Lecture 6:

Efficiently Evaluating Deep Networks

Visual Computing Systems
Stanford CS348K, Spring 2021
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.)
Recall: gradient detection filters

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

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

Note: you can think of a filter as a “detector” of a pattern, and the magnitude of a pixel in the output image as the “response” of the filter to the region surrounding each pixel in the input image.
Applying many filters to an image at once

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

3x3 spatial convolutions on image
3x3 x num\_filters weights

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

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
(3D because they operate on RGB)

96 responses (normalized)
Adding additional layers

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

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

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

post ReLU $W \times H \times \text{num\_filters}$

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

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

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

Sequences of conv + reLU + pool (optional) layers

Example: AlexNet [Krizhevsky12]: 5 convolutional layers + 3 fully connected layers

Another example: VGG-16 [Simonyan15]: 13 convolutional layers

input: 224 x 224 RGB
conv/reLU: 3x3x3x64
conv/reLU: 3x3x64x64
maxpool
conv/reLU: 3x3x64x128
conv/reLU: 3x3x128x128
maxpool
conv/reLU: 3x3x128x256
conv/reLU: 3x3x256x256
maxpool
conv/reLU: 3x3x256x512
conv/reLU: 3x3x512x512
maxpool
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
maxpool
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
maxpool

fully-connected 4096
fully-connected 4096
softmax
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]
More recent image understanding networks

Inception (GoogleLeNet)

ResNet (34 layer version)

Convolution network

Upsampling network

Fully Convolutional Network for image segmentation

Based on the above plain network, we insert shortcut connections (Fig. 1). The image is resized with its shorter side run-length encoded to 224 crop is randomly sampled from an image or its standard color augmentation in [4]. We adopt batch normalization [16] and average the scores for ranking purposes. The challenge uses the top-5 per image area with aspect ratio constrained to the intermediate sizes, they are performed with a stride of 2. The Val set [4] contains [256, 480]. We initialize the weights as in [4] and train all plain/residual nets from scratch. We [21] show that the deeper 34-layer plain performs identity mapping, with extra zero entries padded to match dimensions (done by 1x1+1(S)). The val set [4] is used. We adopt batch normalization (left) we compare their training/validation errors during the training process. The ResNet (34 layer version) is a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions.
Efficiently implementing convolution layers
Dense matrix multiplication

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

// compute C += A * B
#pragma omp parallel for
for (int 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

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

\[ \begin{pmatrix} C \end{pmatrix} = \begin{pmatrix} A \end{pmatrix} \times \begin{pmatrix} B \end{pmatrix} \]

... 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
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++) { // innermost loop items not dependent
            simd_vec bvec = vec_load(&B[kblock+k][iblock+i]);
            for (int kk=0; kk<SIMD_WIDTH; kk++)
                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]);
    }
}
3x3 convolution as matrix-vector product

Construct matrix from elements of input image

Note: 0-pad matrix

$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} \ ...
...
...
...
...

$0 \ 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}
...
...
...

WxH

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

\[ \begin{bmatrix}
X_{00} & X_{01} & X_{02} & X_{03} & \cdots \\
X_{10} & X_{11} & X_{12} & X_{13} & \cdots \\
X_{20} & X_{21} & X_{22} & X_{23} & \cdots \\
X_{30} & X_{31} & X_{32} & X_{33} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \ddots \\
\end{bmatrix} \times \begin{bmatrix}
W_{00} & W_{01} & W_{02} & \cdots & W_{0N} \\
W_{10} & W_{11} & W_{12} & \cdots & W_{1N} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
W_{80} & W_{81} & W_{82} & \cdots & W_{8N} \\
\end{bmatrix} \]
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 (batched)

```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

```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;              // z is num input channels, n is batch 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 of size f_w x f_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!
(p.s. You don’t have to consider, you will!)
Algorithmic improvements

- Direct convolution can be implemented efficiently in Fourier domain (convolution $\rightarrow$ element-wise multiplication)
  - Overhead: FFT to transform inputs into Fourier domain, inverse FFT to get responses back to spatial domain ($N \log N$)
  - Inverse transform amortized over all input channels (due to summation over inputs)

- Direct convolution using work-efficient Winograd convolutions
  1D example: consider producing two outputs of a 3-tap 1D convolution with weights: $w_0 \ w_1 \ w_2$

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

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

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

Direct convolution: 6 multiplies, 4 adds
In 2D can notably reduce multiplications
(3x3 filter: 2.25x fewer multiplies for 2x2 block of output)
Example: CUDNN convolution

```
cudnnStatus_t cudnnConvolutionForward(
    cudnnHandle_t handle,
    const void *alpha,
    const cudnnTensorDescriptor_t xDesc,
    const void *x,
    const cudnnFilterDescriptor_t wDesc,
    const void *w,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionFwdAlgo_t algo,
    void *workspace,
    size_t workspaceSizeInBytes,
    const void *beta,
    const cudnnTensorDescriptor_t yDesc,
    void *y)
```

**Possible algorithms:**

- **CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM**
  This algorithm expresses the convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data.

- **CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM**
  This algorithm expresses convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data, but still needs some memory workspace to precompute some indices in order to facilitate the implicit construction of the matrix that holds the input tensor data.

- **CUDNN_CONVOLUTION_FWD_ALGO_GEMM**
  This algorithm expresses the convolution as an explicit matrix product. A significant memory workspace is needed to store the matrix that holds the input tensor data.

- **CUDNN_CONVOLUTION_FWD_ALGO_DIRECT**
  This algorithm expresses the convolution as a direct convolution (for example, without implicitly or explicitly doing a matrix multiplication).

- **CUDNN_CONVOLUTION_FWD_ALGO_FFT**
  This algorithm uses the Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is needed to store intermediate results.

- **CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING**
  This algorithm uses the Fast-Fourier Transform approach but splits the inputs into tiles. A significant memory workspace is needed to store intermediate results but less than CUDNN_CONVOLUTION_FWD_ALGO_FFT for large size images.

- **CUDNN_CONVOLUTION_FWD_ALGO_WINograd**
  This algorithm uses the Winograd Transform approach to compute the convolution. A reasonably sized workspace is needed to store intermediate results.

- **CUDNN_CONVOLUTION_FWD_ALGO_WINograd_NONFUSED**
  This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results.
Revall: NVIDIA V100 GPU (80 SMs)

- L2 Cache (6 MB)
- GPU memory (HBM) (16 GB)
- 900 GB/sec (4096 bit interface)
Higher performance with “more work”

N=1, P=Q=64 case:
64 x 64 x 128 x 1 = 524K outputs = 2 MB of output data (float32)

N=32, P=Q=256 case:
256 x 256 x 128 x 32 = 256M outputs = 1 GB of output data (float32)
NCHW data layout

- \( N \) is the batch size; 1.
- \( C \) is the number of feature maps (i.e., number of channels); 64.
- \( H \) is the image height; 5.
- \( W \) is the image width; 4.
NHWC data layout

- $N$ is the batch size; 1.
- $C$ is the number of feature maps (i.e., number of channels); 64.
- $H$ is the image height; 5.
- $W$ is the image width; 4.
Another layout (blocked C)

- N is the batch size; 1.
- C is the number of feature maps (i.e., number of channels); 64.
- H is the image height; 5.
- W is the image width; 4.
## Libraries offering high-performance implementations of key DNN layers

### TensorFlow NN ops

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tensorflow::ops::AvgPool</code></td>
<td>Performs average pooling on the input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::AvgPool3D</code></td>
<td>Performs 3D average pooling on the input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::BiasAdd</code></td>
<td>Adds bias to value.</td>
</tr>
<tr>
<td><code>tensorflow::ops::BiasAddGrad</code></td>
<td>The backward operation for &quot;BiasAdd&quot; on the &quot;bias&quot; argument.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Conv2D</code></td>
<td>Computes a 2-D convolution given 4-D input and filter tensors.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Conv2DBackpropFilter</code></td>
<td>Computes the gradients of convolution with respect to the filter.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Conv2DBackpropInput</code></td>
<td>Computes the gradients of convolution with respect to the input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Conv3D</code></td>
<td>Computes a 3-D convolution given 5-D input and filter tensors.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Conv3DBackpropFilterV2</code></td>
<td>Computes the gradients of 3-D convolution with respect to the filter.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Conv3DBackpropInputV2</code></td>
<td>Computes the gradients of 3-D convolution with respect to the input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::DataFormatDimMap</code></td>
<td>Returns the dimension index in the destination data format.</td>
</tr>
<tr>
<td><code>tensorflow::ops::DataFormatVecPermute</code></td>
<td>Permutes input tensor from src_format to dst_format.</td>
</tr>
<tr>
<td><code>tensorflow::ops::DepthwiseConv2dNative</code></td>
<td>Computes a 2-D depthwise convolution given 4-D input and filter tensors.</td>
</tr>
<tr>
<td><code>tensorflow::ops::DepthwiseConv2dNativeBackpropFilter</code></td>
<td>Computes the gradients of depthwise convolution with respect to the filter.</td>
</tr>
<tr>
<td><code>tensorflow::ops::DepthwiseConv2dNativeBackpropInput</code></td>
<td>Computes the gradients of depthwise convolution with respect to the input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Dilation2D</code></td>
<td>Computes the grayscale dilation of 4-D input and filter tensors.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Dilation2DBackpropFilter</code></td>
<td>Computes the gradient of morphological 2-D dilation filter.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Dilation2DBackpropInput</code></td>
<td>Computes the gradient of morphological 2-D dilation input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::Elu</code></td>
<td>Computes exponential linear: ( \exp(\text{features}) - 1 ) otherwise.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FractionalAvgPool</code></td>
<td>Performs fractional average pooling on the input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FractionalMaxPool</code></td>
<td>Performs fractional max pooling on the input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FusedBatchNorm</code></td>
<td>Batch normalization.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FusedBatchNormGrad</code></td>
<td>Gradient for batch normalization.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FusedBatchNormGradV2</code></td>
<td>Gradient for batch normalization.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FusedBatchNormGradV3</code></td>
<td>Gradient for batch normalization.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FusedBatchNormV2</code></td>
<td>Batch normalization.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FusedBatchNormV3</code></td>
<td>Batch normalization.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FusedPadConv2D</code></td>
<td>Performs a padding as a preprocess during a convolution.</td>
</tr>
<tr>
<td><code>tensorflow::ops::FusedResizeAndPadConv2D</code></td>
<td>Performs a resize and padding as a preprocess during a convolution.</td>
</tr>
<tr>
<td><code>tensorflow::ops::InTopK</code></td>
<td>Says whether the targets are in the top K predictions.</td>
</tr>
<tr>
<td><code>tensorflow::ops::InTopKV2</code></td>
<td>Says whether the targets are in the top K predictions.</td>
</tr>
<tr>
<td><code>tensorflow::ops::L2Loss</code></td>
<td>L2 Loss.</td>
</tr>
<tr>
<td><code>tensorflow::ops::LRN</code></td>
<td>Local Response Normalization.</td>
</tr>
<tr>
<td><code>tensorflow::ops::LogSoftmax</code></td>
<td>Computes log softmax activations.</td>
</tr>
<tr>
<td><code>tensorflow::ops::MaxPool</code></td>
<td>Performs max pooling on the input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::MaxPool3D</code></td>
<td>Performs 3D max pooling on the input.</td>
</tr>
<tr>
<td><code>tensorflow::ops::MaxPool3DGrad</code></td>
<td>Computes gradients of 3D max pooling function.</td>
</tr>
<tr>
<td><code>tensorflow::ops::MaxPool3DGradGrad</code></td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td><code>tensorflow::ops::MaxPoolGrad</code></td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td><code>tensorflow::ops::MaxPoolGradGradV2</code></td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td><code>tensorflow::ops::MaxPoolGradGradWithArgmax</code></td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td><code>tensorflow::ops::MaxPoolGradV2</code></td>
<td>Computes gradients of the maxpooling function.</td>
</tr>
<tr>
<td><code>tensorflow::ops::MaxPoolGradWithArgmax</code></td>
<td>Performs max pooling on the input and outputs both max values and indices.</td>
</tr>
<tr>
<td><code>tensorflow::ops::NhElement</code></td>
<td>Finds values of the n-th order statistic for the last dimension.</td>
</tr>
<tr>
<td><code>tensorflow::ops::QuantizedAvgPool</code></td>
<td>Produces the average pool of the input tensor for quantized types.</td>
</tr>
<tr>
<td><code>tensorflow::ops::QuantizedBatchNormalization</code></td>
<td>Quantized Batch normalization.</td>
</tr>
<tr>
<td><code>tensorflow::ops::QuantizedBiasAdd</code></td>
<td>Adds <code>Tensor</code> <code>bias</code> to <code>Tensor</code> <code>input</code> for Quantized types.</td>
</tr>
<tr>
<td><code>tensorflow::ops::QuantizedConv2D</code></td>
<td>Computes a 2D convolution given quantized 4D input and filter tensors.</td>
</tr>
<tr>
<td><code>tensorflow::ops::QuantizedMaxPool</code></td>
<td>Produces the max pool of the input tensor for quantized types.</td>
</tr>
</tbody>
</table>
Libraries offering high-performance implementations of key DNN layers

**TensorFlow** NN ops

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tensorflow.ops.AvgPool</td>
<td>Performs average pooling on the input.</td>
</tr>
<tr>
<td>tensorflow.ops.AvgPool2D</td>
<td>Performs 3D average pooling on the input.</td>
</tr>
<tr>
<td>tensorflow.ops.AvgPool2DGrad</td>
<td>Computes gradients of average pooling function.</td>
</tr>
<tr>
<td>tensorflow.ops.BiasAdd</td>
<td>Adds bias to value.</td>
</tr>
<tr>
<td>tensorflow.ops.BiasAddGrad</td>
<td>The backward operation for &quot;BiasAdd&quot; on the &quot;bias&quot; to be used.</td>
</tr>
<tr>
<td>tensorflow.ops.Conv2D</td>
<td>Computes a 2-D convolution given 4-D input and 2-D filter.</td>
</tr>
<tr>
<td>tensorflow.ops.Conv2DBackpropFilter</td>
<td>Computes gradients of convolution with respect to filters.</td>
</tr>
<tr>
<td>tensorflow.ops.Conv2DBackpropInput</td>
<td>Computes the gradients of convolution with respect to the input.</td>
</tr>
<tr>
<td>tensorflow.ops.Conv3D</td>
<td>Computes a 3-D convolution given 5-D input and 3-D filter.</td>
</tr>
<tr>
<td>tensorflow.ops.Conv3DBackpropFilterV2</td>
<td>Computes the gradients of 3-D convolution with respect to filters.</td>
</tr>
<tr>
<td>tensorflow.ops.Conv3DBackpropInputV2</td>
<td>Computes the gradients of 3-D convolution with respect to the input.</td>
</tr>
<tr>
<td>tensorflow.ops.DataFormatDimMap</td>
<td>Returns the dimension index in the destination data format.</td>
</tr>
<tr>
<td>tensorflow.ops.DataFormatVecPermute</td>
<td>Permutes input tensor from src_format to dst_format.</td>
</tr>
<tr>
<td>tensorflow.ops.DepthwiseConv2DNative</td>
<td>Computes a 2-D depthwise convolution given 4-D input and tensors.</td>
</tr>
<tr>
<td>tensorflow.ops.DepthwiseConv2DNativeBackpropFilter</td>
<td>Computes the gradients of depthwise convolution with respect to filters.</td>
</tr>
<tr>
<td>tensorflow.ops.DepthwiseConv2DNativeBackpropInput</td>
<td>Computes the gradients of depthwise convolution with respect to the input.</td>
</tr>
<tr>
<td>tensorflow.ops.Dilation2D</td>
<td>Computes the grayscale dilation of 4-D input and 3-D filter.</td>
</tr>
<tr>
<td>tensorflow.ops.Dilation2DBackpropFilter</td>
<td>Computes the gradient of morphological 2-D dilation filter.</td>
</tr>
<tr>
<td>tensorflow.ops.Dilation2DBackpropInput</td>
<td>Computes the gradient of morphological 2-D dilation input.</td>
</tr>
<tr>
<td>tensorflow.ops.Elu</td>
<td>Computes exponential linear: ( \text{exp(features)} - 1 ) otherwise.</td>
</tr>
<tr>
<td>tensorflow.ops.FractionalAvgPool</td>
<td>Performs fractional average pooling on the input.</td>
</tr>
<tr>
<td>tensorflow.ops.FractionalMaxPool</td>
<td>Performs fractional max pooling on the input.</td>
</tr>
<tr>
<td>tensorflow.ops.FusedBatchNorm</td>
<td>Batch normalization.</td>
</tr>
</tbody>
</table>

**NVIDIA cuDNN**

**Intel® oneAPI Deep Neural Network Library**
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).

Ug for library implementers!

<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>Conv / s1</td>
<td>3 x 3 x 512 dw</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 512 x 512</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 512 dw</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 512 x 1024</td>
<td>7 x 7 x 512</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 1024 dw</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 1024 x 1024</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>Avg Pool / s1</td>
<td>Pool 7 x 7</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>FC / s1</td>
<td>1024 x 1000</td>
<td>1 x 1 x 1024</td>
</tr>
<tr>
<td>Softmax / s1</td>
<td>Classifier</td>
<td>1 x 1 x 1000</td>
</tr>
</tbody>
</table>
Memory traffic between operations

Consider this sequence:

- Imagine the bandwidth cost of dumping 1 GB of conv outputs to memory, and reading it back in between each op!
- But note that per-element [scale+bias] operation can easily be performed per-element right after each element is computed by conv!
- And max pool's output can be computed once every 2x2 region of output is computed.
Fusing operations with 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++) {
                float tmp = 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)
                            tmp += layer_weights[f][jj][ii][kk] * input[img][j+jj][i+ii][kk];
                output[img][j][i][f] = tmp*scale + bias;
            }
```

**Exercise to class 1:**
Is there a way to eliminate the scale/bias operation completely?

**Exercise to class 2:**
How would you also “fuse” in the max pool?
Old style: hardcoded “fused” ops

```c
#include <cudnn.h>

extern "C" {
    cudnnStatus_t cudnnConvolutionBiasActivationForward(
        cudnnHandle_t
        const void *alpha1,
        const cudnnTensorDescriptor_t xDesc,
        const void *x,
        const cudnnFilterDescriptor_t wDesc,
        const void *w,
        const cudnnConvolutionDescriptor_t convDesc,
        cudnnConvolutionFwdAlgo_t algo,
        void *workSpace,
        size_t workSpaceSizeInBytes,
        const void *alpha2,
        const cudnnTensorDescriptor_t zDesc,
        const void *z,
        const cudnnTensorDescriptor_t biasDesc,
        const void *bias,
        const cudnnActivationDescriptor_t activationDesc,
        const cudnnTensorDescriptor_t yDesc,
        void *y)
    {
        // Function implementation...
    }
}
```

This function applies a bias and then an activation to the convolutions or cross-correlations of `cudnnConvolutionForward()`, returning results in `y`. The full computation follows the equation \( y = \text{act}(\text{alpha1} \ast \text{conv}(x) + \text{alpha2} \ast z + \text{bias}) \).

Tensorflow:

```python
import tensorflow as tf

# Batch normalization.

tf.keras.layers.experimental.preprocessing.FusedBatchNorm

# Performs a resize and padding as a preprocess during a convolution.

tf.keras.layers.experimental.preprocessing.FusedResizeAndPadConv2D
```
Fusion example: CUDNN “backend"

Compiler generate new implementations that “fuse” multiple operations into a single node that executes efficiently (without runtime overhead or communicating intermediate results through memory)

Note: this is Halide “compute at”
Many efforts to automatically schedule key DNN operations
More optimizations

- Low precision
- Sparsification
  - Via automatic mechanisms
  - Via engineering better DNN topologies
  - Via automating engineering of better DNN topologies
- Dynamic execution
- Specialization to input domain (not today)
Use of low precision values

- Many efforts to use low precision values for DNN weights and intermediate activations
- Eight and 16 bit values are common
- 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*
{mohammadr, 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.
Pruning/Sparsification

Automatic?
Hand-engineered?
“Pruning” (sparsifying) a network

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

The diagram shows a neuron with inputs $x_0, x_1, x_2, x_3$ and weights $w_0, w_1, w_2, w_3$. The neuron computes the following function:

$$f \left( \sum_i x_i w_i + b \right)$$

And:

$$f(x) = \max(0, x)$$
“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)
- 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>

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

<table>
<thead>
<tr>
<th>Indices</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1.8</td>
<td>0.5</td>
<td>2.1</td>
<td></td>
</tr>
</tbody>
</table>
Efficiently storing the surviving connections

Step 2: Weight sharing: make surviving connections share a small set of weights
- Cluster weights via k-means clustering
- Compress weights by only storing index of assigned cluster (\(\log(k)\) 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>Weight bits (P+Q)</th>
<th>Weight bits (P+Q+H)</th>
<th>Index bits (P+Q)</th>
<th>Index bits (P+Q+H)</th>
<th>Compress rate (P+Q)</th>
<th>Compress rate (P+Q+H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1_1</td>
<td>2K</td>
<td>58%</td>
<td>8</td>
<td>6.8</td>
<td>5</td>
<td>1.7</td>
<td>40.0%</td>
<td>29.97%</td>
</tr>
<tr>
<td>conv1_2</td>
<td>37K</td>
<td>22%</td>
<td>8</td>
<td>6.5</td>
<td>5</td>
<td>2.6</td>
<td>9.8%</td>
<td>6.99%</td>
</tr>
<tr>
<td>conv2_1</td>
<td>74K</td>
<td>34%</td>
<td>8</td>
<td>5.6</td>
<td>5</td>
<td>2.4</td>
<td>14.3%</td>
<td>8.91%</td>
</tr>
<tr>
<td>conv2_2</td>
<td>148K</td>
<td>36%</td>
<td>8</td>
<td>5.9</td>
<td>5</td>
<td>2.3</td>
<td>14.7%</td>
<td>9.31%</td>
</tr>
<tr>
<td>conv3_1</td>
<td>295K</td>
<td>53%</td>
<td>8</td>
<td>4.8</td>
<td>5</td>
<td>1.8</td>
<td>21.7%</td>
<td>11.15%</td>
</tr>
<tr>
<td>conv3_2</td>
<td>590K</td>
<td>24%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.9</td>
<td>9.7%</td>
<td>5.67%</td>
</tr>
<tr>
<td>conv3_3</td>
<td>590K</td>
<td>42%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.2</td>
<td>17.0%</td>
<td>8.96%</td>
</tr>
<tr>
<td>conv4_1</td>
<td>1M</td>
<td>32%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.6</td>
<td>13.1%</td>
<td>7.29%</td>
</tr>
<tr>
<td>conv4_2</td>
<td>2M</td>
<td>27%</td>
<td>8</td>
<td>4.2</td>
<td>5</td>
<td>2.9</td>
<td>10.9%</td>
<td>5.93%</td>
</tr>
<tr>
<td>conv4_3</td>
<td>2M</td>
<td>34%</td>
<td>8</td>
<td>4.4</td>
<td>5</td>
<td>2.5</td>
<td>14.0%</td>
<td>7.47%</td>
</tr>
<tr>
<td>conv5_1</td>
<td>2M</td>
<td>35%</td>
<td>8</td>
<td>4.7</td>
<td>5</td>
<td>2.5</td>
<td>14.3%</td>
<td>8.00%</td>
</tr>
<tr>
<td>conv5_2</td>
<td>2M</td>
<td>29%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.7</td>
<td>11.7%</td>
<td>6.52%</td>
</tr>
<tr>
<td>conv5_3</td>
<td>2M</td>
<td>36%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.3</td>
<td>14.8%</td>
<td>7.79%</td>
</tr>
<tr>
<td>fc6</td>
<td>103M</td>
<td>4%</td>
<td>5</td>
<td>3.6</td>
<td>5</td>
<td>3.5</td>
<td>1.6%</td>
<td>1.10%</td>
</tr>
<tr>
<td>fc7</td>
<td>17M</td>
<td>4%</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4.3</td>
<td>1.5%</td>
<td>1.25%</td>
</tr>
<tr>
<td>fc8</td>
<td>4M</td>
<td>23%</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3.4</td>
<td>7.1%</td>
<td>5.24%</td>
</tr>
<tr>
<td>Total</td>
<td>138M</td>
<td>7.5% (13⇥)</td>
<td>6.4</td>
<td>4.1</td>
<td>5</td>
<td>3.1</td>
<td>3.2% (31⇥)</td>
<td>2.05% (49⇥)</td>
</tr>
</tbody>
</table>

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

ImageNet Image Classification Performance

<table>
<thead>
<tr>
<th></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
This a great example of non-domain-specific vs. domain-specific approach to innovation.
Leveraging ML domain-knowledge: engineering more efficient topologies (aka better algorithm design)

- Original DNNs for image recognition were heavily 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

**Figure 3.** Example network architectures for ImageNet.

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

ResNet (34 layer version)
Modular network designs

Inception v4

A block

B block
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

We have observed the degradation problem - the plain net. To reveal the reasons, in Fig. we evaluate both top-1 and top-5 error rates.

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

ImageNet Top 1 Accuracy

Flops cost (area of circle is # params)

Accuracy (points) per flop

Figure credit: Canziani et al 2017
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

Depthwise Separable Conv Layer

K_w x K_h x NUM_CHANNELS inputs

K_w x K_h x NUM_CHANNELS weights (for each filter)

K_w x K_h x NUM_CHANNELS work per output pixel (per filter)

NUM_CHANNELS inputs

K_w x K_h weights (for each channel)

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

Image classification (ImageNet)
Comparison to Common DNNs

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 MobileNet-224</td>
<td>70.6%</td>
<td>569</td>
<td>4.2</td>
</tr>
<tr>
<td>GoogleNet</td>
<td>69.8%</td>
<td>1550</td>
<td>6.8</td>
</tr>
<tr>
<td>VGG 16</td>
<td>71.5%</td>
<td>15300</td>
<td>138</td>
</tr>
</tbody>
</table>

Image classification (ImageNet)
Comparison to Other Compressed DNNs

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50 MobileNet-160</td>
<td>60.2%</td>
<td>76</td>
<td>1.32</td>
</tr>
<tr>
<td>Squeezenet</td>
<td>57.5%</td>
<td>1700</td>
<td>1.25</td>
</tr>
<tr>
<td>AlexNet</td>
<td>57.2%</td>
<td>720</td>
<td>60</td>
</tr>
</tbody>
</table>

[Howard et al. 2017]

Table 9. Smaller MobileNet Comparison to Popular Models

Table 10. MobileNet for Stanford Dogs

Table 11. Performance of PlaNet using the MobileNet architecture

"Stanford CS348K, Spring 2021"
Value of improving DNN topology

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

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

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

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

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

[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>Mult-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)
Dynamic Execution
(conditionally execute only parts of the network)
Main idea of dynamic networks

Not all inputs require execution of the full capacity of the network

Example: cat detector

Positive example

Hard negative example
(May require deeper network, with many features per layer to discriminate)

Easy negative example
May be able to detect with smaller number of features.

Small on screen
Some regions of the screen might need more processing than others.
Main idea of dynamic networks

- Not all inputs require execution of the full capacity of the network
- Example 1: “cascade”, terminate early if confident in the result
  - Example 2: given input, compute only a subset of features and use those to perform task
Summary: efficiently evaluating deep nets

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

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

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