Lecture 5:
Efficiently Evaluating Deep Networks

Visual Computing Systems
Stanford CS348K, Spring 2020
Today

- We will discuss the workload created by need to evaluate deep neural networks (performing “inference”) on images
  - This lecture will be heavily biased towards concerns for DNNs that process images

- We will focus on the parallelism challenges of training deep networks in a future class
Review: what does convolution with these filters do?

\[
\begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1
\end{bmatrix}
\]

Extracts horizontal gradients

\[
\begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
1 & 2 & 1
\end{bmatrix}
\]

Extracts vertical gradients
Gradient detection filters

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 x num_filters weights

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

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
  - $3x3 \times \text{num\_filters}$ weights

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

- Filters described by unique set of weights (responds to different image phenomena)

Post ReLU

- Filter responses $W \times H \times \text{num\_filters}$

- ReLU

- Pool (max response in 2x2 region)$W/2 \times H/2 \times \text{num\_filters}$

Post pool

Note data reduction as a result of pooling
Example: “AlexNet” image classification DNN

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: 3x3x1024x512
conv/reLU: 3x3x512x512
maxpool
conv/reLU: 3x3x512x128
conv/reLU: 3x3x128x128
maxpool
conv/reLU: 3x3x128x64
conv/reLU: 3x3x64x64
maxpool
conv/reLU: 3x3x64x32
conv/reLU: 3x3x32x32
maxpool
conv/reLU: 3x3x32x16
conv/reLU: 3x3x16x16
maxpool
conv/reLU: 3x3x16x8
conv/reLU: 3x3x8x8
maxpool
conv/reLU: 3x3x8x4
conv/reLU: 3x3x4x4
maxpool
conv/reLU: 3x3x4x2
conv/reLU: 3x3x2x2
maxpool
conv/reLU: 3x3x2x1
conv/reLU: 3x3x1x1
maxpool
softmax

fully-connected 4096
fully-connected 4096
fully-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?

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

Inception (GoogleLeNet)

ResNet (34 layer version)

Convolution network

Upsampling network

Fully Convolutional Network for image segmentation
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…
Consider SIMD parallelism within a block

\[ \text{BLOCKSIZE}_I \times \text{BLOCKSIZE}_J = \text{BLOCKSIZE}_K \times \text{BLOCKSIZE}_K \]

...for (int j=0; j<\text{BLOCKSIZE}_J; j++) {
    for (int i=0; i<\text{BLOCKSIZE}_I; i+=\text{SIMD}_\text{WIDTH}) {
        \text{simd}_\text{vec} \text{ C}_\text{accum} = \text{vec}\_\text{load}(\&\text{C}[\text{jblock}+j][\text{iblock}+i]);
        for (int k=0; k<\text{BLOCKSIZE}_K; k++) {
            // \text{C} = \text{A*B} + \text{C}
            \text{simd}_\text{vec} \text{ A}_\text{val} = \text{splat}(\&\text{A}[\text{jblock}+j][\text{kblock}+k]); // load a single element in vector register
            \text{simd}\_\text{muladd}\text{A}_\text{val, vec}\_\text{load}(\&\text{B}[\text{kblock}+k][\text{iblock}+i]), \text{C}_\text{accum});
        }
        \text{vec}\_\text{store}(\&\text{C}[\text{jblock}+j][\text{iblock}+i], \text{C}_\text{accum});
    }
}

**Vectorize i loop**

*Good:* also improves spatial locality in access to B

*Bad:* working set increased by \text{SIMD}_\text{WIDTH}, still walking over B in large steps
Blocked dense matrix multiplication (2)

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]);

        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

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>X_{20}</td>
<td>X_{21}</td>
<td>X_{22}</td>
<td>X_{23}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X_{30}</td>
<td>X_{31}</td>
<td>X_{32}</td>
<td>X_{33}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

#### Note: 0-pad matrix

$W \times H$

0(N) storage overhead for filter with N elements

Must construct input data matrix

\[
\begin{bmatrix}
  w_0 \\
  w_1 \\
  \vdots \\
  w_8 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  \begin{array}{cccccc}
    0 & 0 & 0 & x_{00} & x_{01} & x_{10} & x_{11} \\
    0 & 0 & 0 & x_{00} & x_{01} & x_{02} & x_{10} & x_{11} & x_{12} \\
    0 & 0 & 0 & x_{01} & x_{02} & x_{03} & x_{11} & x_{12} & x_{13} \\
    \vdots \\
    x_{00} & x_{01} & x_{02} & x_{10} & x_{11} & x_{12} & x_{20} & x_{21} & x_{22} \\
  \end{array}
\end{bmatrix}
\]
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
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];
```

```c
// 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.
**Convolutional 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, y, z, 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 buffers for layer parameters
Halide::Buffer<float> W(f_w, f_h, in_ch, num_f)
Halide::Buffer<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!
Different layers of a single DNN may benefit from unique scheduling strategies.

![Diagram showing throughput for input-specialized schedules relative to best-on-average schedule.]

[Figure credit: Mullapudi et al. 2016]

Notice sizes of weights and activations in this network: (and consider SIMD widths of modern machines). Ug!

<table>
<thead>
<tr>
<th>Type / Stride</th>
<th>Filter Shape</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv / s1</td>
<td>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</td>
<td>112 x 112 x 32</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 64</td>
<td>112 x 112 x 64</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 64</td>
<td>56 x 56 x 64</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>3 x 3 x 128</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 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>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>
Many efforts to automatically schedule key DNN operations

Open Deep Learning Compiler Stack

TVM is a compiler stack for deep learning systems. It is designed to close the gap between the productivity-focused deep learning frameworks, and the performance- and efficiency-focused hardware backends. TVM works with deep learning frameworks to provide end to end compilation to different backends. Checkout the tvm stack homepage for more information.
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>

Estimates for 45nm process
[Source: Mark Horowitz]
Reducing network footprint

- Early DNN designs: large storage cost for model parameters
  - AlexNet model: ~200 MB
  - VGG-16 model: ~500 MB
  - ResNet-50: 102 MB
  - Inception-v3: 91 MB

- In many modern DNNs, activations (intra-layer intermediate buffers) require more storage than weights
  - So bandwidth is often due to reading/writing intermediate values
Is there an opportunity for compression?
"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)
- 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>
<tr>
<td></td>
<td>0</td>
<td>1.8</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td></td>
<td></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>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 a form of lossy compression

![Diagram of weight sharing and centroid fine-tuning](image)

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

[Figure credit: Han ICLR16]
VGG-16 sparsification

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>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>
</tr>
<tr>
<td>VGG-16 Compressed</td>
<td>31.17%</td>
<td>10.91%</td>
</tr>
</tbody>
</table>

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

Stanford CS348K, Spring 2020
Compressing weights (and activations)

- Many efforts to use low precision values for DNN weights and intermediate activations
- In the extreme case: 1-bit

XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks

Mohammad Rastegari†, Vicente Ordonez†, Joseph Redmon*, Ali Farhadi†*

Allen Institute for AI†, University of Washington*
{mohammadz, vicenteor}@allenai.org
{pjreddie, ali}@cs.washington.edu

Abstract. We propose two efficient approximations to standard convolutional neural networks: Binary-Weight-Networks and XNOR-Networks. In Binary-Weight-Networks, the filters are approximated with binary values resulting in 32× memory saving. In XNOR-Networks, both the filters and the input to convolutional layers are binary. XNOR-Networks approximate convolutions using primarily binary operations. This results in 58× faster convolutional operations (in terms of number of the high precision operations) and 32× memory savings. XNOR-Nets offer the possibility of running state-of-the-art networks on CPUs (rather than GPUs) in real-time. Our binary networks are simple, accurate, efficient, and work on challenging visual tasks. We evaluate our approach on the ImageNet classification task. The classification accuracy with a Binary-Weight-Network version of AlexNet is the same as the full-precision AlexNet. We compare our method with recent network binarization methods, BinaryConnect and BinaryNets, and outperform these methods by large margins on ImageNet, more than 16% in top-1 accuracy. Our code is available at: http://allenai.org/plato/xnornet.
This a great example of non-domain-specific vs. domain-specific approach to innovation
Leveraging domain-knowledge: more efficient topologies (aka better algorithm design)

- Original DNNs for image recognition where over-provisioned
  - Large filters, many filters
- Modern DNNs designs are hand-designed to be sparser

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

Inception v1 (GoogleLeNet) — 27 total layers, 7M parameters

ResNet (34 layer version)
Modular network designs

Inception v4

Input (299x299x3)

Stem

Output: 35x35x384

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

B block

Avg Pooling

Output: 384

1x1 Conv (384)

1x7 Conv (224)

7x1 Conv (224)

1x7 Conv (224)

7x1 Conv (192)

1x1 Conv (192)

Filter concat

Output: 256

1x7 Conv (256)

1x1 Conv (192)

7x1 Conv (256)

1x1 Conv (128)

B block

Filter concat

Output: 64

1x1 Conv (64)

1x1 Conv (64)

3x3 Conv (96)

3x3 Conv (96)

3x3 Conv (96)

A block

Filter concat

Output: 96

1x1 Conv (96)

1x1 Conv (96)

3x3 Conv (96)
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.
The results in Table 1 show that the deeper 34-layer plain network achieves the best result on the 100k test images, reported by the test server.

Our implementation for ImageNet follows the practice in section 3.4. Implementation. When the dimensions increase (dotted line shortcuts in Fig. 11), we use SGD with a mini-batch size of 256. The learning rate is set to 0.1, and it is divided by 10 after every 8 epochs.


Figure 10. The schema for 35 × 35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.
Effect of topology innovation

ImageNet Top 1 Accuracy

Flops cost (area of circle is # params)

Accuracy (points) per flop
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)
Depthwise separable convolution

Main idea: 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

Convolution Layer

- NUM_CHANNELS inputs
- \( K_w \times K_h \times \text{NUM\_CHANNELS} \) weights (for each filter)
- \( K_w \times K_h \times \text{NUM\_CHANNELS} \) work per output pixel (per filter)

Depthwise Separable Conv Layer

- NUM_CHANNELS inputs
- \( K_w \times K_h \) weights (for each channel)
- Results of filtering each of \( \text{NUM\_CHANNELS} \) independently
- \( \text{NUM\_CHANNELS} \) weights (for each filter)
- \( \text{NUM\_CHANNELS} \) work per output pixel (per filter)

Image credit: Eli Bendersky
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>
<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 3 x 64</td>
<td>112 x 112 x 32</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>3 x 1 x 128 x 256</td>
<td>28 x 28 x 128</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>1 x 1 x 256 x 256</td>
<td>28 x 28 x 256</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 3 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>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 / s1</td>
<td>3 x 3 x 512 dw</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>1 x 1 x 512 x 1024</td>
<td>7 x 7 x 512</td>
</tr>
<tr>
<td>Conv / s1</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>

---

Image classification (ImageNet)
Comparison to Common DNNs

<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)
Comparison to Other Compressed DNNs

<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>

[Howard et al. 2017]
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 of executing DNN inference?
  - Number of ops? (often measured in multiply adds)
  - Bandwidth?
    - Loading model weights + loading/storing intermediate activations
  - Careful! Certain layers are bandwidth bound, e.g., batch norm

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

Implication: number of math ops can be a poor predictor of run time of network! (too small to utilize processor, bandwidth bound, etc.)

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

\[
\begin{align*}
\mu_B & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean} \\
\sigma_B^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{// mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{// normalize} \\
y_i & \leftarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\end{align*}
\]
Model optimization techniques

- Manually designing better models
  - Common parameters: depth of network, width of filters, number of filters per layer, convolutional stride, etc.

- Good scheduling of performance-critical operations (layers)
  - Loop blocking/tiling, fusion
  - Typically optimized manually by humans (but significant research efforts to automate scheduling)

- Compressing models
  - Lower bit precision
  - Automatic sparsification/pruning

- Automatically discovering efficient model topologies (architecture search)
DNN architecture search

- Learn an efficient DNN topology along with associated weights
- Example: progressive neural architecture search [Liu et al. 18]

“Block” = (input1, input2, op1, op2)

Eight possible operations:

- 3x3 depthwise-separable conv
- 5x5 depthwise-separable conv
- 7x7 depthwise-separable conv
- 1x7 followed by 7x1 conv
- identity
- 3x3 average pool
- 3x3 max pool
- 3x3 dilated conv
Architecture search space

Cells are DAGs of $B$ blocks

Cells have one output, can receive input from all prior cells

DNNs are sequences of $N$ cells

[Liu et al. 18]
Progressive neural architecture search results

- Automatic search was able to find model architectures that yielded similar/better accuracy to hand designed models (and comparable costs)

<table>
<thead>
<tr>
<th>Model</th>
<th>Params</th>
<th>Multi-Adds</th>
<th>Top-1</th>
<th>Top-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNet-224 [14]</td>
<td>4.2M</td>
<td>569M</td>
<td>70.6</td>
<td>89.5</td>
</tr>
<tr>
<td>ShuffleNet (2x) [37]</td>
<td>5M</td>
<td>524M</td>
<td>70.9</td>
<td>89.8</td>
</tr>
<tr>
<td>NASNet-A ((N = 4, F = 44) [41])</td>
<td>5.3M</td>
<td>564M</td>
<td>74.0</td>
<td>91.6</td>
</tr>
<tr>
<td>AmoebaNet-B ((N = 3, F = 62) [27])</td>
<td>5.3M</td>
<td>555M</td>
<td>74.0</td>
<td>91.5</td>
</tr>
<tr>
<td>AmoebaNet-A ((N = 4, F = 50) [27])</td>
<td>5.1M</td>
<td>555M</td>
<td>74.5</td>
<td>92.0</td>
</tr>
<tr>
<td>AmoebaNet-C ((N = 4, F = 50) [27])</td>
<td>6.4M</td>
<td>570M</td>
<td>75.7</td>
<td>92.4</td>
</tr>
<tr>
<td>PNASNet-5 ((N = 3, F = 54))</td>
<td>5.1M</td>
<td>588M</td>
<td>74.2</td>
<td>91.9</td>
</tr>
</tbody>
</table>

- Forms of architecture search implemented by Cloud-based ML hosting services (user provides training data, service searches for good model)
Summary: efficiently evaluating deep nets

- Workload characteristics:
  - Convlayers: high arithmetic intensity, significant portion of cost when evaluating DNNs for computer vision
  - Similar data access patterns to dense-matrix multiplication (exploiting temporal reuse is key), but direct implementation as matrix-matrix multiplication is sub-optimal

- Significant interest in reducing size of DNNs for more efficiency evaluation

- Algorithmic techniques (better DNN model architectures) are responsible for significant speedups in recent years
  - Expect increasing use of automated model search techniques