Lecture 19:
Efficiently Evaluating DNNs

Parallel Computing
Stanford CS149, Fall 2019
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)
  - Which admittedly is not the majority of DNN evaluation in the world right now (text processing, speech, ads, etc.)
Consider the following expression

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

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

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

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

Basic computational interpretation:
It is just a circuit!

Biological inspiration:
unit output corresponds loosely to activation of neuron

Machine learning interpretation:
binary classifier: interpret output as the probability of one class
\[
f(x) = \frac{1}{1 + e^{-x}}
\]
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 \( i_i \) loop)
Strided 3x3 convolution

```c
int WIDTH = 1024;
int HEIGHT = 1024;
int STRIDE = 2;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[(WIDTH/STRIDE) * (HEIGHT/STRIDE)];

float weights[] = {1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9};

for (int j=0; j<HEIGHT; j+=STRIDE) {
    for (int i=0; i<WIDTH; i+=STRIDE) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            for (int ii=0; ii<3; ii++) {
                tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
            }
        output[(j/STRIDE)*WIDTH + (i/STRIDE)] = tmp;
    }
}
```
What does convolution using these filter weights do? 

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

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

\[
\begin{bmatrix}
0.075 & 0.124 & 0.075 \\
0.124 & 0.204 & 0.124 \\
0.075 & 0.124 & 0.075 \\
\end{bmatrix}
\]

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

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

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 
\[ 3x3 \times \text{num\_filters weights} \]

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

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

Filter response maps (num\_filters of them)
Applying many filters to an image at once

Input RGB image (W x H x 3)

96 11x11x3 filters
(operate on RGB)

96 responses (normalized)
Adding additional layers

Input: image (single channel)
\( W \times H \)

3x3 spatial convolutions
\( 3x3 \times \text{num\_filters} \text{ weights} \)

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

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

Filter responses

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

ReLU

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

(max response in 2x2 region)

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

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
- maxpool
- 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
- 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?

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

Fig. 3. Example network architectures for ImageNet.

- **Left**: a standard convolutional network with 40 million parameters (19.6 billion FLOPs) as a reference.
- **Mid-**: a fully convolutional network for image segmentation.
- **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs).
Deep networks learn useful representations

- 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)**
float A[M][K];
float B[K][N];
float C[M][N];

// compute C += A * B
#pragma omp parallel for
for (int jblock=0; jblock<M; jblock+=BLOCKSIZE_J)
    for (int iblock=0; iblock<N; iblock+=BLOCKSIZE_I)
        for (int kblock=0; kblock<K; kblock+=BLOCKSIZE_K)
            for (int j=0; j<BLOCKSIZE_J; j++)
                for (int i=0; i<BLOCKSIZE_I; i++)
                    for (int k=0; k<BLOCKSIZE_K; k++)
                        C[jblock+j][iblock+i] += A[jblock+j][kblock+k] * B[kblock+k][iblock+i];

Idea: compute partial result for block of C while required blocks of A and B remain in cache
(Assumes BLOCKSIZE chosen to allow block of A, B, and C to remain resident)

Self check: do you want as big a BLOCKSIZE as possible? Why?
Hierarchical blocked matrix mult

Exploit multiple levels of memory hierarchy

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

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

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

Consider SIMD parallelism within a block

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

Vectorize i loop

Good: also improves spatial locality in access to B
Bad: working set increased by SIMD_WIDTH, still walking over B in large steps
Blocked dense matrix multiplication (2)

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

... 
for (int j=0; j<\text{BLOCKSIZE}_J; j++)
   for (int i=0; i<\text{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<\text{BLOCKSIZE}_K; k+=SIMD\_WIDTH) {
         C\_scalar += simd\_dot(vec\_load(A[jblock+j][kblock+k]), vec\_load(B\_trans[iblock+i][kblock+k]));
      }
      C[jblock+j][iblock+i] = C\_scalar;
   }

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

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

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

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

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

Construct matrix from elements of input image

Note: 0-pad matrix

<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>

$3 \times 3 = 9$

$O(N)$ storage overhead 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

Construct matrix from elements of input image

Note: 0-pad matrix

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

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

\[
\begin{array}{cccc}
0 & 0 & 0 & x00 \\
0 & 0 & 0 & x01 \\
0 & 0 & 0 & x02 \\
0 & 0 & 0 & x03 \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & x12 \\
0 & 0 & 0 & x13 \\
0 & 0 & 0 & x14 \\
0 & 0 & 0 & x15 \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & x22 \\
0 & 0 & 0 & x23 \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & x32 \\
0 & 0 & 0 & x33 \\
\vdots & \vdots & \vdots & \vdots \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 0 & x10 \\
0 & 0 & 0 & x11 \\
0 & 0 & 0 & x12 \\
0 & 0 & 0 & x13 \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & x20 \\
0 & 0 & 0 & x21 \\
0 & 0 & 0 & x22 \\
0 & 0 & 0 & x23 \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & x30 \\
0 & 0 & 0 & x31 \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & x32 \\
0 & 0 & 0 & x33 \\
\vdots & \vdots & \vdots & \vdots \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots \\
0 & 0 & 0 & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
\end{array}
\]
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.
```
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.

![Throughput: Input-Specialized Schedules (relative to best-on-average schedule)](image)

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 dw</td>
<td>112 x 112 x 32</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 / s2</td>
<td>3 x 3 x 64</td>
<td>112 x 112 x 64</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 128 x 128</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>1 x 1 x 128 x 128</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv / s2</td>
<td>3 x 3 x 128 dw</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>1 x 1 x 128 x 256</td>
<td>28 x 28 x 128</td>
</tr>
<tr>
<td>Conv / s2</td>
<td>3 x 3 x 256 dw</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>1 x 1 x 256 x 512</td>
<td>14 x 14 x 256</td>
</tr>
<tr>
<td>Conv / s2</td>
<td>3 x 3 x 512 dw</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>

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

TVM: 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.
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 many modern DNNs, activations (intra-layer intermediate buffers) require more storage than weights
  - So bandwidth is often due to reading/writing intermediates
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

Indices   1   4   9   ...
Value     1.8  0.5  2.1

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

Indices   1   3   5   ...
Value     1.8  0.5  2.1
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

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</th>
<th>Weights% (P)</th>
<th>Weights% (P+Q)</th>
<th>Weights% (P+Q+H)</th>
<th>Index bits (P)</th>
<th>Index bits (P+Q)</th>
<th>Index bits (P+Q+H)</th>
<th>Compress rate (P)</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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>138M</strong></td>
<td><strong>7.5%(13×)</strong></td>
<td><strong>6.4</strong></td>
<td><strong>4.1</strong></td>
<td><strong>5</strong></td>
<td><strong>3.1</strong></td>
<td><strong>3.2% (31×)</strong></td>
<td><strong>2.05% (49×)</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P = connection pruning (prune low weight connections)
Q = quantize surviving weights (using shared weights)
H = Huffman encode

**ImageNet Image Classification Performance**

<table>
<thead>
<tr>
<th></th>
<th>Top-1 Error</th>
<th>Top-5 Error</th>
<th>Model size</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-16 Ref</td>
<td>31.50%</td>
<td>11.32%</td>
<td>552 MB</td>
</tr>
<tr>
<td>VGG-16 Compressed</td>
<td>31.17%</td>
<td>10.91%</td>
<td>11.3 MB</td>
</tr>
</tbody>
</table>

* Benefits of automatic pruning apply mainly to fully connected layers, but 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
Modular network designs

Inception v4

Stem
Input (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

A block

Filter concat
3x3 Conv (96)
3x3 Conv (96)
3x3 Conv (96)

1x1 Conv (96)
1x1 Conv (64)
1x1 Conv (64)

Avg Pooling

B block

Filter concat
7x1 Conv (256)
7x1 Conv (224)
7x1 Conv (224)
7x1 Conv (192)
1x1 Conv (192)
1x1 Conv (192)

1x1 Conv (128)
1x7 Conv (256)
1x7 Conv (224)
1x7 Conv (224)

1x1 Conv (384)
1x1 Conv (224)
1x1 Conv (224)

1x7 Conv (224)
1x1 Conv (192)
1x1 Conv (192)

1x1 Conv (224)
1x1 Conv (192)
1x1 Conv (192)

1x1 Conv (256)
1x7 Conv (224)
1x7 Conv (224)

1x1 Conv (192)
1x1 Conv (192)
1x1 Conv (192)

1x1 Conv (64)
1x1 Conv (64)
1x1 Conv (64)

Softmax
Output: 1000

Dropout (keep 0.8)
Output: 1536

Avarage Pooling
Output: 1536

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.

Figure 10. The schema for 35x35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

Figure 11. The schema for 17x17 grid (Inception-ResNet-B) module of Inception-ResNet-v1 network.

Figure 12. "Reduction-B" 17x17 to 8x8 grid-reduction module. This module used by the smaller Inception-ResNet-v1 network in Figure 15.

Figure 13. The schema for 35x35 grid modules of the pure Inception-v4 network. This is the Inception-A block of Figure 9.

Figure 14. The schema for 17x17 grid modules of the pure Inception-v4 network. This is the Inception-B block of Figure 9.

Figure 15. The schema for 8x8 grid modules of the pure Inception-v4 network. This is the Inception-C block of Figure 9.

Figure 16. The schema for 35x35 to 17x17 reduction module. Different variants of this blocks (with various number of filters) are used in Figure 9, and 15 in each of the new Inception(-v4, -ResNet-v1, -ResNet-v2) variants presented in this paper. The k, l, m, n numbers represent filter bank sizes which can be looked up in Table 1.
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.
ResNet

Figure 3. Example network architectures for ImageNet.

VGG-19 model

34-layer plain

34-layer residual

Table 1 shows: the

procedure. We have observed the degradation problem - the

plain net. To reveal the reasons, in Fig.

net has higher validation error than the shallower 18-layer

tailed architectures.

The 34-layer plain net is in Fig.

We also obtain a final

10-crop testing

do not use dropout

A 224 crop is randomly sampled from an image or its

detailed structure of the various components.

detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the

Figure 9. The overall schema of the Inception-v4 network. For the

Figure 10. The schema for 35 × 35 grid (Inception-ResNet-A)

module of Inception-ResNet-v1 network.

Figure 11. The schema for

Figure 12. "Reduction-B"

module of Inception-ResNet-v1 network.

Figure 13. "Reduction-A"

module of Inception-ResNet-v1 network.

Figure 14. "Reduction-C"

module of Inception-ResNet-v1 network.
In this section we report our results and comparisons. We analysed the following DDNs: AlexNet, GoogLeNet, ResNet-50, ResNet-101, ResNet-152, Inception-v3, Inception-v4, and VGG-16. We use a JetPack-2.3 NVIDIA Jetson TX1 board for inference time and memory usage measurements. For measuring the power consumption, a Keysight 1146B Hall effect current probe has been used with a Keysight MSO-X 2024A digital oscilloscope.

Our goal is to provide insights into the design choices that can lead to efficient neural networks for practical application, and optimisation of the often-limited resources in actual deployments, which lead us to the creation of ENet — or Efficient-Network — for ImageNet.

DNNs are known to be highly inefficient in utilising their full learning power (number of parameters through hyperbolic dependency on the amount of computations that a network requires. Therefore, VGG-16 will perform worse than GoogLeNet, using a single central-crop. For this reason, top-1 accuracy vs. operations, size, and power consumption. Our goal is to provide insights into the design choices that can lead to efficient architectures that can be run on an embedded system.

 accuracy and error rate always sum to 100%. From a given image multiple patches are extracted: four corners plus central crop and their horizontal and vertical flips. For inference time and memory usage measurements we have used Torch7.

Accuracy per parameter

In the original paper this network is called VGG-D, which is the best performing network. Here we prefer to call it VGG-16, to highlight the number of layer utilised, so we will call it VGG-16 in this publication.

Inception-v3 (Szegedy et al., 2015) challenge, in terms of accuracy, memory footprint, parameters, operations count, inference time and size up to 60%. ENet (Paszke et al., 2016) for ImageNet (Culurciello, 2016), — which we have specifically designed to be highly efficient and “squeezing” all their neurons to learn the given task, and are the winners of this section.

\[ \text{Accuracy per parameter} \]

\[ \text{Flops cost (area of circle is \# params)} \]

\[ \text{Accuracy (points) per flop} \]

\[ \text{Top-1 accuracy per parameter} \]

\[ \text{Operations [G-Ops]} \]
## Improving accuracy/cost (image classification)

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

<table>
<thead>
<tr>
<th></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
- $K_w \times K_h \times NUM\_CHANNELS$ weights (for each filter)
- $K_w \times K_h \times NUM\_CHANNELS$ 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 NUM_CHANNELS independently
- NUM_CHANNELS weights (for each filter)
- 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 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 128 dw</td>
<td>112 x 112 x 128</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 1 x 128 x 256</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 512</td>
</tr>
<tr>
<td>5 x Conv dw / s1</td>
<td>3 x 3 x 512 dw</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 512 x 512</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 1024 dw</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 1024 x 1024</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>Avg Pool / s1</td>
<td>Pool 7 x 7</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>FC / s1</td>
<td>1024 x 1000</td>
<td>1 x 1 x 1024</td>
</tr>
<tr>
<td>Softmax / s1</td>
<td>Classifier</td>
<td>1 x 1 x 1000</td>
</tr>
</tbody>
</table>

Table 9. Smaller MobileNet Comparison to Popular Models

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>

Table 11. Performance of PlaNet using the MobileNet architecture

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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>

Image classification (ImageNet)
Comparison to Common DNNs

Image classification (ImageNet)
Comparison to Other Compressed DNNs
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)

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

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

\[
\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{// mini-batch variance}
\]

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{// normalize}
\]

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

Implication: number of ops can be a poor predictor of run time of network (too small to utilize processor, bandwidth bound, etc.)!
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 CS149, Fall 2019] [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)
Hardware acceleration for DNNs

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

Slide credit: Xuan Yang
And many more...

<table>
<thead>
<tr>
<th>Category</th>
<th>Companies</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC Giants</td>
<td>Intel, Qualcomm, Nvidia, Samsung, AMD, Apple, Xilinx, IBM, STMicroelectronics, NXP, MediaTek, HiSilicon</td>
<td>12</td>
</tr>
<tr>
<td>Cloud/HPC</td>
<td>Google, Amazon.AWS, Microsoft, Aliyun, Tencent Cloud, Baidu, Baidu Cloud, HUAWEI Cloud, Fujitsu</td>
<td>9</td>
</tr>
<tr>
<td>IP Vendors</td>
<td>ARM, Synopsys, Imagination, CEVA, Cadence, VeriSilicon</td>
<td>6</td>
</tr>
<tr>
<td>Startups in China</td>
<td>Cambricon, Horizon Robotics, DeePhi, Bitmain, Chipintelli, Thinkforce</td>
<td>6</td>
</tr>
</tbody>
</table>

Image credit: Shan Tang [https://medium.com/@shan.tang.g/a-list-of-chip-ip-for-deep-learning-48d05f1759ae]
Modern NVIDIA GPU
(Volta)
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 \(x = A \text{ dot } B\)
- 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
\[ A \times B + C \text{ for 4x4 matrices } A,B,C \]
A, B stored as fp16, accumulation with fp32 C

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

\[ = 15.7 \text{ TFLOPs fp32} \]
\[ = 125 \text{ TFLOPs (fp16/32 mixed) in tensor cores} \]
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
Google TPU (version 1)
Google’s TPU

Figure credit: Jouppi et al. 2017
TPU area proportionality

Local Unified Buffer for activations (96Kx256x8b = 24 MiB) 29% of chip

Matrix Multiply Unit (256x256x8b = 64K MAC) 24%

Host Interf. 2%

Accumulators (4Kx256x32b = 4 MiB) 6%

Control 2%

Activation Pipeline 6%

PCle Interface 3%

Misc. I/O 1%

DRAM port ddr3 3%

Compute ~ 30% of chip
Note low area footprint of control

Key instructions:
read host memory
write host memory
read weights
matrix_multiply / convolve activate

Figure credit: Jouppi et al. 2017

Stanford CS149, Fall 2019
Systolic array

(matrix vector multiplication example: $y=Wx$)
Systolic array

(matrix vector multiplication example: \( y = Wx \))
Systolic array

(matrix vector multiplication example: $y = Wx$)

Weights FIFO

Accumulators (32-bit)
Systolic array

(matrix vector multiplication example: $y = Wx$)
Systolic array

(matrix vector multiplication example: $y = Wx$)
Systolic array

(matrix vector multiplication example: $y = Wx$)

Weights FIFO

PE

PE

PE

PE

+ PE

+ PE

+ PE

+ PE

Accumulators (32-bit)
Systolic array

(matrix vector multiplication example: $y = Wx$)

Notice: need multiple 4x32bit accumulators to hold output columns
Building larger matrix-matrix multiplies

Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$

Assume 4096 accumulators
Building larger matrix-matrix multiplies

Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$

Assume 4096 accumulators
Building larger matrix-matrix multiplies

Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$

Assume 4096 accumulators
Building larger matrix-matrix multiplies

Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$

Assume 4096 accumulators
TPU Performance/Watt

GM = geometric mean over all apps
WM = weighted mean over all apps

Figure credit: Jouppi et al. 2017
Course Wrap Up

(Students)
For the foreseeable future, the primary way to obtain higher performance computing hardware is through a combination of increased parallelism and hardware specialization.

Intel Xeon Phi
72 cores, 16-wide SIMD, 4-way multi-threading

NVIDIA Maxwell GPU
(single SMM core)
32 wide SIMD
2048 CUDA/core threads per SMM

FPGA
(reconfigurable logic)

Apple A9
Heterogeneous SoC
multi-core CPU + multi-core GPU + media ASICs

Integrated Gen9 GPU
graphics + media

Cyclone IV

Figure 3: GM204 SM Diagram (GM204 also features 4 DP units per SMM, which are not depicted on this diagram)
Today’s software is surprisingly inefficient compared to the capability of modern machines

A lot of performance is currently left on the table (increasingly so as machines get more complex, and parallel processing capability grows)

Extracting this performance stands to provide a notable impact on many compute-intensive fields (or, more importantly enable new applications of computing!)

Given current software programming systems and tools, understanding how a parallel machine works is important to achieving high performance.

A major challenge going forward is making it simpler for programmers to extract performance on these complex machines.
This is very important given how exciting (and efficiency-critical) the next generation of computing applications are likely to be.
Key issues we have addressed in this course

Identifying parallelism
(or conversely, identifying dependencies)

Efficiently scheduling parallelism

1. Achieving good workload balance
2. Overcoming communication constraints:
   Bandwidth limits, dealing with latency, synchronization
   Exploiting data/computation locality = efficiently managing state!
3. Scheduling under heterogeneity (using the right processor for the job)

We discussed these issues at many scales and in many contexts

- Heterogeneous mobile SoC
- Single chip, multi-core CPU
  - Multi-core GPU
- CPU+GPU connected via bus
- Clusters of machines
- Large scale, multi-node supercomputers
Key issues we have addressed in this course

Abstractions for thinking about efficient code
- Data parallel thinking
- Functional parallelism
- Transactions
- Tasks

How throughput-oriented hardware works
- Multiple cores, hardware-threads, SIMD
- Specialization
After taking this course, you are ready to try undergraduate research in parallel computing!
Why research (or independent study)?

- You will learn **way more** about a topic than in any class.

- You think your undergrad friends are very smart? Come hang out with Stanford Ph.D. students! (you get to work side-by-side with them and with faculty). Imagine what level you might rise to.

- It’s way more fun to be on the cutting edge. Industry might not even know about what you are working on. (imagine how much more valuable you are if you can teach them)

- It widens your mind as to what is possible.
Example: what my own Ph.D. students are working on these days...

- Generating efficient code from image processing or deep learning DSLs, and compiling these applications directly to FGPAs

- Designing a new shading language for future real-time 3D graphics pipelines (collaboration with NVIDIA)

- Parallel computing platforms that make it simpler and more efficient to analyzing large video collections (Scanner project: “Spark for video”)

- Designing programming models for querying video collections (e.g., find frames with “three people around a table” or where DNN1 disagrees with DNN2)

- Designing more efficient DNNs to accelerate image processing on video

- Parallel rendering using 1000’s of CPU cores in the cloud

- Analyzing 230,000 hours of news video for biases in representation.
Thanks for being a great class!

Good luck on your finals!

p.s. See you in a week!