Lecture 6:

# Efficiently Evaluating Deep Networks

Visual Computing Systems
Stanford CS348K, Spring 2021

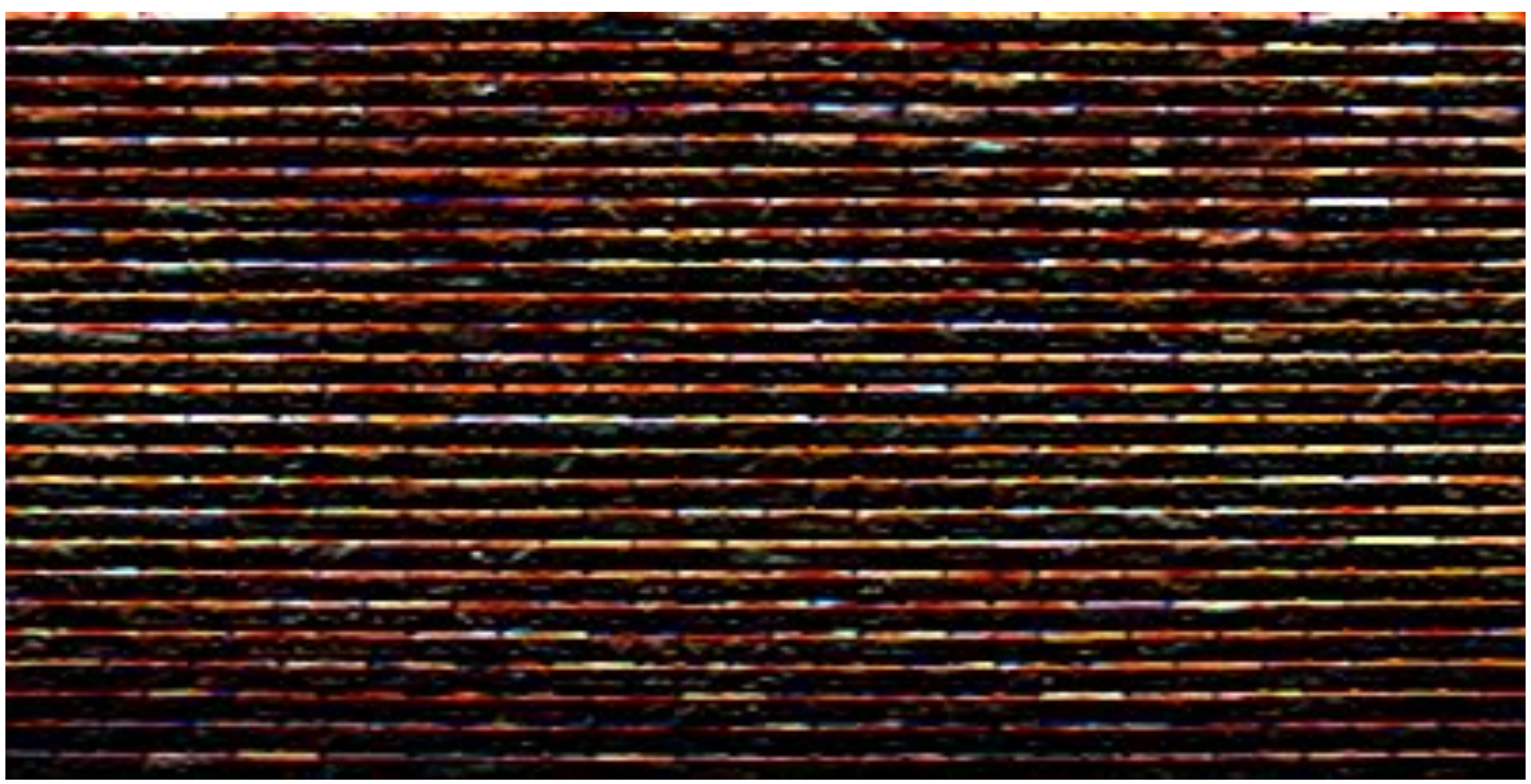
# Today

- We will discuss the workload of <u>evaluating</u> 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**

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

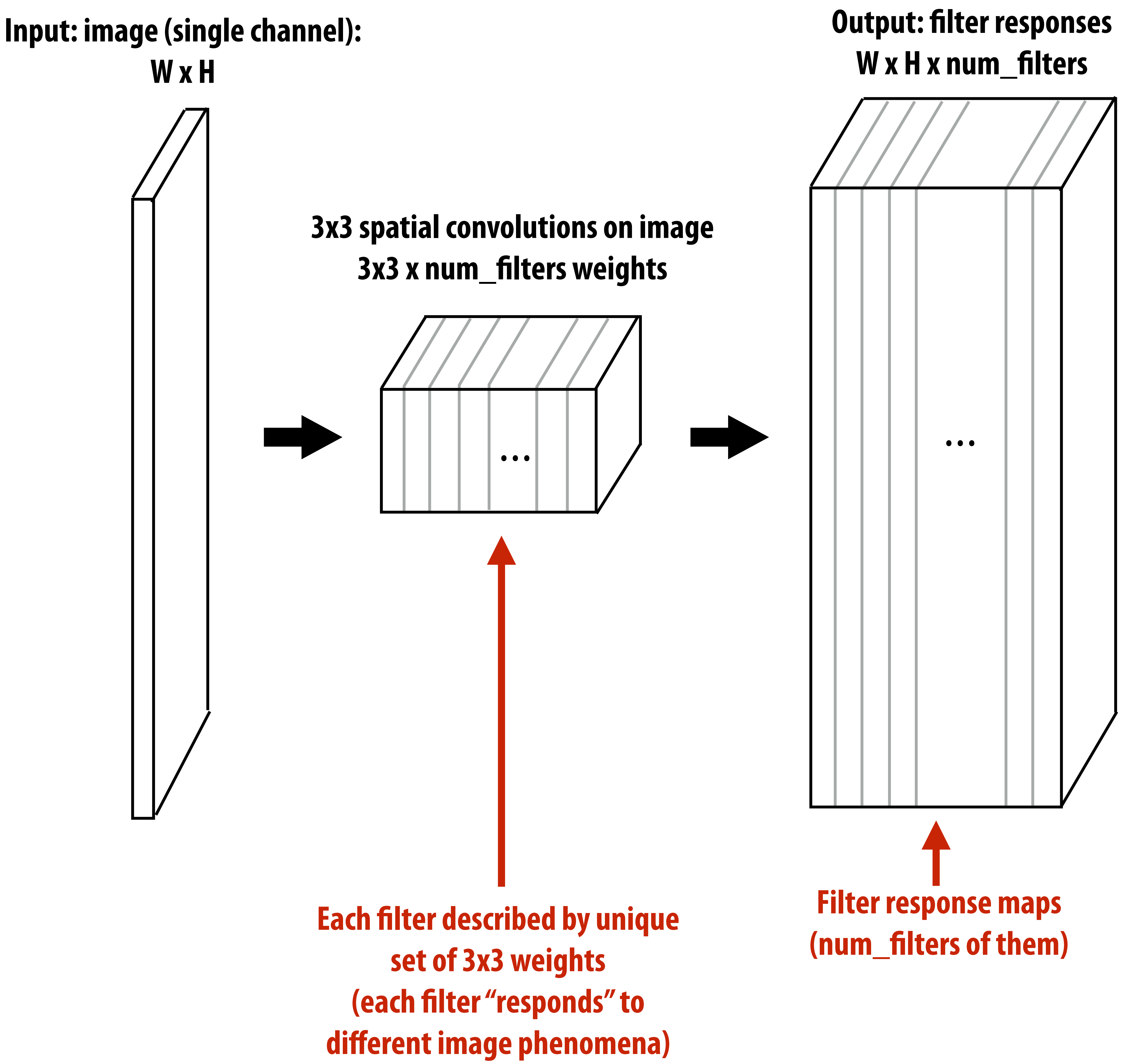


**Vertical gradients**

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

**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**

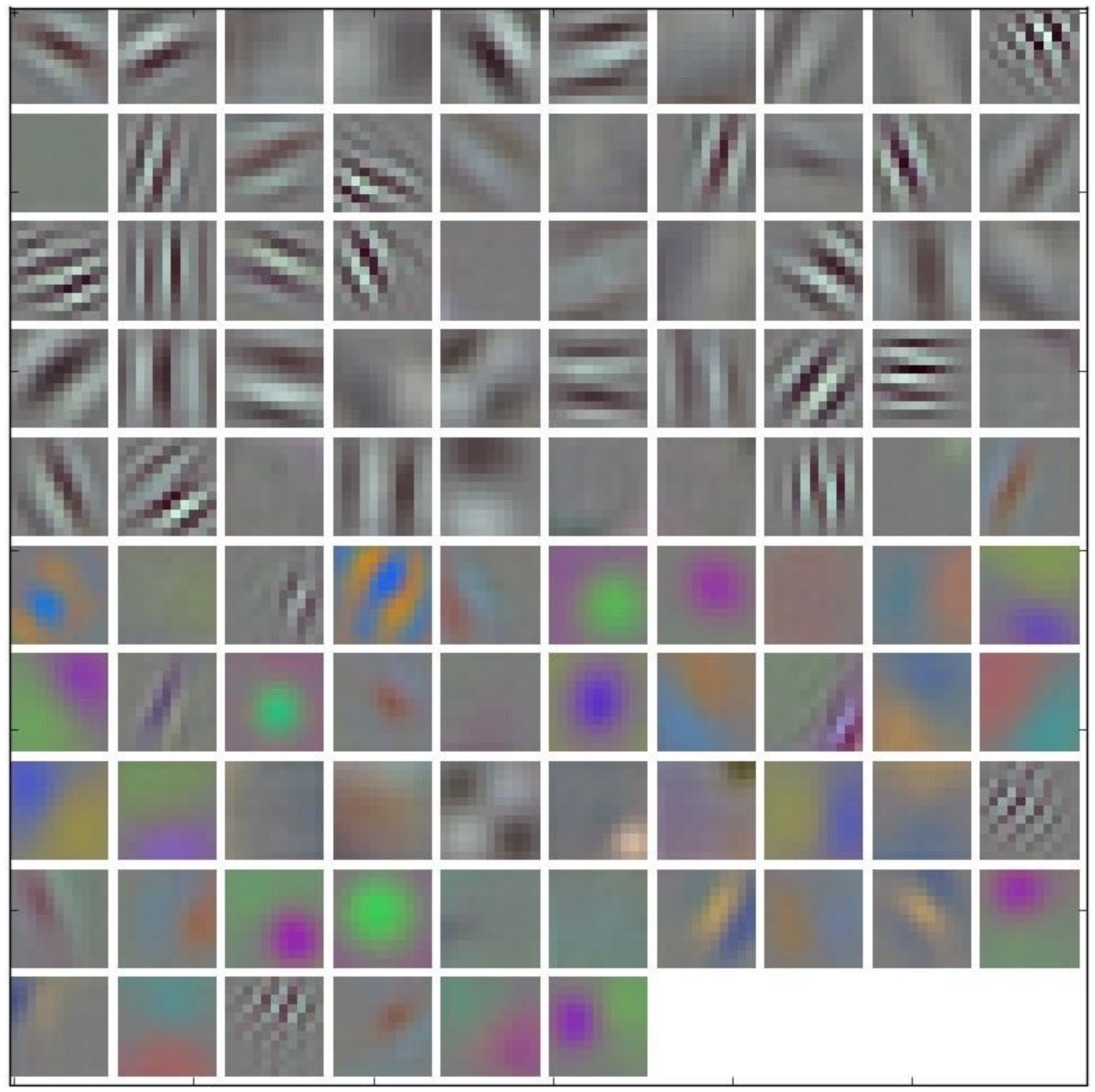
# Applying many filters to an image at once



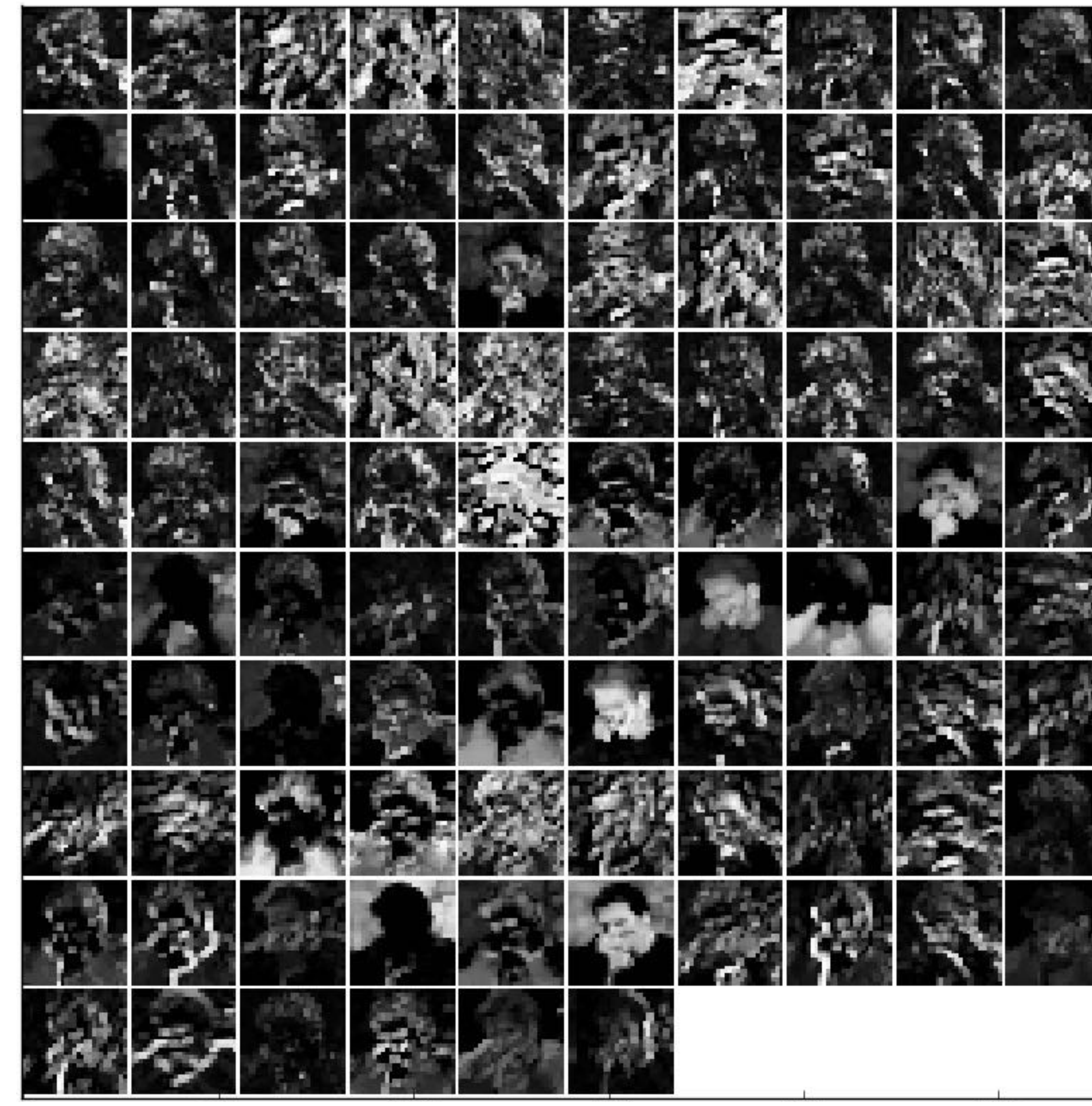
# Applying many filters to an image at once

Input RGB image (W x H x 3)

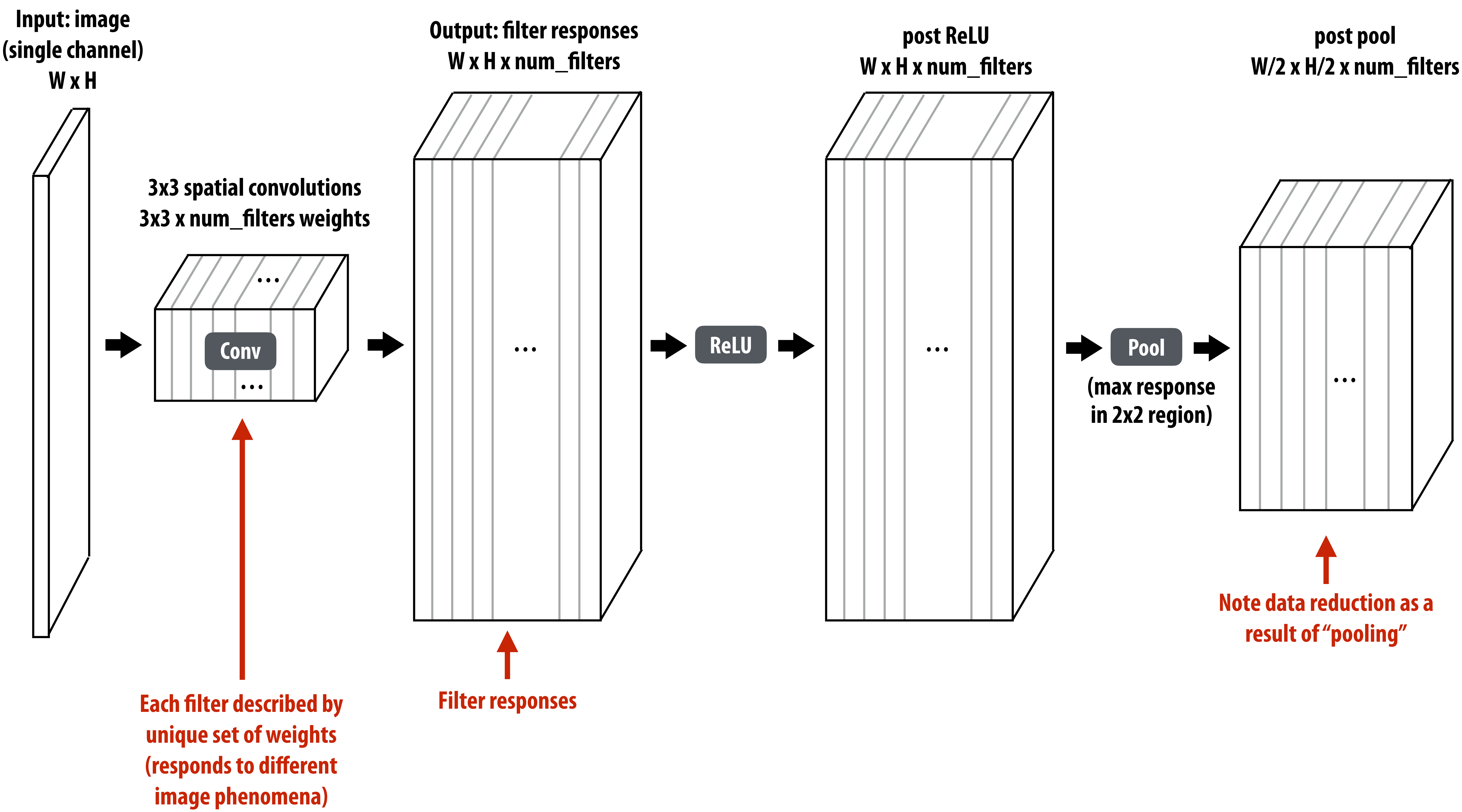
96 11x11x3 filters
(3D because they operate on RGB)



96 responses (normalized)



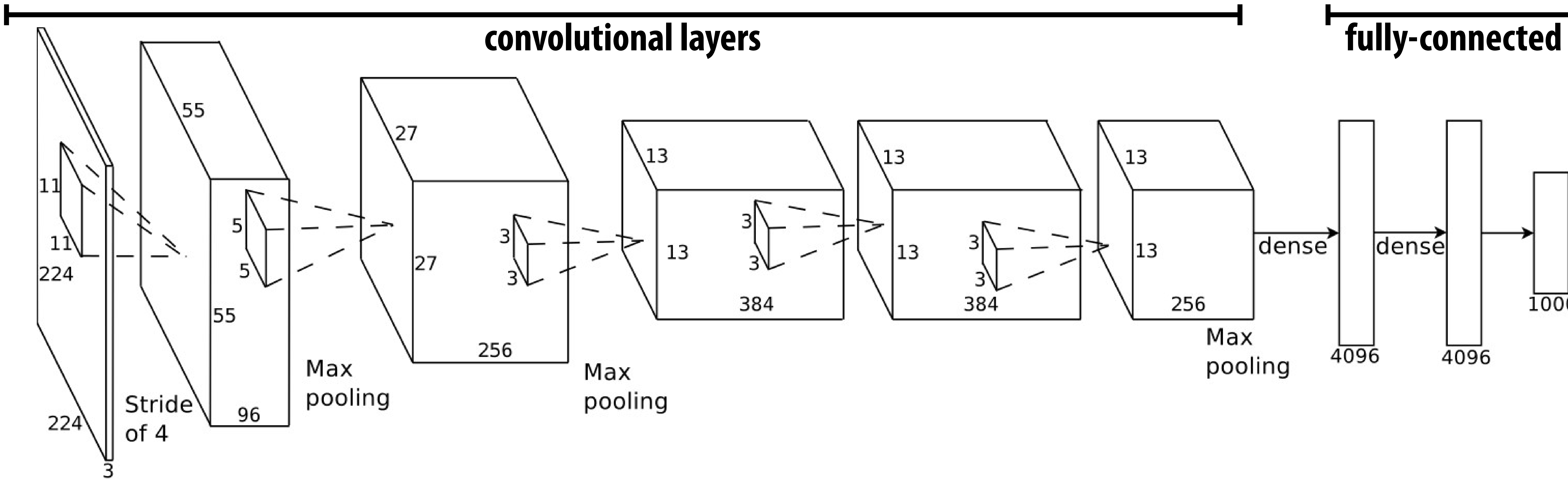
# Adding additional layers



# Example: "AlexNet" image classification DNN

## Sequences of conv + reLU + pool (optional) layers

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



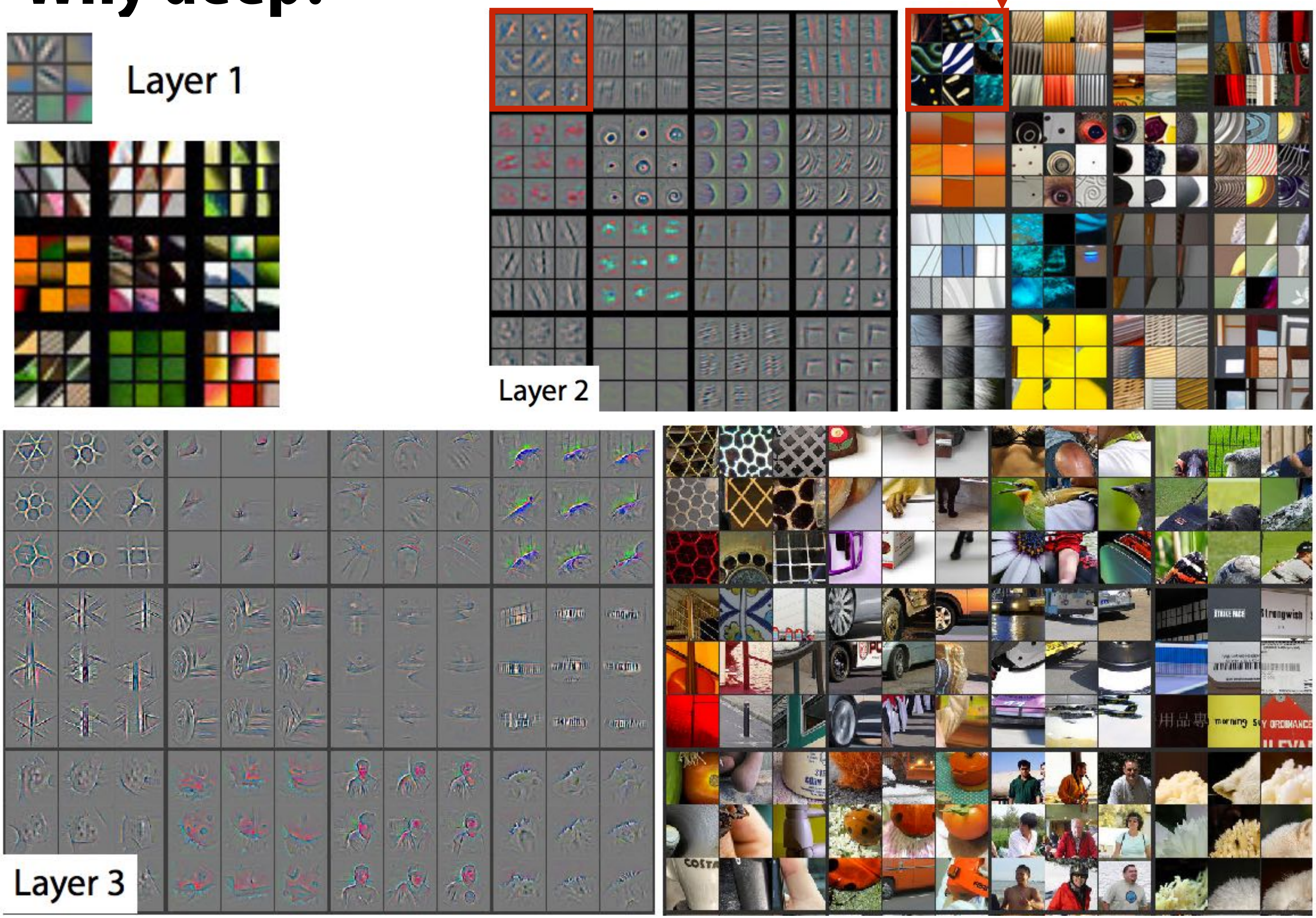

### Another example: VGG-16 [Simonyan15]: 13 convolutional layers

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
conv/reLU: 3x3x256x256
maxpool
conv/reLU: 3x3x256x512
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
maxpool
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
maxpool
fully-connected 4096
fully-connected 4096
fully-connected 1000
soft-max

[VGG illustration credit: Yang et al.]

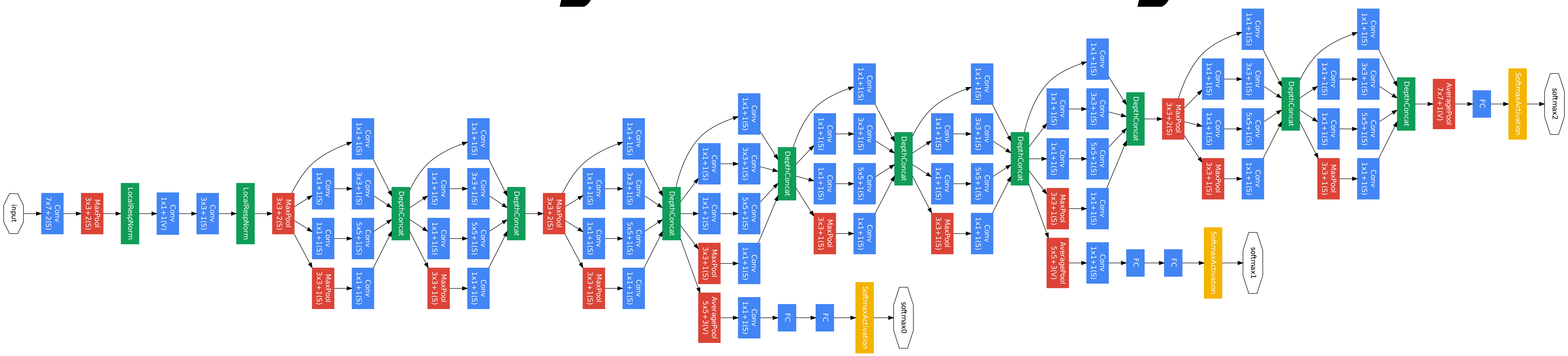
# Why deep?

Left: what pixels trigger the response
Right: images that generate strongest response for filters at each layer

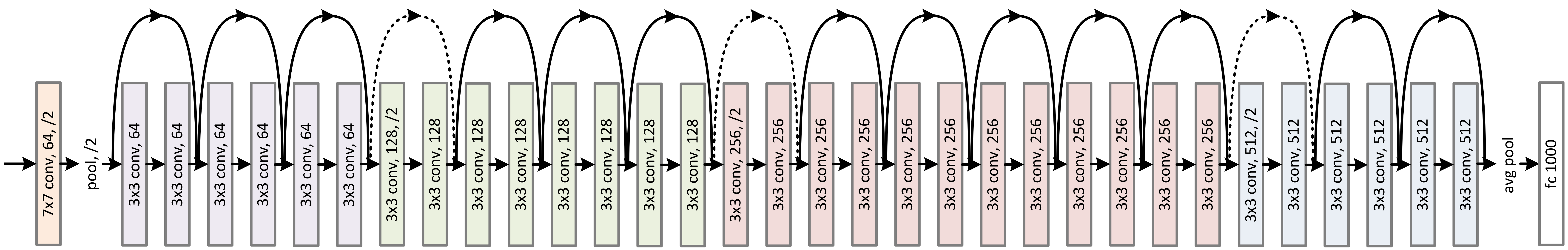



[image credit: Zeiler 14]

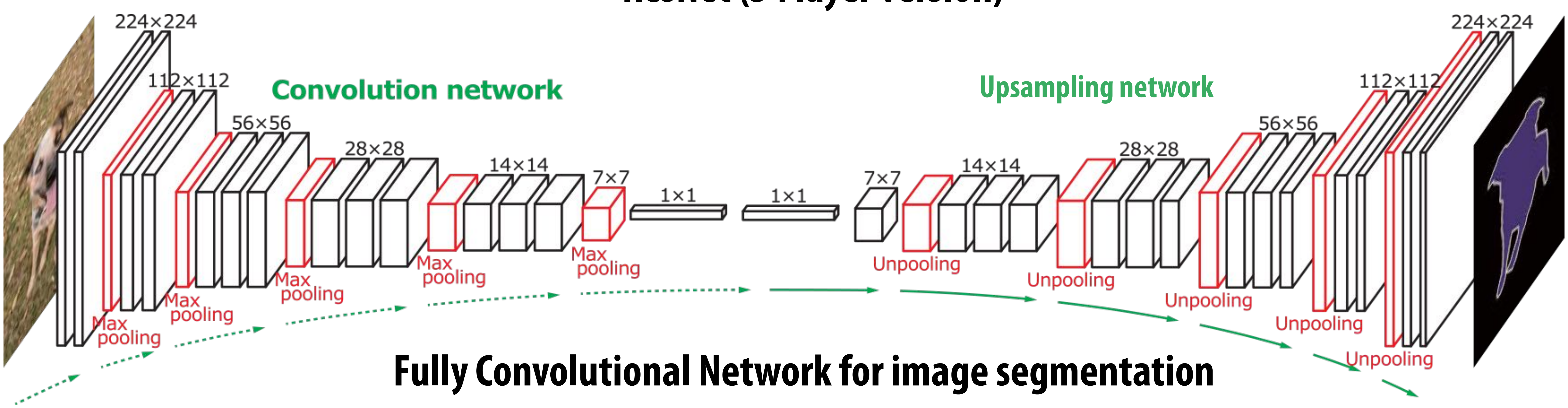
# More recent image understanding networks



Inception (GoogleLeNet)




ResNet (34 layer version)

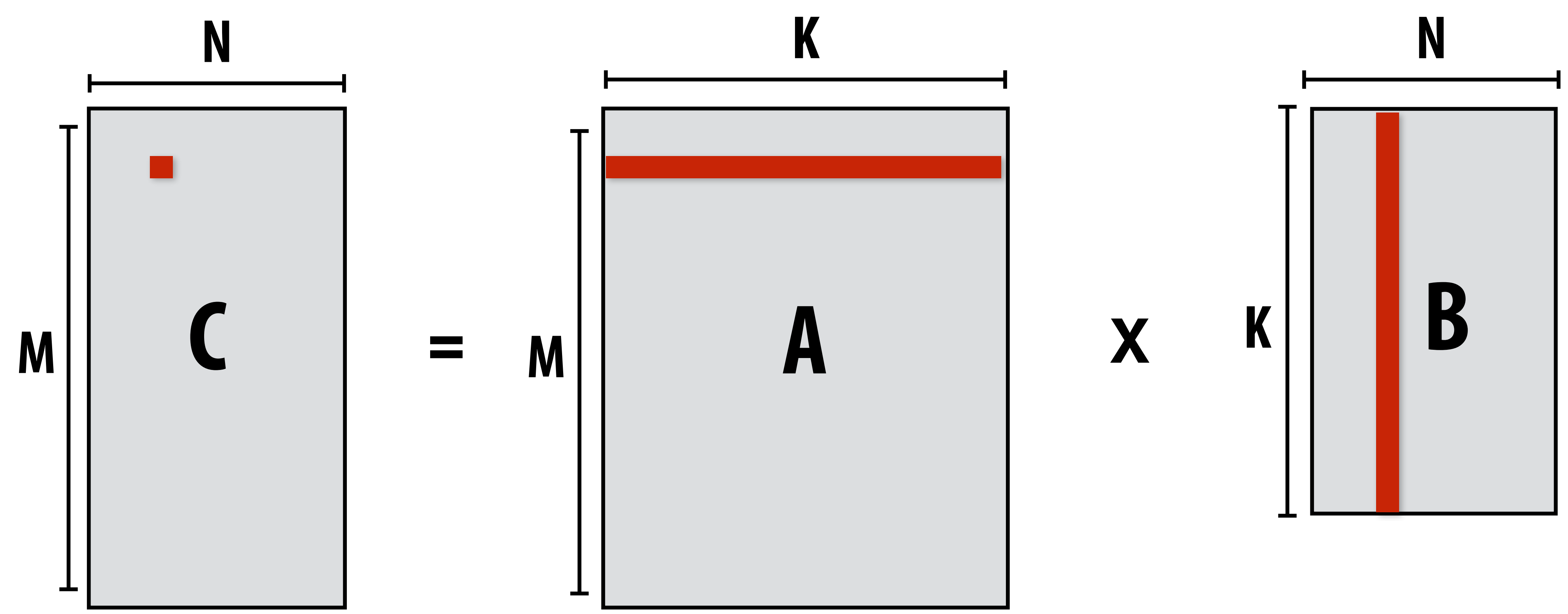



Fully Convolutional Network for image segmentation

# Efficiently implementing convolution layers

# Dense matrix multiplication

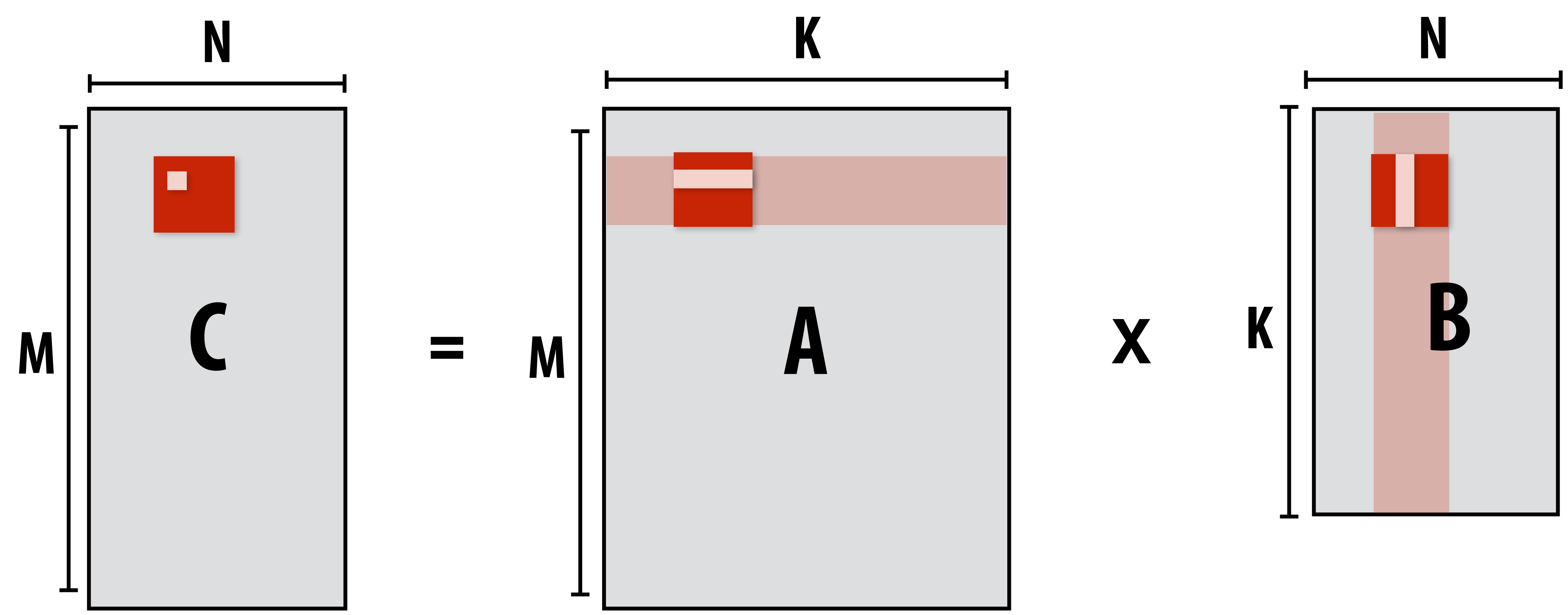
```
float A[M][K];
float B[K][N];
float C[M][N];

// 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




```
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

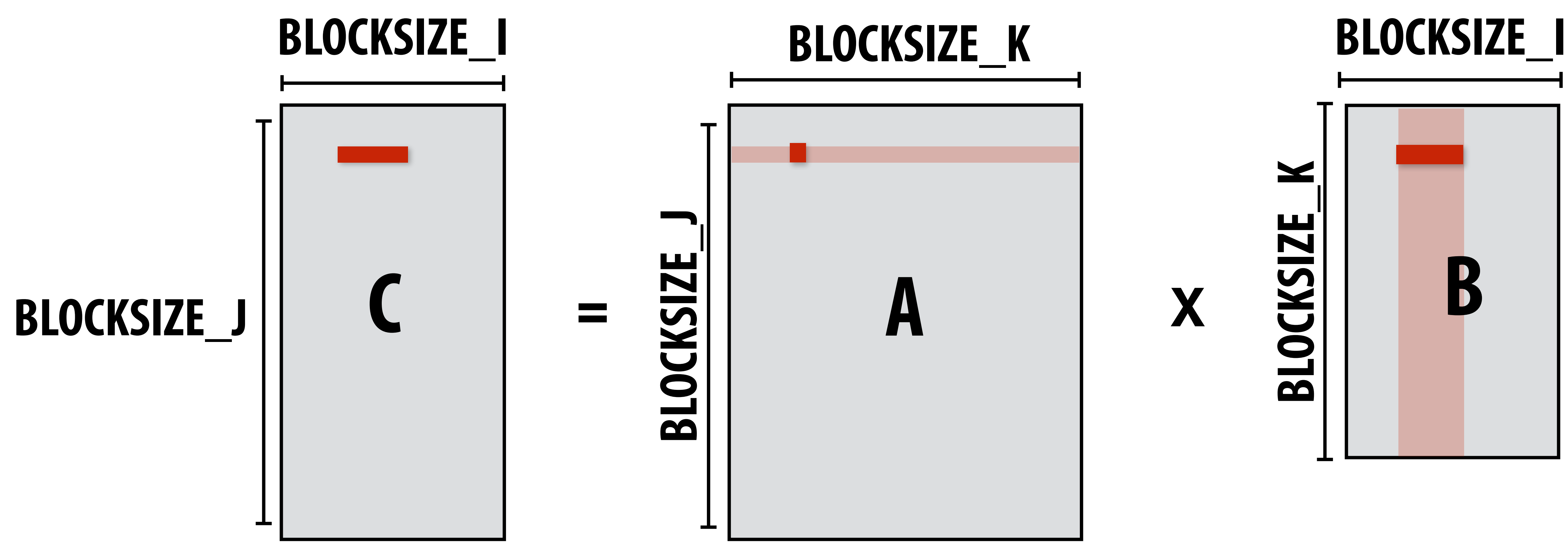
## Exploit multiple levels of memory hierarchy

```
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**



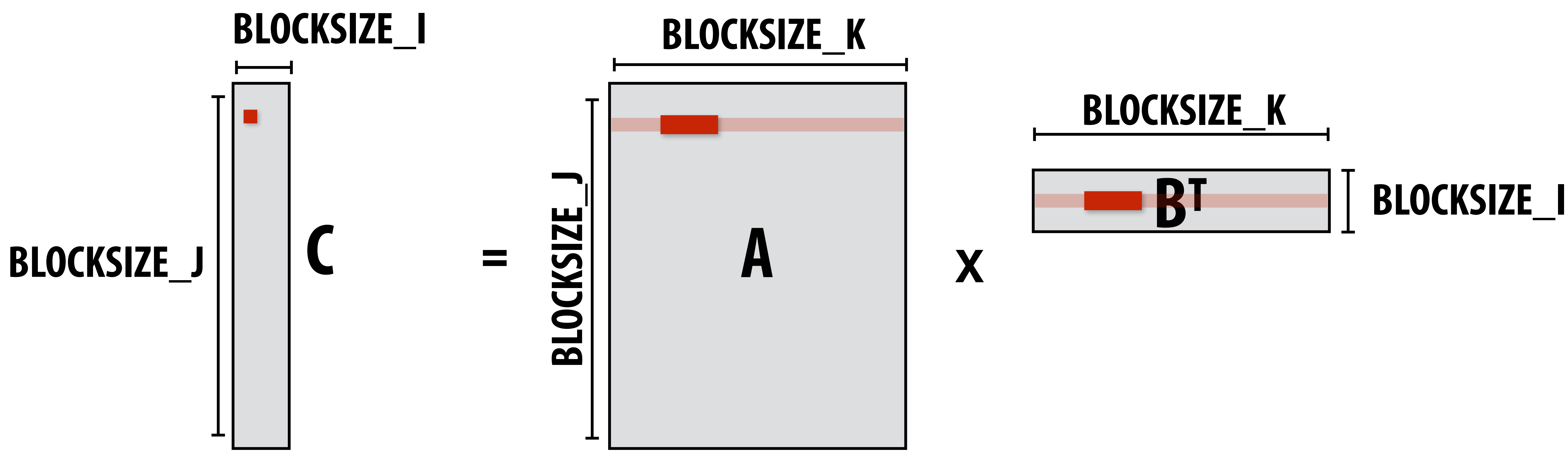

```
...
for (int j=0; j<BLOCKSIZE_J; j++) {
   for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {
      simd_vec C_accum = vec_load(&C[jblock+j][iblock+i]);
      for (int k=0; k<BLOCKSIZE_K; k++) {
         // C = A*B + C
         simd_vec A_val = splat(&A[jblock+j][kblock+k]); // load a single element in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



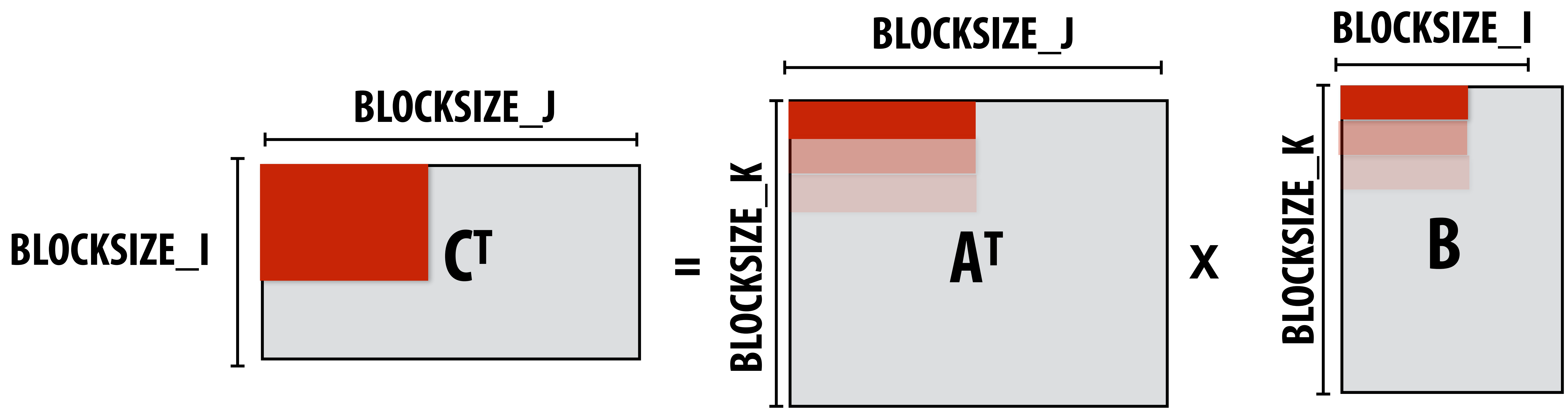

```
...
for (int j=0; j<BLOCKSIZE_J; j++)
   for (int i=0; i<BLOCKSIZE_I; i++) {
      float C_scalar = C[jblock+j][iblock+i];
      // C_scalar += dot(row of A,row of B)
      for (int k=0; k<BLOCKSIZE_K; k+=SIMD_WIDTH) {
        C_scalar += simd_dot(vec_load(&A[jblock+j][kblock+k]), vec_load(&Btrans[iblock+i][[kblock+k]);
      }
      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**

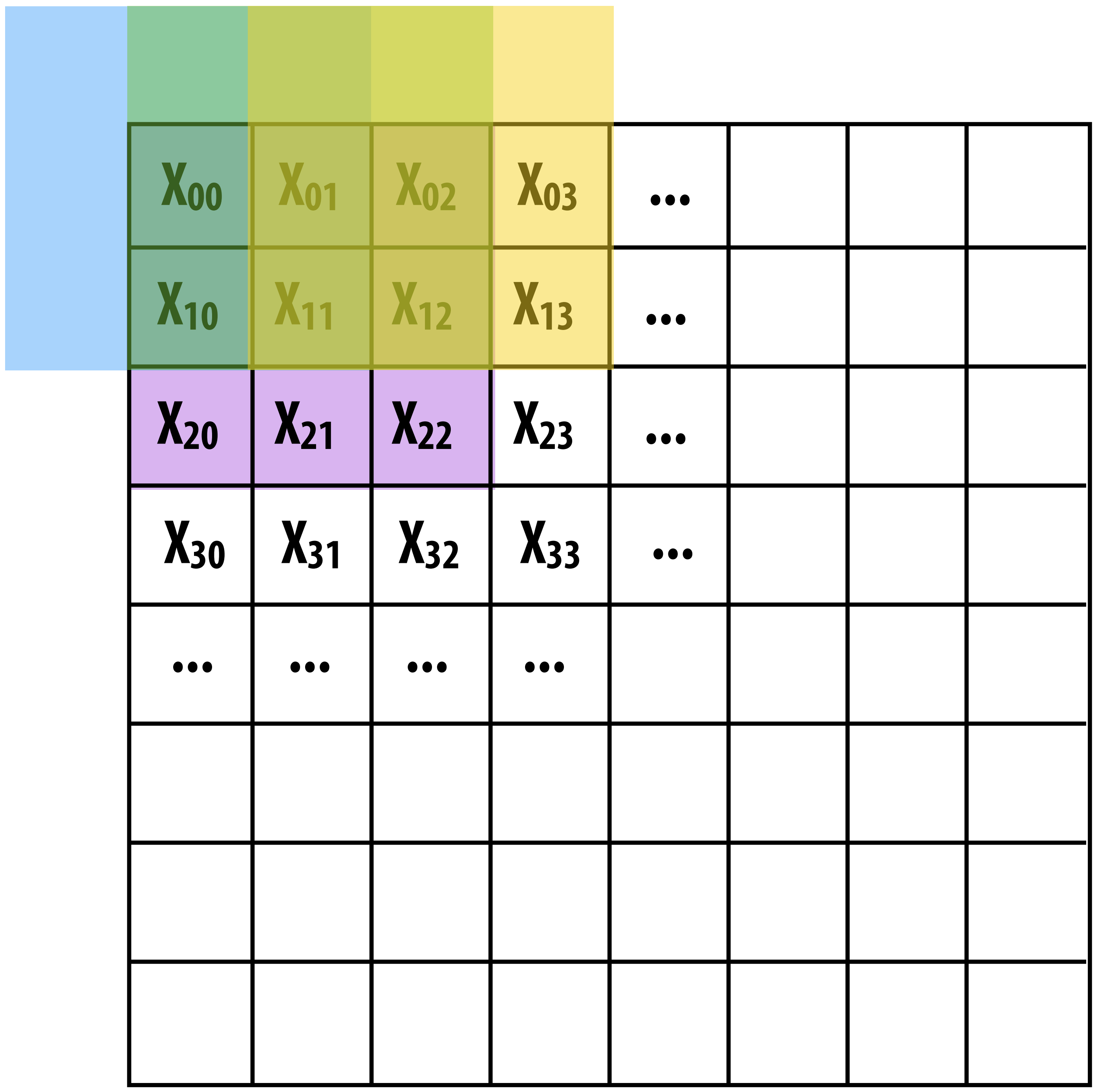
# Blocked dense matrix multiplication (3)




```
// 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++) {
        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]);
   }
}
```

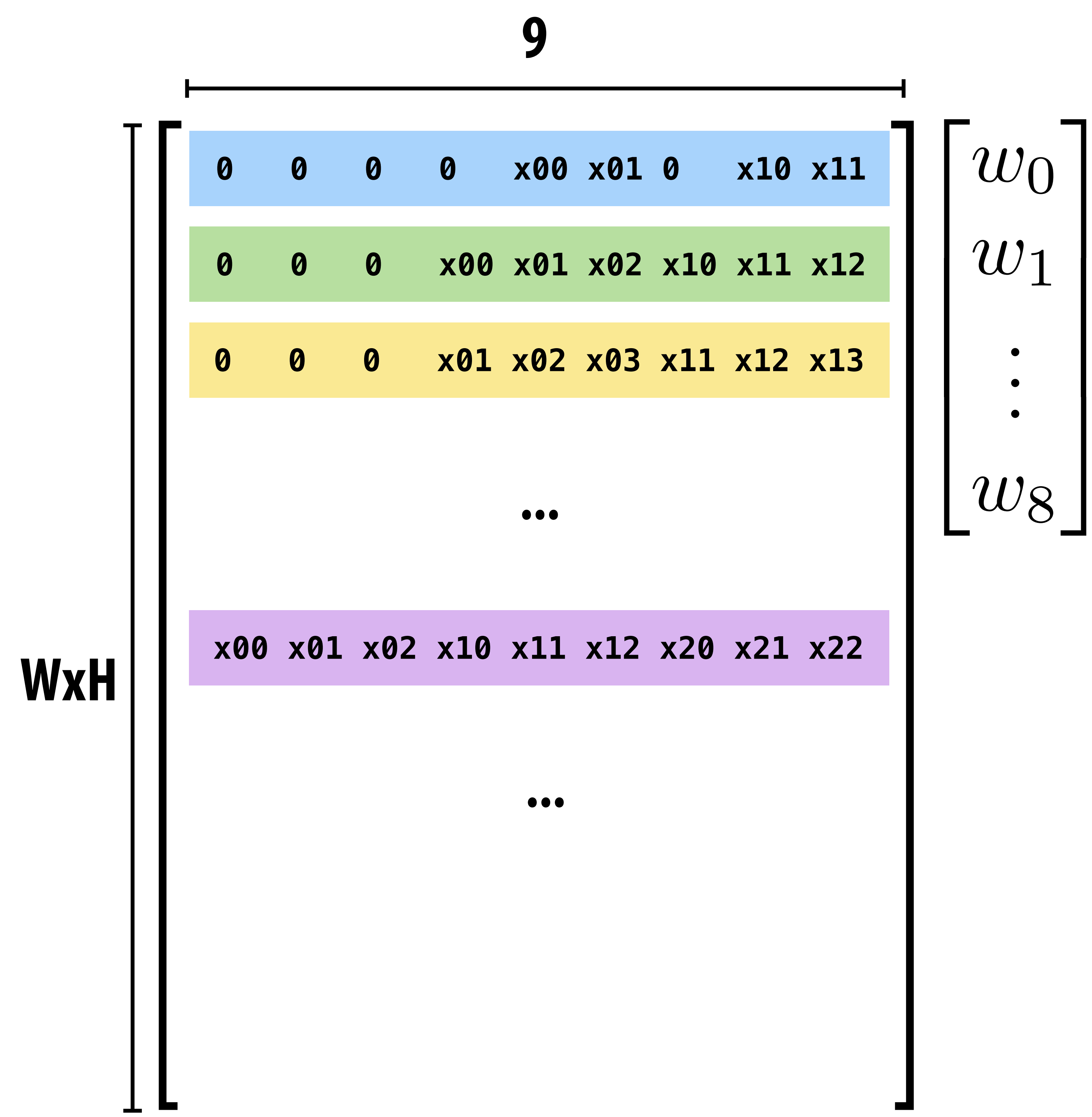
# 3x3 convolution as matrix-vector product

**Construct matrix from elements of input image**



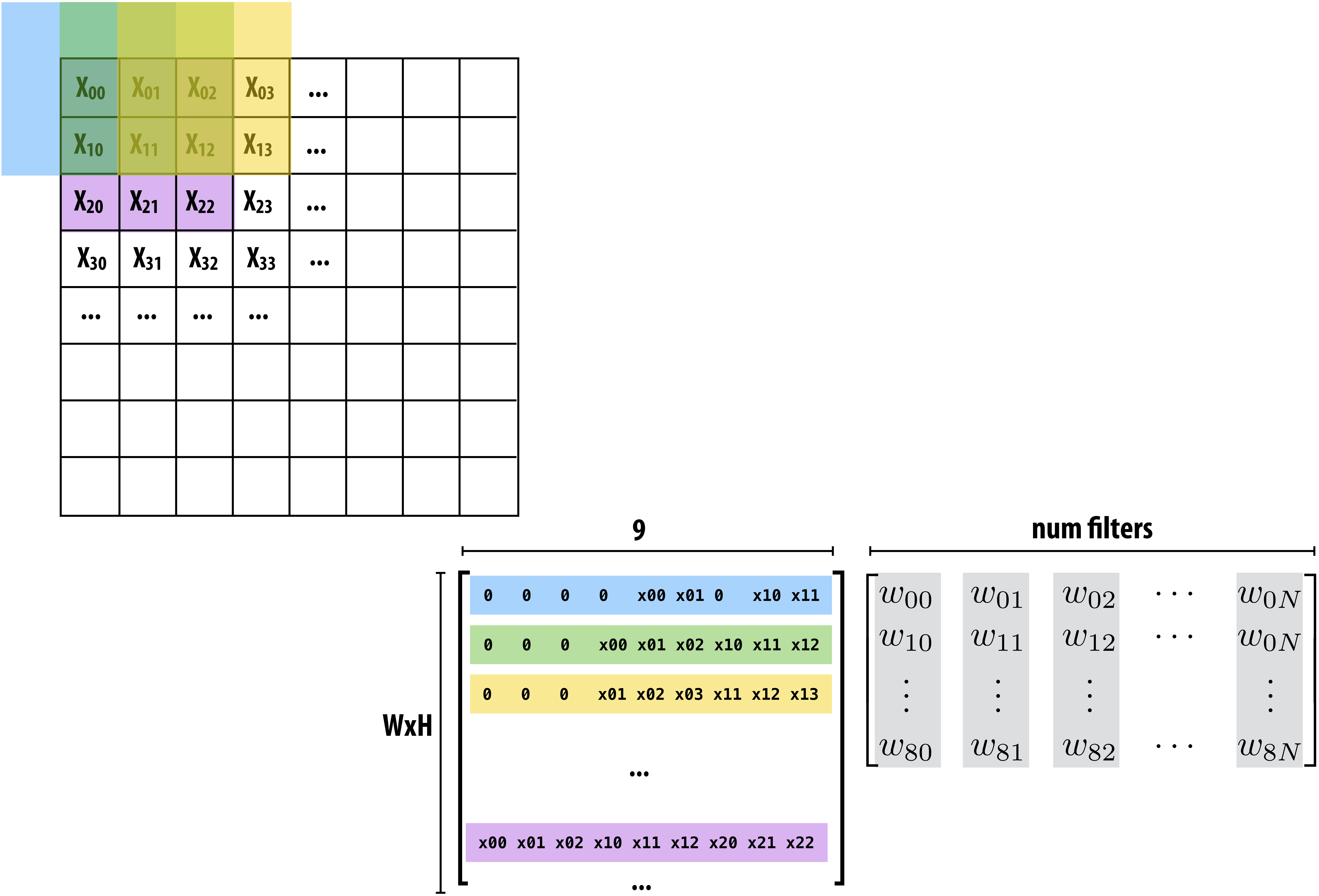

**O(N) storage overhead for filter with N elements**

**Must construct input data matrix**

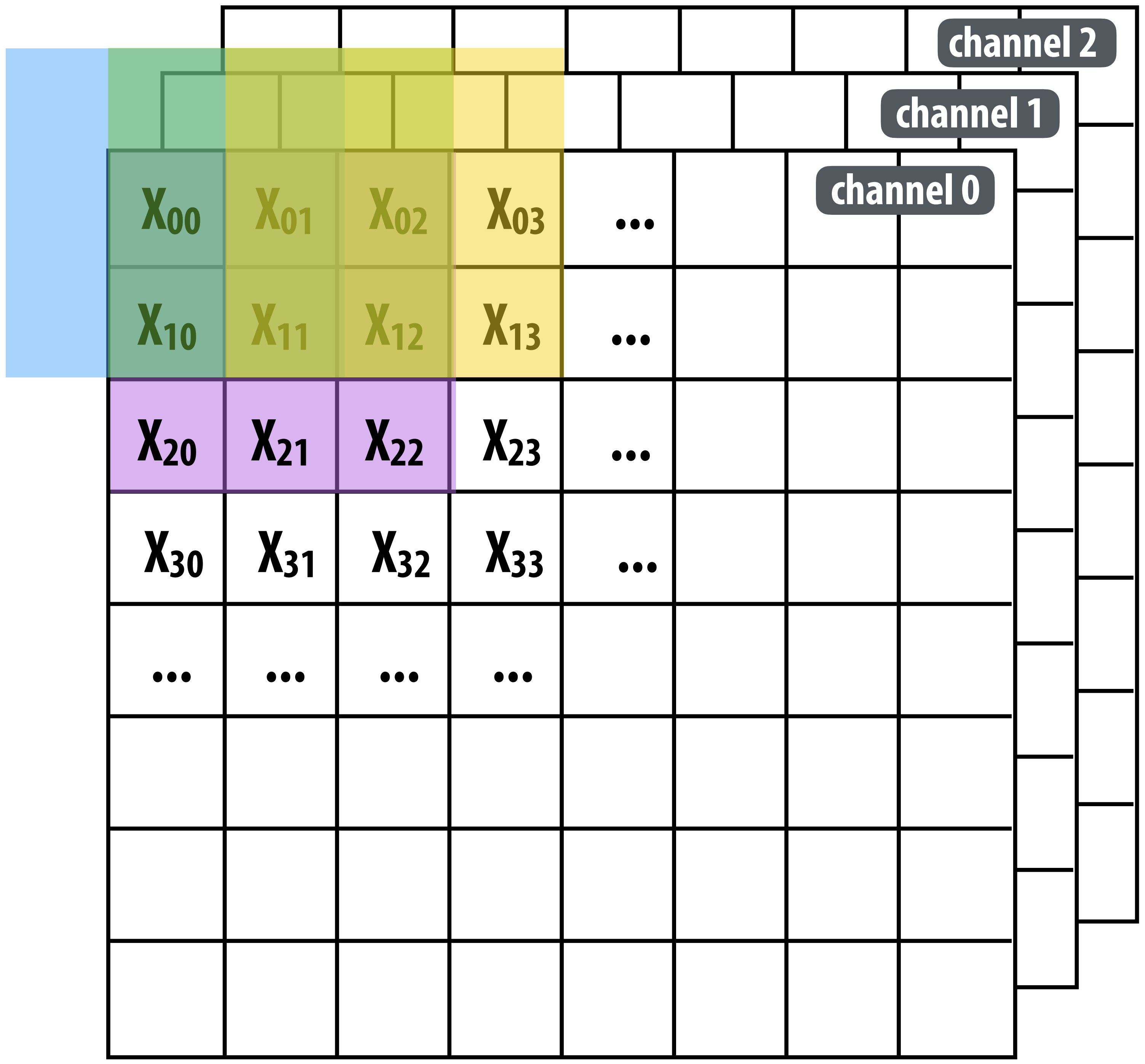



**Note: 0-pad matrix**

# Multiple convolutions as matrix-matrix mult

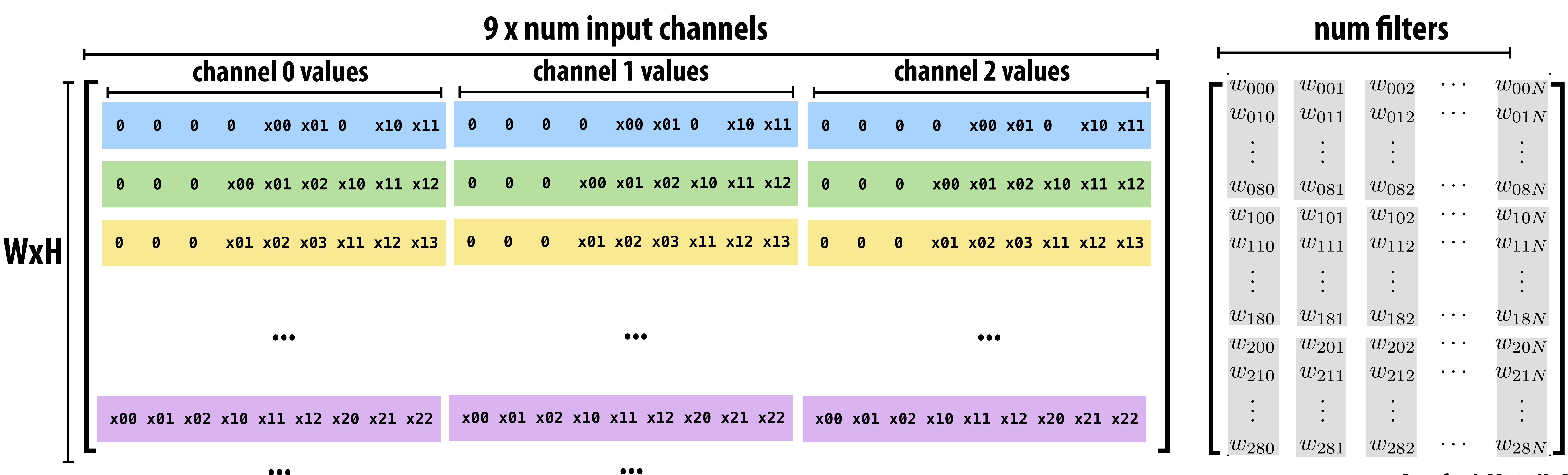


# Multiple convolutions on multiple input channels




For each filter, sum responses over input channels

Equivalent to (3 x 3 x num_channels) convolution on (W x H x num_channels) input data



# Direct implementation of conv layer (batched)

```
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

```
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);
```

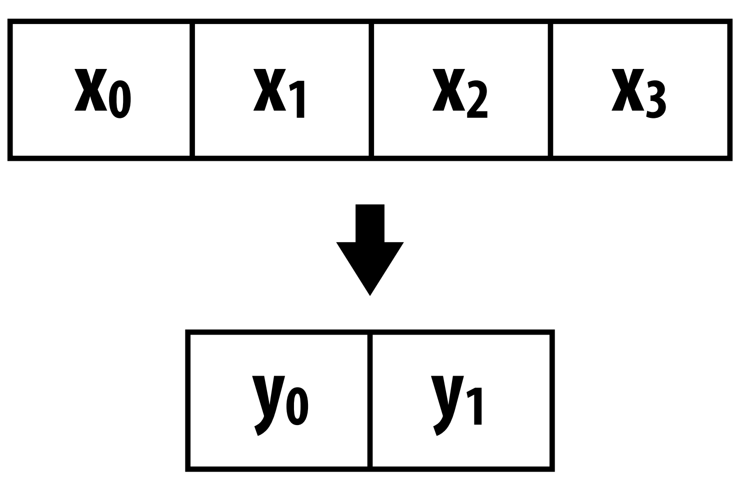
**Consider scheduling this seven-dimensional loop nest!**

**(p.s. You don’t have to consider, you will!)**

# Algorithmic improvements

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

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




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

$$m_1 = (x_0 - x_1)w_0$$

$$m_2 = (x_1 + x_2)\frac{w_0 + w_1 + w_2}{2}$$

$$m_3 = (x_2 - x_1)\frac{w_0 - w_1 + w_2}{2}$$

$$m_4 = (x_1 - x_3)w_2$$

Filter dependent
(can be precomputed)

**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 multiples 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 in 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 construct 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 tl 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).

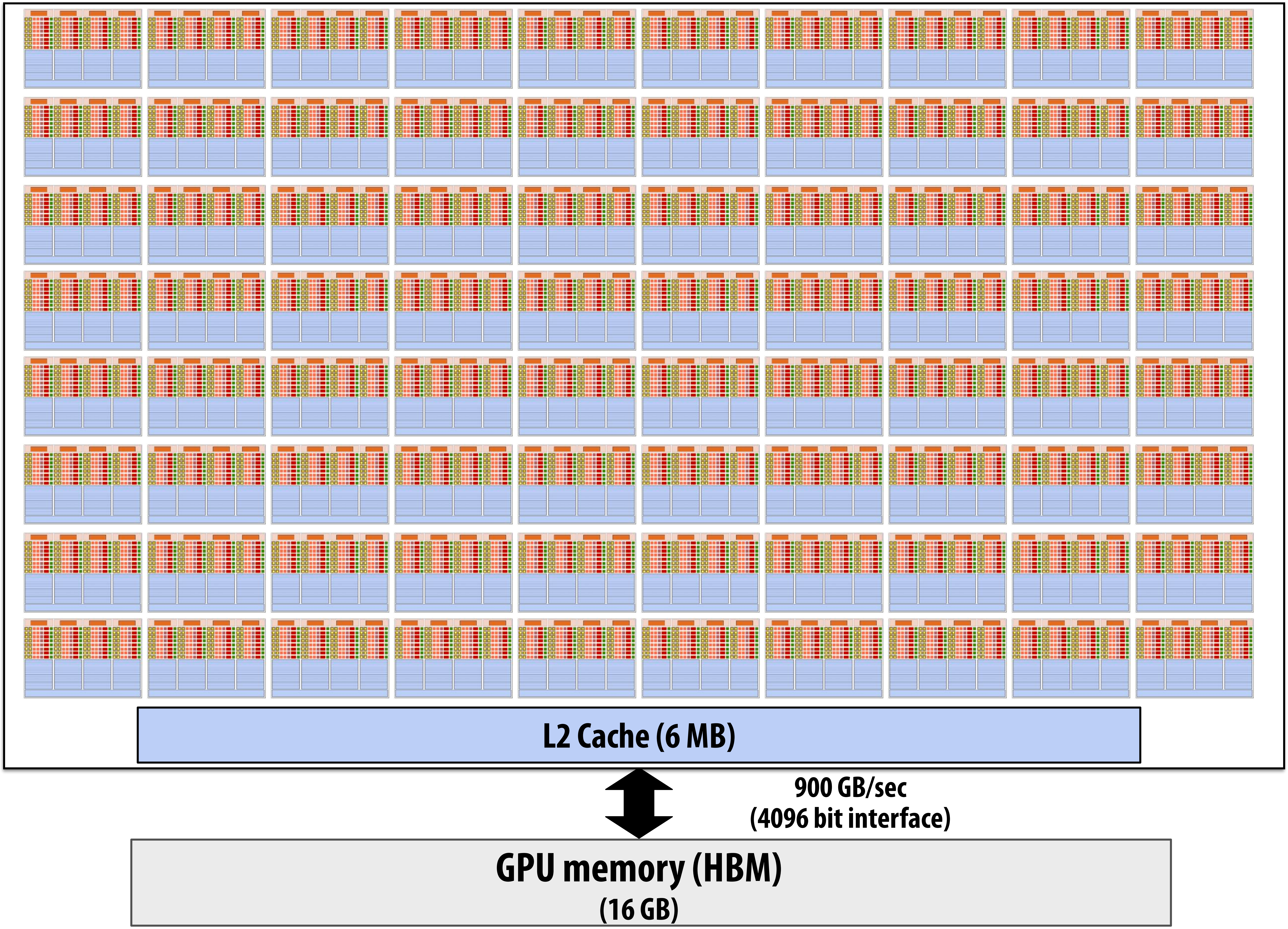
`CUDNN_CONVOLUTION_FWD_ALGO_FFT`
This algorithm uses the Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is nee to store intermediate results.

`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 t 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 t store intermediate results.

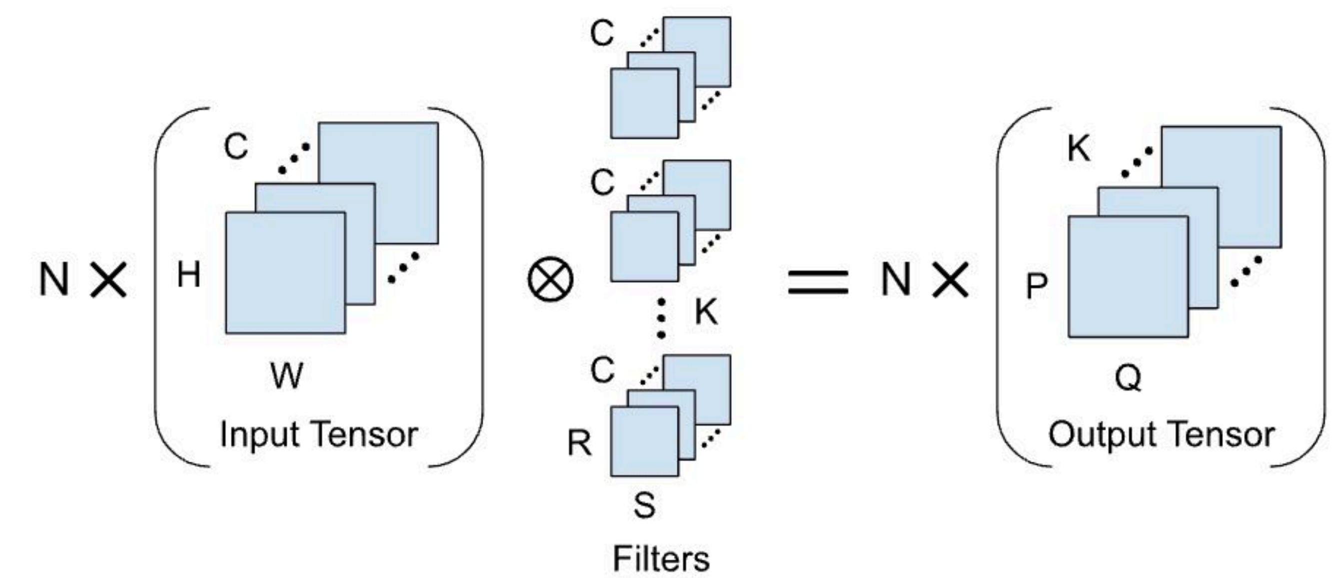
# Revall: NVIDIA V100 GPU (80 SMs)

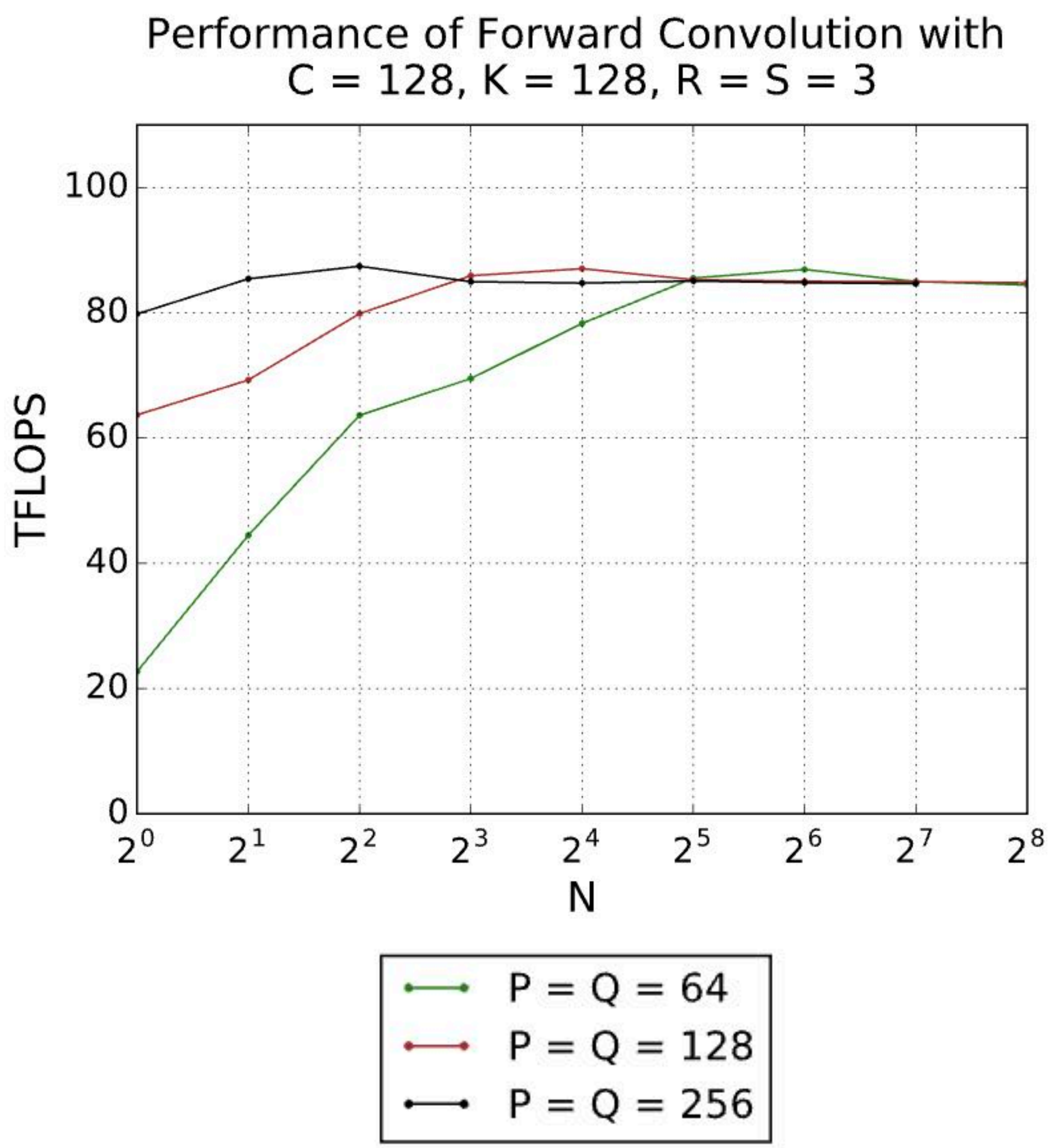


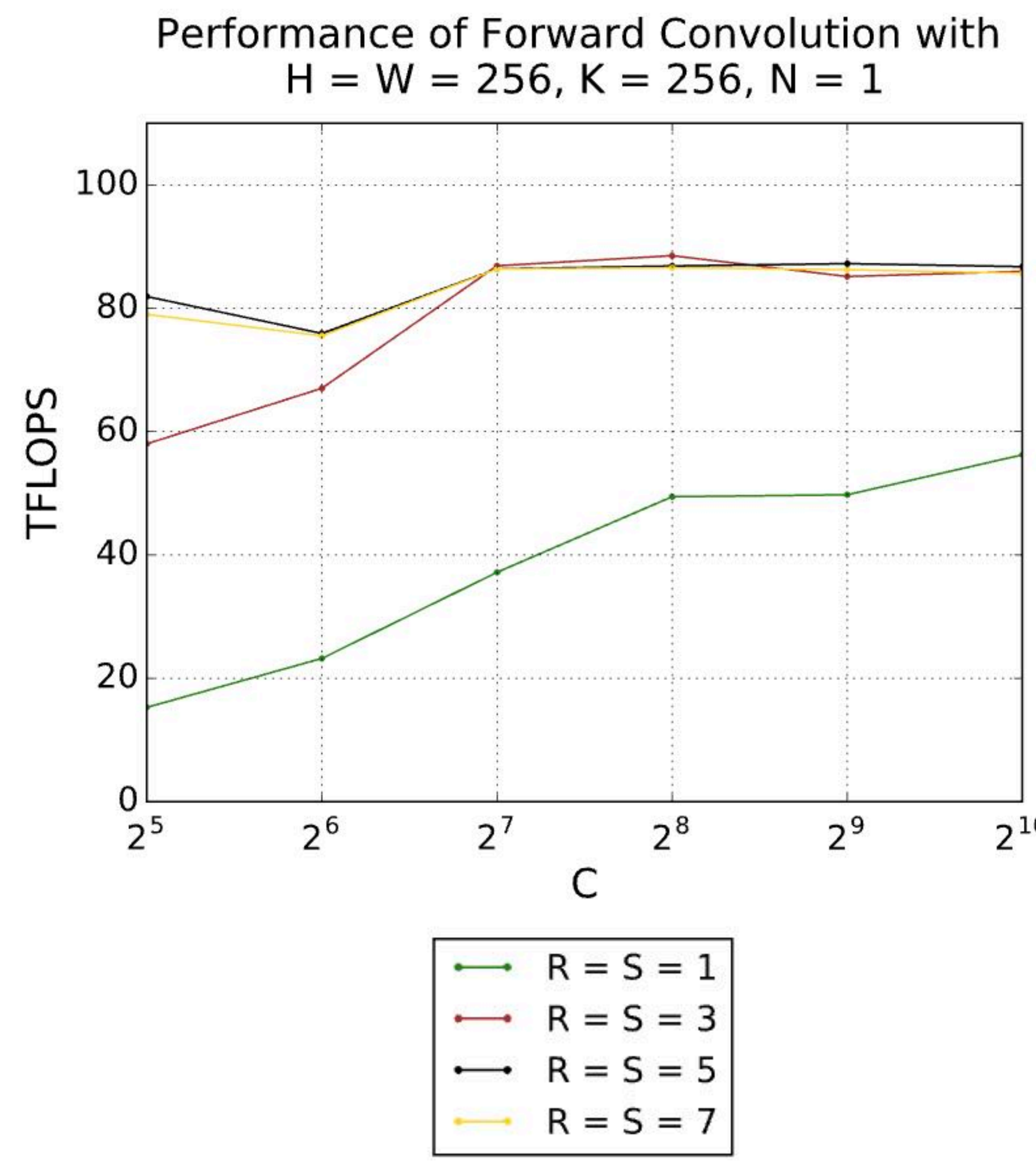
# Higher performance with “more work”

**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)**

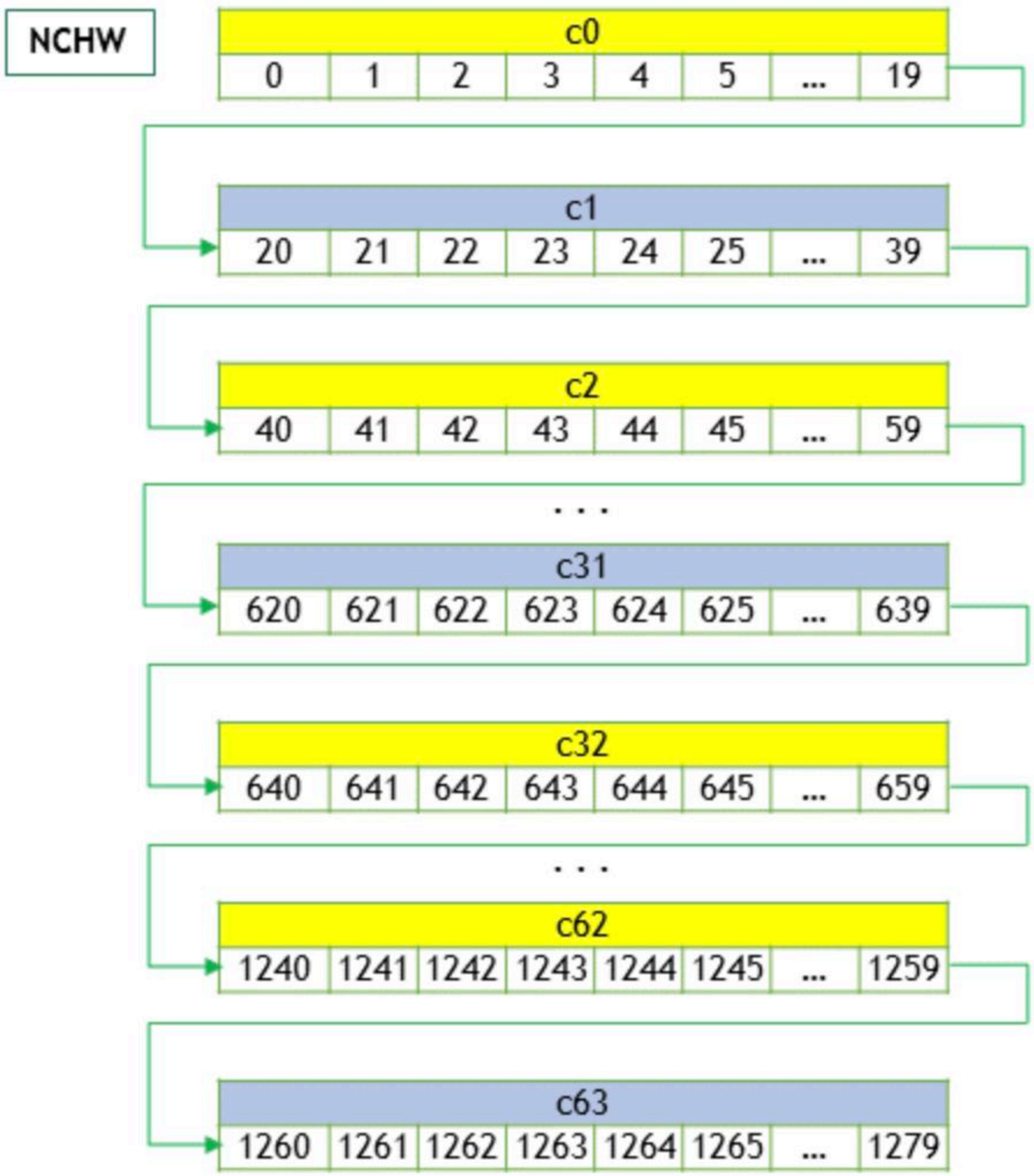


# NCHW data layout

- **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.




**c = 0**

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |

**c = 1**

| | | | |
|---|---|---|---|
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 |
| 36 | 37 | 38 | 39 |

**c = 2**

| | | | |
|---|---|---|---|
| 40 | 41 | 42 | 43 |
| 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 |
| 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 |

. . .

. . .

**c = 62**

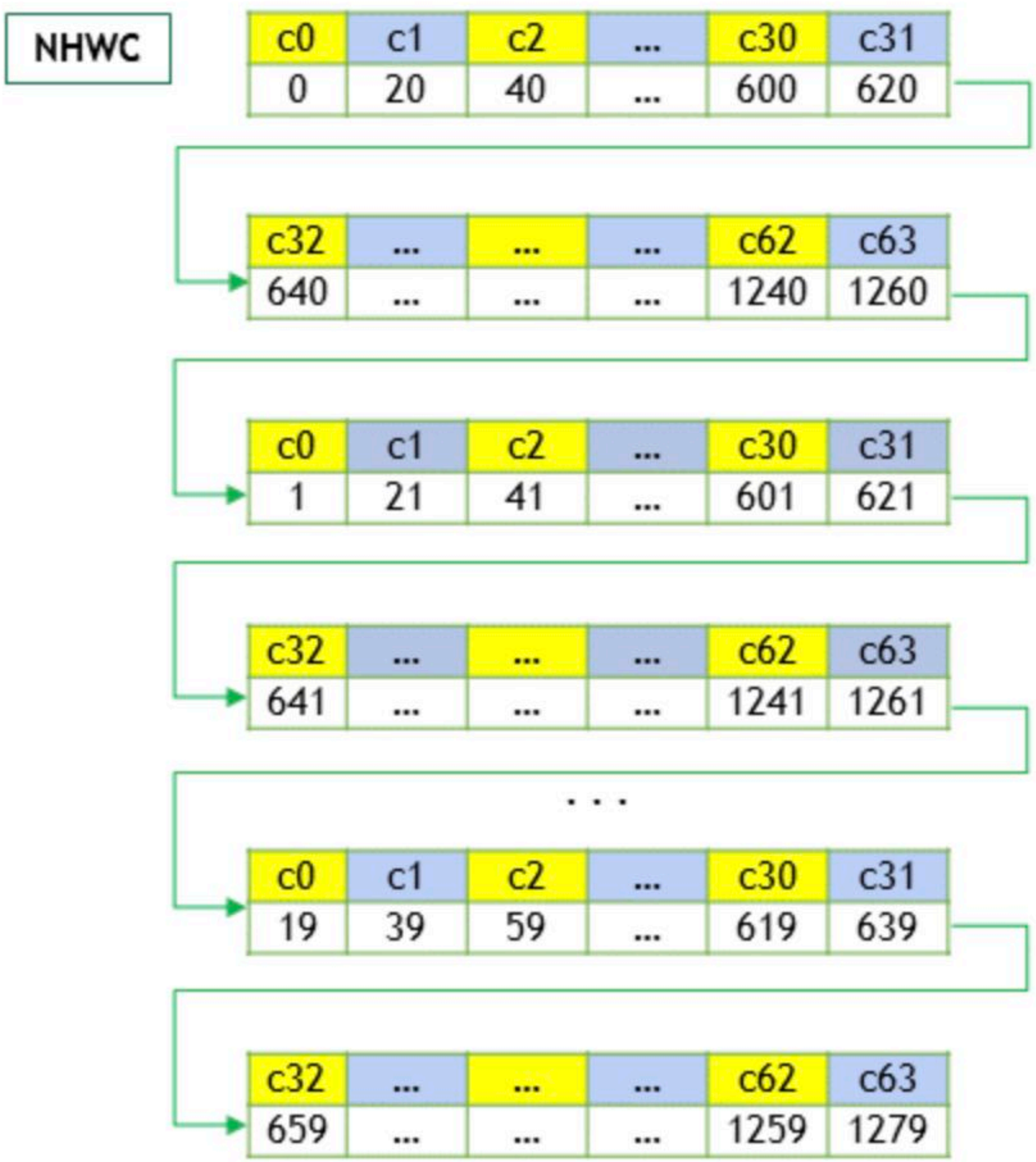
| | | | |
|---|---|---|---|
| 1240 | 1241 | 1242 | 1243 |
| 1244 | 1245 | 1246 | 1247 |
| 1248 | 1249 | 1250 | 1251 |
| 1252 | 1253 | 1254 | 1255 |
| 1256 | 1257 | 1258 | 1259 |

**c = 63**

| | | | |
|---|---|---|---|
| 1260 | 1261 | 1262 | 1263 |
| 1264 | 1265 | 1266 | 1267 |
| 1268 | 1269 | 1270 | 1271 |
| 1272 | 1273 | 1274 | 1275 |
| 1276 | 1277 | 1278 | 1279 |

# 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.




**c = 0**

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |

**c = 1**

| | | | |
|---|---|---|---|
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 |
| 36 | 37 | 38 | 39 |

**c = 2**

| | | | |
|---|---|---|---|
| 40 | 41 | 42 | 43 |
| 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 |
| 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 |

. . .

**c = 62**

| | | | |
|---|---|---|---|
| 1240 | 1241 | 1242 | 1243 |
| 1244 | 1245 | 1246 | 1247 |
| 1248 | 1249 | 1250 | 1251 |
| 1252 | 1253 | 1254 | 1255 |
| 1256 | 1257 | 1258 | 1259 |

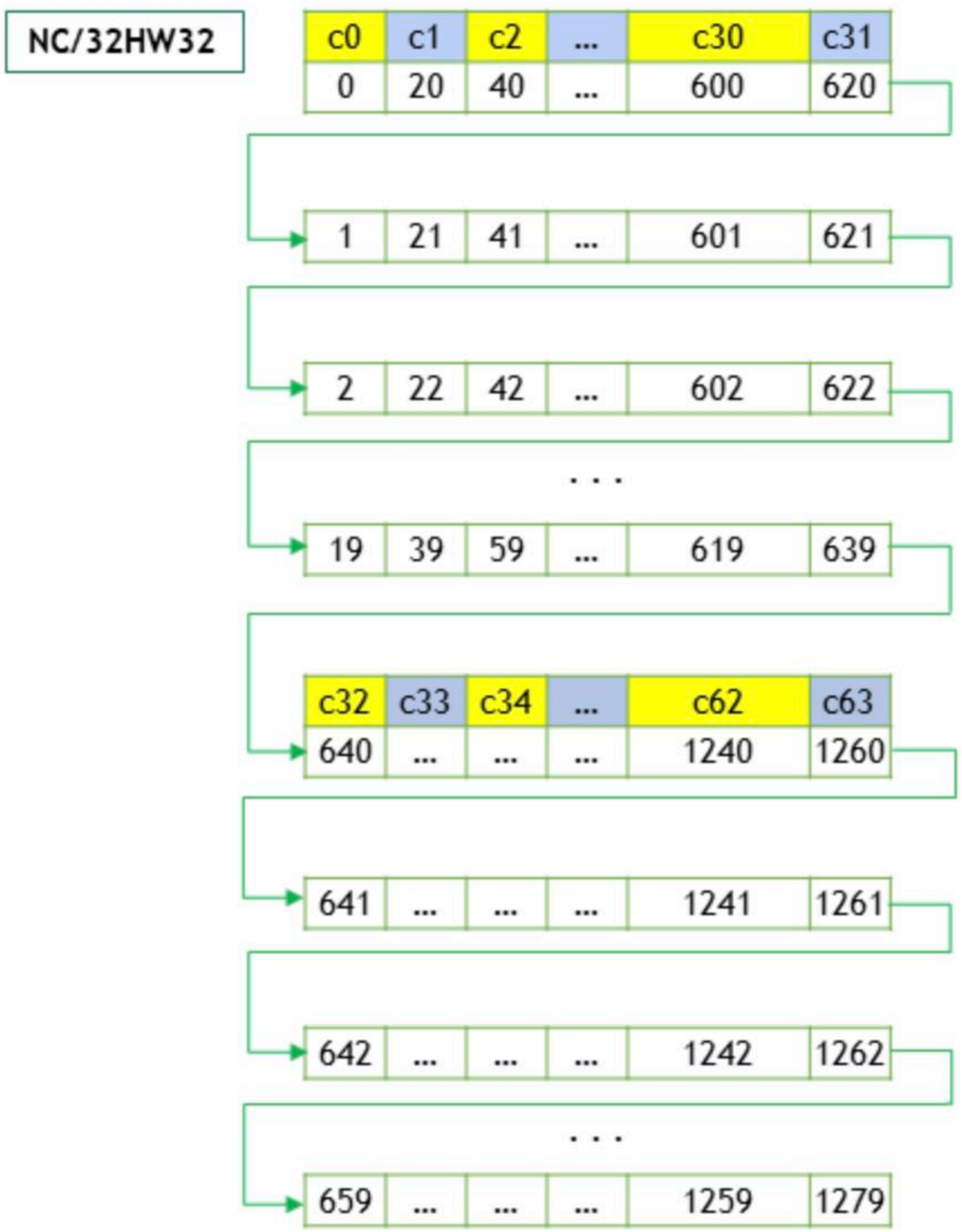
**c = 63**

| | | | |
|---|---|---|---|
| 1260 | 1261 | 1262 | 1263 |
| 1264 | 1265 | 1266 | 1267 |
| 1268 | 1269 | 1270 | 1271 |
| 1272 | 1273 | 1274 | 1275 |
| 1276 | 1277 | 1278 | 1279 |

. . .

# 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.




**c = 0**

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |

**c = 1**

| | | | |
|---|---|---|---|
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 |
| 36 | 37 | 38 | 39 |

**c = 2**

| | | | |
|---|---|---|---|
| 40 | 41 | 42 | 43 |
| 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 |
| 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 |

. . .

. . .

**c = 62**

| | | | |
|---|---|---|---|
| 1240 | 1241 | 1242 | 1243 |
| 1244 | 1245 | 1246 | 1247 |
| 1248 | 1249 | 1250 | 1251 |
| 1252 | 1253 | 1254 | 1255 |
| 1256 | 1257 | 1258 | 1259 |

**c = 63**

| | | | |
|---|---|---|---|
| 1260 | 1261 | 1262 | 1263 |
| 1264 | 1265 | 1266 | 1267 |
| 1268 | 1269 | 1270 | 1271 |
| 1272 | 1273 | 1274 | 1275 |
| 1276 | 1277 | 1278 | 1279 |

# Libraries offering high-performance implementations of key DNN layers

TensorFlow **NN ops**

| | |
|---|---|
| tensorflow::ops::AvgPool | Performs average pooling on the input. |
| tensorflow::ops::AvgPool3D | Performs 3D average pooling on the input. |
| tensorflow::ops::AvgPool3DGrad | Computes gradients of average pooling function. |
| tensorflow::ops::BiasAdd | Adds `bias` to `value`. |
| tensorflow::ops::BiasAddGrad | The backward operation for "BiasAdd" on the "bias" te |
| tensorflow::ops::Conv2D | Computes a 2-D convolution given 4-D `input` and `fi` |
| tensorflow::ops::Conv2DBackpropFilter | Computes the gradients of convolution with respect t |
| tensorflow::ops::Conv2DBackpropInput | Computes the gradients of convolution with respect t |
| tensorflow::ops::Conv3D | Computes a 3-D convolution given 5-D `input` and `fi` |
| tensorflow::ops::Conv3DBackpropFilterV2 | Computes the gradients of 3-D convolution with respe |
| tensorflow::ops::Conv3DBackpropInputV2 | Computes the gradients of 3-D convolution with respe |
| tensorflow::ops::DataFormatDimMap | Returns the dimension index in the destination data f |
| tensorflow::ops::DataFormatVecPermute | Permute input tensor from `src_format` to `dst_for` |
| tensorflow::ops::DepthwiseConv2dNative | Computes a 2-D depthwise convolution given 4-D inp tensors. |
| tensorflow::ops::DepthwiseConv2dNativeBackpropFilter | Computes the gradients of depthwise convolution wi |
| tensorflow::ops::DepthwiseConv2dNativeBackpropInput | Computes the gradients of depthwise convolution wi |
| tensorflow::ops::Dilation2D | Computes the grayscale dilation of 4-D `input` and 3- |
| tensorflow::ops::Dilation2DBackpropFilter | Computes the gradient of morphological 2-D dilation filter. |
| tensorflow::ops::Dilation2DBackpropInput | Computes the gradient of morphological 2-D dilation input. |
| tensorflow::ops::Elu | Computes exponential linear: `exp(features) - 1` otherwise. |
| tensorflow::ops::FractionalAvgPool | Performs fractional average pooling on the input. |
| tensorflow::ops::FractionalMaxPool | Performs fractional max pooling on the input. |
| tensorflow::ops::FusedBatchNorm | Batch normalization. |
| tensorflow::ops::FusedBatchNormGrad | Gradient for batch normalization. |
| tensorflow::ops::FusedBatchNormGradV2 | Gradient for batch normalization. |
| tensorflow::ops::FusedBatchNormGradV3 | Gradient for batch normalization. |
| tensorflow::ops::FusedBatchNormV2 | Batch normalization. |
| tensorflow::ops::FusedBatchNormV3 | Batch normalization. |
| tensorflow::ops::FusedPadConv2D | Performs a padding as a preprocess during a convolution. |
| tensorflow::ops::FusedResizeAndPadConv2D | Performs a resize and padding as a preprocess during a convolution. |
| tensorflow::ops::InTopK | Says whether the targets are in the top K predictions. |
| tensorflow::ops::InTopKV2 | Says whether the targets are in the top K predictions. |
| tensorflow::ops::L2Loss | L2 Loss. |
| tensorflow::ops::LRN | Local Response Normalization. |
| tensorflow::ops::LogSoftmax | Computes log softmax activations. |
| tensorflow::ops::MaxPool | Performs max pooling on the input. |
| tensorflow::ops::MaxPool3D | Performs 3D max pooling on the input. |
| tensorflow::ops::MaxPool3DGrad | Computes gradients of 3D max pooling function. |
| tensorflow::ops::MaxPool3DGradGrad | Computes second-order gradients of the maxpooling function. |
| tensorflow::ops::MaxPoolGradGrad | Computes second-order gradients of the maxpooling function. |
| tensorflow::ops::MaxPoolGradGradV2 | Computes second-order gradients of the maxpooling function. |
| tensorflow::ops::MaxPoolGradGradWithArgmax | Computes second-order gradients of the maxpooling function. |
| tensorflow::ops::MaxPoolGradV2 | Computes gradients of the maxpooling function. |
| tensorflow::ops::MaxPoolV2 | Performs max pooling on the input. |
| tensorflow::ops::MaxPoolWithArgmax | Performs max pooling on the input and outputs both max values and indices. |
| tensorflow::ops::NthElement | Finds values of the n-th order statistic for the last dimension. |
| tensorflow::ops::QuantizedAvgPool | Produces the average pool of the input tensor for quantized types. |
| tensorflow::ops:: QuantizedBatchNormWithGlobalNormalization | Quantized Batch normalization. |
| tensorflow::ops::QuantizedBiasAdd | Adds Tensor 'bias' to Tensor 'input' for Quantized types. |
| tensorflow::ops::QuantizedConv2D | Computes a 2D convolution given quantized 4D input and filter tensors. |
| tensorflow::ops::QuantizedMaxPool | Produces the max pool of the input tensor for quantized types. |

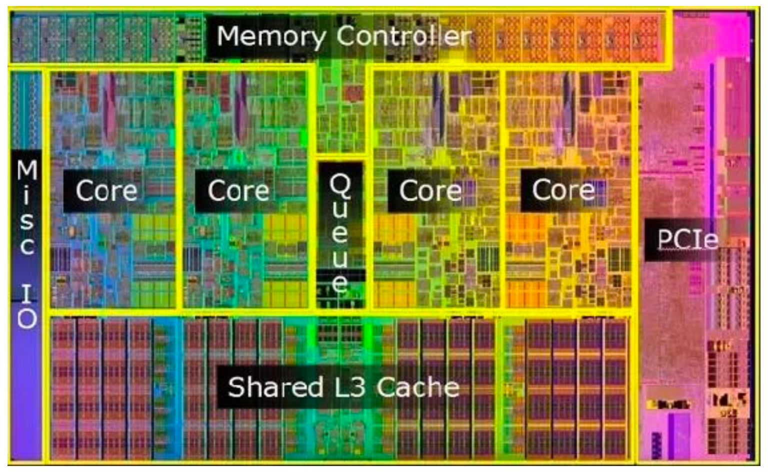
# Libraries offering high-performance implementations of key DNN layers

TensorFlow **NN ops**

| | |
|---|---|
| tensorflow::ops::AvgPool | Performs average pooling on the input. |
| tensorflow::ops::AvgPool3D | Performs 3D average pooling on the input. |
| tensorflow::ops::AvgPool3DGrad | Computes gradients of average pooling function. |
| tensorflow::ops::BiasAdd | Adds `bias` to `value`. |
| tensorflow::ops::BiasAddGrad | The backward operation for "BiasAdd" on the "bias" te |
| tensorflow::ops::Conv2D | Computes a 2-D convolution given 4-D `input` and `fi` |
| tensorflow::ops::Conv2DBackpropFilter | Computes the gradients of convolution with respect t |
| tensorflow::ops::Conv2DBackpropInput | Computes the gradients of convolution with respect t |
| tensorflow::ops::Conv3D | Computes a 3-D convolution given 5-D `input` and `fi` |
| tensorflow::ops::Conv3DBackpropFilterV2 | Computes the gradients of 3-D convolution with resp |
| tensorflow::ops::Conv3DBackpropInputV2 | Computes the gradients of 3-D convolution with resp |
| tensorflow::ops::DataFormatDimMap | Returns the dimension index in the destination data f |
| tensorflow::ops::DataFormatVecPermute | Permute input tensor from `src_format` to `dst_for` |
| tensorflow::ops::DepthwiseConv2dNative | Computes a 2-D depthwise convolution given 4-D `inp` tensors. |
| tensorflow::ops::DepthwiseConv2dNativeBackpropFilter | Computes the gradients of depthwise convolution wi |
| tensorflow::ops::DepthwiseConv2dNativeBackpropInput | Computes the gradients of depthwise convolution wi |
| tensorflow::ops::Dilation2D | Computes the grayscale dilation of 4-D `input` and 3- |
| tensorflow::ops::Dilation2DBackpropFilter | Computes the gradient of morphological 2-D dilation filter. |
| tensorflow::ops::Dilation2DBackpropInput | Computes the gradient of morphological 2-D dilation input. |
| tensorflow::ops::Elu | Computes exponential linear: `exp(features) - 1` otherwise. |
| tensorflow::ops::FractionalAvgPool | Performs fractional average pooling on the input. |
| tensorflow::ops::FractionalMaxPool | Performs fractional max pooling on the input. |
| tensorflow::ops::FusedBatchNorm | Batch normalization. |

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 1. MobileNet Body Architecture

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

# 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

```
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

```
cudnnStatus_t cudnnConvolutionBiasActivationForward(
    cudnnHandle_t                      handle,
    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)
```

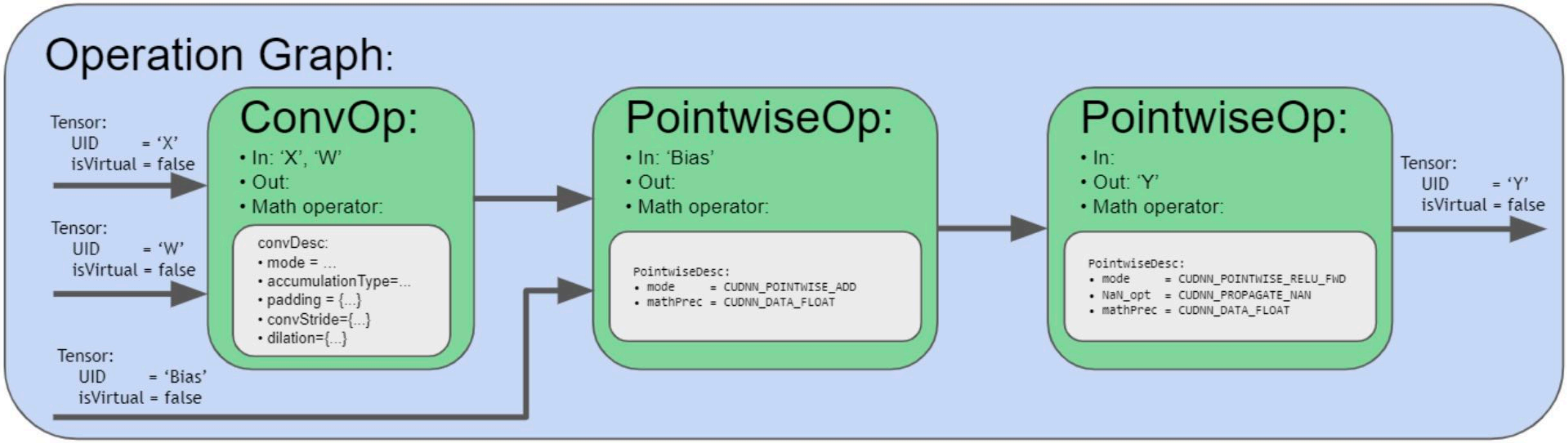
This function applies a bias and then an activation to the convolutions or cross-correlations of cudnnConvolutionForward(), returning results in `y`. The full computation follows the equation `y = act ( alpha1 * conv(x) + alpha2 * z + bias )`.

## Tensorflow:

| | |
|---|---|
| tensorflow::ops::FusedBatchNorm | Batch normalization. |

| | |
|---|---|
| tensorflow::ops::FusedResizeAndPadConv2D | Performs a resize and padding as a preprocess during a convolution. |

# Fusion example: CUDNN “backend"

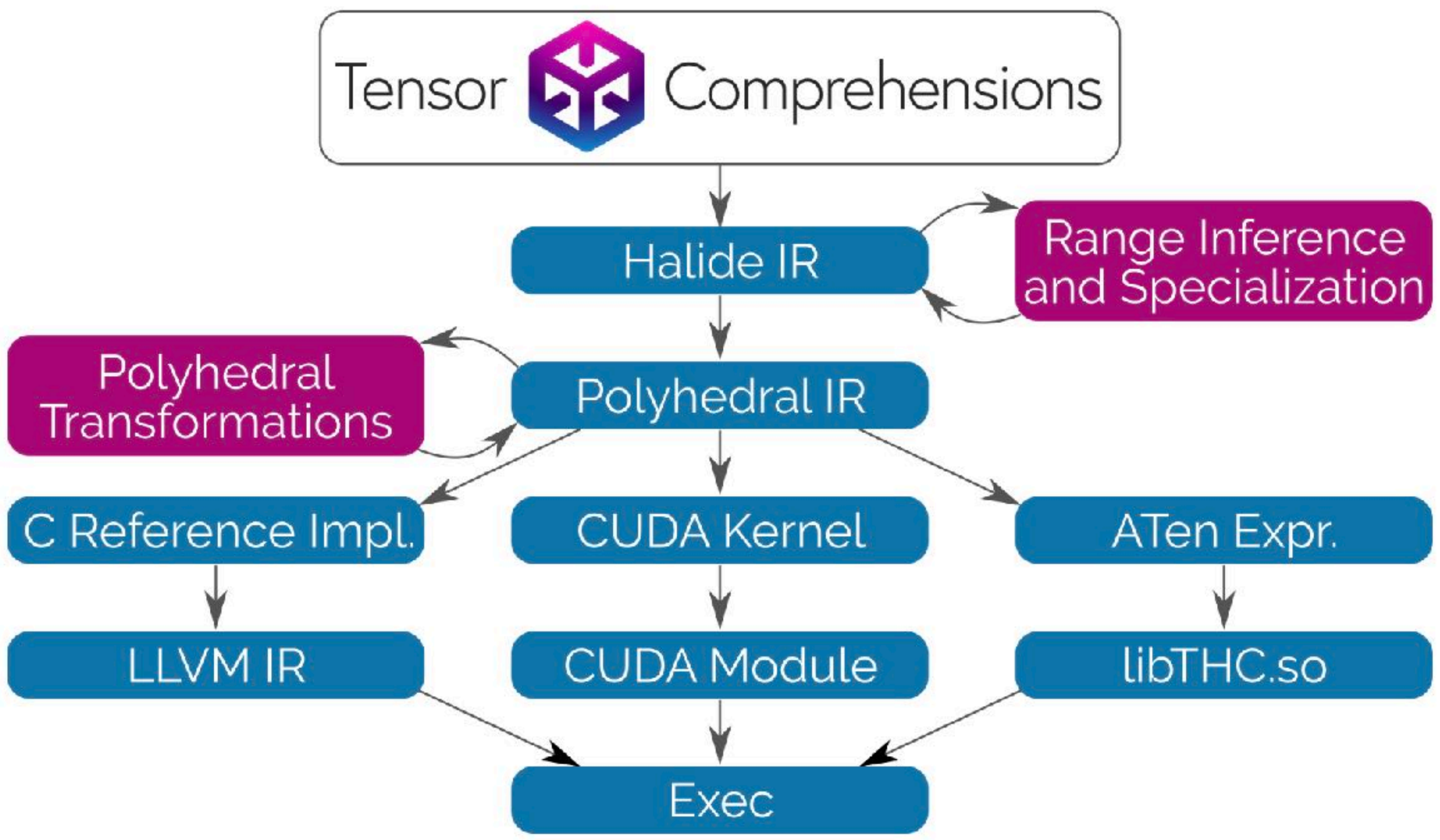



Note for operation fusion use cases, there are two different mechanisms in cuDNN to support them. First, there are engines containing offline compiled kernels that can support certain fusion patterns. These engines try to match the user provided operation graph with their supported fusion pattern. If there is a match, then that particular engine is deemed suitable for this use case. In addition, there are also runtime fusion engines to be made available in the upcoming releases. Instead of passively matching the user graph, such engines actively walk the graph and assemble code blocks to form a CUDA kernel and compile on the fly. Such runtime fusion engines are much more flexible in its range of support. However, because the construction of the execution plans requires runtime compilation, the one-time CPU overhead is higher than the other engines.

**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




**MLIR**

Multi-Level IR Compiler Framework

**tvm** Open Deep Learning Compiler Stack

license Apache 2.0 | build passing

Documentation | Contributors | Community | Release Notes

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.

Deep Learning Compiler

nGraph

NVIDIA TensorRT

Programmable Inference Accelerator

# 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†*

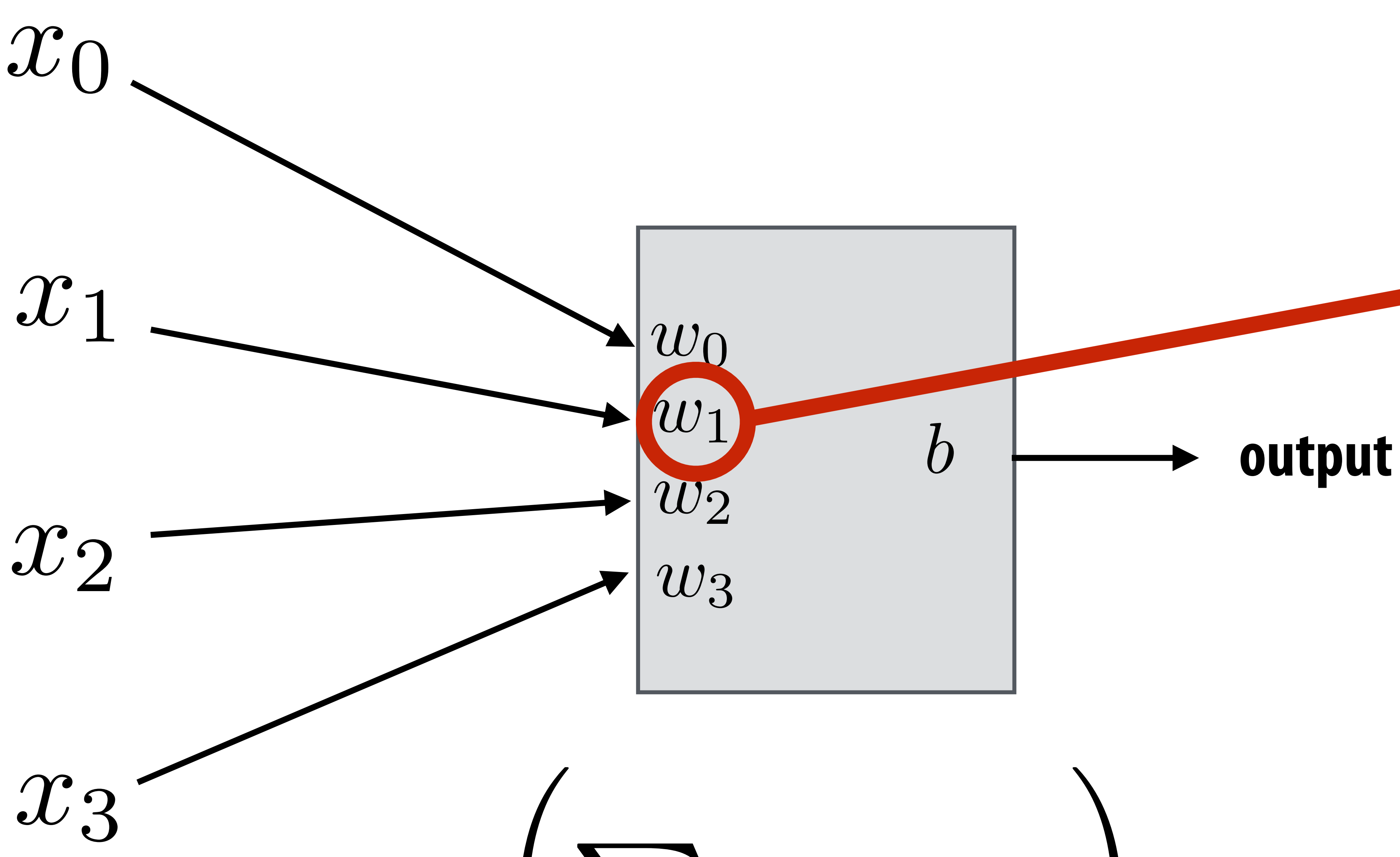
Allen Institute for AI†, University of Washington*
{mohammadr,vicenteor}@allenai.org
{pjreddie,ali}@cs.washington.edu

**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

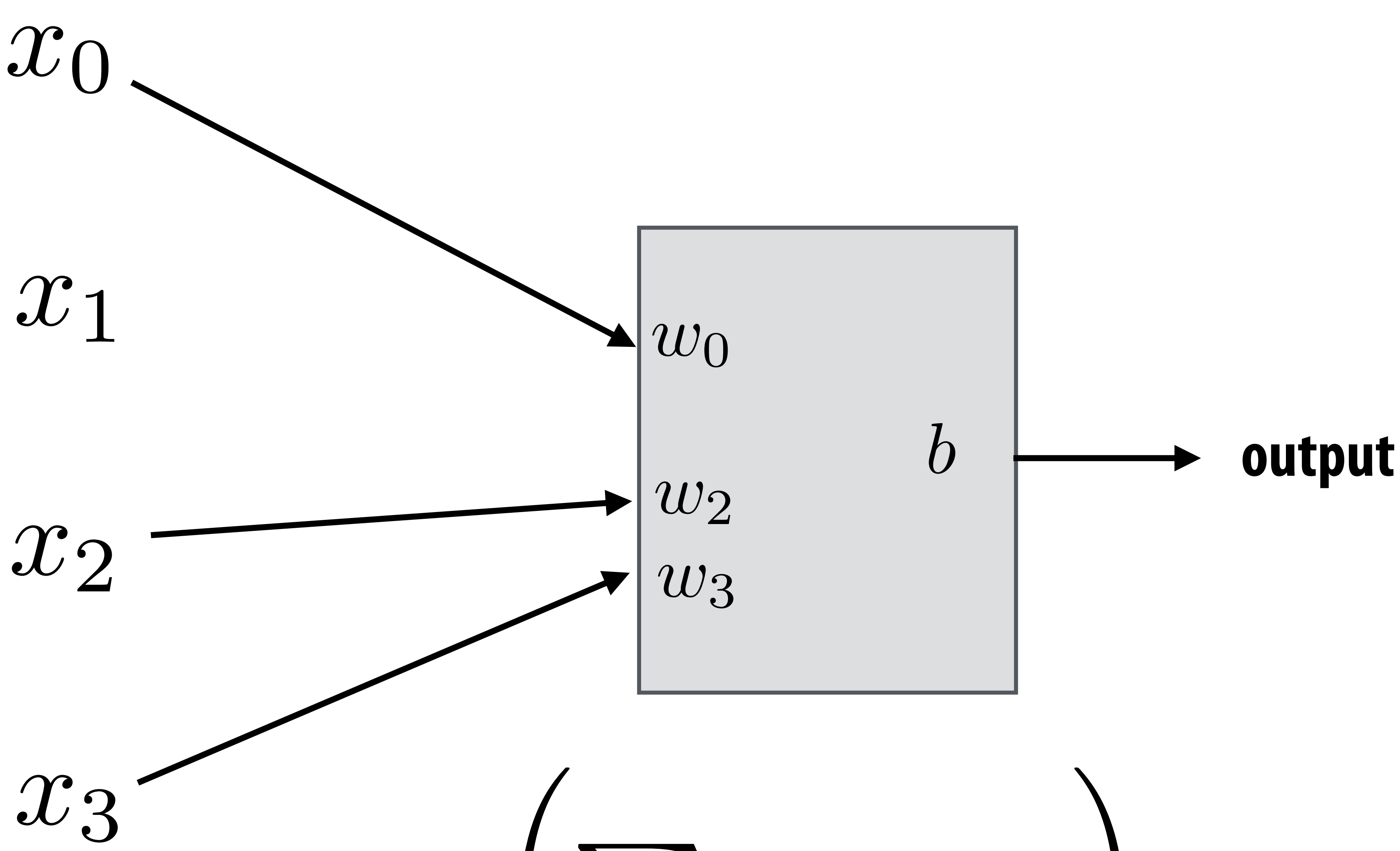



If weight is near zero, then corresponding input has little impact on output of neuron.

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

$$f(x) = max(0, x)$$

# "Pruning" (sparsifying) a network




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

$$f(x) = max(0, x)$$

**Idea: prune connections with near zero weight**

**Remove entire units if all connections are pruned.**

# 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
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.8 | 0 | 0 | 0.5 | 0 | 0 | 0 | 0 | 1.1 | ... |

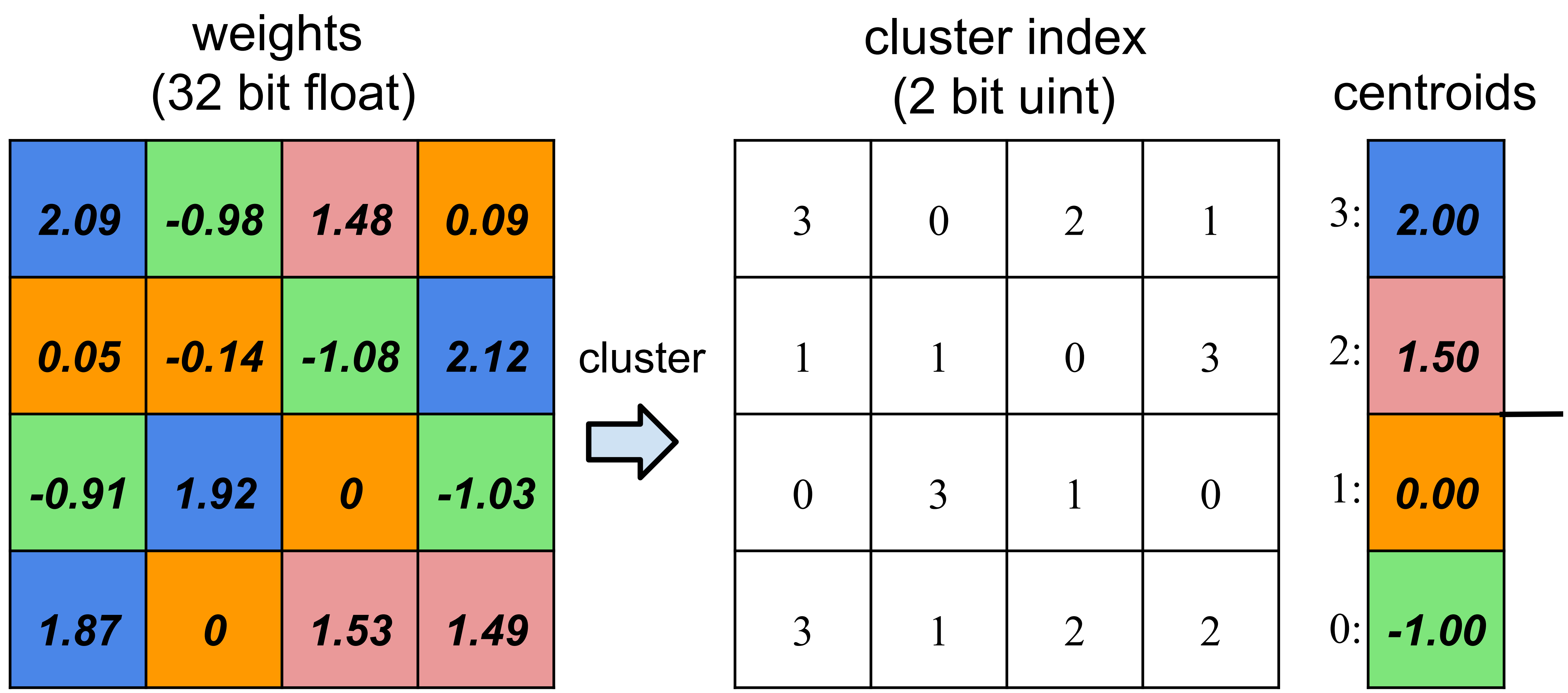
**Reduce storage over head of indices by delta encoding them to fit in 8 bits**

```
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)**

[Figure credit: Han ICLR16]

# VGG-16 sparsification

[Han ICLR16]

**Large savings in fully connected layers due to combination of pruning, quantization, Huffman encoding ***

| Layer | #Weights | Weights% (P) | Weigh bits (P+Q) | Weight bits (P+Q+H) | Index bits (P+Q) | Index bits (P+Q+H) | Compress rate (P+Q) | Compress rate (P+Q+H) |
|---|---|---|---|---|---|---|---|---|
| conv1_1 | 2K | 58% | 8 | 6.8 | 5 | 1.7 | 40.0% | 29.97% |
| conv1_2 | 37K | 22% | 8 | 6.5 | 5 | 2.6 | 9.8% | 6.99% |
| conv2_1 | 74K | 34% | 8 | 5.6 | 5 | 2.4 | 14.3% | 8.91% |
| conv2_2 | 148K | 36% | 8 | 5.9 | 5 | 2.3 | 14.7% | 9.31% |
| conv3_1 | 295K | 53% | 8 | 4.8 | 5 | 1.8 | 21.7% | 11.15% |
| conv3_2 | 590K | 24% | 8 | 4.6 | 5 | 2.9 | 9.7% | 5.67% |
| conv3_3 | 590K | 42% | 8 | 4.6 | 5 | 2.2 | 17.0% | 8.96% |
| conv4_1 | 1M | 32% | 8 | 4.6 | 5 | 2.6 | 13.1% | 7.29% |
| conv4_2 | 2M | 27% | 8 | 4.2 | 5 | 2.9 | 10.9% | 5.93% |
| conv4_3 | 2M | 34% | 8 | 4.4 | 5 | 2.5 | 14.0% | 7.47% |
| conv5_1 | 2M | 35% | 8 | 4.7 | 5 | 2.5 | 14.3% | 8.00% |
| conv5_2 | 2M | 29% | 8 | 4.6 | 5 | 2.7 | 11.7% | 6.52% |
| conv5_3 | 2M | 36% | 8 | 4.6 | 5 | 2.3 | 14.8% | 7.79% |
| fc6 | 103M | 4% | 5 | 3.6 | 5 | 3.5 | 1.6% | 1.10% |
| fc7 | 17M | 4% | 5 | 4 | 5 | 4.3 | 1.5% | 1.25% |
| fc8 | 4M | 23% | 5 | 4 | 5 | 3.4 | 7.1% | 5.24% |
| Total | 138M | 7.5%(13×) | 6.4 | 4.1 | 5 | 3.1 | 3.2% (**31×**) | 2.05% (**49×**) |

**P = connection pruning (prune low weight connections)**

**Q = quantize surviving weights (using shared weights)**

**H = Huffman encode**

**ImageNet Image Classification Performance**

| | Top-1 Error | Top-5 Error | Model size | |
|---|---|---|---|---|
| VGG-16 Ref | 31.50% | 11.32% | 552 MB | |
| VGG-16 Compressed | 31.17% | 10.91% | **11.3 MB** | 49× |

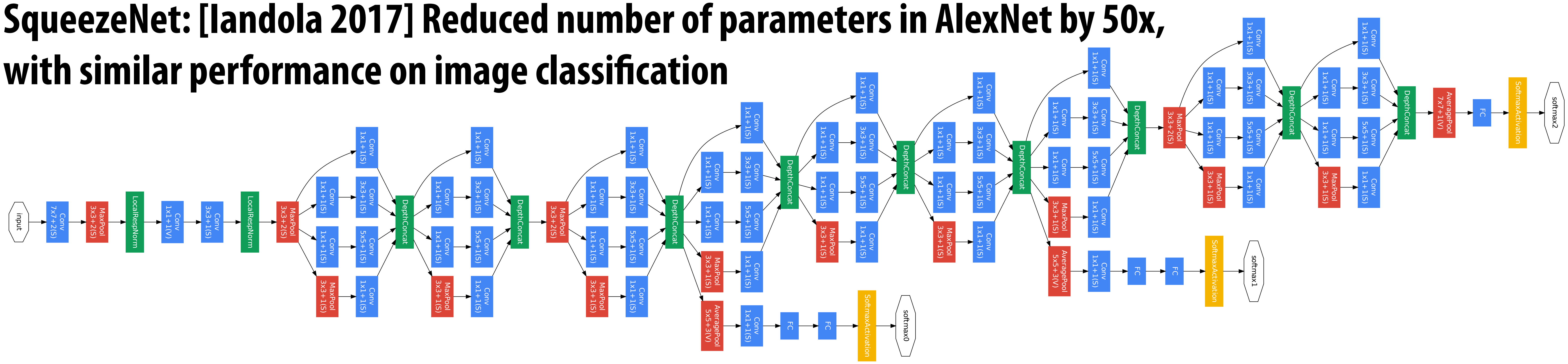
*** Benefits of automatic pruning apply mainly to fully connected layers, but unfortunately many modern networks are dominated by costs of convolutional layers**

# This a great example of non-domain-specific vs. domain-specific approach to innovation

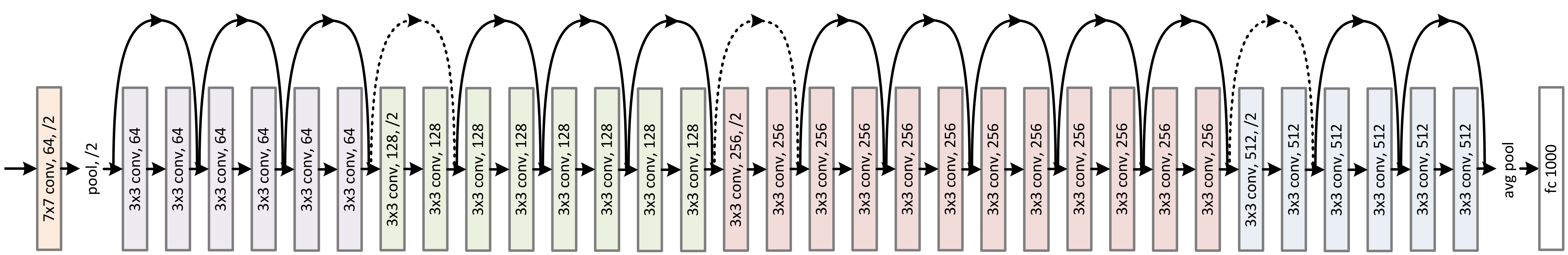
# Leveraging ML domain-knowledge: engineering more efficient topologies (aka better algorithm design)

- 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

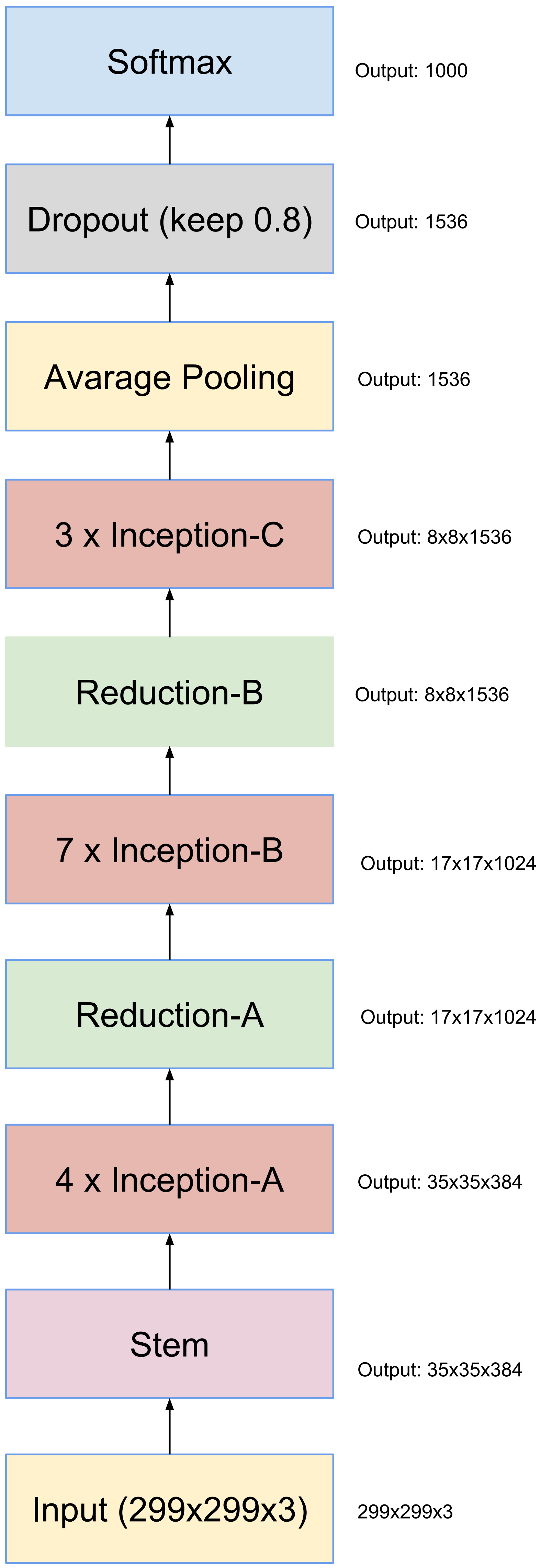


Inception v1 (GoogleLeNet) — 27 total layers, 7M parameters

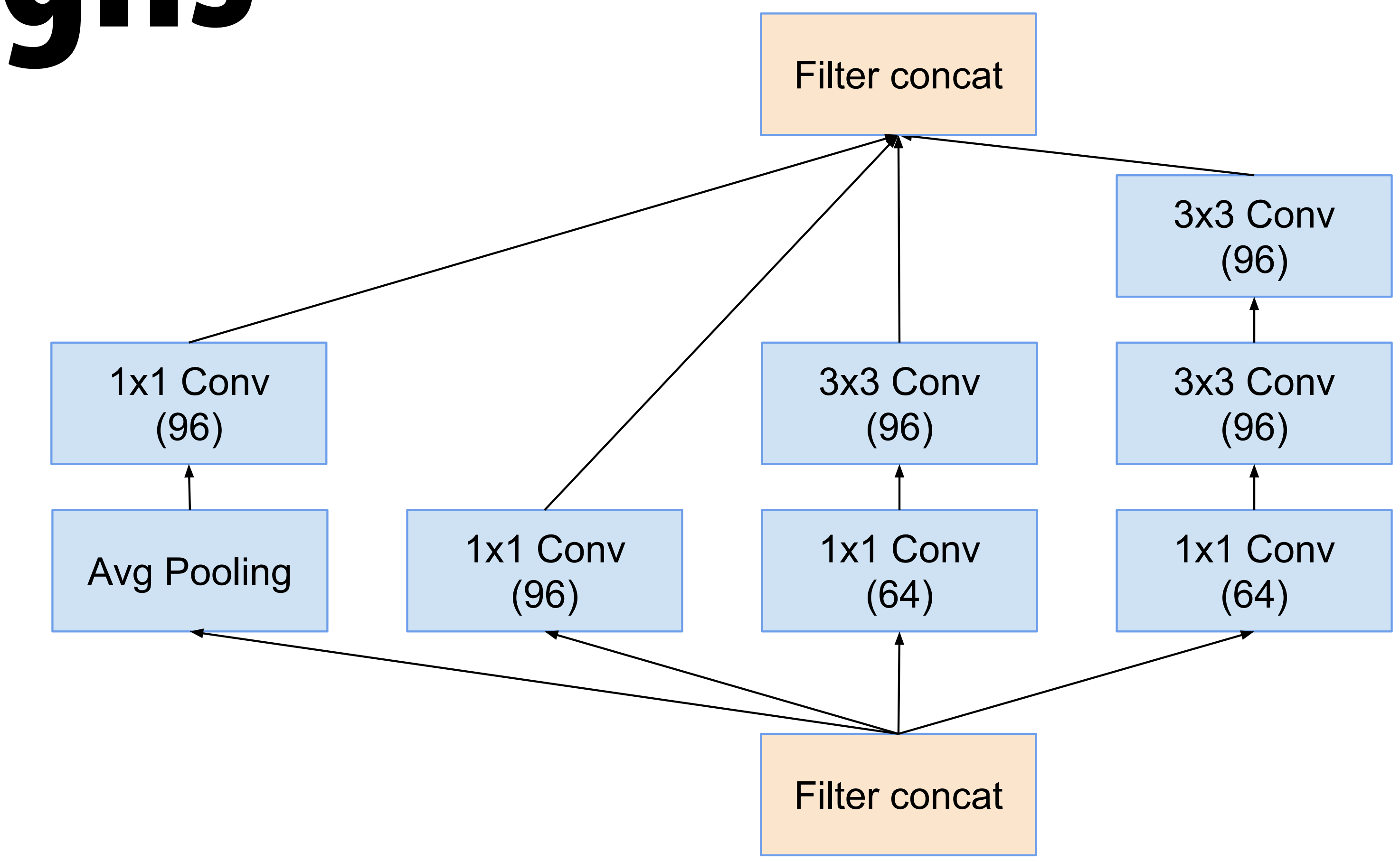



ResNet (34 layer version)

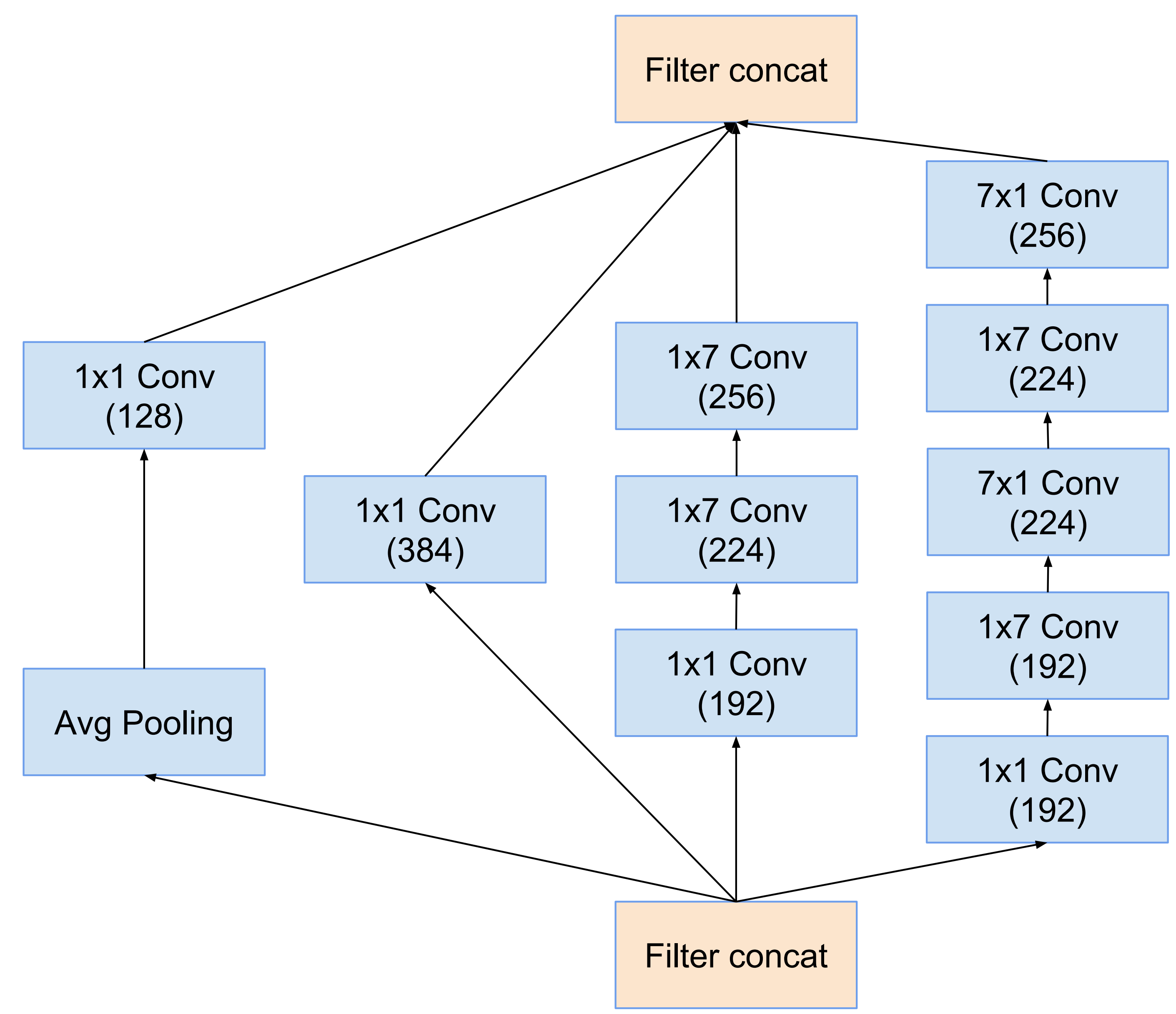
# Modular network designs




Inception v4

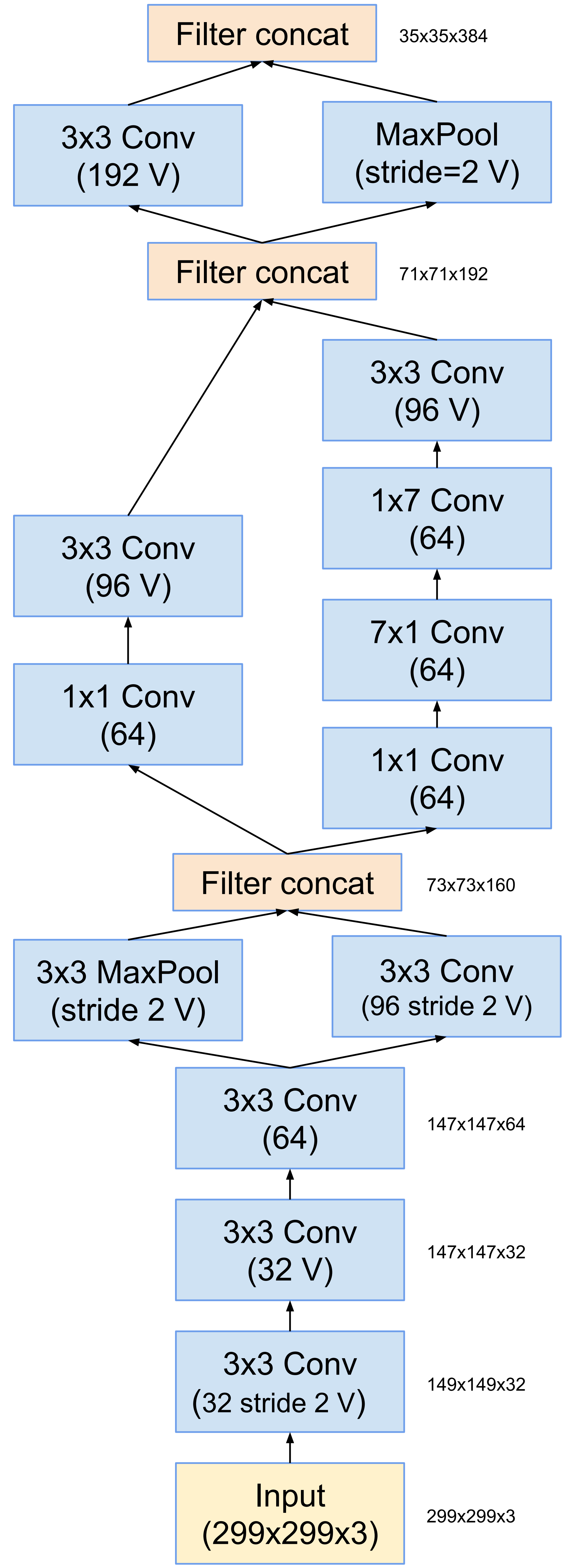



A block

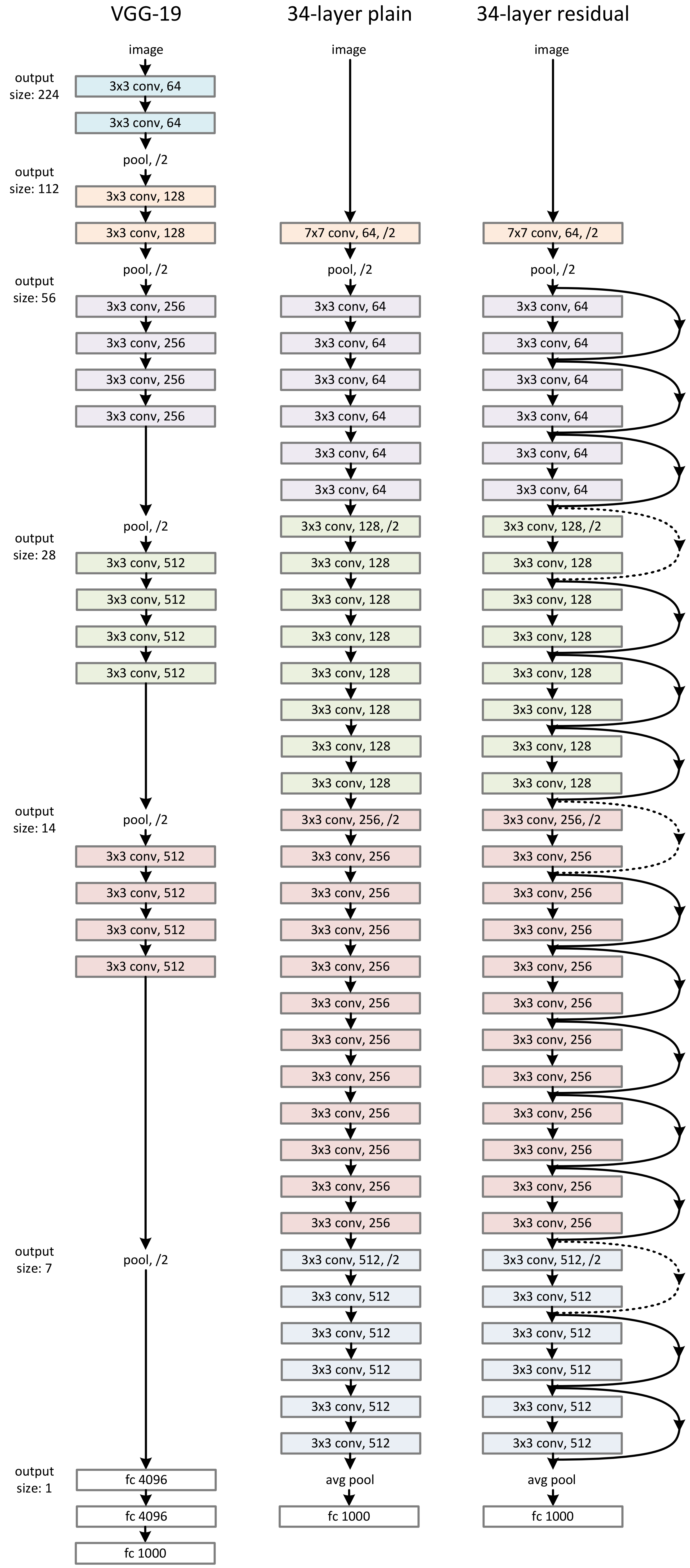



B block

# Inception stem



# ResNet



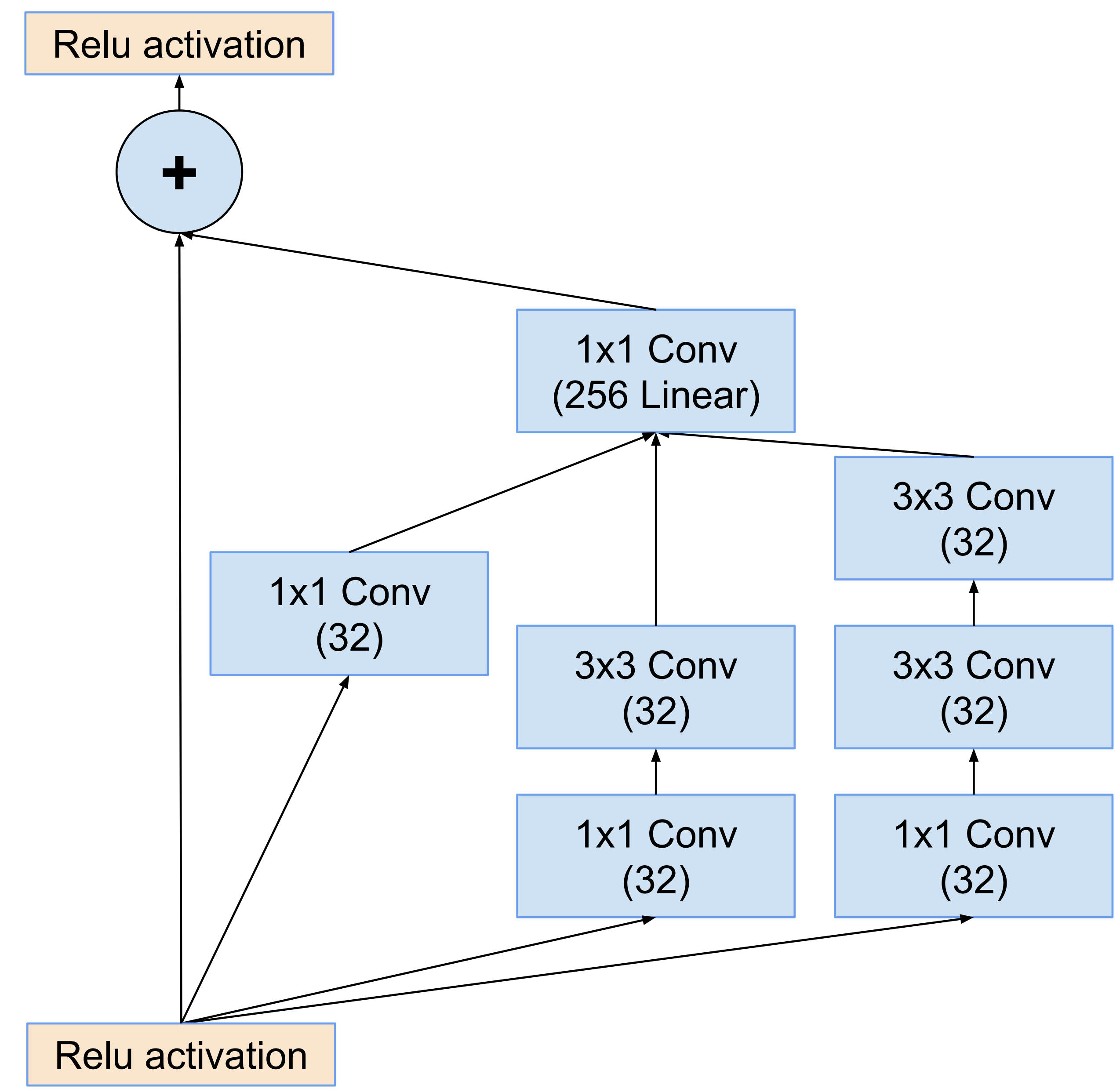
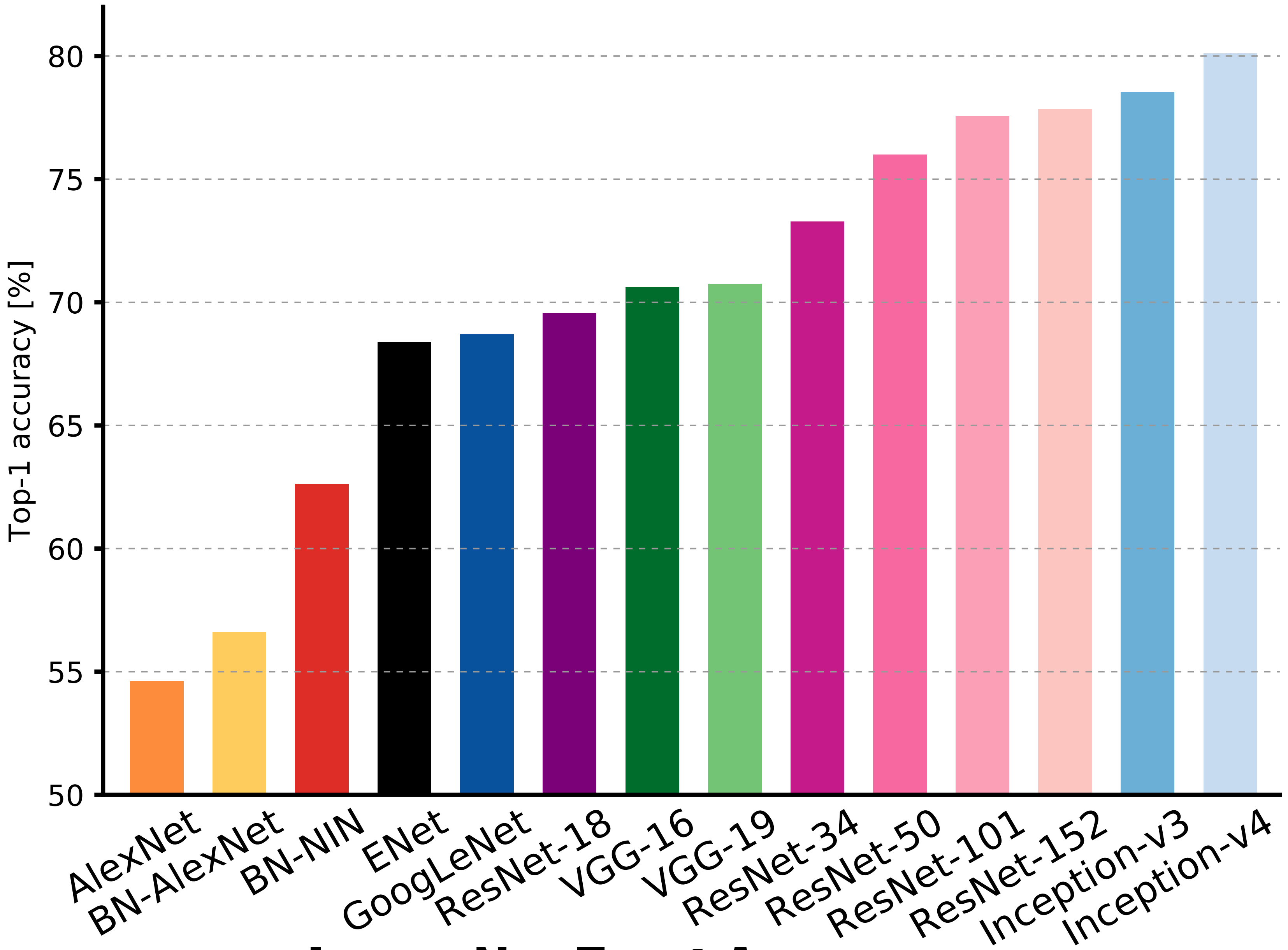
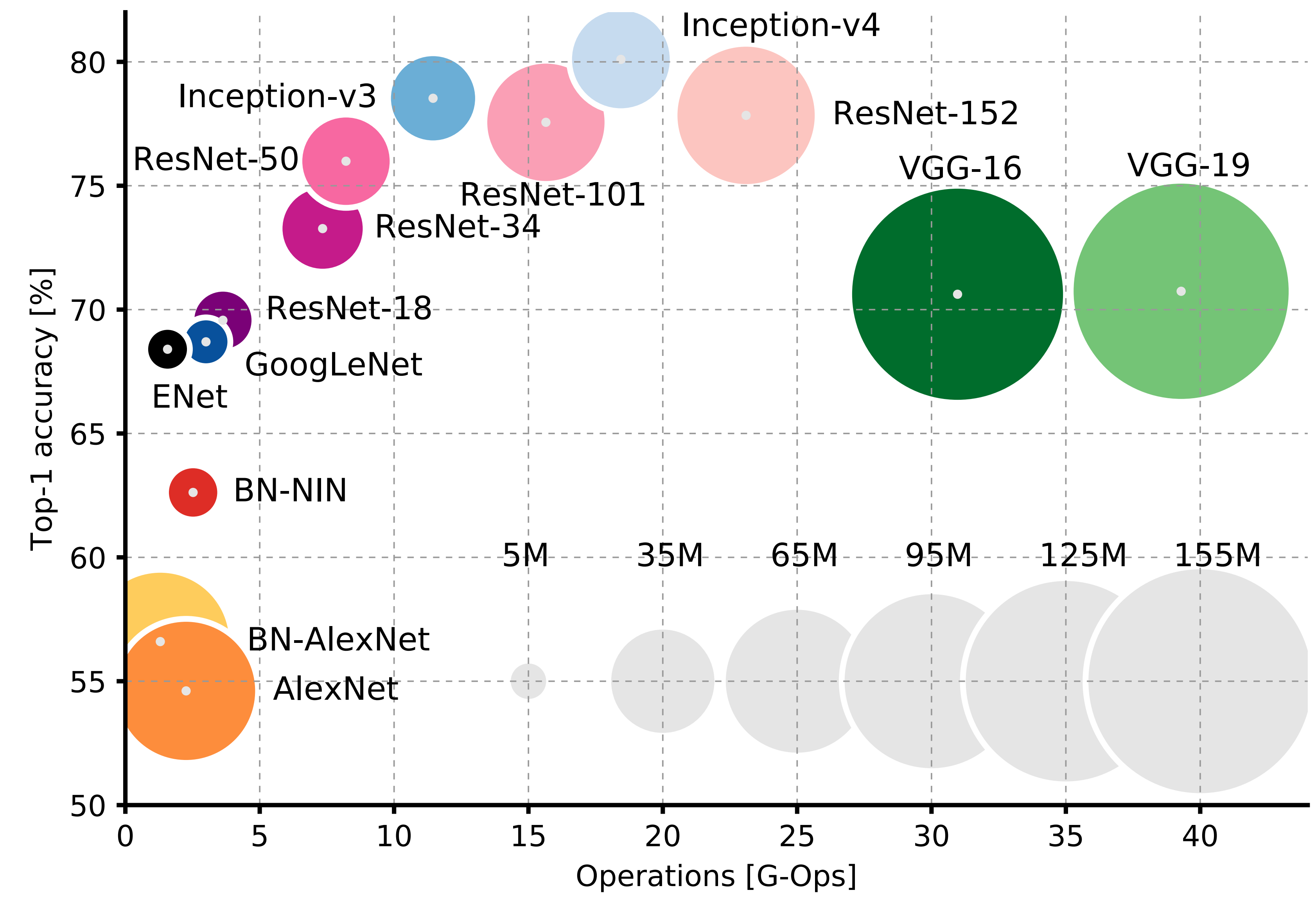




Figure 10. The schema for 35 × 35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

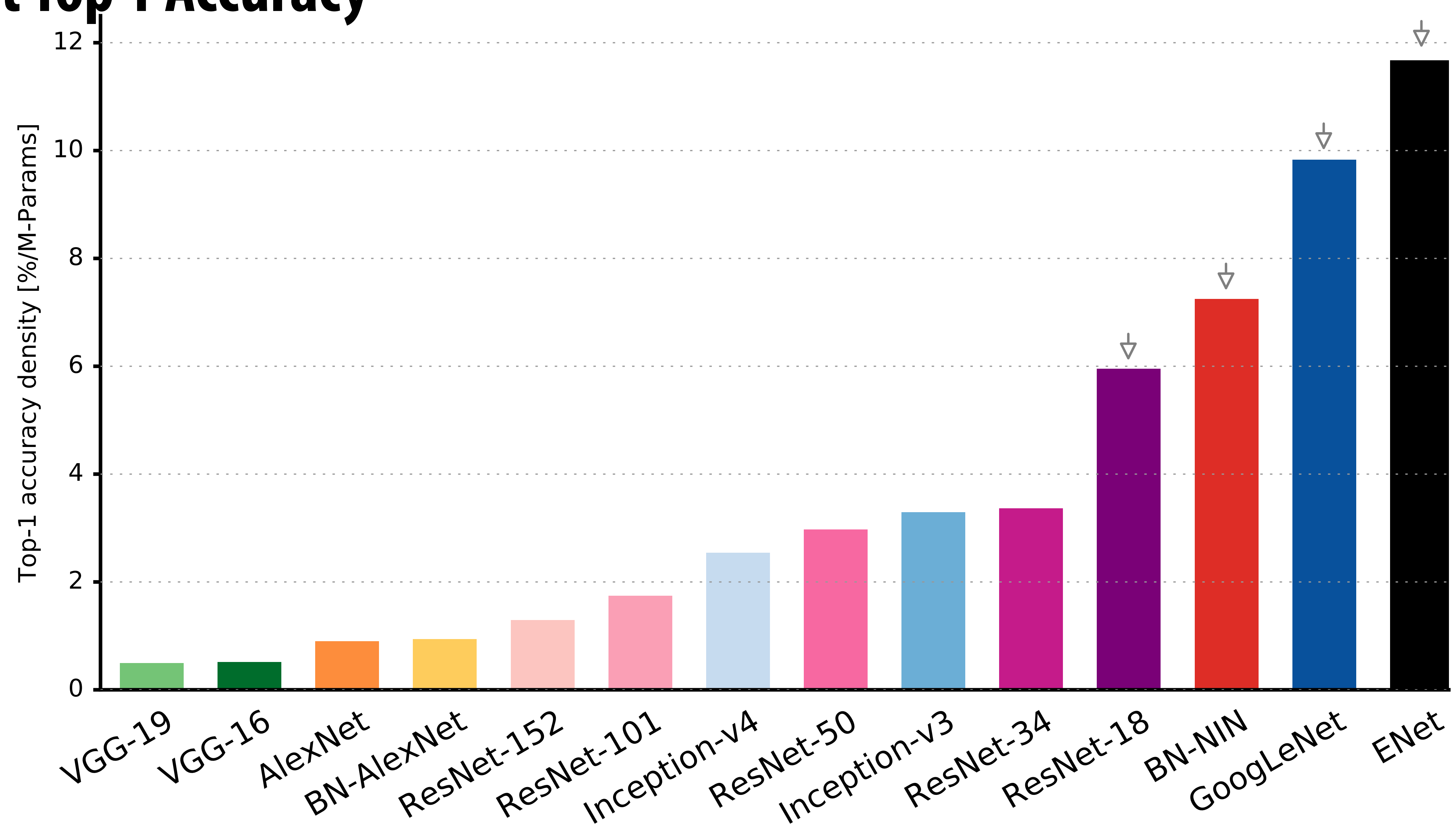
# Effect of topology innovation




**ImageNet Top 1 Accuracy**




**Flops cost (area of circle is # params)**




**Accuracy (points) per flop**

**Figure credit: Canziani et al 2017**

# Improving accuracy/cost (image classification)

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

| | ImageNet Top-1 Accuracy | Num Params | Cost/image (MADDs) | |
|---|---|---|---|---|
| VGG-16 | 71.5% | 138M | 15B | [2014] |
| GoogleNet | 70% | 6.8M | 1.5B | [2015] |
| ResNet-18 | 73% * | 11.7M | 1.8B | [2016] |
| MobileNet-224 | 70.5% | 4.2M | 0.6B | [2017] |

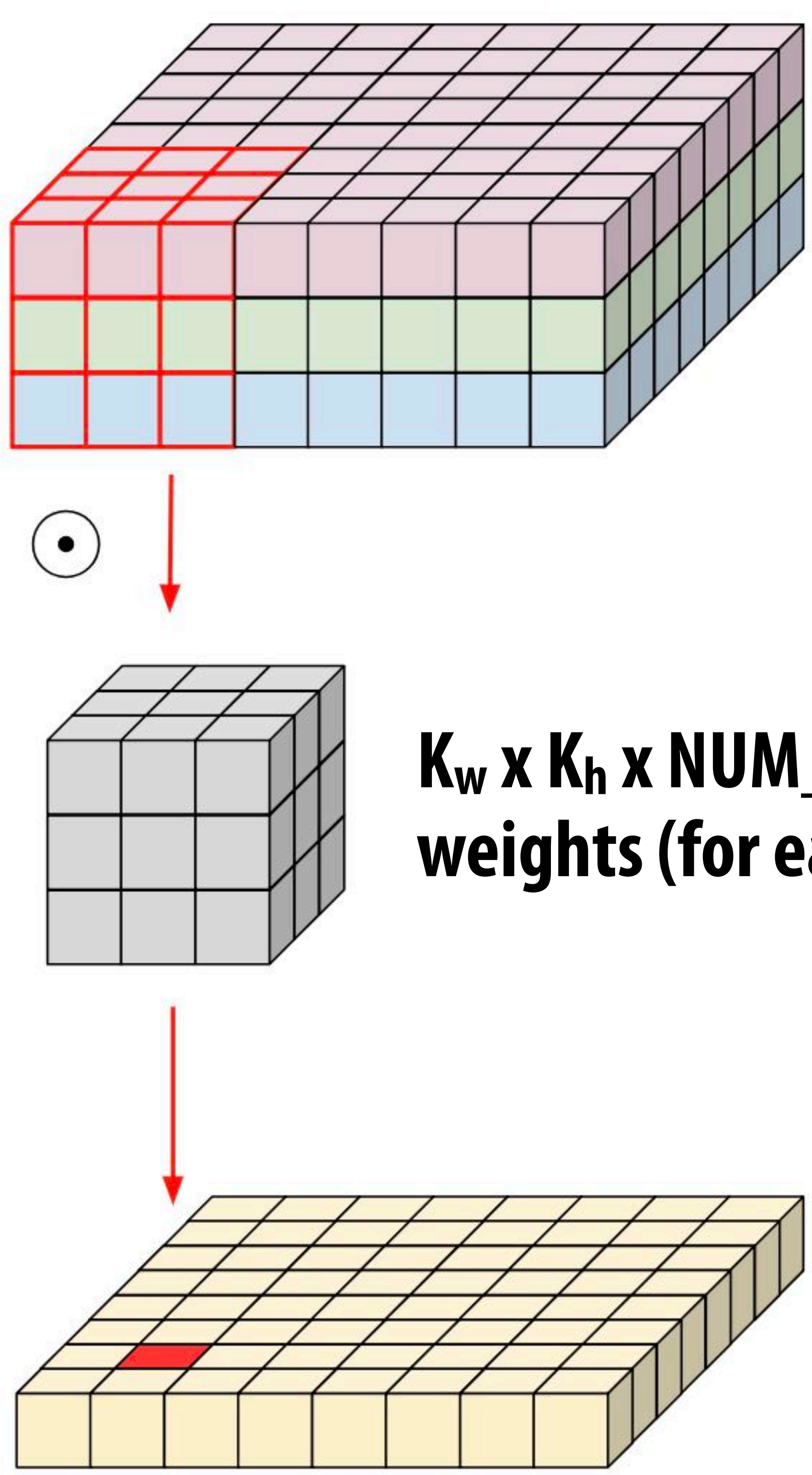
* 10-crop results (ResNet 1-crop results are similar to other DNNs in this table)

# 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

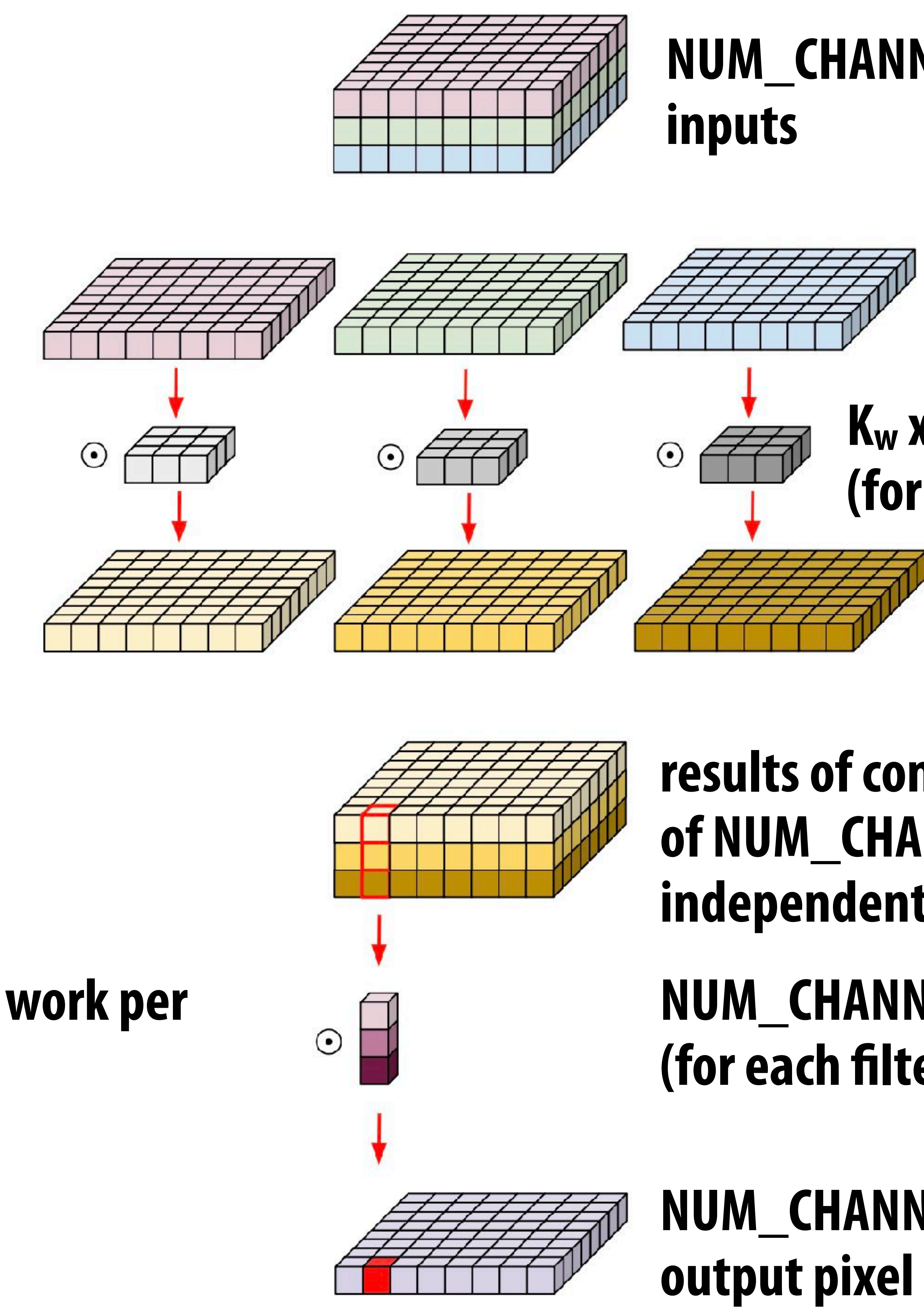
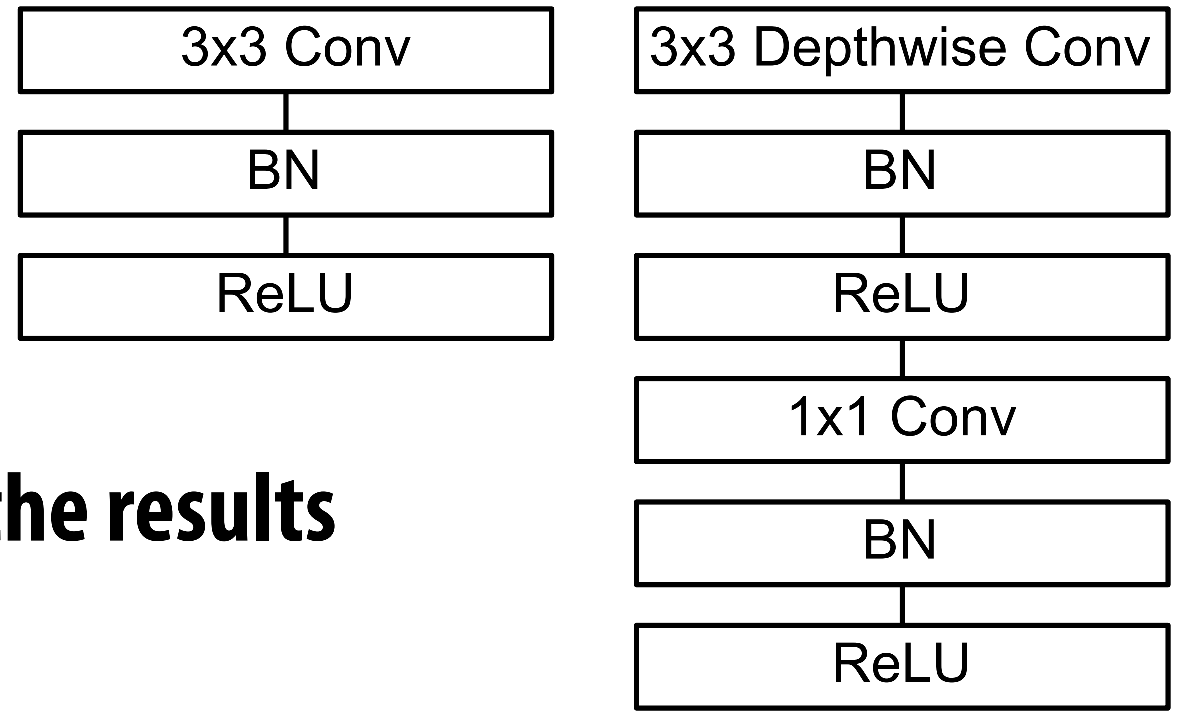



Image credit: Eli Bendersky
https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/

# MobileNet

[Howard et al. 2017]




**Factor NUM_FILTERS 3x3xNUM_CHANNELS convolutions into:**

- **NUM_CHANNELS 3x3x1 convolutions for each input channel**
- **And NUM_FILTERS 1x1xNUM_CHANNELS convolutions to combine the results**

Table 1. MobileNet Body Architecture

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

**Image classification (ImageNet)**
**Comparison to Common DNNs**

| Model | ImageNet Accuracy | Million Mult-Adds | Million Parameters |
|---|---|---|---|
| 1.0 MobileNet-224 | 70.6% | 569 | 4.2 |
| GoogleNet | 69.8% | 1550 | 6.8 |
| VGG 16 | 71.5% | 15300 | 138 |

**Image classification (ImageNet)**
**Comparison to Other Compressed DNNs**

| Model | ImageNet Accuracy | Million Mult-Adds | Million Parameters |
|---|---|---|---|
| 0.50 MobileNet-160 | 60.2% | 76 | 1.32 |
| Squeezenet | 57.5% | 1700 | 1.25 |
| AlexNet | 57.2% | 720 | 60 |

# 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)**

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: $\gamma$, $\beta$

**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

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

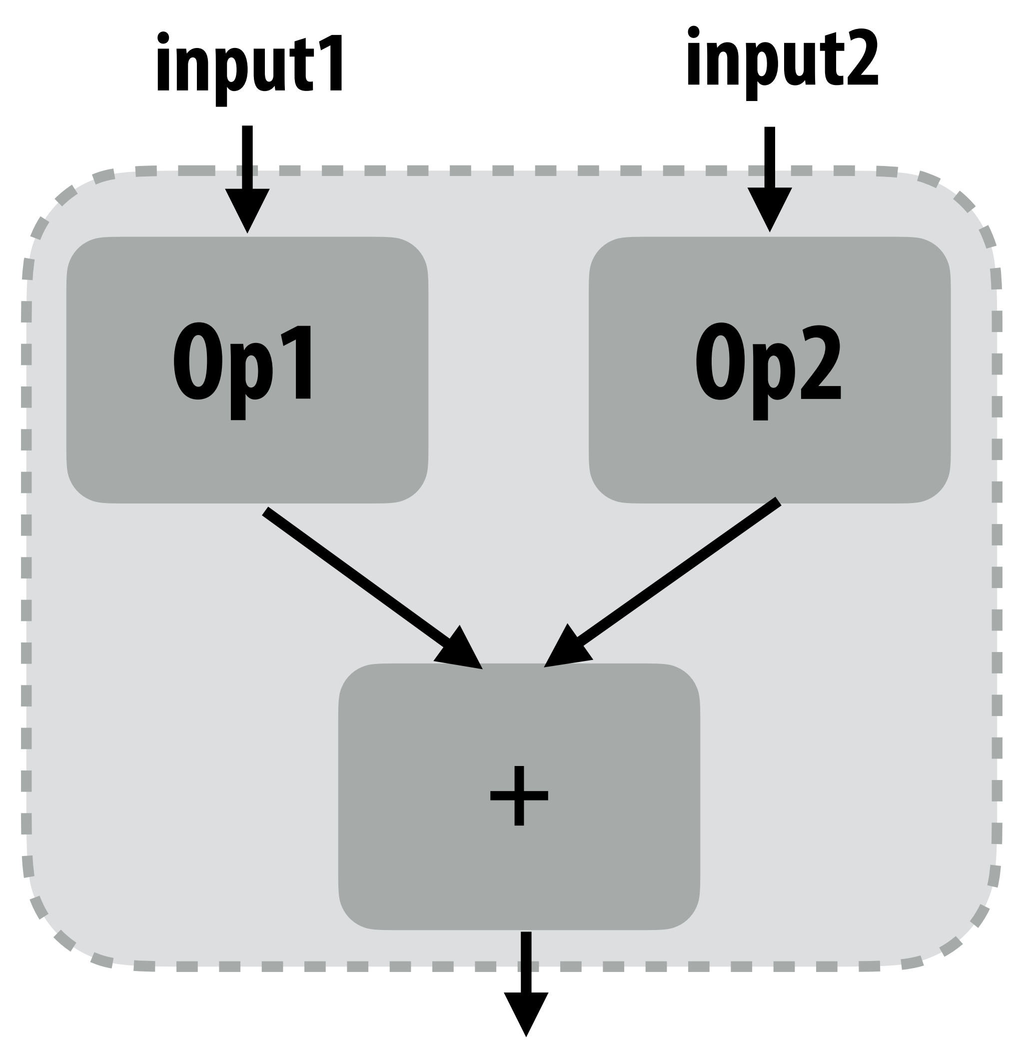
# Model optimization techniques

- 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)



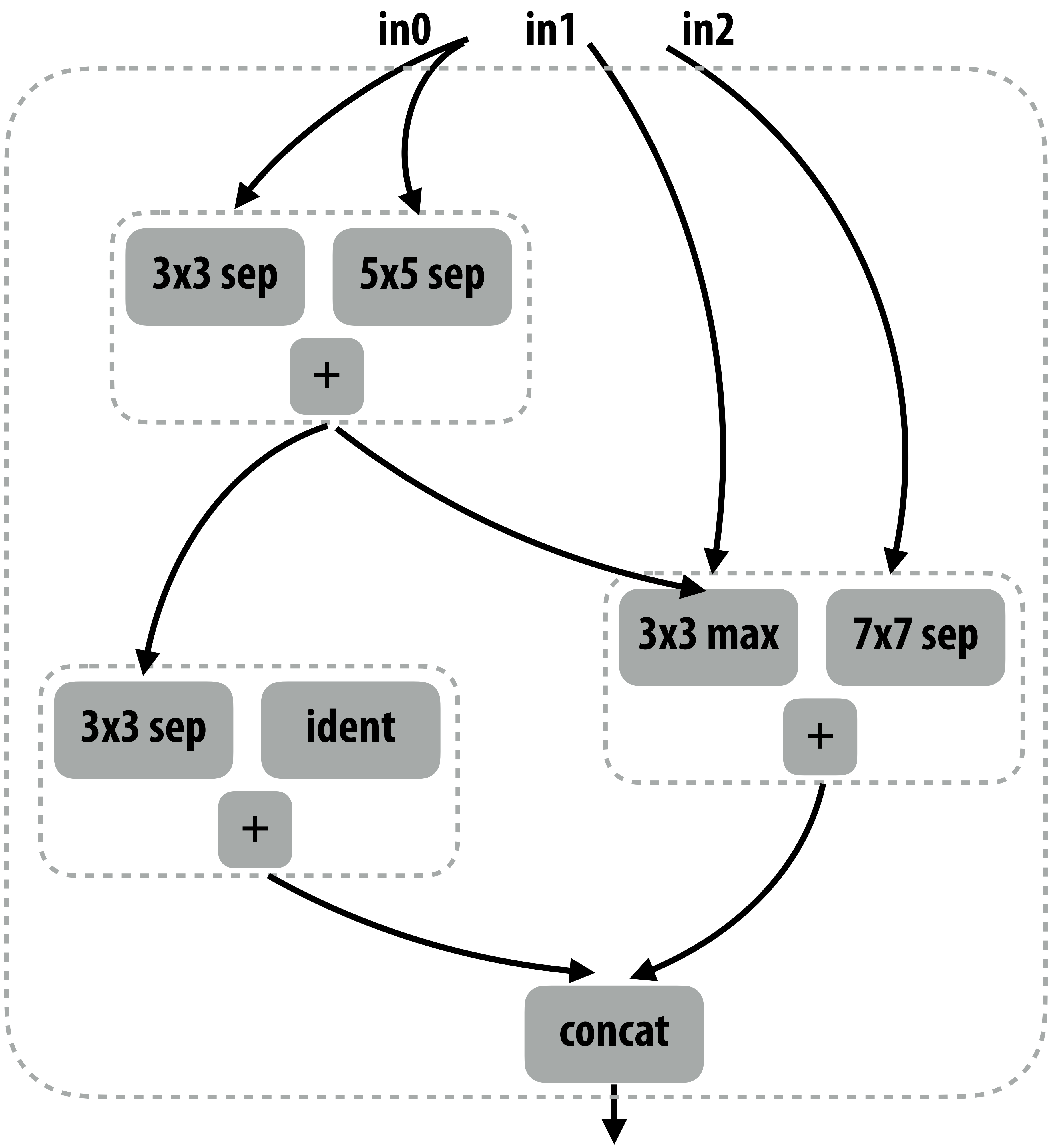

**Eight possible operations:**

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

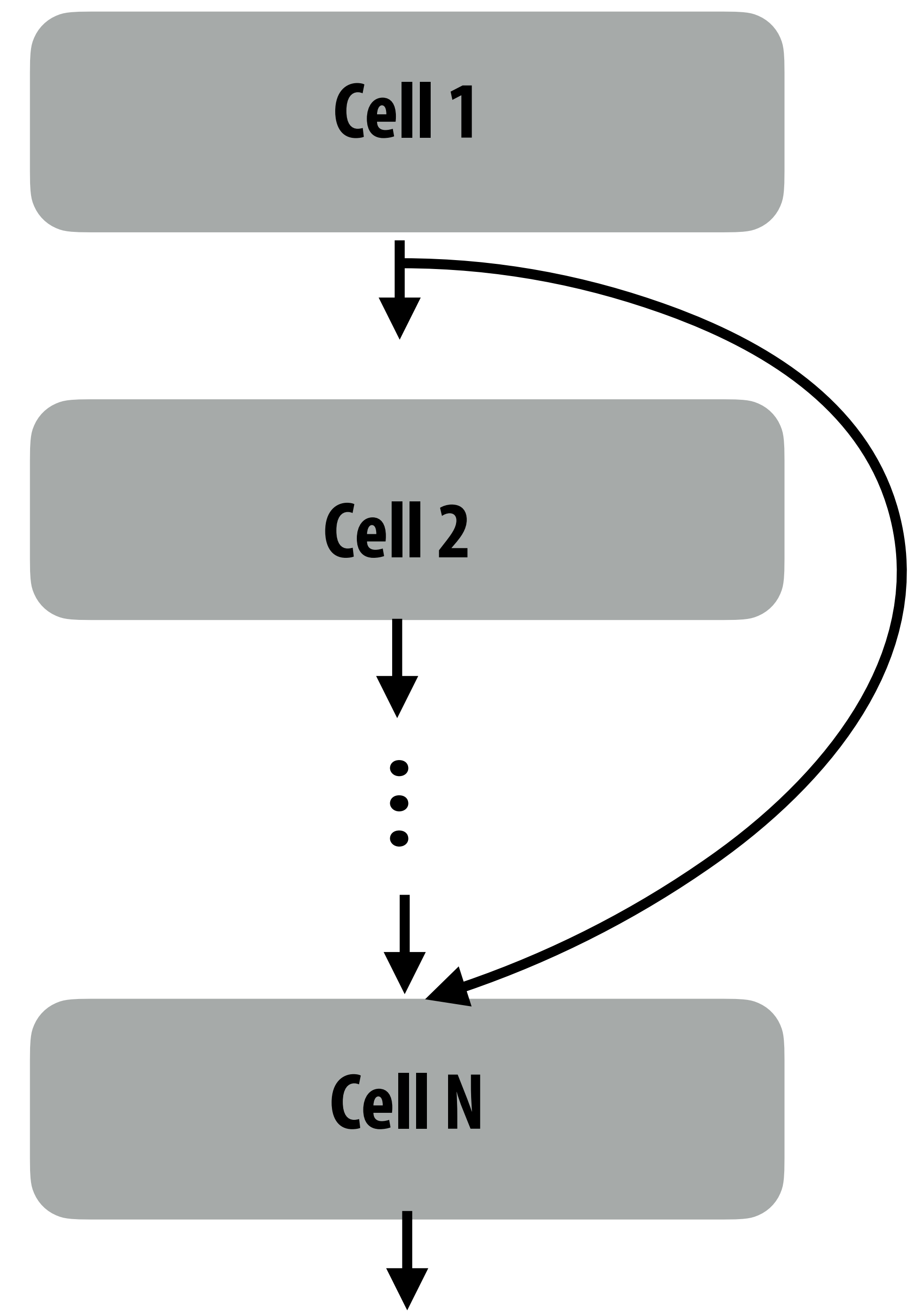
[Liu et al. 18]

# 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**

# 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)**

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

- **Forms of architecture search implemented by Cloud-based ML hosting services (user provides training data, service searches for good model)**

Google Cloud
Cloud AutoML BETA

Amazon SageMaker
Build, train, and deploy machine learning models at scale

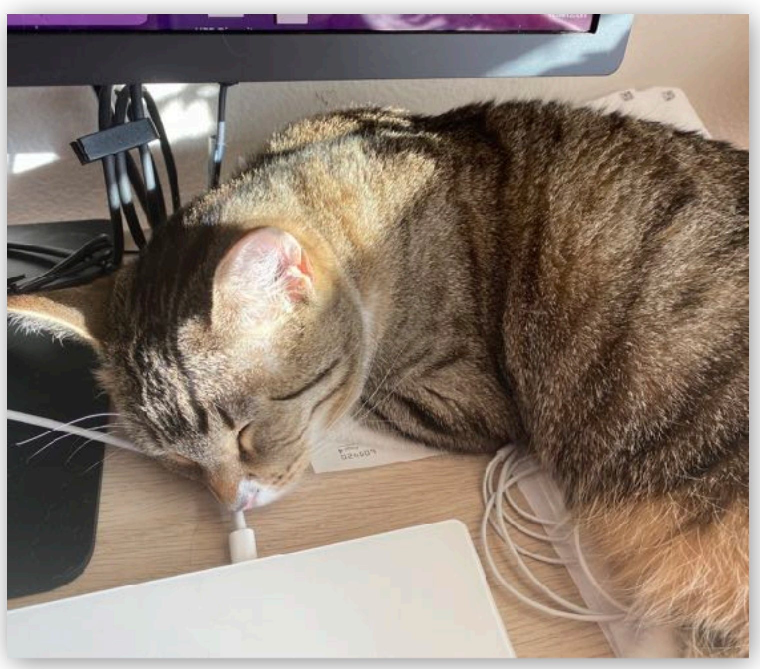
# Dynamic Execution

## (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



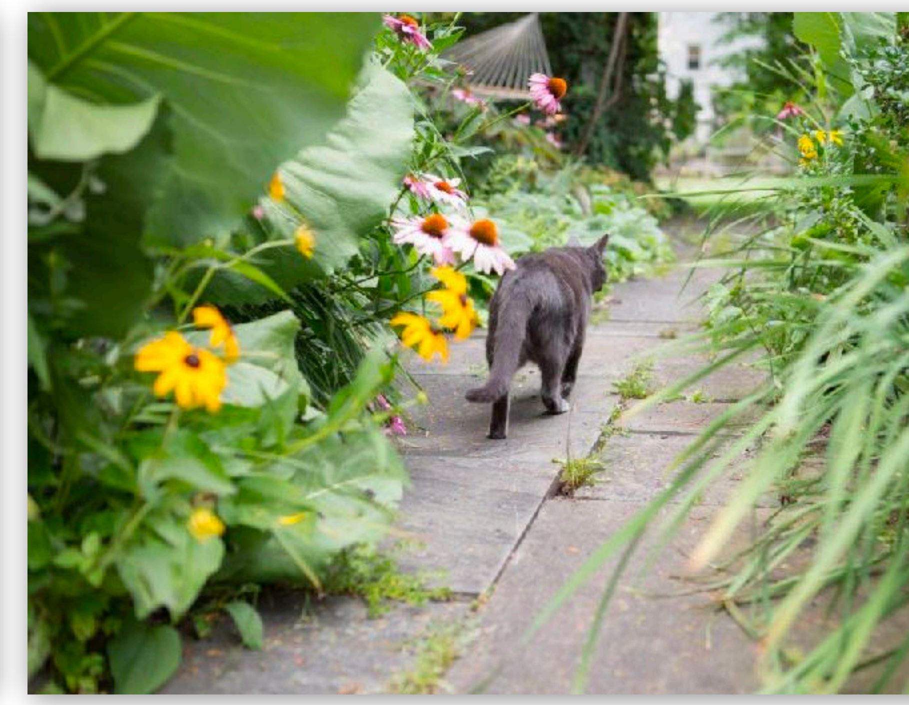
**Positive example**

**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.

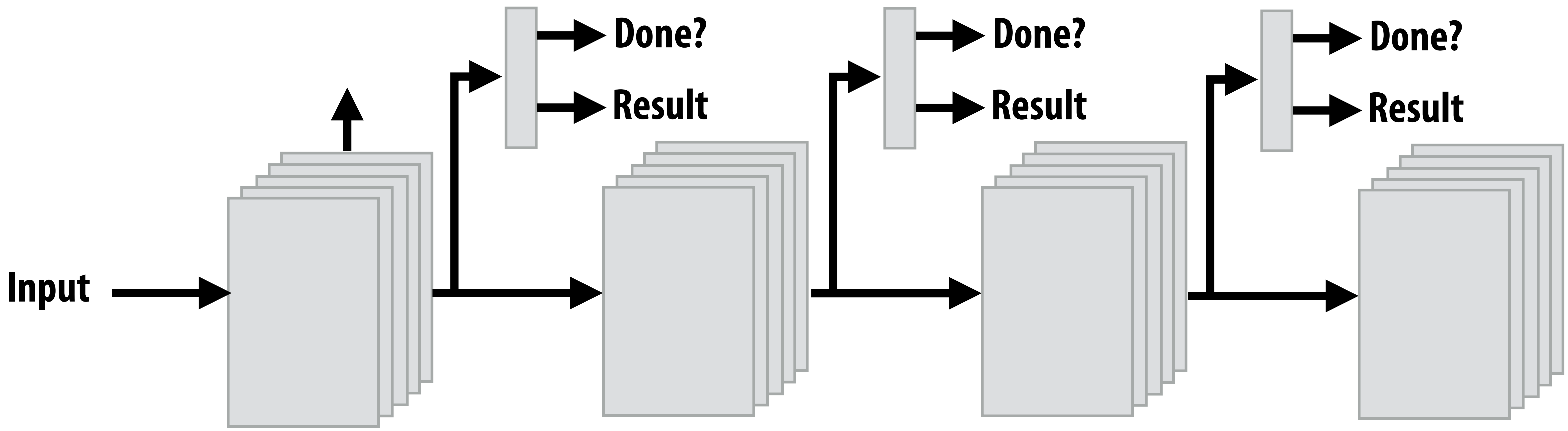


**Small on screen**

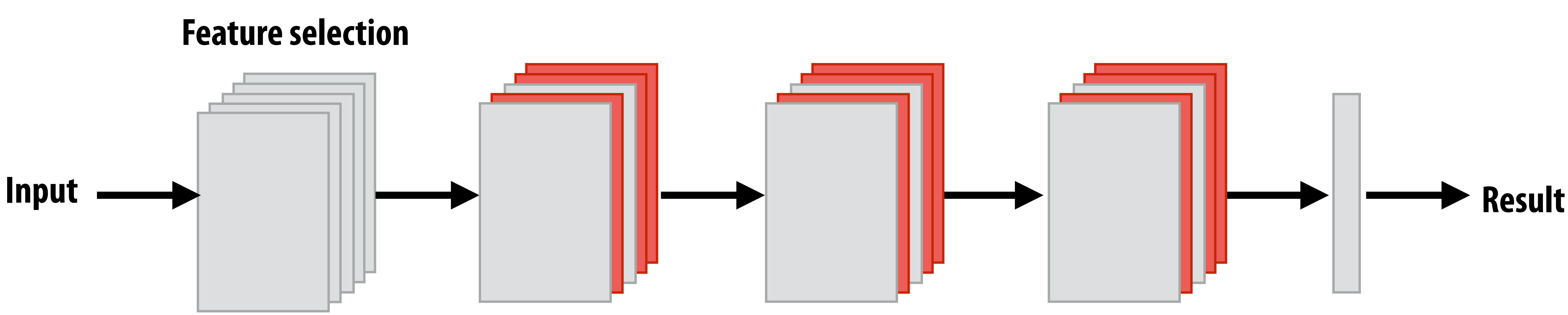
Some regions of the screen might need more processing than others.

# 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