Lecture 8:
Efficiently Evaluating Deep Networks

Visual Computing Systems
Stanford CS348V, Winter 2018
Today

- We will discuss the workload created by need to evaluate deep neural networks (performing “inference”) on image datasets

- We will focus on the parallelism challenges of training deep networks next time
Consider the following expression:

\[
\max(\max(0, (a \times b) + (c \times d)) + (e \times f) + (g \times h), i \times j)
\]
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? 

\[
\begin{bmatrix}
.111 & .111 & .111 \\
.111 & .111 & .111 \\
.111 & .111 & .111 \\
\end{bmatrix}
\]

“Box blur”
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 $3 \times 3 \times \text{num\_filters}$ weights

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

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\_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

ReLU

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

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

(max response in 2x2 region)

Note data reduction as a result of pooling

Stanford CS348V, Winter 2018
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
cnv/ReLU: 3x3x64x64
maxpool
conv/ReLU: 3x3x64x128
cnv/ReLU: 3x3x128x128
maxpool
conv/ReLU: 3x3x128x256
cnv/ReLU: 3x3x256x256
maxpool
cnv/ReLU: 3x3x256x512
cnv/ReLU: 3x3x512x512
maxpool
cnv/ReLU: 3x3x512x512
cnv/ReLU: 3x3x512x512
maxpool
full-connected 4096
full-connected 4096
full-connected 1000
softmax

[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?

Layer 4

Layer 5

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

Inception (GoogleLeNet)

ResNet (34 layer version)

Convolution network

Upsampling network

FCN for image segmentation

Figure 3. Example network architectures for ImageNet.

Left: a residual network with 34 parameter layers (3.6 billion)

Right: a residual network with 34 parameter layers (3.6 billion)

Table 1 (left) we com-

De-
Deep networks learn useful representations

- Simultaneous, multi-scale learning of useful features 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

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?
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

... 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
// 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

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

Construct matrix from elements of input image

Note: 0-pad matrix

$X_{00} \quad X_{01} \quad X_{02} \quad X_{03} \quad ...$

$X_{10} \quad X_{11} \quad X_{12} \quad X_{13} \quad ...$

$X_{20} \quad X_{21} \quad X_{22} \quad X_{23} \quad ...$

$X_{30} \quad X_{31} \quad X_{32} \quad X_{33} \quad ...$

$... \quad ... \quad ... \quad ... \quad ...$

$W \times H$

$9$

$\begin{bmatrix}
0 & 0 & 0 & x00 & x01 & x10 & x11 \\
0 & 0 & 0 & x00 & x01 & x02 & x10 & x11 & x12 \\
0 & 0 & 0 & x01 & x02 & x03 & x11 & x12 & x13 \\
... \\
x00 & x01 & x02 & x10 & x11 & x12 & x20 & x21 & x22 \\
... 
\end{bmatrix}$

$w_0$

$w_1$

$w_8$

$O(N)$ storage overhead for filter with N elements
Must construct input data matrix
Multiple convolutions as matrix-matrix mult
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 Description</th>
<th>Weights Mem (per image)</th>
<th>(mem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>input: 224 x 224 RGB image</td>
<td>224x224</td>
<td>150K</td>
</tr>
<tr>
<td>conv: (3x3x3) x 64</td>
<td>224x224x64</td>
<td>12.3 MB</td>
</tr>
<tr>
<td>conv: (3x3x64) x 64</td>
<td>224x224x64</td>
<td>12.3 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>112x112x64</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>conv: (3x3x64) x 128</td>
<td>112x112x128</td>
<td>6.2 MB</td>
</tr>
<tr>
<td>conv: (3x3x128) x 128</td>
<td>112x112x128</td>
<td>6.2 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>56x56x128</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>conv: (3x3x128) x 256</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>conv: (3x3x256) x 256</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>conv: (3x3x256) x 256</td>
<td>56x56x256</td>
<td>3.1 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>28x28x256</td>
<td>766 KB</td>
</tr>
<tr>
<td>conv: (3x3x256) x 512</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>28x28x512</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>maxpool</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>conv: (3x3x512) x 512</td>
<td>14x14x512</td>
<td>383 KB</td>
</tr>
<tr>
<td>maxpool</td>
<td>7x7x512</td>
<td>98 KB</td>
</tr>
<tr>
<td>fully-connected 4096</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>fully-connected 4096</td>
<td>4096</td>
<td>16 KB</td>
</tr>
<tr>
<td>fully-connected 1000</td>
<td>1000</td>
<td>4 KB</td>
</tr>
<tr>
<td>soft-max</td>
<td>1000</td>
<td>4 KB</td>
</tr>
</tbody>
</table>
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

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.
Each layer benefits from having a unique schedule

Throughput: Input-Specialized Schedules (relative to best-on-average schedule)

Relative Throughput

VGG-16 conv layers

L1  L2  L3  L4  L5  L6  L7  L8  L9

[Figure credit: Mullapudi et al. 2016]
Algorithmic improvements

Direct convolution can be implemented efficiently in Fourier domain (convolution → element-wise multiplication)
- Overhead: FFT to transform inputs into Fourier domain, inverse FFT to get responses back to spatial domain (N\lg 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: \( w_0 \ w_1 \ w_2 \)

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

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

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

Filter dependent (can be precomputed)

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><strong>32 bit DRAM Memory</strong></td>
<td><strong>640</strong></td>
<td><strong>6400</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Estimates for 45nm process</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Source: Mark Horowitz]</td>
</tr>
</tbody>
</table>

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 of early DNN designs
  - 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
  - \((20 \times 1\text{B} \times 640\text{pJ}) = 12.8\text{W}\) for DRAM access
    (more than power budget of any modern smartphone)
Is this an opportunity for compression?
“Pruning” (sparsifying) a network

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

\[ f(x) = \max(0, x) \]
“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>

| Indices | 0   | 1.8 | 0   | 0   | 0   | 0   | 0   | 0   | 1.1 | ... |

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>5</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 (lg(k) bits)
- This is lossy compression

![Cluster weights via k-means clustering](image)

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>0.05</td>
<td>-0.14</td>
<td>-1.08</td>
<td>2.12</td>
<td></td>
</tr>
<tr>
<td>-0.91</td>
<td>1.92</td>
<td>0</td>
<td>-1.03</td>
<td></td>
</tr>
<tr>
<td>1.87</td>
<td>0</td>
<td>1.53</td>
<td>1.49</td>
<td></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>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

centroids

<table>
<thead>
<tr>
<th></th>
<th>3:</th>
<th>2:</th>
<th>1:</th>
<th>0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>1.50</td>
<td>0.00</td>
<td>-1.00</td>
<td></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>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>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>Model</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

Stanford CS348V, Winter 2018
But wait... This a great example of non-domain-specific vs. domain-specific approach to innovation
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

Inception (GoogleLeNet) — 27 total layers, 7M parameters

ResNet (34 layer version)
Modular network designs

Inception v4
Input (299x299x3)

Stem
Output (299x299x3)

4 x Inception-A
Output: 35x35x384

Reduction-A
Output: 17x17x1024

7 x Inception-B
Output: 17x17x1024

Reduction-B
Output: 8x8x1536

3 x Inception-C
Output: 8x8x1536

Avarage Pooling
Output: 1536

Dropout (keep 0.8)
Output: 1536

Softmax
Output: 1000

Figure 9. The overall schema of the Inception-v4 network. For the detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.
Inception stem

Historically, we have been relatively conservative about changing the architectural choices and restricted our experiments to varying isolated network components while keeping the rest of the network stable. Not simplifying earlier choices resulted in networks that looked more complicated than they needed to be. In our newer experiments, for Inception-v4 we decided to shed this unnecessary baggage and made uniform choices for the Inception blocks for each grid size. Please refer to Figure 9 for the large scale structure of the Inception-v4 network and Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of its components. All the convolutions not marked with "V" in the figures are same-padded meaning that their output grid matches the size of their input. Convolutions marked with "V" are valid padded, meaning that input patch of each unit is fully contained in the previous layer and the grid size of the output activation map is reduced accordingly.

3.2. Residual Inception Blocks

For the residual versions of the Inception networks, we use cheaper Inception blocks than the original Inception. Each Inception block is followed by filter-expansion layer (1 × 1 convolution without activation) which is used for scaling up the dimensionality of the filter bank before the addition to match the depth of the input. This is needed to compensate for the dimensionality reduction induced by the Inception block.

We tried several versions of the residual version of Inception. Only two of them are detailed here. The first one "Inception-ResNet-v1" roughly the computational cost of Inception-v3, while "Inception-ResNet-v2" matches the raw cost of the newly introduced Inception-v4 network. See Figure 15 for the large scale structure of both variants. (However, the step time of Inception-v4 proved to be significantly slower in practice, probably due to the larger number of layers.)

Another small technical difference between our residual and non-residual Inception variants is that in the case of Inception-ResNet, we used batch-normalization only on top of the traditional layers, but not on top of the summations. It is reasonable to expect that a thorough use of batch-normalization should be advantageous, but we wanted to keep each model replica trainable on a single GPU. It turned out that the memory footprint of layers with large activation size was consuming disproportionate amount of GPU-memory. By omitting the batch-normalization on top of those layers, we were able to increase the overall number of Inception blocks substantially. We hope that with better utilization of computing resources, making this trade-off will become unnecessary.
Figure 10. The schema for $35 \times 35$ grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.
In this paper we analysed multiple state-of-the-art deep neural networks submitted to the ImageNet
challenge. We introduce with this chart our choice of network architecture, but similar results can be
generated on most recent GPUs, such as the NVIDIA K40 or Titan X, to name a few. Operation counts were
obtained using an open-source tool that we created to handle the complexity of the different
networks.

ImageNet Top 1 Accuracy

Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the
blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner.

Accuracy (points) per flop

The accuracy of a network is inversely proportional to the number of operations required for its forward
pass. This means that more efficient architectures will perform better when compared on the basis of the
number of operations required to reach the same accuracy, which is a significant metric for practical
applications where computational resources are limited.

Figure credit: Canziani et al 2017

Stanford CS348V, Winter 2018
## Improving accuracy/cost (image classification)

2014 → 2017  ~ 25x improvement in cost at similar accuracy

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Top-1 Accuracy</th>
<th>Num Params</th>
<th>Cost/image (MADDs)</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-16</td>
<td>71.5%</td>
<td>138M</td>
<td>15B</td>
<td>[2014]</td>
</tr>
<tr>
<td>GoogleNet</td>
<td>70%</td>
<td>6.8M</td>
<td>1.5B</td>
<td>[2015]</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>73% *</td>
<td>11.7M</td>
<td>1.8B</td>
<td>[2016]</td>
</tr>
<tr>
<td>MobileNet-224</td>
<td>70.5%</td>
<td>4.2M</td>
<td>0.6B</td>
<td>[2017]</td>
</tr>
</tbody>
</table>

* 10-crop results (ResNet 1-crop results are similar to other DNNs in this table)
MobileNet

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

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 x 3 x 3 x 32</td>
<td>224 x 224 x 3</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>3 x 3 x 32 dw</td>
<td>112 x 112 x 32</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 32 x 64</td>
<td>112 x 112 x 32</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 64 dw</td>
<td>112 x 112 x 64</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 64 x 128</td>
<td>56 x 56 x 64</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>3 x 3 x 128 dw</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 128 x 128</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 128 dw</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 128 x 256</td>
<td>28 x 28 x 128</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>3 x 3 x 256 dw</td>
<td>28 x 28 x 256</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 256 x 256</td>
<td>28 x 28 x 256</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 256 dw</td>
<td>28 x 28 x 256</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 256 x 512</td>
<td>14 x 14 x 256</td>
</tr>
<tr>
<td>5 x Conv dw / s1</td>
<td>3 x 3 x 512 dw</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 512 x 512</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 512 dw</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 512 x 1024</td>
<td>7 x 7 x 512</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 1024 dw</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 1024 x 1024</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>Avg Pool / s1</td>
<td>Pool 7 x 7</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>FC / s1</td>
<td>1024 x 1000</td>
<td>1 x 1 x 1024</td>
</tr>
<tr>
<td>Softmax / s1</td>
<td>Classifier</td>
<td>1 x 1 x 1000</td>
</tr>
</tbody>
</table>

Table 9. Smaller MobileNet Comparison to Popular Models

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million</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>
<td></td>
</tr>
<tr>
<td>GoogleNet</td>
<td>69.8%</td>
<td>1550</td>
<td>6.8</td>
<td></td>
</tr>
<tr>
<td>VGG 16</td>
<td>71.5%</td>
<td>15300</td>
<td>138</td>
<td></td>
</tr>
</tbody>
</table>

Table 11. Performance of PlaNet using the MobileNet architecture

<table>
<thead>
<tr>
<th>Model ImageNet</th>
<th>Million</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>PlaNet 32</td>
<td>10%</td>
<td>94.86%</td>
<td>0.75 MobileNet-224</td>
</tr>
<tr>
<td>PlaNet 64</td>
<td>18%</td>
<td>93.55%</td>
<td>1.0 MobileNet-224</td>
</tr>
<tr>
<td>PlaNet 128</td>
<td>32%</td>
<td>88.34%</td>
<td>5.0 MobileNet-224</td>
</tr>
</tbody>
</table>

Stanford CS348V, Winter 2018
Value of improving DNN topology

- Increasing overall accuracy on a task (often primary goal of CV/ML papers)
- Increasing accuracy/unit cost
- What is cost?
  - Ops (often measured in multiply adds)
  - Bandwidth!
    - Reading model weights + intermediate activations
    - Careful! Certain layers are bandwidth bound, e.g., batch norm

Depthwise separable convolutions add additional batch norm to network (after each step of depthwise conv layer)

Input: Values of \( x \) over a mini-batch: \( \mathcal{B} = \{x_1...m\} \); Parameters to be learned: \( \gamma, \beta \)
Output: \( \{y_i = \text{BN}_{\gamma,\beta}(x_i)\} \)

\[
\mu_\mathcal{B} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean}
\]

\[
\sigma^2_\mathcal{B} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \quad \text{// mini-batch variance}
\]

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}} \quad \text{// normalize}
\]

\[
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\]
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!
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)

- **Significant interest in reducing size of networks for both training and evaluation**

- **Algorithmic techniques (better algorithms) are responsible for huge speedups in recent years**
  - This work is complemented and extended by much ongoing work on efficient mapping of key layers to CPUs/GPUs and on custom hardware for evaluation