Lecture 18:

Efficiently Evaluating DNNs

Parallel Computing
Stanford CS149, Fall 2020
Today

- We will discuss the workload of evaluating deep neural networks (performing “inference”)
  - This lecture will be heavily biased towards concerns of DNNs that process images (to be honest, because that is what your instructor knows best)
  - But, image processing is not the application driving the majority of DNN evaluation in the world right now (its text processing, speech, ads, etc.)
Consider the following expression

\[
\max( \max(0, (a*b) + (c*d)) + (e*f) + (g*h), i*j) 
\]
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
$$f(x) = \frac{1}{1 + e^{-x}}$$
Deep neural network: topology

- Fully connected layer
- Sparsely (locally) connected layer (each unit only received inputs from three input nodes)

Stanford CS149, Fall 2020
Recall image convolution (3x3 conv)

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)
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 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

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

Output: filter responses
\( W \times H \times n_{\text{num\_filters}} \)

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

ReLU

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

Pool (max response in 2x2 region)

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

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

Filter responses

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: 3x3x128x256
- conv/reLU: 3x3x256x256
- conv/reLU: 3x3x256x256
- maxpool
- conv/reLU: 3x3x64x128
- conv/reLU: 3x3x128x128
- maxpool

- conv/reLU: 3x3x128x256
- conv/reLU: 3x3x256x256
- conv/reLU: 3x3x256x256
- maxpool
- conv/reLU: 3x3x64x128
- conv/reLU: 3x3x128x128
- maxpool

- conv/reLU: 3x3x512x512
- conv/reLU: 3x3x512x512
- conv/reLU: 3x3x512x512
- maxpool
- conv/reLU: 3x3x512x512
- conv/reLU: 3x3x512x512
- conv/reLU: 3x3x512x512
- maxpool

- fully-connected 4096
- fully-connected 4096
- fully-connected 1000
- soft-max
Why deep?

Left: what pixels in image patch 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

- **Residual Network.**

- Based on the above plain network, we correctly classified if the ground truth is among the top-5, against the first 5 predicted classes: an image is deemed output are of the same dimensions (solid line shortcuts in Fig. 3).

- The plain nets. The 34-layer plain net is in Fig. 3. Example network architectures for ImageNet. The results in Table 36 show that the deeper 34-layer plain net has higher validation error than the shallower 18-layer plain net. To reveal the reasons, in Fig. 4.

- Right: a residual network with 34 parameter layers (3.6 billion FLOPs) as a reference.

- The dotted shortcuts increase dimensions. The 10-crop testing at multiple scales (images are resized such that the shorter side is in $2144, 1600, 1024$).

- Upsampling network

- Fully Convolutional Network for image segmentation

- Stanford CS149, Fall 2020
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)

... 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<BLOCKSIZE_K; 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<BLOCKSIZE_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<BLOCKSIZE_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>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>
</tr>
<tr>
<td>X_20</td>
<td>X_21</td>
<td>X_22</td>
<td>X_23</td>
<td>...</td>
</tr>
<tr>
<td>X_30</td>
<td>X_31</td>
<td>X_32</td>
<td>X_33</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Note:** 0-pad matrix

**0(N) storage overhead for filter with N elements**

**Must construct input data matrix**

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

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

### Construct Matrix from Elements of Input Image

- \(X_{00}, X_{01}, X_{02}, X_{03}, \ldots\)
- \(X_{10}, X_{11}, X_{12}, X_{13}, \ldots\)
- \(X_{20}, X_{21}, X_{22}, X_{23}, \ldots\)
- \(X_{30}, X_{31}, X_{32}, X_{33}, \ldots\)
- \(\ldots, \ldots, \ldots, \ldots\)

### Note: 0-pad matrix
Multiple convolutions as matrix-matrix mult

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

\[
WxH
\]

\[
\begin{bmatrix}
0 & 0 & 0 & x_{00} & x_{01} & 0 & x_{10} & x_{11} \\
0 & 0 & 0 & x_{00} & x_{01} & x_{02} & x_{10} & x_{11} & x_{12} \\
0 & 0 & 0 & x_{01} & x_{02} & x_{03} & x_{11} & x_{12} & x_{13} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
w_{00} & w_{01} & w_{02} & \cdots & w_{0N} \\
w_{10} & w_{11} & w_{12} & \cdots & w_{1N} \\
w_{20} & w_{21} & w_{22} & \cdots & w_{2N} \\
w_{30} & w_{31} & w_{32} & \cdots & w_{3N} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\end{bmatrix}
\]

num filters
Multiple convolutions on multiple input channels

For each filter, sum responses over input channels

Equivalent to (3 x 3 x num_channels) convolution on (W x H x num_channels) input data
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 input matrix materialization
But must roll your own highly optimized implementation of complicated loop nest.
Convolutional 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");                    // n is minibatch dimension
Var x, y, z, n;

// 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.

![Figure credit: Mullapudi et al. 2016]

Table 1. MobileNet Body Architecture

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

Notice sizes of weights and activations in this network: (and consider SIMD widths of modern machines). Ug!
Many efforts to automatically schedule key DNN operations
Halide autoscheduler produces efficient DNN layer implementations (for CPUs)

Use large database of programs to learn to predict performance of Halide program+schedule. Then search for best performing programs.

Resnet50 throughput in images/sec (higher is better)
16 threads, AVX-512, batch size 1

<table>
<thead>
<tr>
<th>Framework</th>
<th>Images/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch</td>
<td>16.0</td>
</tr>
<tr>
<td>Caffe2</td>
<td>24.2</td>
</tr>
<tr>
<td>TensorFlow 1.13</td>
<td>49.4</td>
</tr>
<tr>
<td>Halide Autosched</td>
<td>62.6</td>
</tr>
<tr>
<td>MXNet</td>
<td>64.5</td>
</tr>
</tbody>
</table>

[Adams et al. 2019]
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 modern DNNs, activations (intra-layer intermediate buffers) require much more storage than weights
  - Consider 28 x 28 x 256 channels x batch size 64 = 52 MB
  - So bandwidth is often due to reading/writing intermediate values to memory
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.

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

\[
f(x) = \text{max}(0, x)
\]
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>
</tbody>
</table>

<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>

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

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

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

![Huffman encoding](image)
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>14.1%</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 unfortunately 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 $32 \times$ 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 $58 \times$ faster convolutional operations (in terms of number of the high precision operations) and $32 \times$ 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
- Dropout (keep 0.8): Output 1536
- Average Pooling: Output 1536
- Softmax: Output 1000

A block

- Filter concat
- 1x1 Conv (96)
- Avg Pooling
- 1x1 Conv (96)
- 3x3 Conv (96)

B block

- Filter concat
- 1x7 Conv (224)
- 1x1 Conv (192)
- 1x7 Conv (224)
- 1x7 Conv (192)
- 1x1 Conv (256)
- 7x1 Conv (256)
- 1x7 Conv (224)
- 1x7 Conv (224)
- 7x1 Conv (256)
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 section we report our results and comparisons. We analysed the following DDNs: AlexNet, BN-AlexNet, ENet, GoogLeNet, ResNet-18, BN-ResNet-18, VGG-16, VGG-19, ResNet-34, ResNet-50, ResNet-101, ResNet-152, Inception-v3, Inception-v4. Moreover, ENet (Paszke et al., 2016) — which we have specifically designed to be highly efficient and it has been adapted and retrained on ImageNet (Culurciello, 2016) for this work — achieves the best accuracy and the best accuracy per parameter of 55.6% and 4.94%, respectively, revealing that VGG-16 performs better than GoogLeNet. When comparing accuracy (Top-1) versus parameters, we notice that Inception-v3 has a high accuracy but consumes twice as many parameters as AlexNet. However, it's worth noticing that, using more efficient architectures to begin with may produce even more compact file size up to 10\(^{\text{th}}\) degrees of freedom. Prominent work (Han et al., 2015) exploits this flaw to reduce network complexity, spanning from 10\(^{\text{th}}\) to 50\(^{\text{th}}\) parameters are sufficient to provide state-of-the-art results.

Accuracy and error rate always sum to 100\%.

From a given image multiple patches are extracted: four corners plus central crop and their horizontal axis, and the grey dots highlight the centre of the blobs.

Figure 1: Accuracy and error rate always sum to 100\%.

Figure 2: 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)
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 |
|-----------------|-----------------|-----------------|
| Type / Stride | Filter Shape | Input Size |
| Conv / s2 | 3 x 3 x 3 x 32 | 224 x 224 x 3 |
| Conv dw / s1 | 3 x 3 x 32 dw | 112 x 112 x 32 |
| Conv / s1 | 1 x 1 x 32 x 64 | 112 x 112 x 32 |
| Conv dw / s2 | 3 x 3 x 64 dw | 112 x 112 x 64 |
| Conv / s1 | 1 x 1 x 64 x 128 | 56 x 56 x 64 |
| Conv dw / s1 | 3 x 3 x 128 dw | 56 x 56 x 128 |
| Conv / s1 | 1 x 1 x 128 x 128 | 56 x 56 x 128 |
| Conv dw / s2 | 3 x 3 x 128 dw | 56 x 56 x 128 |
| Conv / s1 | 1 x 1 x 128 x 256 | 28 x 28 x 128 |
| Conv dw / s1 | 3 x 3 x 256 dw | 28 x 28 x 256 |
| Conv / s1 | 1 x 1 x 256 x 256 | 28 x 28 x 256 |
| Conv dw / s2 | 3 x 3 x 256 dw | 28 x 28 x 256 |
| Conv / s1 | 1 x 1 x 256 x 512 | 14 x 14 x 256 |
| 5 x Conv / s1 | 3 x 3 x 512 dw | 14 x 14 x 512 |
| Conv / s1 | 1 x 1 x 512 x 512 | 14 x 14 x 512 |
| Conv dw / s2 | 3 x 3 x 512 dw | 14 x 14 x 512 |
| Conv / s1 | 1 x 1 x 512 x 1024 | 7 x 7 x 512 |
| Conv dw / s2 | 3 x 3 x 1024 dw | 7 x 7 x 1024 |
| Conv / s1 | 1 x 1 x 1024 x 1024 | 7 x 7 x 1024 |
| Avg Pool / s1 | Pool 7 x 7 | 7 x 7 x 1024 |
| FC / s1 | 1024 x 1000 | 1 x 1 x 1024 |
| Softmax / s1 | Classifier | 1 x 1 x 1000 |

Image classification (ImageNet)
Comparison to Common DNNs

<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>

Image classification (ImageNet)
Comparison to Other Compressed DNNs

<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>0.50 MobileNet-160</td>
<td>60.2%</td>
<td>76</td>
<td>1.32</td>
<td></td>
</tr>
<tr>
<td>SqueezeNet</td>
<td>57.5%</td>
<td>1700</td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td>AlexNet</td>
<td>57.2%</td>
<td>720</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

Stanford CS149, Fall 2020
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)
Why might a GPU be a good platform for DNN evaluation?

consider: arithmetic intensity, SIMD, data-parallelism, memory bandwidth requirements
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)
Why might a GPU be a sub-optimal platform for DNN evaluation?

consider: is a general purpose processor needed?
Hardware acceleration for DNNs

- Google TPU
- MIT Eyeriss
- Huawei Kirin NPU
- SambaNova Systems
- Apple Neural Engine
- Intel Lake Crest
- Deep Learning Accelerator
- Volta GPU with Tensor Cores

Slide credit: Xuan Yang
Recall: properties of GPUs

- "Compute rich": packed densely with processing elements
  - Good for compute-bound applications

- Good, because dense-matrix multiplication and DNN convolutional layers (when implemented properly) are compute bound

- But recall cost of instruction stream processing and control in a programmable processor:

Note: these figures are estimates for a CPU:

Efficient Embedded Computing [Daily et al. 08]
[Figure credit Eric Chung]
One solution: more complex instructions

- Fused multiply add \((ax + b)\)
- 4-component dot product \(x = A \cdot B\)
- 4x4 matrix multiply
  - \(AB + C\) for 4x4 matrices \(A, B, C\)

- Key principle: amortize cost of instruction stream processing across many operations of a single complex instruction
Volta GPU

Each SM core has:
- 64 fp32 ALUs (mul-add)
- 32 fp64 ALUs
- 8 “tensor cores”

Execute 4x4 matrix mul-add instr
A x B + C for 4x4 matrices A,B,C
A, B stored as fp16, accumulation with fp32 C

There are 80 SM cores in the GV100 GPU:
- 5,120 fp32 mul-add ALUs
- 640 tensor cores
- 6 MB of L2 cache
- 1.5 GHz max clock

= 15.7 TFLOPs fp32
= 125 TFLOPs (fp16/32 mixed) in tensor cores
Efficiency estimates *

- Estimated overhead of programmability (instruction stream, control, etc.)
  - Half-precision FMA (fused multiply-add) 2000%
  - Half-precision DP4 (vec4 dot product) 500%
  - Half-precision MMA (matrix-matrix multiply + accumulate) 27%

NVIDIA Xavier (SoC for automotive domain)

Features a Computer Vision Accelerator (CVA), a custom module for deep learning acceleration (large matrix multiply unit)

But only 2x more efficient than Volta MMA instruction despite being highly specialized component. (includes optimization of gating multipliers if either operand is zero)

* Estimates by Bill Dally using academic numbers, SysML talk, Feb 2018
Summary: efficiently evaluating deep nets

- Workload characteristics for image processing DNNs:
  - Convlayers: high arithmetic intensity, significant portion of cost when evaluating DNNs for computer vision

- 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

- Huge innovation in specialized hardware accelerators