladd(A_val, vec_load(&B[kblock+k][iblock+i]), C_accum);
        }
        vec_store(&C[jblock+j][iblock+i], C_accum);
    }
}

Vectorize i loop

Good: also improves spatial locality in access to B
Bad: working set increased by SIMD_WIDTH, still walking over B in large steps
... for (int j=0; j<BLOCKSIZE_J; j++)
    for (int i=0; i<BLOCKSIZE_I; i++) {
        float C_scalar = C[jblock+j][iblock+i];
        // C_scalar += dot(row of A, row of B)
        for (int k=0; k<BLOCKSIZE_K; k+=SIMD_WIDTH) {
            C_scalar += simd_dot(vec_load(&A[jblock+j][kblock+k]), vec_load(&Btrans[iblock+i][kblock+k]));
        }
        C[jblock+j][iblock+i] = C_scalar;
    }

Assume i dimension is small. Previous vectorization scheme (1) would not work well. Pre-transpose block of B (copy block of B to temp buffer in transposed form) Vectorize innermost loop
Blocked dense matrix multiplication (3)

// assume blocks of A and C are pre-transposed as Atrans and Ctrans
for (int j=0; j<BLOCKSIZE_J; j+=SIMD_WIDTH) {
    for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {

        simd_vec C_accum[SIMD_WIDTH];
        for (int k=0; k<SIMD_WIDTH; k++) // load C_accum for a SIMD_WIDTH x SIMD_WIDTH chunk of C^T
            C_accum[k] = vec_load(&Ctrans[iblock+i+k][jblock+j]);

        for (int k=0; k<BLOCKSIZE_K; k++) {
            simd_vec bvec = vec_load(&B[kblock+k][iblock+i]);
            for (int kk=0; kk<SIMD_WIDTH; kk++) // innermost loop items not dependent
                simd_muladd(vec_load(&Atrans[kblock+k][jblock+j], splat(bvec[kk]), C_accum[kk]);
        }

        for (int k=0; k<SIMD_WIDTH; k++)
            vec_store(&Ctrans[iblock+i+k][jblock+j], C_accum[k]);
    }
}
Convolution as matrix-vector product

Construct matrix from elements of input image

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

0(N) storage multiplier 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

| \( x_{00} \) | \( x_{01} \) | \( x_{02} \) | \( x_{03} \) | ... |
| \( x_{10} \) | \( x_{11} \) | \( x_{12} \) | \( x_{13} \) | ... |
| \( x_{20} \) | \( x_{21} \) | \( x_{22} \) | \( x_{23} \) | ... |
| \( x_{30} \) | \( x_{31} \) | \( x_{32} \) | \( x_{33} \) | ... |
| ... | ... | ... | ... | ... |

\[ W \times H \]

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & x_{00} & x_{01} & 0 & x_{10} & x_{11} \\
0 & 0 & 0 & x_{00} & x_{01} & x_{02} & x_{10} & x_{11} & x_{12} \\
0 & 0 & 0 & x_{01} & x_{02} & x_{03} & x_{11} & x_{12} & x_{13} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
w_{00} & w_{01} & w_{02} & \cdots & w_{0N} \\
w_{10} & w_{11} & w_{12} & \cdots & w_{1N} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
w_{80} & w_{81} & w_{82} & \cdots & w_{8N}
\end{pmatrix}
\]

\( 9 \) \hspace{2cm} \text{num filters}
Multiple convolutions on multiple input channels

For each filter, sum responses over input channels

Equivalent to (3 x 3 x num_channels) convolution on (W x H x num_channels) input data
# VGG memory footprint

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

<table>
<thead>
<tr>
<th>Layer Type</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>

Inputs/outputs get multiplied by image batch size!!! (for VGG's 3x3 convolutions, this is 9x data amplification)
Direct implementation of conv layer

```c
float input[IMAGE_BATCH_SIZE][INPUT_HEIGHT][INPUT_WIDTH][INPUT_DEPTH];
float output[IMAGE_BATCH_SIZE][INPUT_HEIGHT][INPUT_WIDTH][LAYER_NUM_FILTERS];
float layer_weights[LAYER_NUM_FILTERS][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[img][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_FILTER_Y; jj++)
            // spatial convolution (Y)
            for (int ii=0; ii<LAYER_FILTER_X; ii++)
                // spatial convolution (X)
                output[img][j][i][f] += layer_weights[f][jj][ii][kk] * input[img][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)

Avoids $O(N)$ footprint increase by avoiding materializing input matrix
In theory loads $O(N)$ times less data (potentially higher arithmetic intensity... but matrix mult is typically compute-bound)
But must roll your own highly optimized implementation of complicated loop nest.
**Conv layer in Halide**

```c
int in_w, in_h, in_ch = 4; // input params: assume initialized

Func in_func; // assume input function is initialized

int num_f, f_w, f_h, pad, stride; // parameters of the conv layer

Func forward = Func("conv");
Var x("x"), y("y"), z("z"), n("n"); // n is minibatch dimension

// This creates a padded input to avoid checking boundary
// conditions while computing the actual convolution
f_in_bound = BoundaryConditions::repeat_edge(in_func, 0, in_w, 0, in_h);

// Create image buffers for layer parameters
Image<float> W(f_w, f_h, in_ch, num_f)
Image<float> b(num_f);

// domain of summation for filter with W x H x in_ch
RDom r(0, f_w, 0, f_h, 0, in_ch);

// Initialize to bias
forward(x, y, z, n) = b(z);
forward(x, y, z, n) += W(r.x, r.y, r.z, z) *
    f_in_bound(x*stride + r.x - pad, y*stride + r.y - pad, r.z, n);
```

Consider scheduling this seven-dimensional loop nest.
Algorithmic improvements

- Direct convolution can be implemented efficiently in Fourier domain (convolution $\rightarrow$ element-wise multiplication)
  - Overhead: FFT to transform inputs into Fourier domain, inverse FFT to get responses back to spatial domain ($N\log N$)
  - Inverse transform amortized over all input channels (due to summation over inputs)

- Direct convolution using work-efficient Winograd convolutions

  1D example: consider producing two outputs of a 3-tap 1D convolution with weights: $w_0 \ w_1 \ w_2$

  \[
  \begin{bmatrix}
  x_0 & x_1 & x_2 & x_3 \\
  \downarrow & & & \\
  y_0 & y_1
  \end{bmatrix}
  \]

  \[
  \begin{bmatrix}
  y_0 \\
  y_1
  \end{bmatrix} = \begin{bmatrix}
  x_0 & x_1 & x_2 \\
  x_1 & x_2 & x_3
  \end{bmatrix}\begin{bmatrix}
  w_0 \\
  w_1 \\
  w_2
  \end{bmatrix} = \begin{bmatrix}
  m_1 + m_2 + m_3 \\
  m_2 - m_3 - m_4
  \end{bmatrix}
  \]

  \[
  m_1 = (x_0 - x_1)w_0 \\
  m_2 = (x_1 + x_2)\frac{w_0 + w_1 + w_2}{2} \\
  m_3 = (x_2 - x_1)\frac{w_0 - w_1 + w_2}{2} \\
  m_4 = (x_1 - x_3)w_2
  \]

  Winograd 1D 3-element filter:
  - 4 multiplies
  - 8 additions
  - (4 to compute m’s + 4 to reduce final result)

  Direct convolution: 6 multiplies, 4 adds

  In 2D can notably reduce multiplications
  (3x3 filter: 2.25x fewer multiplies for 2x2 block of output)
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
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)
"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.

$$f(x) = \max(0, x)$$
Representing “sparsified” networks

Step 1: prune low-weight links (iteratively retrain network, then prune)
- Over 90% of weights in fully connected layers 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>

Reduce storage overhead 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

Step 2: 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 (\(\log(k)\) bits)
- This is lossy compression

```
weights (32 bit float)

<table>
<thead>
<tr>
<th></th>
<th>2.09</th>
<th>-0.98</th>
<th>1.48</th>
<th>0.09</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.05</td>
<td>-0.14</td>
<td>-1.08</td>
<td>2.12</td>
</tr>
<tr>
<td></td>
<td>-0.91</td>
<td>1.92</td>
<td>0</td>
<td>-1.03</td>
</tr>
<tr>
<td></td>
<td>1.87</td>
<td>0</td>
<td>1.53</td>
<td>1.49</td>
</tr>
</tbody>
</table>

cluster index (2 bit uint)

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>0</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Step 3: Huffman encode quantized weights and CSR indices (lossless compression)
VGG-16 compression

Large savings in fully connected layers 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>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>

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>

* Benefits of automatic pruning apply mainly to fully connected layers, but many more modern networks are dominated by costs of convolutional layers.

CMU / 清华大学, Summer 2017
More efficient topologies

- Original DNNs for image recognition where overprovisioned
  - Large filters, many filters
- Modern DNNs designs as hand-designed to be sparser

SqueezeNet: [Iandola 2017] Reduced number of parameters in AlexNet by 50x, with similar performance on image classification
MobileNet

[Howard et al. 2017]

**Factor NUM_FILTERS 3x3xNUM_CHANNELS convolutions into:**
- NUM_CHANNELS 3x3x1 depth-wise separable convolutions for each input channel
- And NUM_FILTERS 1x1xNUM_CHANNELS convolutions to combine the results

---

**Image classification (ImageNet)**

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 MobileNet-224</td>
<td>70.6%</td>
<td>569</td>
<td>4.2</td>
</tr>
<tr>
<td>GoogleNet</td>
<td>69.8%</td>
<td>1550</td>
<td>6.8</td>
</tr>
<tr>
<td>VGG 16</td>
<td>71.5%</td>
<td>15300</td>
<td>138</td>
</tr>
</tbody>
</table>

**Image classification (ImageNet)**

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50 MobileNet-160</td>
<td>60.2%</td>
<td>76</td>
<td>1.32</td>
</tr>
<tr>
<td>Squeezenet</td>
<td>57.5%</td>
<td>1700</td>
<td>1.25</td>
</tr>
<tr>
<td>AlexNet</td>
<td>57.2%</td>
<td>720</td>
<td>60</td>
</tr>
</tbody>
</table>

---

**Table 1. MobileNet Body Architecture**

<table>
<thead>
<tr>
<th>Type / Stride</th>
<th>Filter Shape</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv / s2</td>
<td>$3 \times 3 \times 3 \times 32$</td>
<td>$224 \times 224 \times 3$</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>$3 \times 3 \times 32 \ dw$</td>
<td>$112 \times 112 \times 32$</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>$1 \times 1 \times 32 \times 64$</td>
<td>$112 \times 112 \times 64$</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>$3 \times 3 \times 64 \ dw$</td>
<td>$112 \times 112 \times 64$</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>$1 \times 1 \times 64 \times 128$</td>
<td>$56 \times 56 \times 128$</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>$3 \times 3 \times 128 \ dw$</td>
<td>$56 \times 56 \times 128$</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>$1 \times 1 \times 128 \times 128$</td>
<td>$56 \times 56 \times 128$</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>$3 \times 3 \times 128 \ dw$</td>
<td>$56 \times 56 \times 128$</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>$1 \times 1 \times 128 \times 256$</td>
<td>$28 \times 28 \times 256$</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>$3 \times 3 \times 256 \ dw$</td>
<td>$28 \times 28 \times 256$</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>$1 \times 1 \times 256 \times 256$</td>
<td>$28 \times 28 \times 256$</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>$3 \times 3 \times 256 \ dw$</td>
<td>$28 \times 28 \times 256$</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>$1 \times 1 \times 256 \times 512$</td>
<td>$14 \times 14 \times 512$</td>
</tr>
<tr>
<td>$5 \times$</td>
<td>Conv dw / s1</td>
<td>$3 \times 3 \times 512 \ dw$</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>$1 \times 1 \times 512 \times 512$</td>
<td>$14 \times 14 \times 512$</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>$3 \times 3 \times 512 \ dw$</td>
<td>$14 \times 14 \times 512$</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>$1 \times 1 \times 512 \times 1024$</td>
<td>$7 \times 7 \times 512$</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>$3 \times 3 \times 1024 \ dw$</td>
<td>$7 \times 7 \times 1024$</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>$1 \times 1 \times 1024 \times 1024$</td>
<td>$7 \times 7 \times 1024$</td>
</tr>
<tr>
<td>Avg Pool / s1</td>
<td>Pool 7 \times 7</td>
<td>$7 \times 7 \times 1024$</td>
</tr>
<tr>
<td>FC / s1</td>
<td>$1024 \times 1000$</td>
<td>$1 \times 1 \times 1024$</td>
</tr>
<tr>
<td>Softmax / s1</td>
<td>Classifier</td>
<td>$1 \times 1 \times 1000$</td>
</tr>
</tbody>
</table>
Deep neural networks on GPUs

- Many high-performance 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!)
Increasing efficiency through specialization

Example: Google’s Tensor Processing Unit (TPU) Accelerates deep learning operations in Google datacenter

Intel has announced Lake Crest ML accelerator (formerly called Nervana)
Emerging architectures for deep learning

- **NVIDIA Pascal (NVIDIA’s latest GPU)**
  - Adds double-throughput 16-bit floating point ops
  - This feature is already common on mobile GPUs

- **Intel Xeon Phi (Knights Landing)**
  - Flop rich 72-core x86 processor for scientific computing and machine learning

- **FPGAs, ASICs**
  - Not new: FPGA solutions have been explored for years
  - Significant amount of ongoing industry and academic research
  - Many efforts around the world (both big companies and startups) seek to produce ASIC accelerators for evaluating deep networks!
Programming frameworks for deep learning

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

- Popular domain-specific frameworks use these kernel libraries
  - Caffe2, Torch, TensorFlow, MxNet

- DNN application development involves constructing novel network topologies
Still early stages of designing productive/performant programming systems for deep learning

Current status quo: wire up data flow graph of nodes corresponding to common layer types (conv layers, maxpool, batch norm, etc.)
Summary: efficiently evaluating deep nets

- **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**

- **Many ongoing studies of specialized hardware architectures for efficient evaluation**
  - Future CPUs/GPUs, ASICs, FPGS, ...
  - Specialization will be important to achieving “always on” applications