Lecture 8:

Efficiently Evaluating Deep Networks

Visual Computing Systems
Stanford CS348K, Fall 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
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
\[f(x) = \frac{1}{1 + e^{-x}}\]
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;
    }
}
```
Recall: what does convolution with these filters do?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Extracts horizontal gradients

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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 x num_filters weights

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

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

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

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

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

Filter responses

ReLU

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

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

Note data reduction as a result of pooling
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: 3x3x1024x512
conv/reLU: 3x3x512x512
maxpool
conv/reLU: 3x3x512x256
conv/reLU: 3x3x256x256
maxpool
conv/reLU: 3x3x256x128
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
soft-max

[VGG illustration credit: Yang et al.]

Stanford CS348K, Fall 2018
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

Residual Network.

VGG-19 34-layer plain which turn the image network into its counterpart residual version. The identity correctly classified if the ground truth is among the top-5, 3x3 conv, 64 output are of the same dimensions (solid line shortcuts in Fig. 7x7 conv, 64, /2 in Fig. 4), we consider two options: (A) The shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no extra lion images for training, 50,000 for validation and 100,000 per image whose size is distributed evenly between 8% and 100% of the image area with aspect ratio constrained to the inter-

Setup and Results

Our implementation for ImageNet follows the practice of Andrew Howard [8] were useful to combat overfitting to val

4

setup 

4

3

60

4

10

\begin{figure}
\centering
\includegraphics[width=\textwidth]{image}
\caption{Example network architectures for ImageNet.}
\end{figure}

A 224 414

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
Layer & Output Size & Description \\
\hline
7x7 conv, 64 & 112x112 & AveragePool 480, 480 \\
3x3 conv, 64 & 56x56 & DepthConcat 256, 256 \\
2x2 conv, 64 & 28x28 & DepthConcat 128, 128 \\
1x1 conv, 128 & 14x14 & DepthConcat 64, 64 \\
1x1 conv, 256 & 7x7 & DepthConcat 64, 64 \\
3x3 conv, 256 & 7x7 & SoftmaxActivation \\
1x1 conv, 512 & 7x7 & SoftmaxActivation \\
3x3 conv, 512 & 7x7 & SoftmaxActivation \\
\hline
\end{tabular}
\caption{Network Architectures for ImageNet}
\end{table}

The ILSVRC 2014 classification challenge involves the training of ImageNet, which contains 1.28 million images for training, 50,000 for validation and 100,000 for testing. To facilitate the training of these large networks, we use 10-crop testing [12] that consists of 1000 classes. The models are trained on the 1.28 million training images, and evaluated on the 50k validation images. We also obtain a final result on the 100k test images, reported by the test server. Also, we found that the photometric distortions, includes sampling of various sized patches of the image. 224x224

\begin{figure}
\centering
\includegraphics[width=\textwidth]{image}
\caption{A schematic view of the resulting network is depicted in Fig. 3. Example network architectures for ImageNet.}
\end{figure}

Inception (GoogleLeNet)
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?
  - 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

```c
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

```
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
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]);

        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

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

$$W \times H$$

$$(3 \times 3 = 9)$$

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & x_{00} & x_{01} & 0 & x_{10} & x_{11} \\
\end{bmatrix}
\]

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

O(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

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

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

\[
\begin{bmatrix}
W_{00} & W_{01} & W_{02} & \cdots & W_{0N} \\
W_{10} & W_{11} & W_{12} & \cdots & W_{1N} \\
W_{20} & W_{21} & W_{22} & \cdots & W_{2N} \\
W_{30} & W_{31} & W_{32} & \cdots & W_{3N} \\
\vdots & \vdots & \vdots & \cdots & \vdots \\
W_{80} & W_{81} & W_{82} & \cdots & W_{8N} \\
\end{bmatrix}
\]
Multiple convolutions on multiple input channels

For each filter, sum responses over input channels

Equivalent to \((3 \times 3 \times \text{num\_channels})\) convolution on \((W \times H \times \text{num\_channels})\) input data.
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

```cpp
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.

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

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

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 / 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 128 dw</td>
<td>112 x 112 x 32</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 / 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 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 / 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 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 / 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 1024</td>
<td>7 x 7 x 512</td>
</tr>
<tr>
<td>Conv dw / 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>
Significant number of efforts to automatically schedule key DNN operations
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 (NlgN)
  - 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: \( w_0 \ w_1 \ w_2 \)

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

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

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

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>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 of early DNN designs
  - AlexNet model: ~200 MB
  - VGG-16 model: ~500 MB
  - ResNet-50: 102 MB
  - Inception-v3: 91 MB
  - In most modern DNNs, activations require more storage than weights)

- 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)
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.

\[ f \left( \sum_i x_i w_i + b \right) \]

\[ f(x) = \max(0, x) \]
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 4 9 ...</th>
<th>0 1.8 0 0 0.5 0 0 0 0 1.1 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1.8 0.5 2.1</td>
<td>0 1.8 0 0 0.5 0 0 0 0 1.1 ...</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 3 5 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1.8 0.5 2.1</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

<table>
<thead>
<tr>
<th>weights (32 bit float)</th>
<th>cluster index (2 bit uint)</th>
<th>centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.09 -0.98 1.48 0.09</td>
<td>3 0 2 1</td>
<td>3: 2.00</td>
</tr>
<tr>
<td>0.05 -0.14 -1.08 2.12</td>
<td>1 1 0 3</td>
<td>2: 1.50</td>
</tr>
<tr>
<td>-0.91 1.92 0 -1.03</td>
<td>0 3 1 0</td>
<td>1: 0.00</td>
</tr>
<tr>
<td>1.87 0 1.53 1.49</td>
<td>3 1 2 2</td>
<td>0: -1.00</td>
</tr>
</tbody>
</table>

Step 3: Huffman encode quantized weights and CSR indices (lossless compression)
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 (P)</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>Network</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 modern networks are dominated by costs of convolutional layers
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 overprovisioned
  - 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)
Inception v4

A block

B block

Stem

Input (299x299x3)

299x299x3

4 x Inception-A

Output: 35x35x384

Output: 35x35x384

Reduction-A

Output: 17x17x1024

7 x Inception-B

Output: 17x17x1024

Reduction-B

Output: 8x8x1536

3 x Inception-C

Output: 8x8x1536

Average 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

- **Input**: (299x299x3)
- **3x3 Conv (32 stride 2 V)**
- **3x3 Conv (32 V)**
- **3x3 Conv (32 stride 2 V)**
- **Input (299x299x3)**

**Filter concat**

- **3x3 Conv (96 V)**
- **3x3 MaxPool (stride 2 V)**
- **3x3 Conv (96 V)**
- **1x1 Conv (64)**
- **1x7 Conv (64)**
- **7x1 Conv (64)**
- **1x1 Conv (64)**

**Filter concat**

- **3x3 Conv (192 V)**
- **MaxPool (stride=2 V)**
- **3x3 Conv (96 V)**
- **1x7 Conv (64)**
- **7x1 Conv (64)**
- **1x1 Conv (64)**

**Filter concat**

- **35x35x384**
- **71x71x192**
- **73x73x160**

**Figure 3. The schema for stem of the pure Inception-v4 and Inception-ResNet-v2 networks. This is the input part of those networks. Cf. Figures 9 and 15**
Figure 10. The schema for $35 \times 35$ grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.
In this section we report our results and comparisons. We analysed the following DDNs: AlexNet, batch normalised AlexNet, BN-AlexNet, GoogLeNet, ResNet-18, BN-ResNet-18, ENet, VGG-16, VGG-19, ResNet-50, ResNet-101, ResNet-152, Inception-v3, Inception-v4, Inception-v3R, 3R, 2R, 3R, 1R, 3R2, 1R2, 3R1, 1R1, 3R3, 1R3, 3R4, 1R4.

The results are shown in the graph below. The x-axis represents the operations (in G-Ops), the y-axis represents the top-1 accuracy. The bars show the accuracy per parameter. The circles represent the flops cost (area of circle is # params).

From the graph, we can see that ENet is the best architecture in terms of parameters space utilisation, squeezing the number of operations in a network model, and it has been adapted and retrained on ImageNet (Culurciello, 2016) for this work — achieves the highest score, showing that ENet (Paszke et al., 2016) is the winner of this section.

Moreover, ENet (Paszke et al., 2016) is the best architecture in terms of parameters, operations count, inference time. On the far right, ResNet-18, BN-NIN, GoogLeNet and ENet (marked by grey arrows) do a better job at efficiency. On the far left, AlexNet, VGG-16, VGG-19, ResNet-50, ResNet-101, ResNet-152, Inception-v3, Inception-v4, Inception-v3R, 3R, 2R, 3R, 1R, 3R2, 1R2, 3R1, 1R1, 3R3, 1R3, 3R4, 1R4 do a better job at accuracy, memory footprint, parameters.

DNNs are known to be highly inefficient in utilising their full learning power (number of parameters / size / flop vs. accuracy). From the graph, we can see that accuracy and inference time are in a hyperbolic relationship: a little increment in accuracy costs a lot of computational time. We show that number of operations in a network model is directly proportional to the number of parameters (size / flop vs. accuracy / size / flop). Further, and obtain an upper bound in accuracy even for an energetic constraint, which could possibly be exploited in practical application.

We show that accuracy and inference time are in a hyperbolic relationship: a little increment in accuracy costs a lot of computational time. We show that number of operations in a network model is directly proportional to the number of parameters (size / flop vs. accuracy / size / flop). Further, and obtain an upper bound in accuracy even for an energetic constraint, which could possibly be exploited in practical application.

Finally, we show that ENet is the best architecture in terms of parameters space utilisation, squeezing the number of operations in a network model, and it has been adapted and retrained on ImageNet (Culurciello, 2016) for this work — achieves the highest score, showing that ENet (Paszke et al., 2016) is the winner of this section.

Moreover, ENet (Paszke et al., 2016) is the best architecture in terms of parameters, operations count, inference time. On the far right, ResNet-18, BN-NIN, GoogLeNet and ENet (marked by grey arrows) do a better job at efficiency. On the far left, AlexNet, VGG-16, VGG-19, ResNet-50, ResNet-101, ResNet-152, Inception-v3, Inception-v4, Inception-v3R, 3R, 2R, 3R, 1R, 3R2, 1R2, 3R1, 1R1, 3R3, 1R3, 3R4, 1R4 do a better job at accuracy, memory footprint, parameters.

DNNs are known to be highly inefficient in utilising their full learning power (number of parameters / size / flop vs. accuracy). From the graph, we can see that accuracy and inference time are in a hyperbolic relationship: a little increment in accuracy costs a lot of computational time. We show that number of operations in a network model is directly proportional to the number of parameters (size / flop vs. accuracy / size / flop). Further, and obtain an upper bound in accuracy even for an energetic constraint, which could possibly be exploited in practical application.

Finally, we show that ENet is the best architecture in terms of parameters space utilisation, squeezing the number of operations in a network model, and it has been adapted and retrained on ImageNet (Culurciello, 2016) for this work — achieves the highest score, showing that ENet (Paszke et al., 2016) is the winner of this section.

Moreover, ENet (Paszke et al., 2016) is the best architecture in terms of parameters, operations count, inference time. On the far right, ResNet-18, BN-NIN, GoogLeNet and ENet (marked by grey arrows) do a better job at efficiency. On the far left, AlexNet, VGG-16, VGG-19, ResNet-50, ResNet-101, ResNet-152, Inception-v3, Inception-v4, Inception-v3R, 3R, 2R, 3R, 1R, 3R2, 1R2, 3R1, 1R1, 3R3, 1R3, 3R4, 1R4 do a better job at accuracy, memory footprint, parameters.

DNNs are known to be highly inefficient in utilising their full learning power (number of parameters / size / flop vs. accuracy). From the graph, we can see that accuracy and inference time are in a hyperbolic relationship: a little increment in accuracy costs a lot of computational time. We show that number of operations in a network model is directly proportional to the number of parameters (size / flop vs. accuracy / size / flop). Further, and obtain an upper bound in accuracy even for an energetic constraint, which could possibly be exploited in practical application.

Finally, we show that ENet is the best architecture in terms of parameters space utilisation, squeezing the number of operations in a network model, and it has been adapted and retrained on ImageNet (Culurciello, 2016) for this work — achieves the highest score, showing that ENet (Paszke et al., 2016) is the winner of this section.

Moreover, ENet (Paszke et al., 2016) is the best architecture in terms of parameters, operations count, inference time. On the far right, ResNet-18, BN-NIN, GoogLeNet and ENet (marked by grey arrows) do a better job at efficiency. On the far left, AlexNet, VGG-16, VGG-19, ResNet-50, ResNet-101, ResNet-152, Inception-v3, Inception-v4, Inception-v3R, 3R, 2R, 3R, 1R, 3R2, 1R2, 3R1, 1R1, 3R3, 1R3, 3R4, 1R4 do a better job at accuracy, memory footprint, parameters.

DNNs are known to be highly inefficient in utilising their full learning power (number of parameters / size / flop vs. accuracy). From the graph, we can see that accuracy and inference time are in a hyperbolic relationship: a little increment in accuracy costs a lot of computational time. We show that number of operations in a network model is directly proportional to the number of parameters (size / flop vs. accuracy / size / flop). Further, and obtain an upper bound in accuracy even for an energetic constraint, which could possibly be exploited in practical application.

Finally, we show that ENet is the best architecture in terms of parameters space utilisation, squeezing the number of operations in a network model, and it has been adapted and retrained on ImageNet (Culurciello, 2016) for this work — achieves the highest score, showing that ENet (Paszke et al., 2016) is the winner of this section.
# 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>5x Conv dw / s1</td>
<td>3 x 3 x 512 dw</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>5x 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>

Image classification (ImageNet)

Comparison to Common DNNs

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million</th>
<th>Million</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</th>
<th>Million</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>Squeezennet</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>
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 evaluating DNN?
  - 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 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

DNNs are sequences of $N$ cells

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

[Stanford CS348K, Fall 2018] [Liu et al. 18]
Progressive neural architecture search

- Parameters:
  - $N =$ number of cells in DNN
  - $B =$ max blocks per cell

- Let $S$ be the set of 1-block cells (8 choices)
- train $N$-cell DNNs using cells given by $s$ in $S$
- keep $k$ best performing cells on test set
- for $i=2$ to $B$:
  - for each $s$ in $S$, add new cells to $S$ by adding additional block to $s$ (many possible ways to add new block)
  - train $N$-cell DNNs for all cells in $S$
  - keep $k$ best performing cells on test set

[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)
Why might a GPU be a good platform for DNN evaluation?
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!)
Why might a GPU be a sub-optimal platform for DNN evaluation?
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)
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 implementation as matrix-matrix multiplication is sub-optimal

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

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

- Model innovation complemented and extended by much ongoing work on efficient mapping of key layers to CPUs/GPUs and on custom hardware for evaluation
  - Specialized hardware is a topic of a coming lecture