Lecture 25:

Efficiently Evaluating Deep Networks

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2016

Zoobi Doobi

Sonu Nigam, Anuradha Paudwal, Shreya Ghoshal & Kumar Sanu

(Three Idiots Soundtrack)

"Pursue excellence, and success will follow, pants down."

- Rancho, on the final weeks of 418.

Today

- High-performance evaluation of deep neural networks
- We will focus on the parallelism challenges of training deep networks next time

Training/evaluating deep neural networks

Technique leading to many high-profile AI advances in recent years

Speech recognition/natural language processing

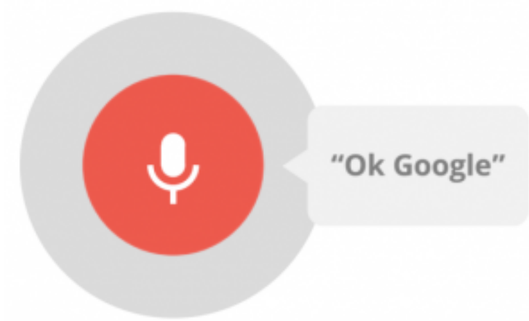
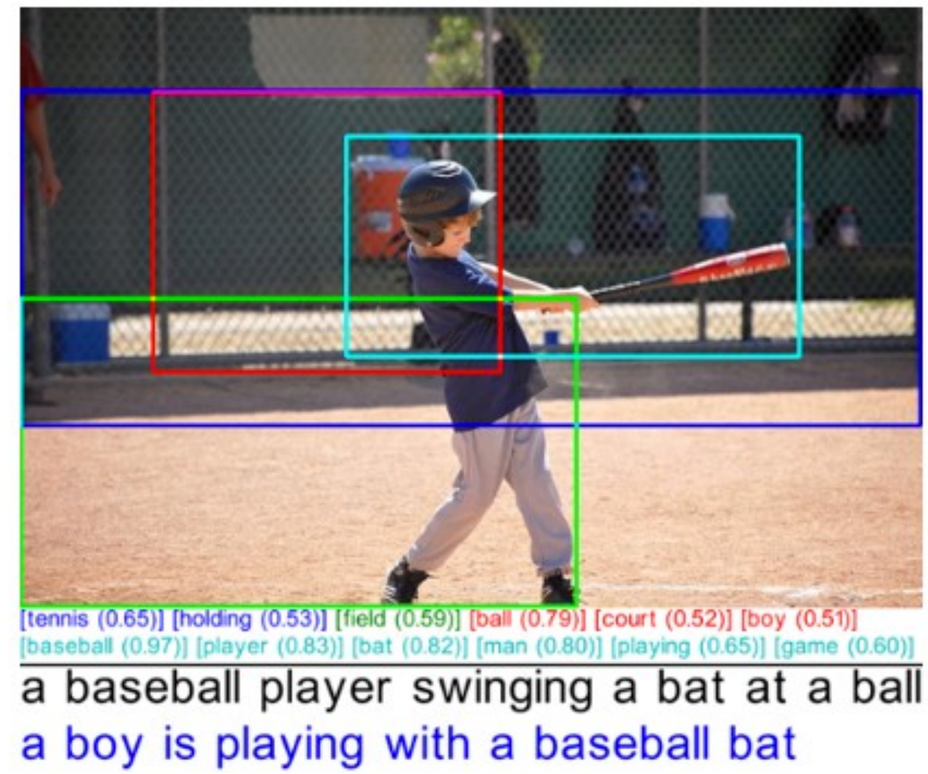
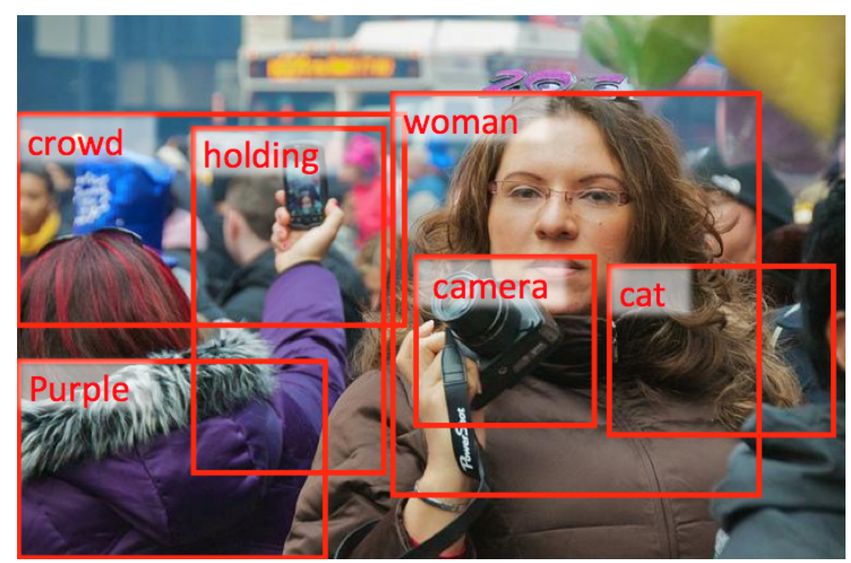


Image interpretation and understanding

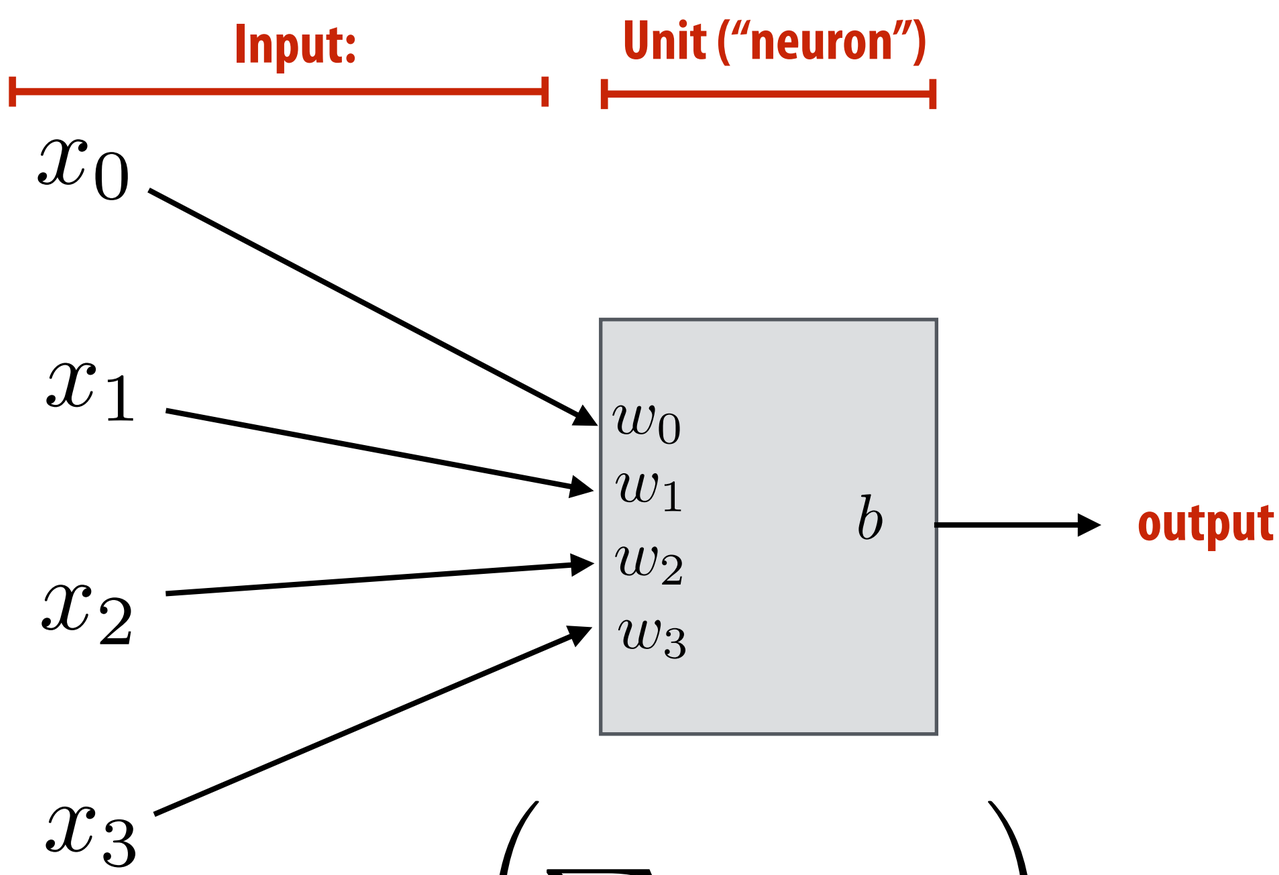




What is a deep neural network?

A basic unit:

Unit with *n* inputs described by *n+1* parameters (weights + bias)



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

Example: rectified linear unit (ReLU)

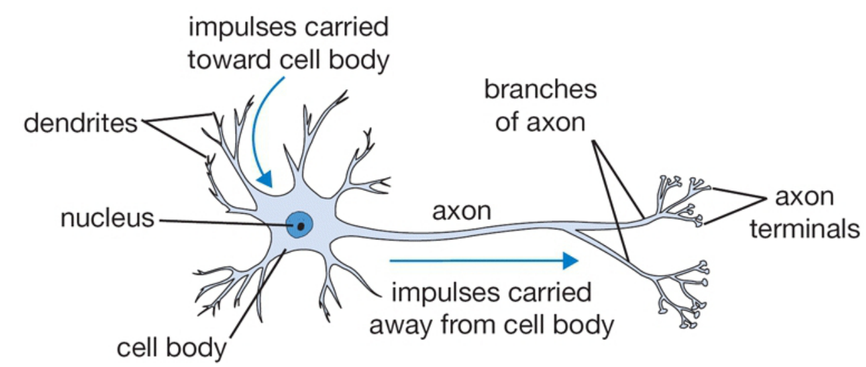
$$f(x) = max(0, x)$$

Basic computational interpretation:

It's just a circuit!

Biological inspiration:

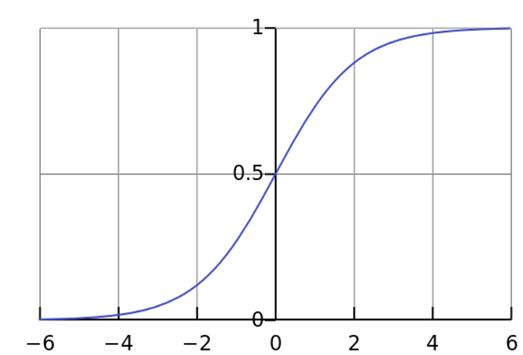
unit output corresponds loosely to activation of neuron



Machine learning interpretation:

binary classifier: interpret output as the probability of one class

$$f(x) = \frac{1}{1 + e^{-x}}$$



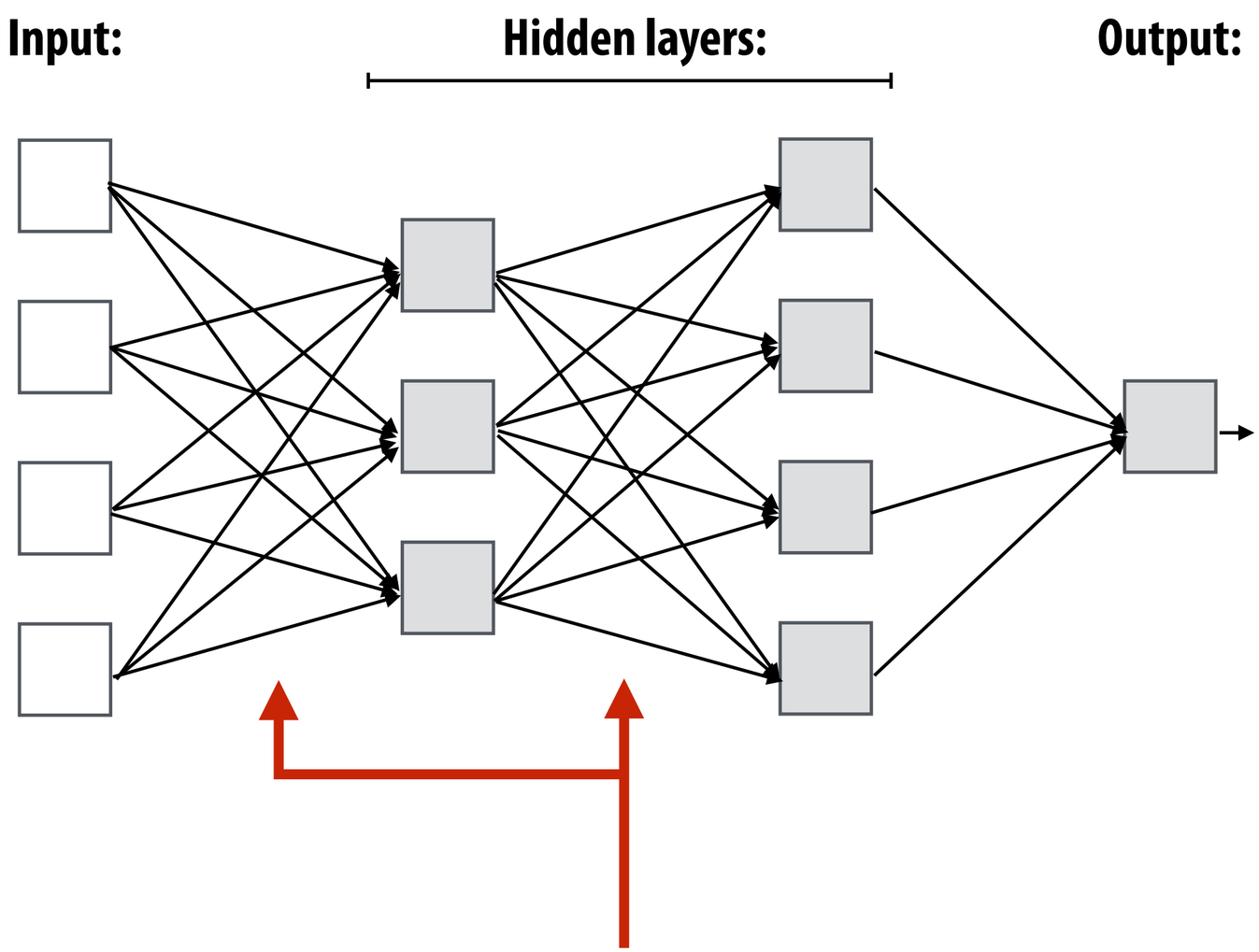
What is a deep neural network? topology

This network has: 4 inputs, 1 output, 7 hidden units

"Deep" = at least one hidden layer

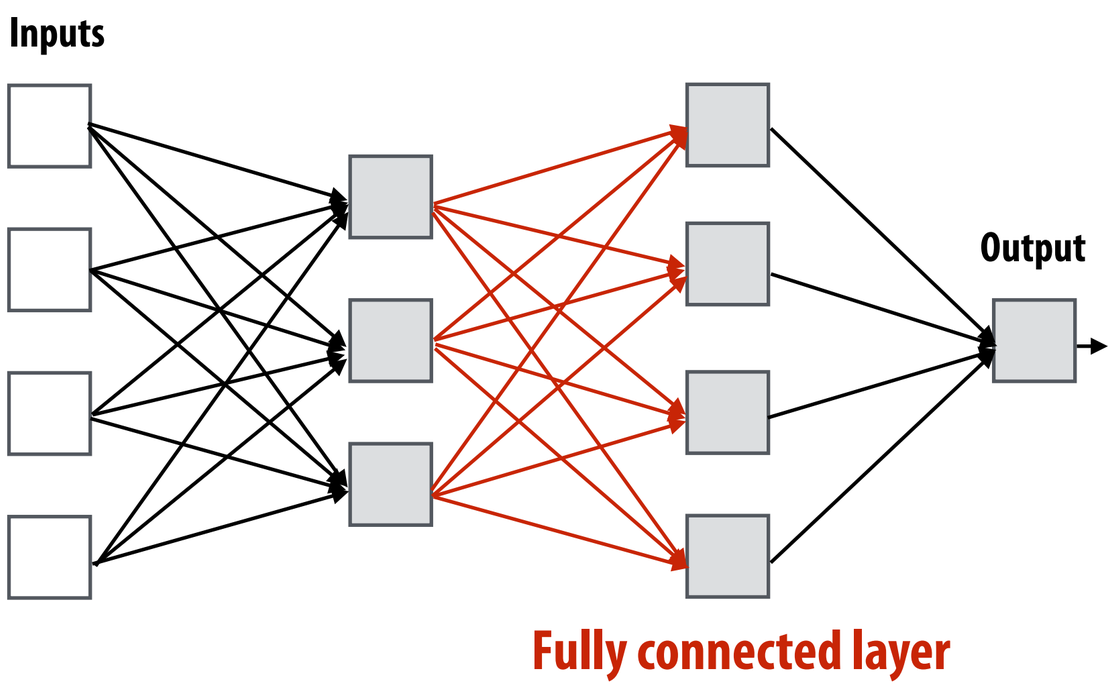
Hidden layer 1: 3 units x (4 weights + 1 bias) = 15 parameters

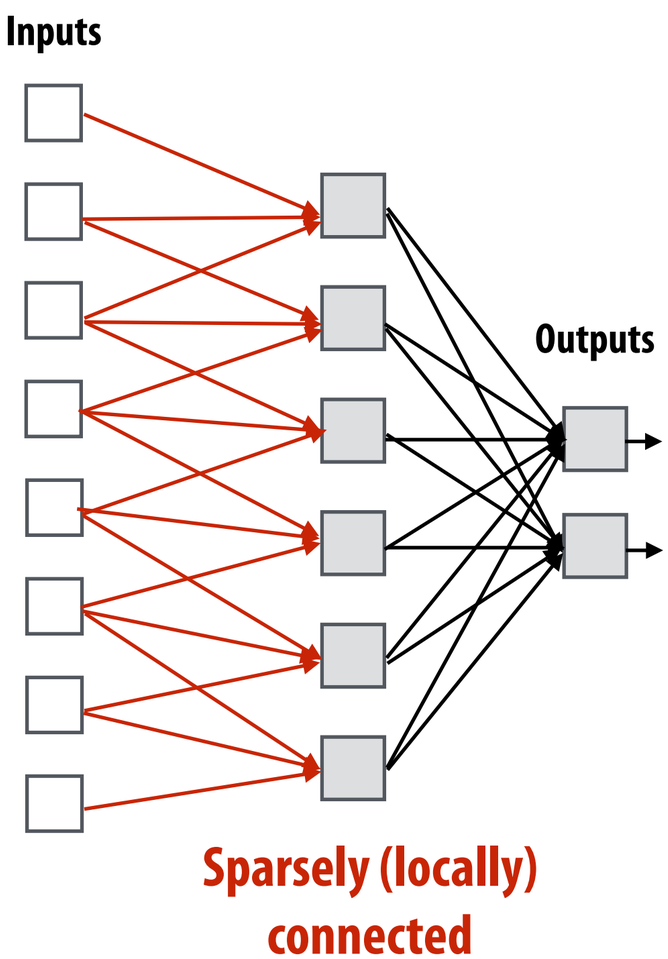
Hidden layer 2: 4 units x (3 weights + 1 bias) = 16 parameters



Note fully-connected topology in this example

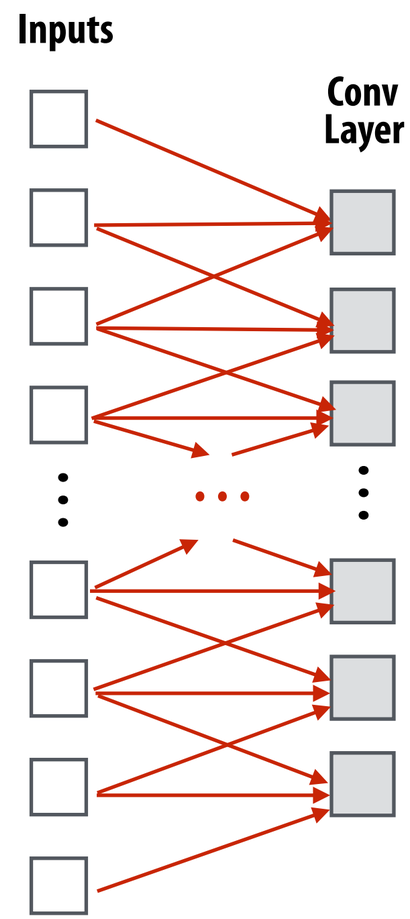
What is a deep neural network? topology

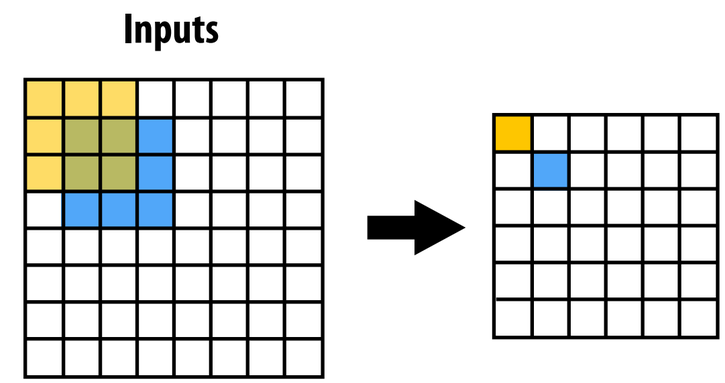




Recall image convolution (3x3 conv)

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


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


for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
    output[j*WIDTH + i] = tmp;
  }
}
```

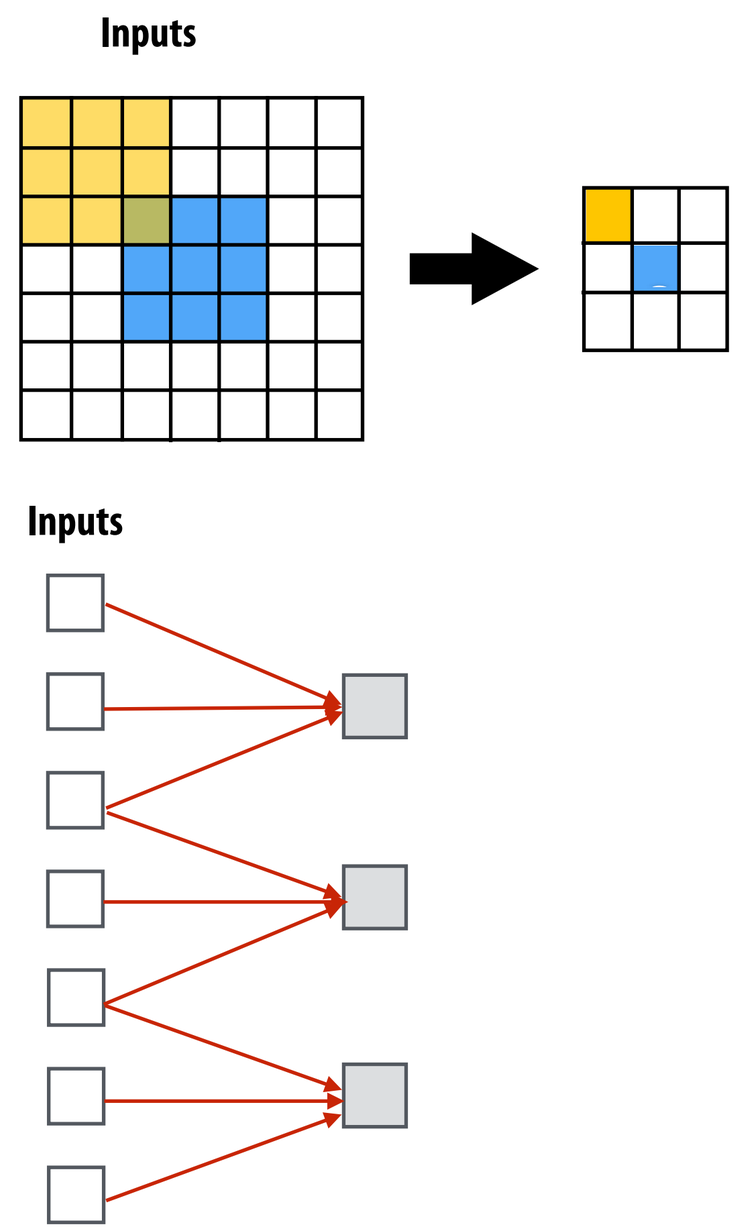




Convolutional layer: locally connected AND all units in layer share the same parameters (same weights + same bias):
(note: network diagram only shows links due to one iteration of `ii` loop)

Strided 3x3 convolution

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


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


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

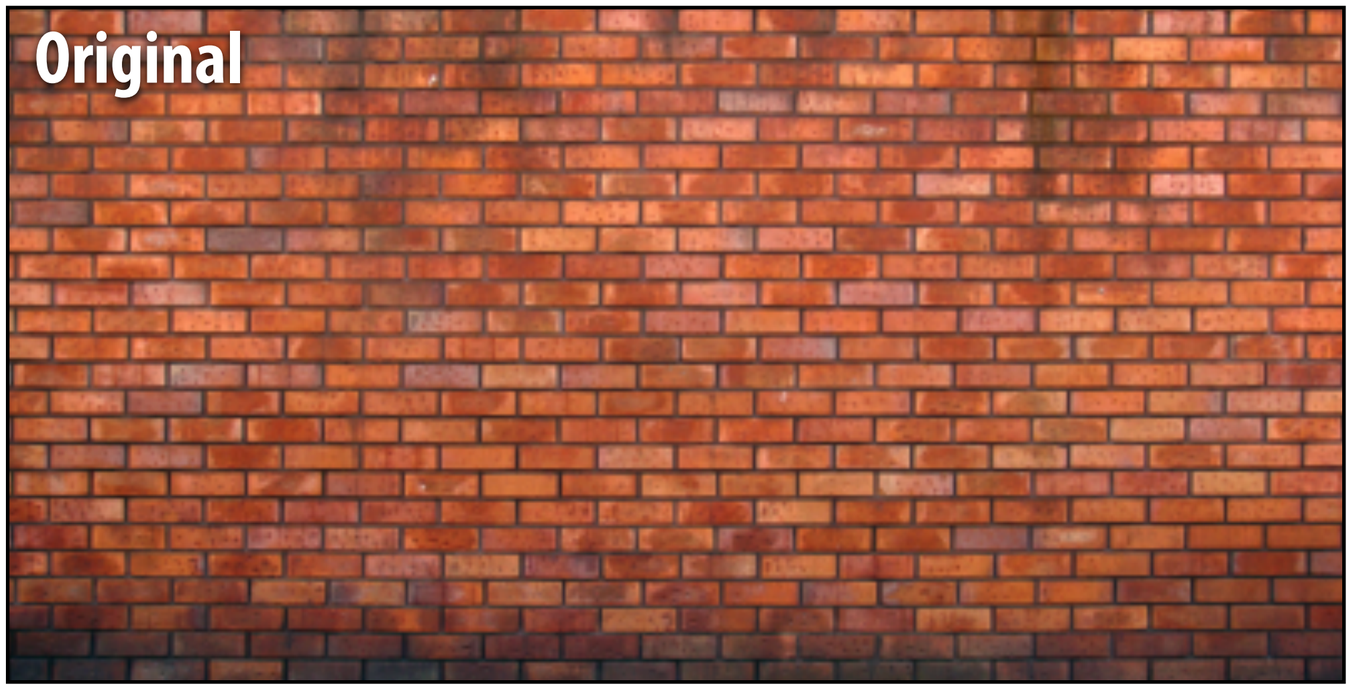


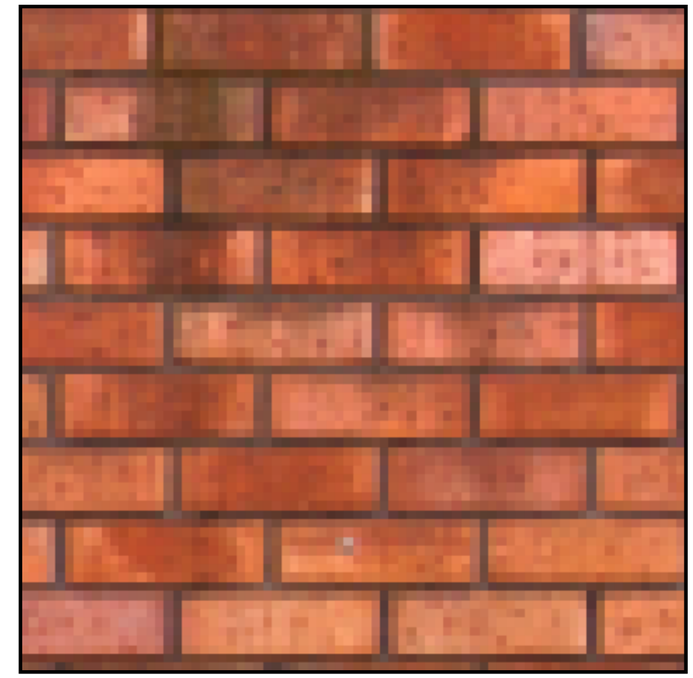
Convolutional layer with stride 2

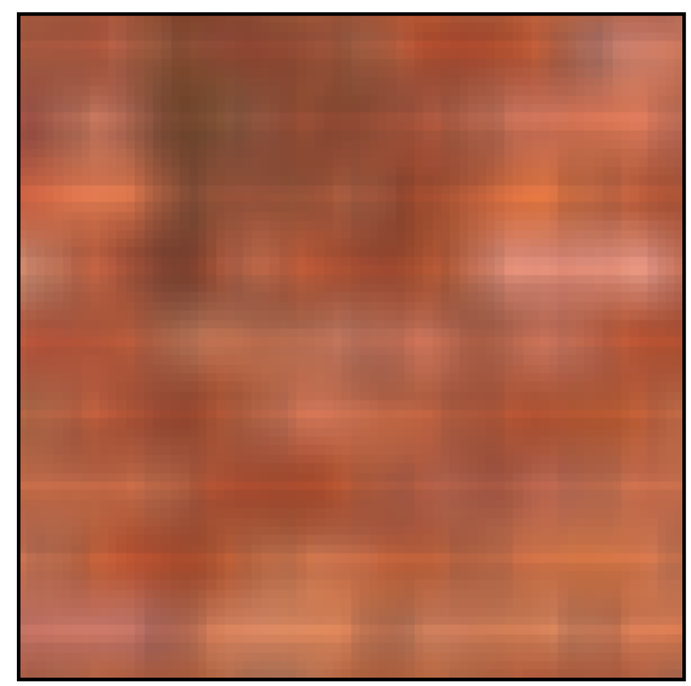
What does convolution using these filter weights do?

$$\begin{bmatrix} .111 & .111 & .111 \\ .111 & .111 & .111 \\ .111 & .111 & .111 \end{bmatrix}$$

"Box blur"



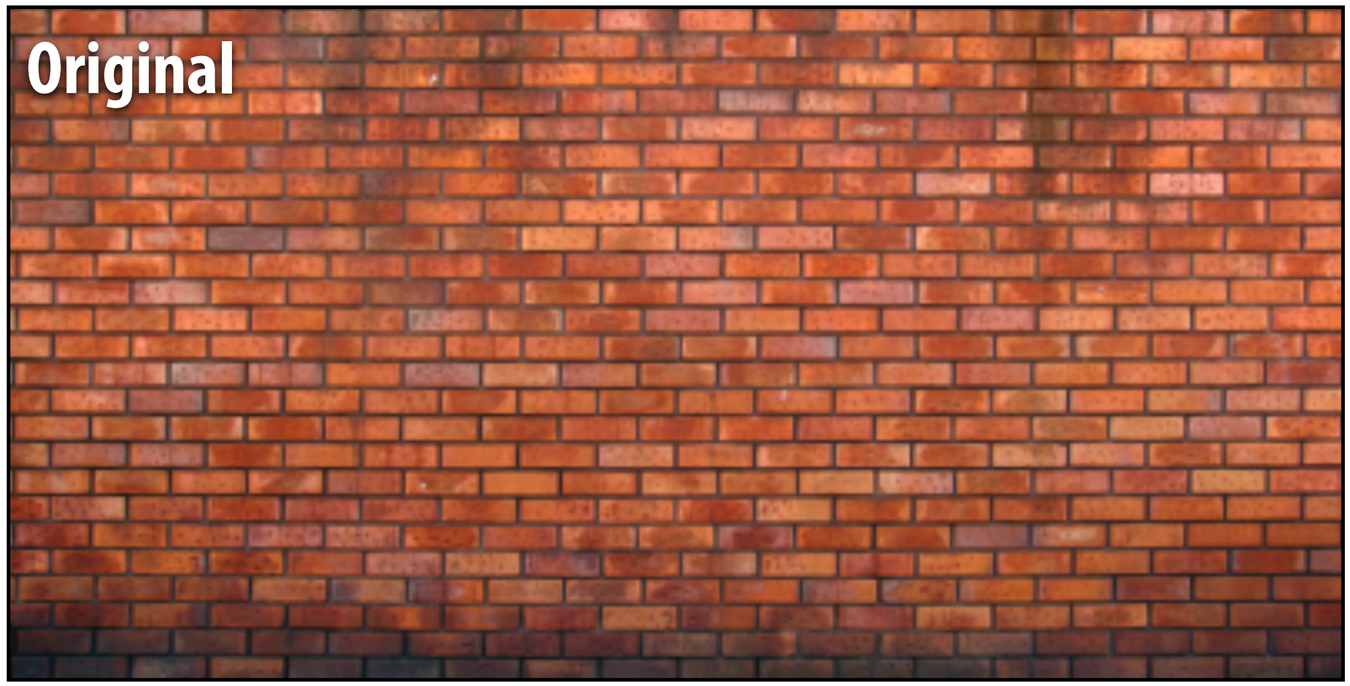


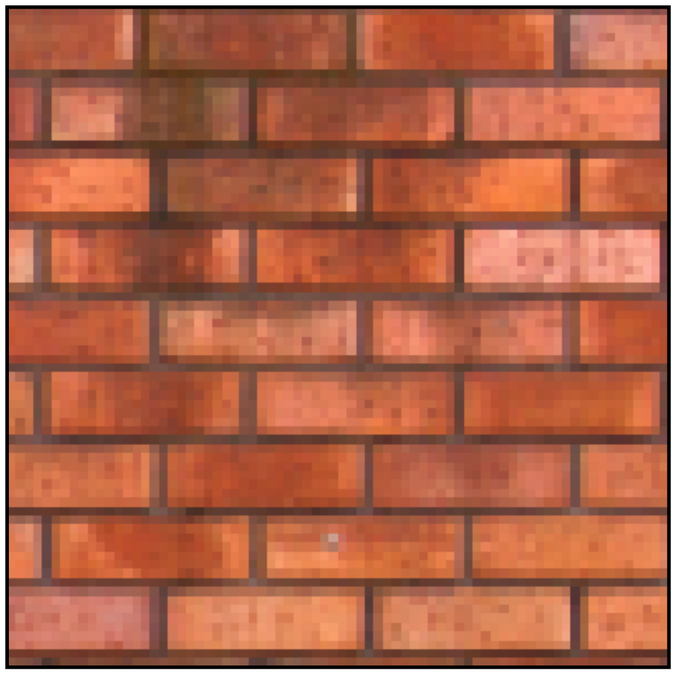


What does convolution using these filter weights do?

$$\begin{bmatrix} .075 & .124 & .075 \\ .124 & .204 & .124 \\ .075 & .124 & .075 \end{bmatrix}$$

"Gaussian Blur"





What does convolution with these filters do?

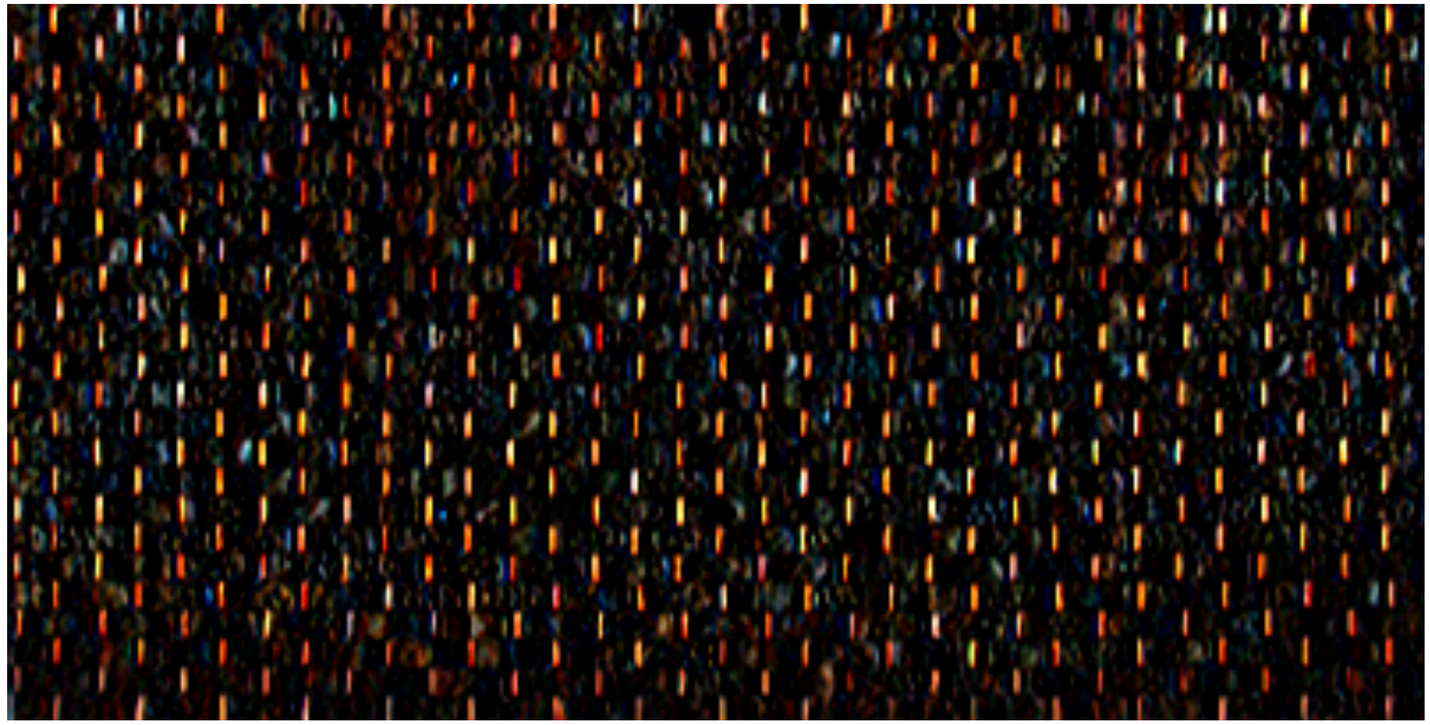
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Extracts horizontal gradients

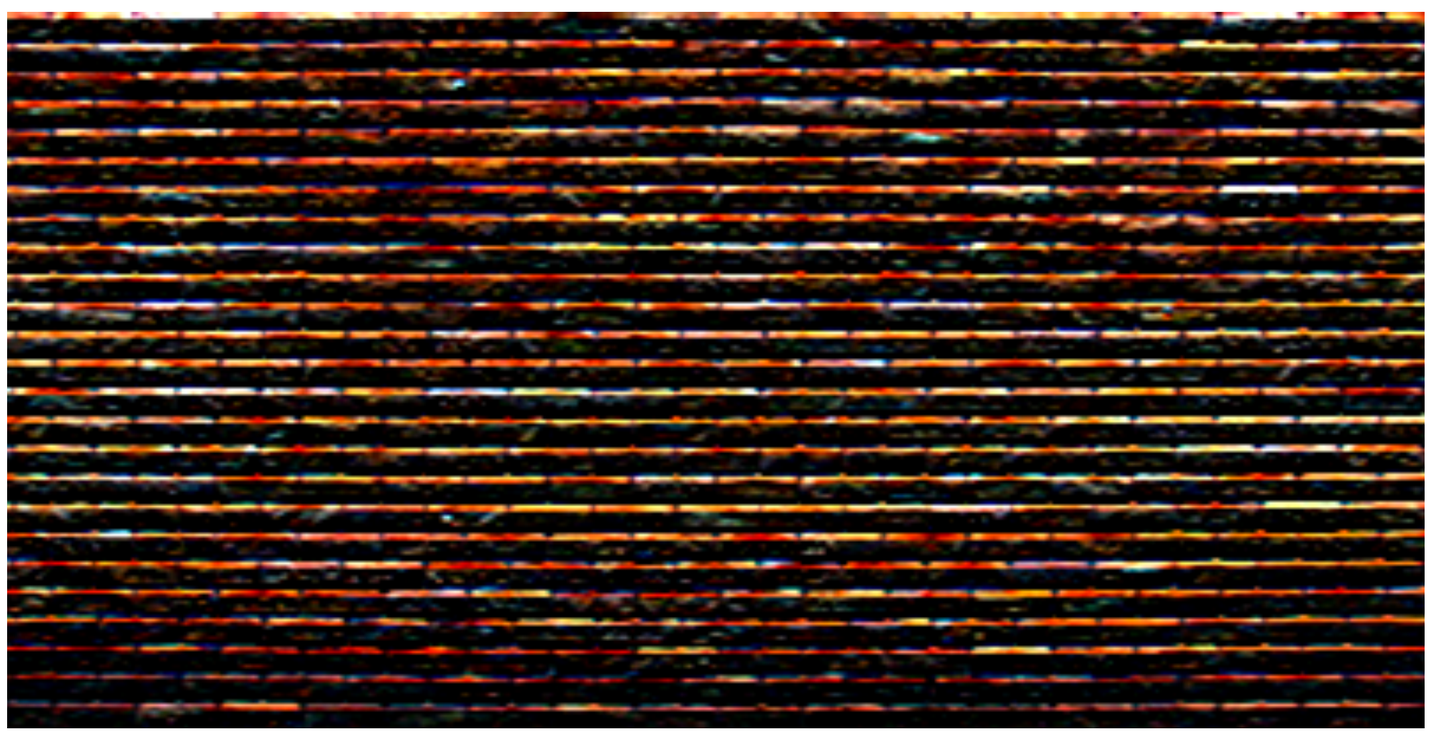
$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Extracts vertical gradients

Gradient detection filters



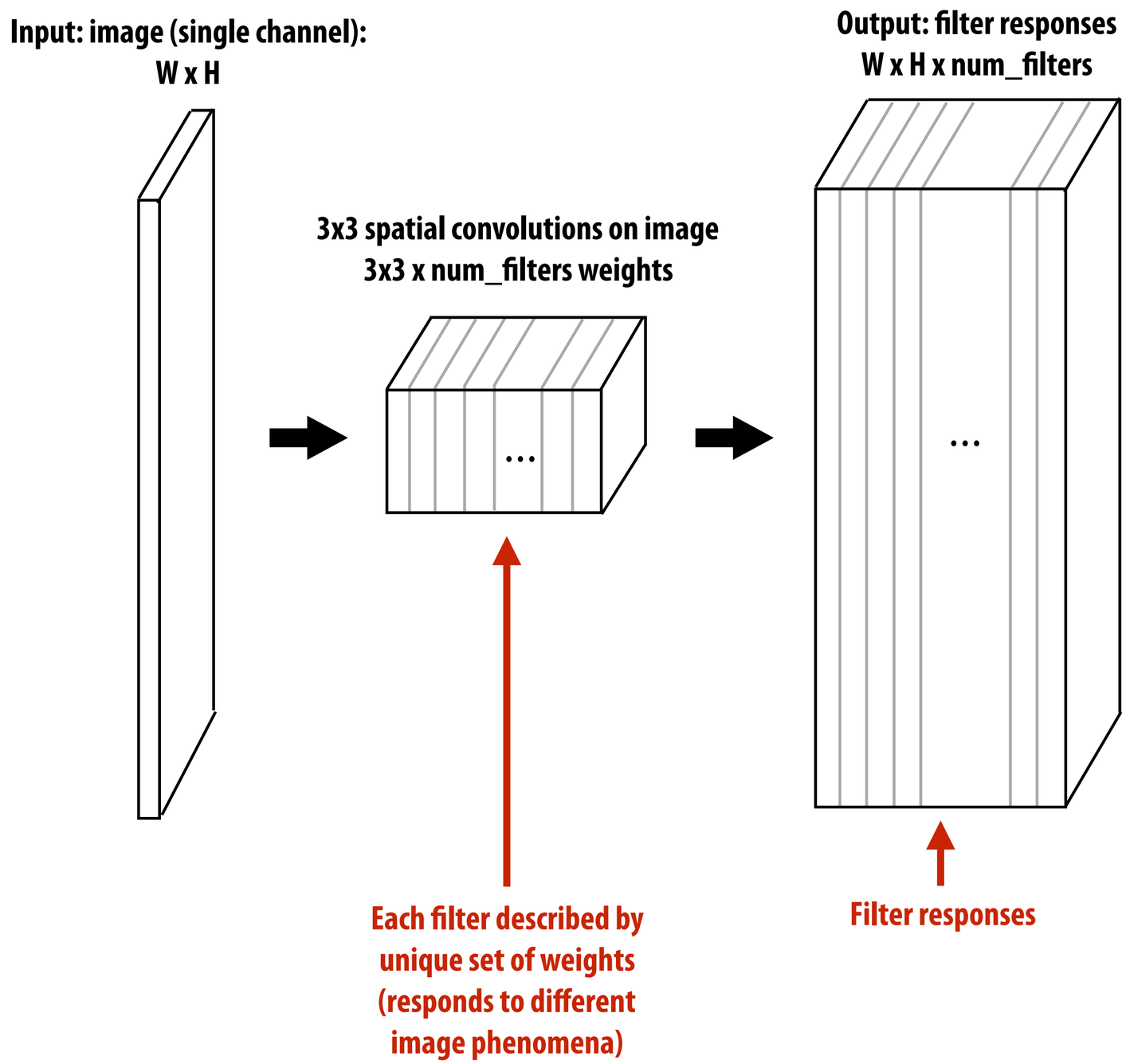
Horizontal gradients



Vertical gradients

Note: you can think of a filter as a "detector" of a pattern, and the magnitude of a pixel in the output image as the "response" of the filter to the region surrounding each pixel in the input image

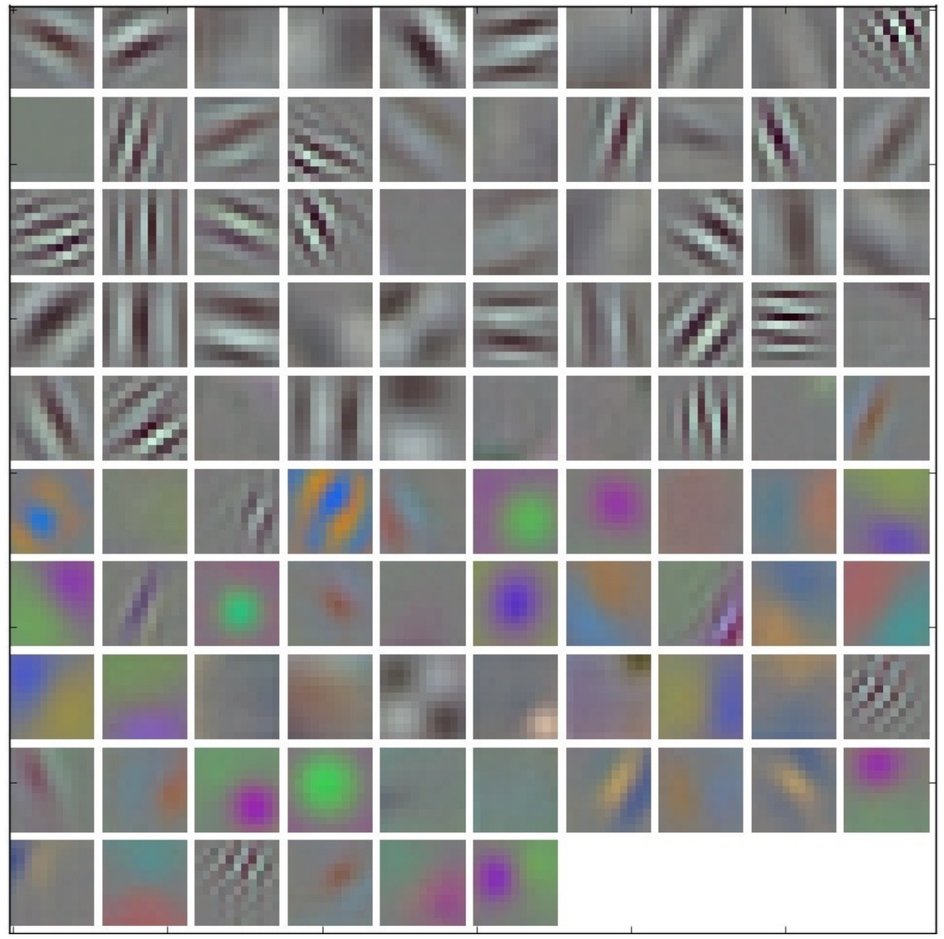
Applying many filters to an image at once



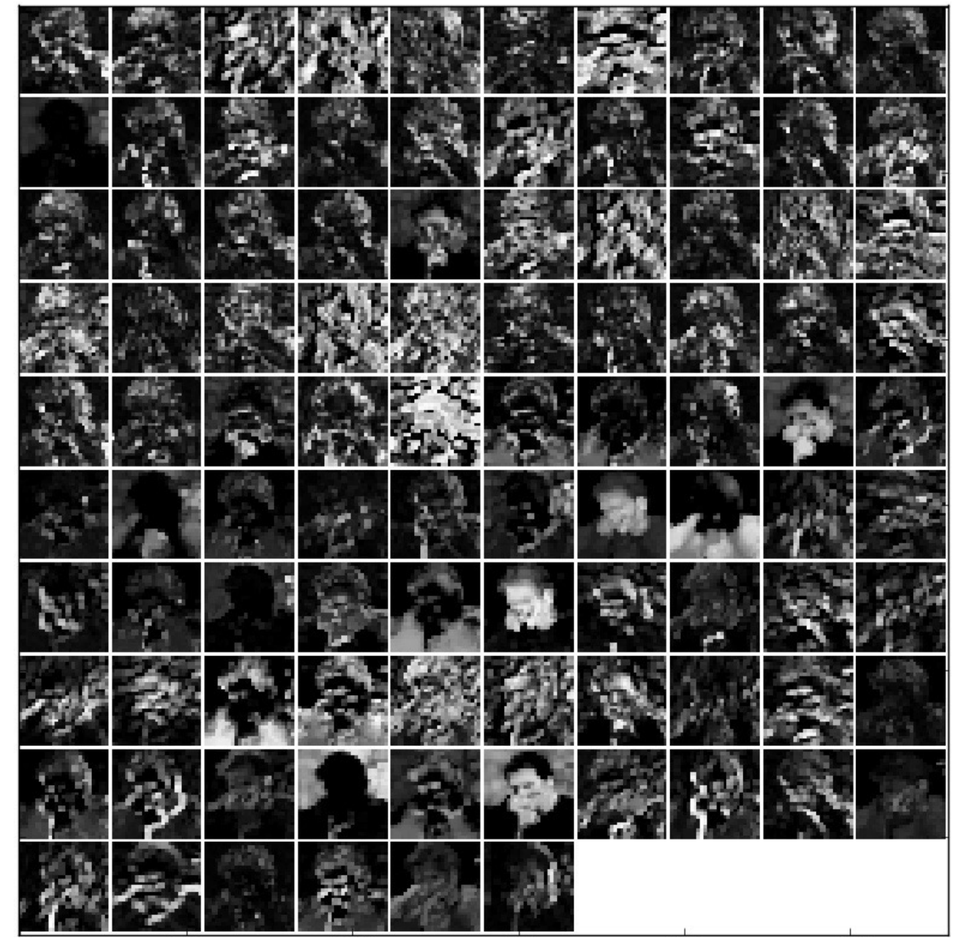
Applying many filters to an image at once

Input RGB image (W x H x 3)

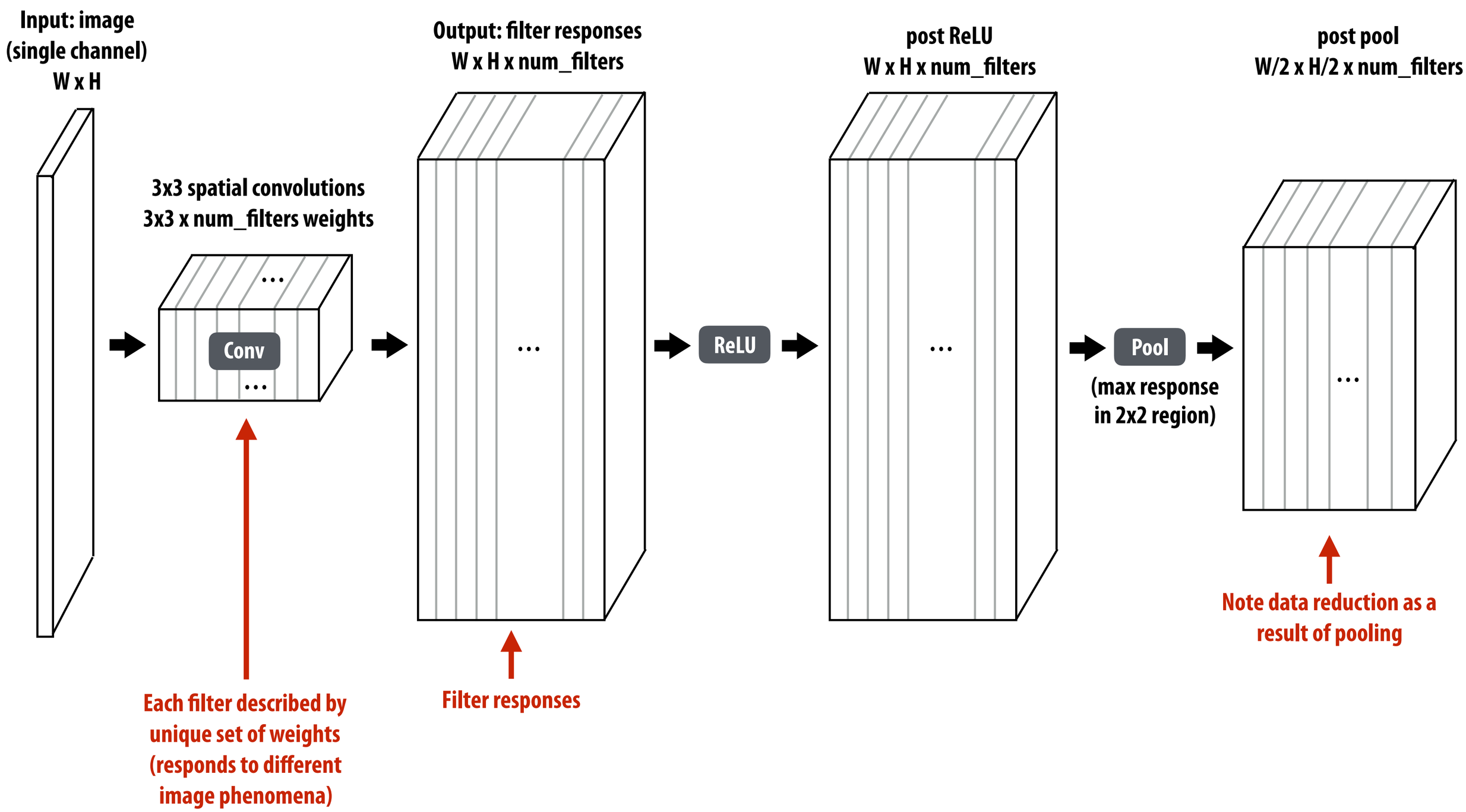
96 11x11x3 filters
(operate on RGB)



96 responses (normalized)



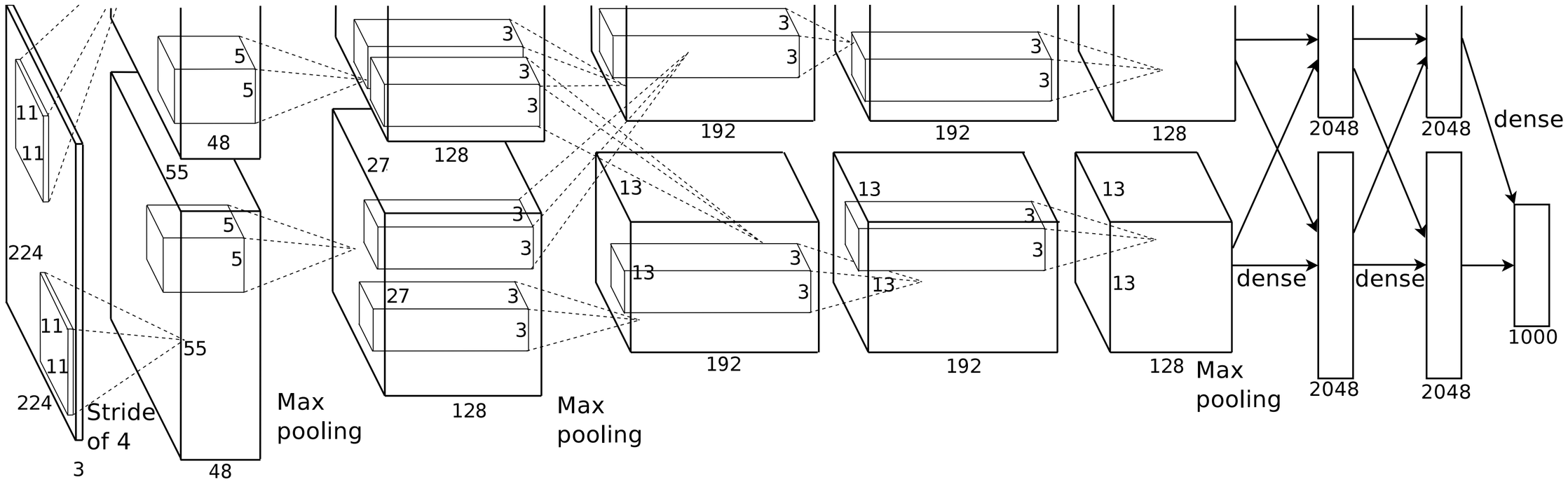
Adding additional layers



Modern object detection networks

Sequences of cont + reLU + (optional) pool layers

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



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.]

"Training a network"

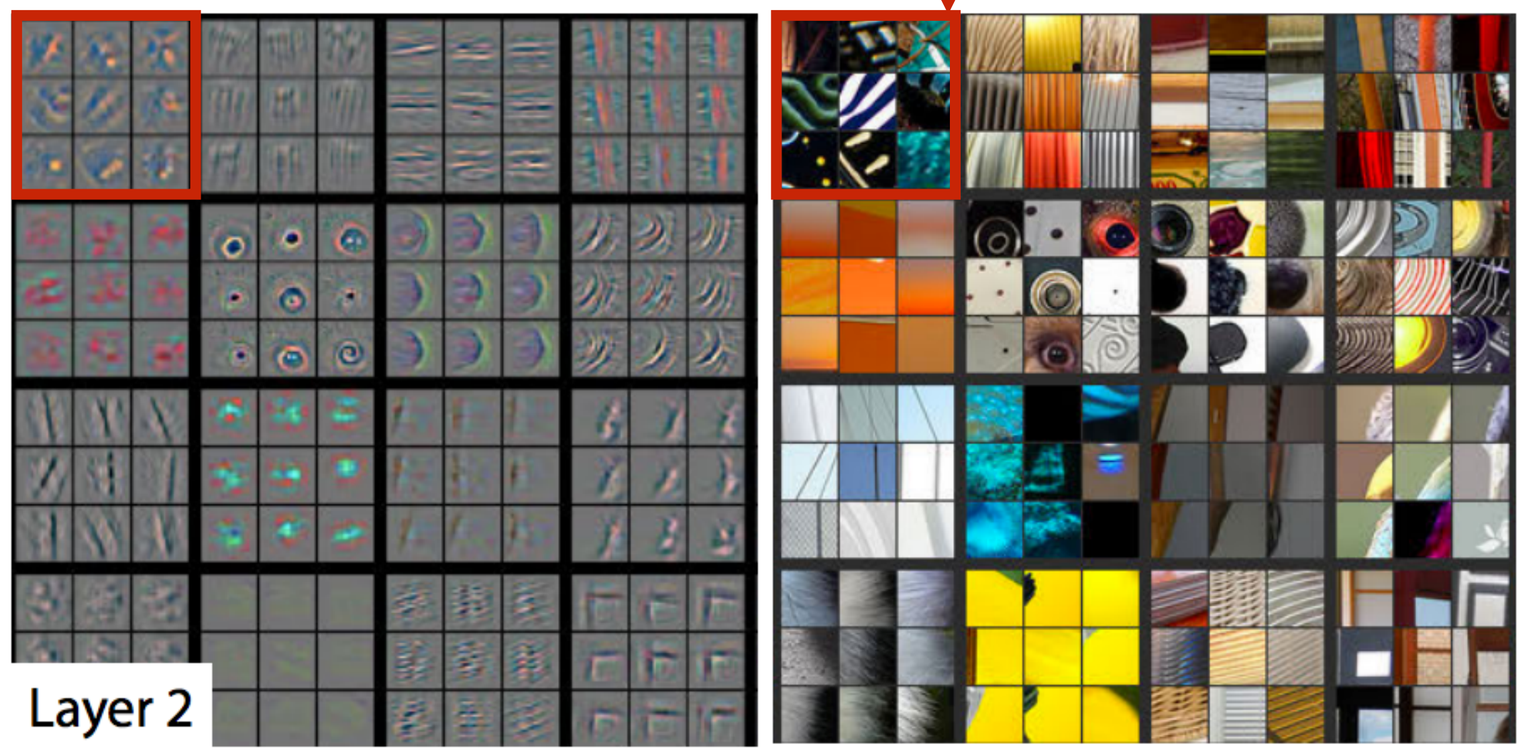
- Training a network is the process of learning the value of network parameters so that output of the network provides the desired result for a task
  - [Krizhevsky12] task = object classification
    - input 224 x 224 x 3 RGB image
    - output probability of 1000 ImageNet object classes: "dog", "cat", etc…
    - ~ 60M weights

- Will discuss how to train networks in parallel next time

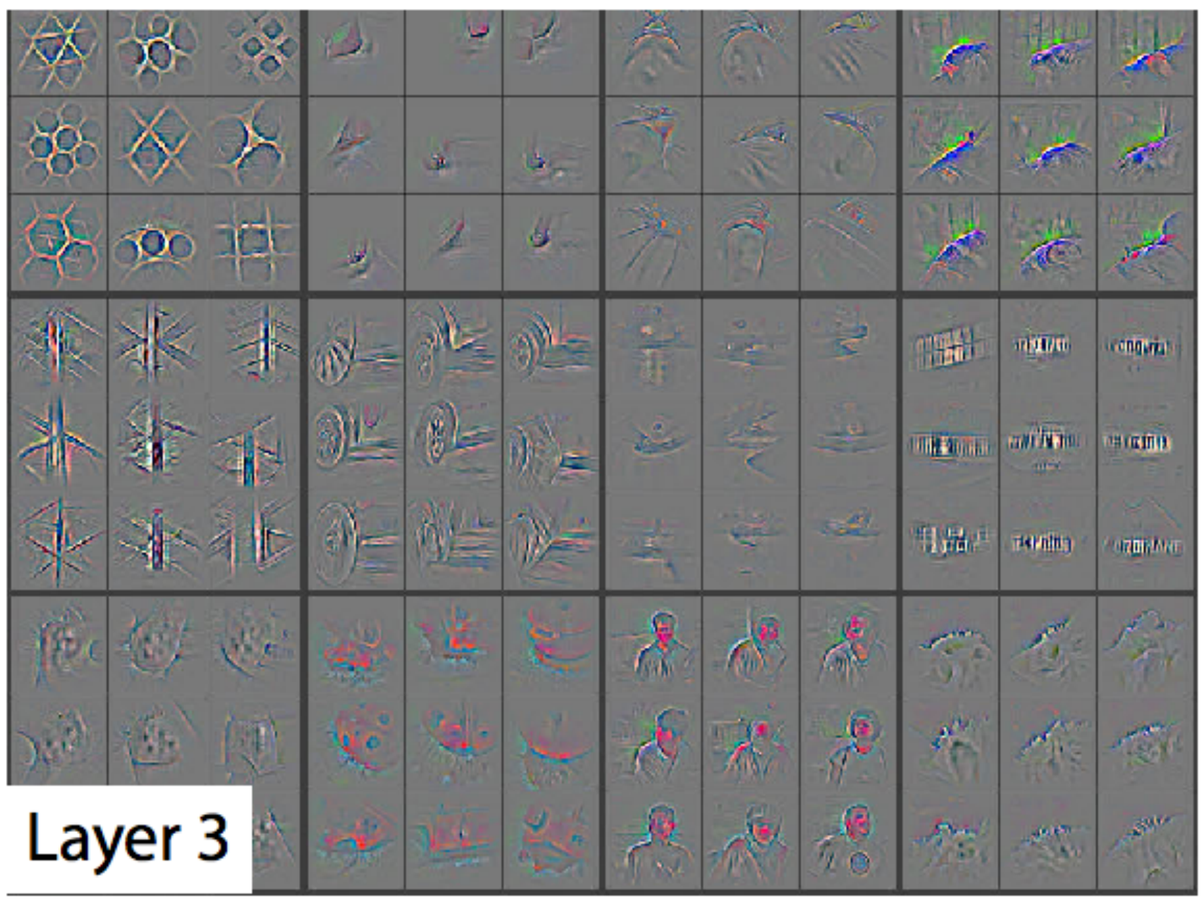
Why deep?

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

Layer 1



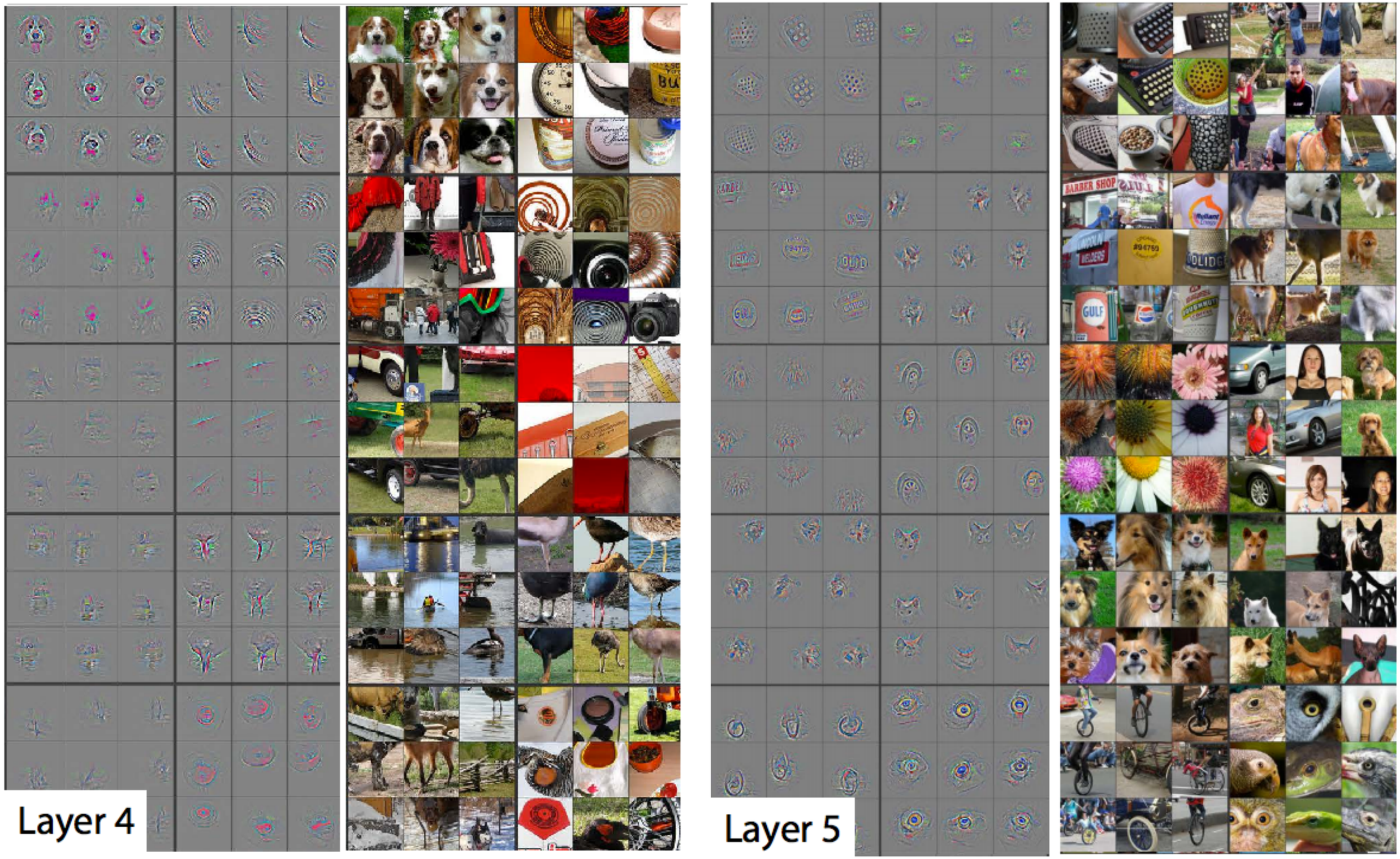
Layer 2



Layer 3

[image credit: Zeiler 14]

Why deep?

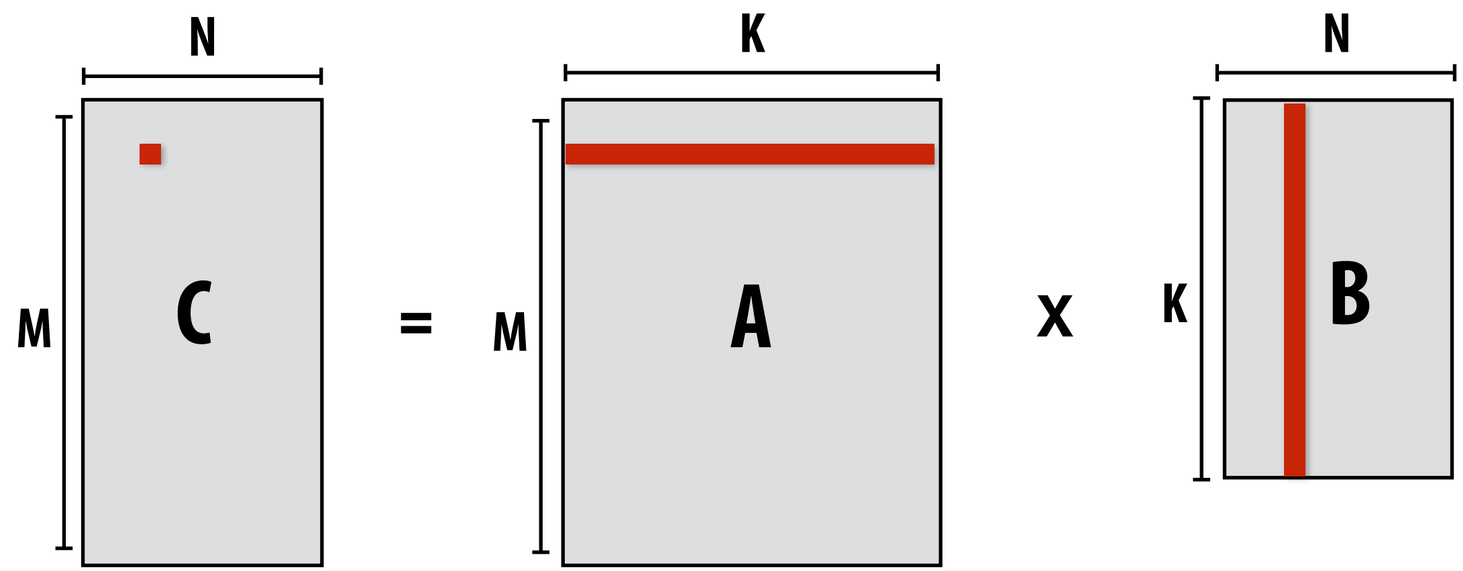


[image credit: Zeiler 14]

Efficiently implementing convolution layers

Dense matrix multiplication

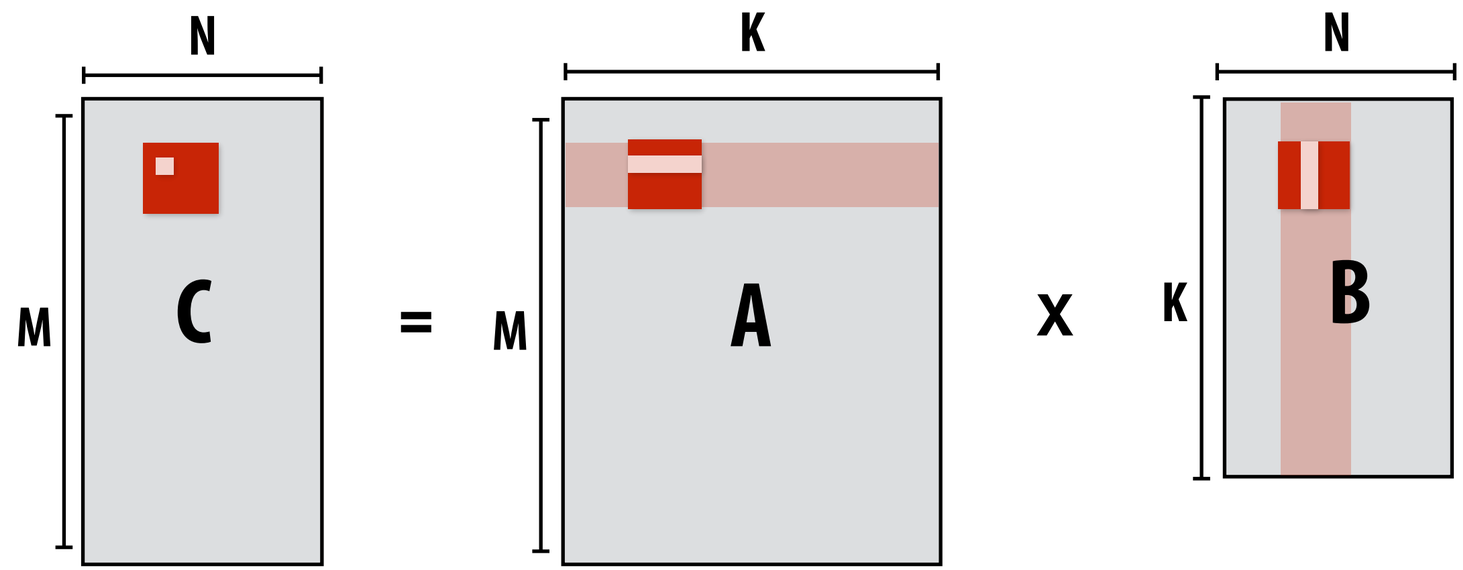
```
float A[M][K];
float B[K][N];
float C[M][N];

// 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

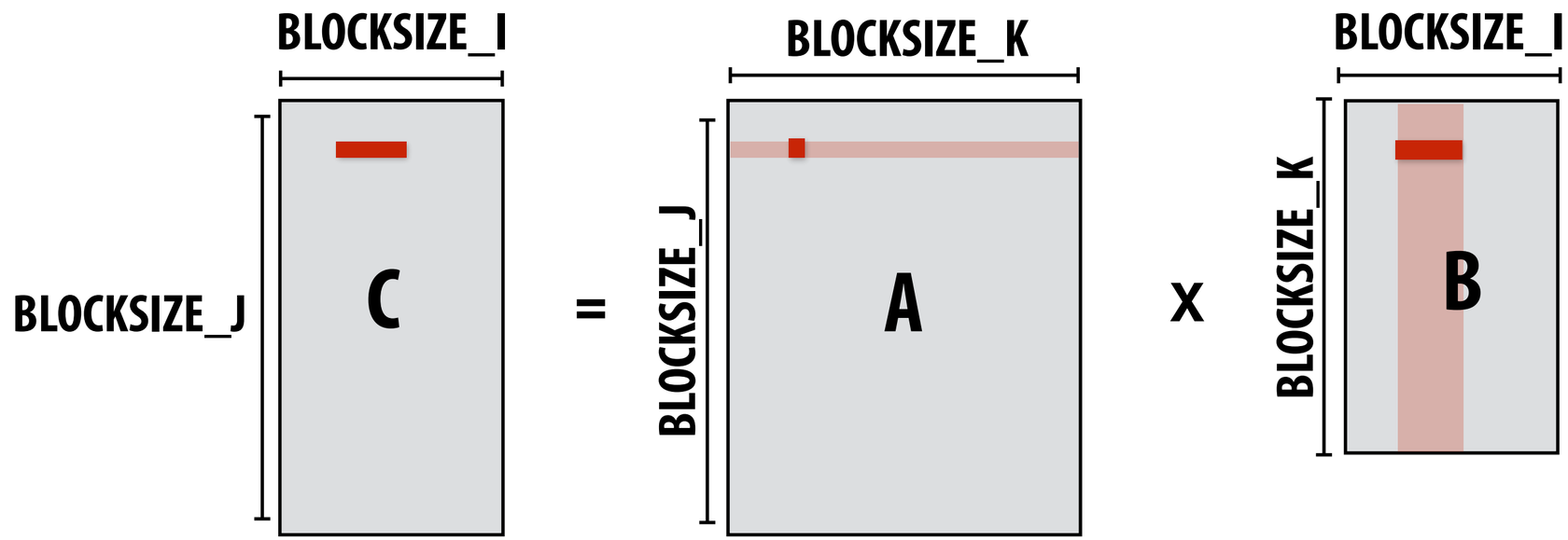
Exploit multi-level of memory hierarchy

```
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);
   }
}
```

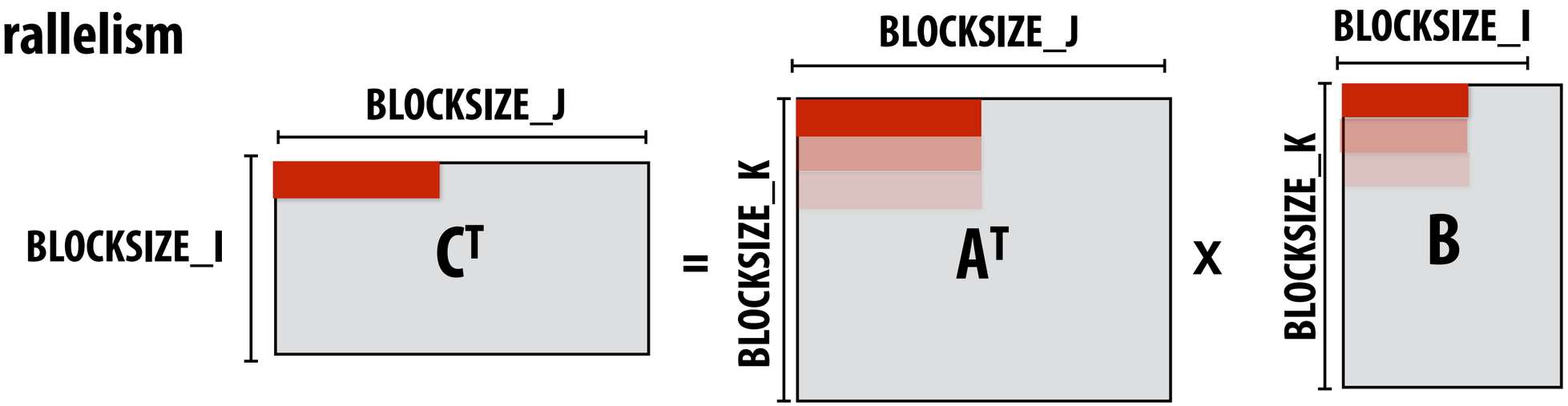
Vectorize i loop

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)

Consider SIMD parallelism within a block

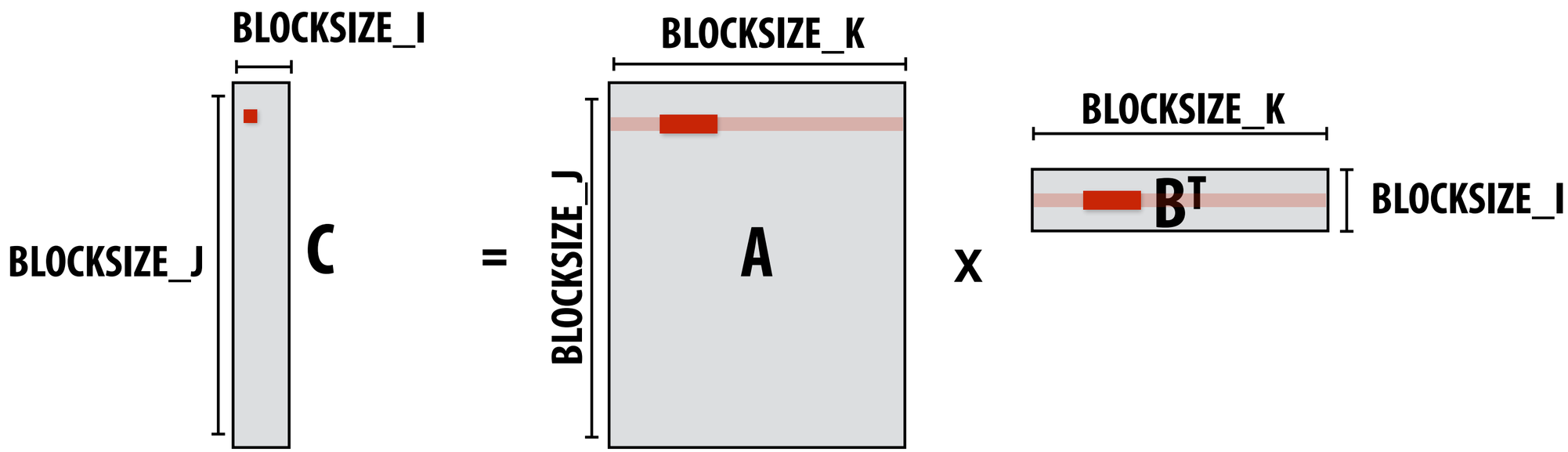



```
// assume blocks of A and C are pre-transposed as Atrans
for (int j=0; j<BLOCKSIZE_J; j+=SIMD_WIDTH) {
   for (int i=0; i<BLOCKSIZE_I; i++) {
      simd_vec C_accum = vec_load(&Ctrans[iblock+i][jblock+j]);
      for (int k=0; k<BLOCKSIZE_K; k++) {
         // C = A*B + C
         simd_vec A_val =);
         simd_muladd(vec_load(&Atrans[kblock+k][jblock+j], vec_load(&B[kblock+k][iblock+i]), C_accum);
      }
      vec_store(&Ctrans[iblock+i][jblock+j], C_accum);
   }
}
```

Vectorize *j* loop

All loads are now SIMD vector loads (removed single element load from A + splat)

Blocked dense matrix multiplication (3)




```
...
for (int j=0; j<BLOCKSIZE_J; j++)
   for (int i=0; i<BLOCKSIZE_I; i++) {
      float C_scalar = C[jblock+j][iblock+i];
      for (int k=0; k<BLOCKSIZE_K; k+=SIMD_WIDTH) {
         // C_scalar = dot(A,B) + C_scalar
         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

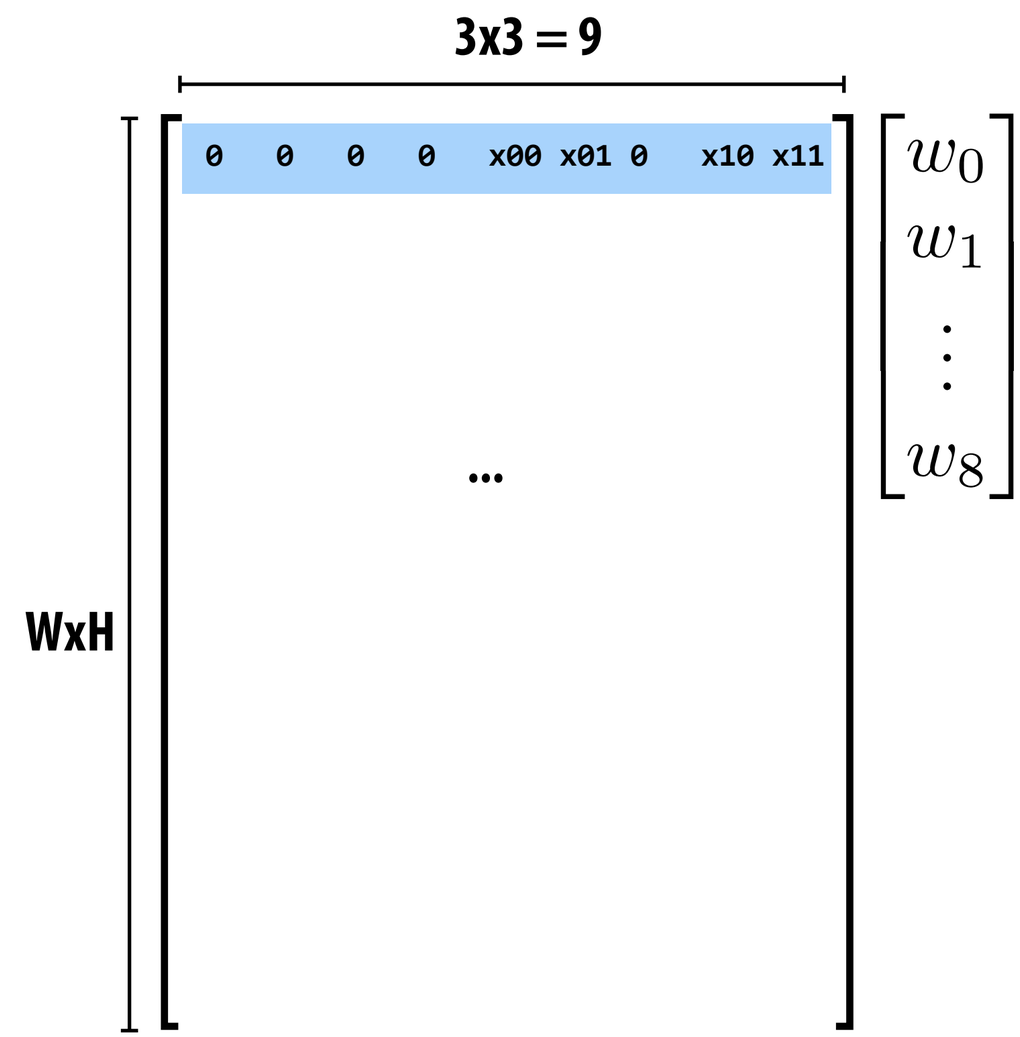
Convolution as matrix-vector product

Construct matrix from elements of input image

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}	...			
...	...	...	...				

O(N) storage overhead for filter with N elements

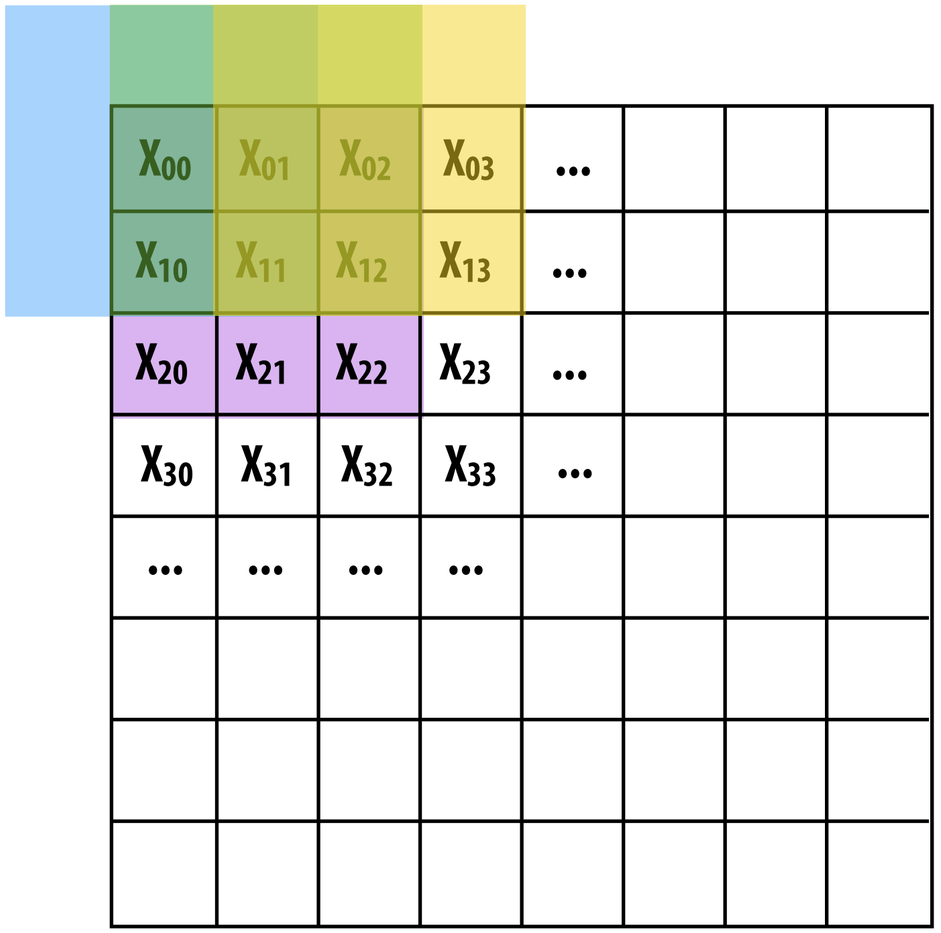
Must construct input data matrix



Note: 0-pad matrix

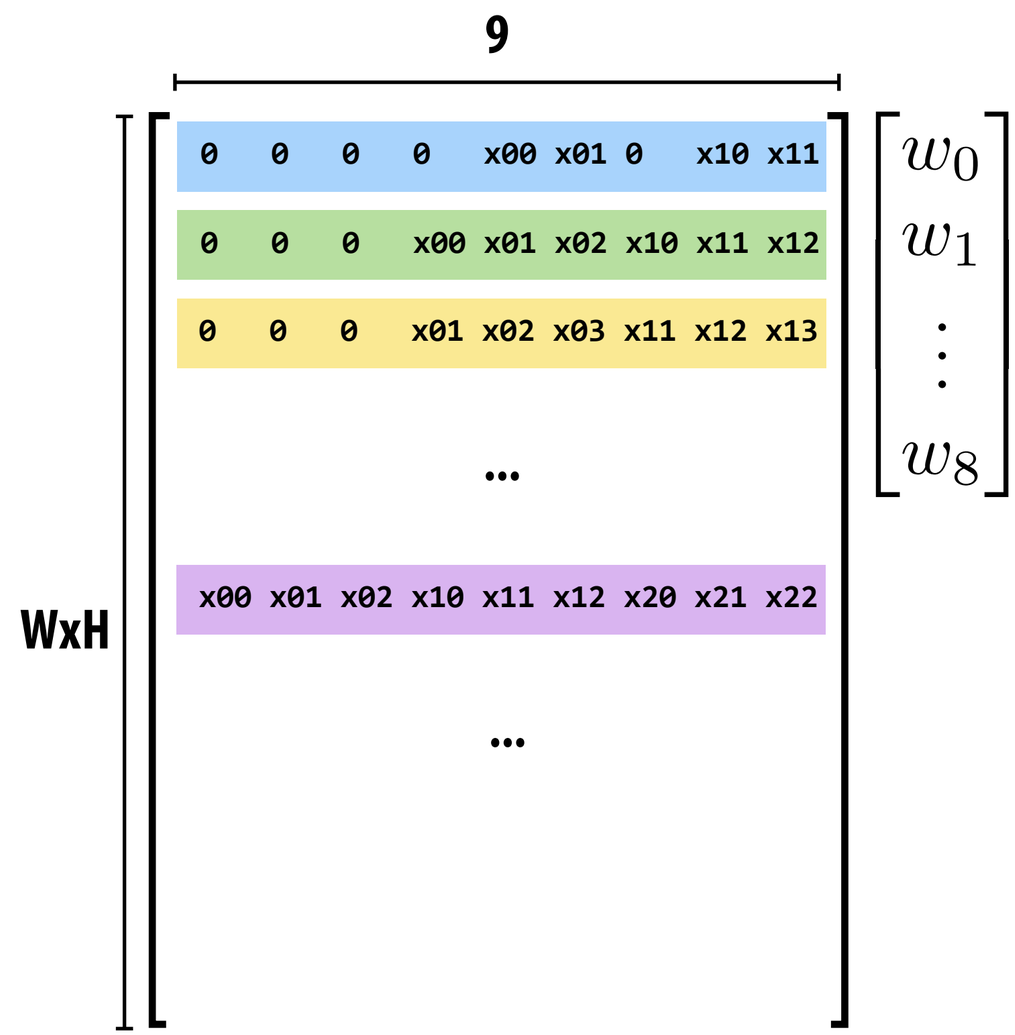
3x3 convolution as matrix-vector product

Construct matrix from elements of input image



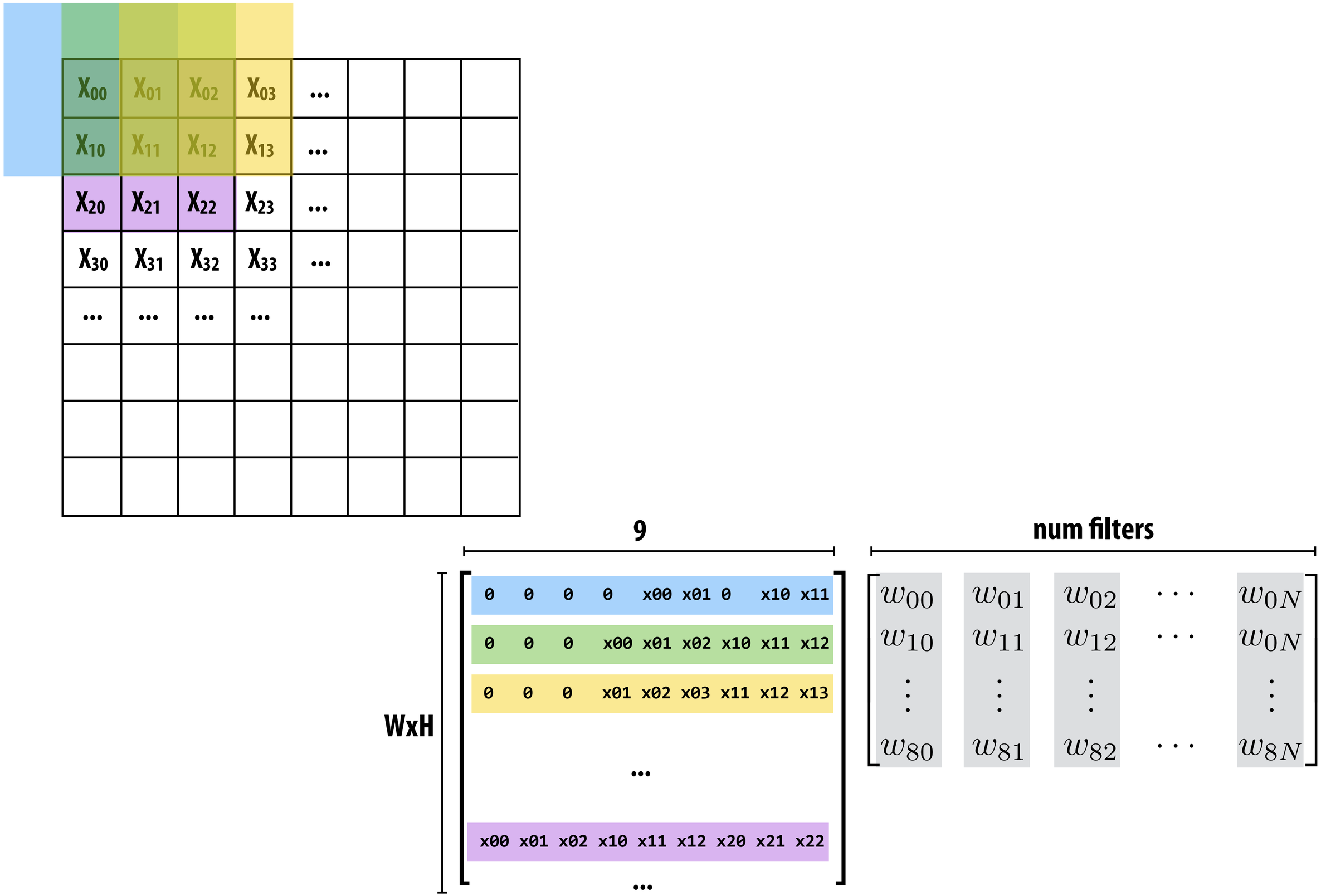
O(N) storage overhead for filter with N elements

Must construct input data matrix

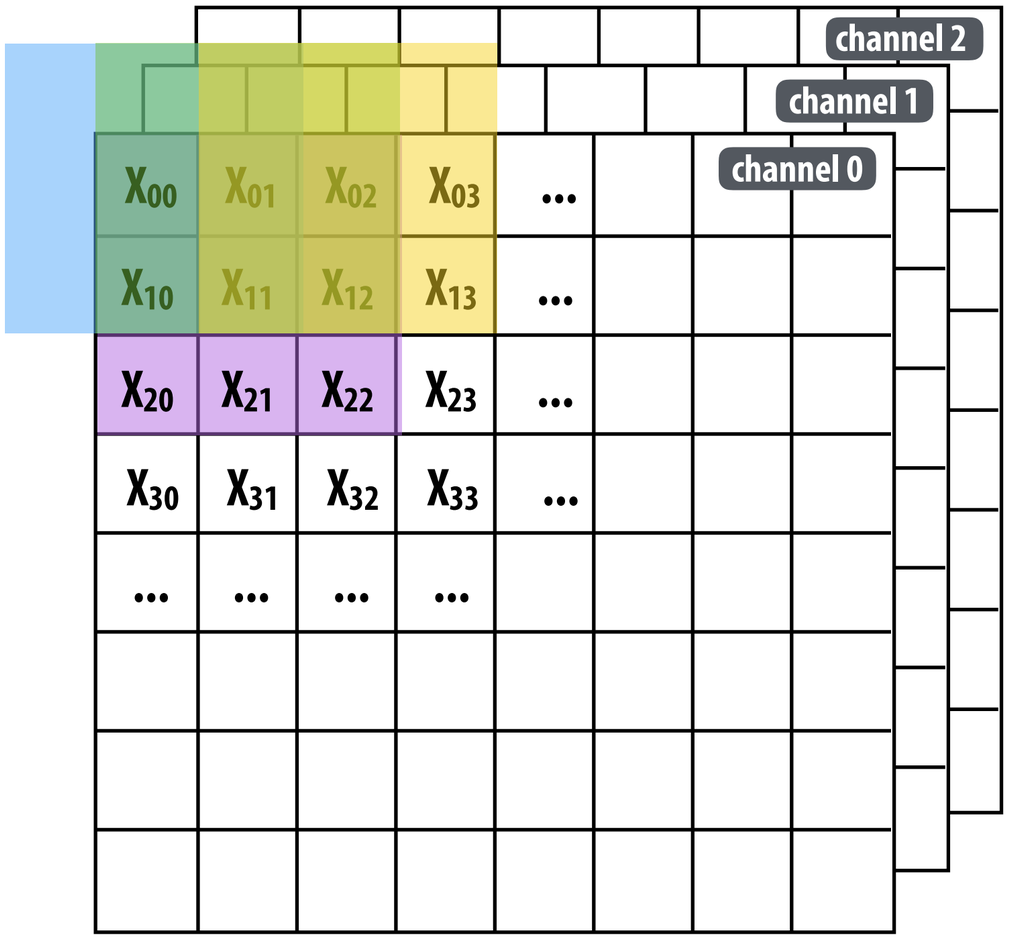


Note: 0-pad matrix

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

9 x num input channels

channel 0 values | **channel 1 values** | **channel 2 values**

WxH

```
0   0   0   0   x00 x01 0   x10 x11
0   0   0   x00 x01 x02 x10 x11 x12
0   0   0   x01 x02 x03 x11 x12 x13
...
x00 x01 x02 x10 x11 x12 x20 x21 x22
...
```

num filters

$$\begin{bmatrix} w_{000} & w_{001} & w_{002} & \cdots & w_{00N} \\ w_{010} & w_{011} & w_{012} & \cdots & w_{01N} \\ \vdots & \vdots & \vdots & & \vdots \\ w_{080} & w_{081} & w_{082} & \cdots & w_{08N} \\ w_{100} & w_{101} & w_{102} & \cdots & w_{10N} \\ w_{110} & w_{111} & w_{112} & \cdots & w_{11N} \\ \vdots & \vdots & \vdots & & \vdots \\ w_{180} & w_{181} & w_{182} & \cdots & w_{18N} \\ w_{200} & w_{201} & w_{202} & \cdots & w_{20N} \\ w_{210} & w_{211} & w_{212} & \cdots & w_{21N} \\ \vdots & \vdots & \vdots & & \vdots \\ w_{280} & w_{281} & w_{282} & \cdots & w_{28N} \end{bmatrix}$$

VGG memory footprint

Calculations assume 32-bit values (image batch size = 1)

inputs/outputs get multiplied by image batch size

multiply by next layer's conv window size to form input matrix to next conv layer!!! (for VGG, this is a 9x data amplification)

	weights mem:	output size (per image)	(mem)
input: 224 x 224 RGB image	—	224x224x3	150K
conv: (3x3x3) x 64	6.5 KB	224x224x64	12.3 MB
conv: (3x3x64) x 64	144 KB	224x224x64	12.3 MB
maxpool	—	112x112x64	3.1 MB
conv: (3x3x64) x 128	228 KB	112x112x128	6.2 MB
conv: (3x3x128) x 128	576 KB	112x112x128	6.2 MB
maxpool	—	56x56x128	1.5 MB
conv: (3x3x128) x 256	1.1 MB	56x56x256	3.1 MB
conv: (3x3x256) x 256	2.3 MB	56x56x256	3.1 MB
maxpool	—	28x28x256	766 KB
conv: (3x3x256) x 512	4.5 MB	28x28x512	1.5 MB
conv: (3x3x512) x 512	9 MB	28x28x512	1.5 MB
conv: (3x3x512) x 512	9 MB	28x28x512	1.5 MB
maxpool	—	14x14x512	383 KB
conv: (3x3x512) x 512	9 MB	14x14x512	383 KB
conv: (3x3x512) x 512	9 MB	14x14x512	383 KB
conv: (3x3x512) x 512	9 MB	14x14x512	383 KB
maxpool	—	7x7x512	98 KB
fully-connected 4096	392 MB	4096	16 KB
fully-connected 4096	64 MB	4096	16 KB
fully-connected 1000	15.6 MB	1000	4 KB
soft-max		1000	4 KB

Direct implementation of conv layer

```
float input[INPUT_HEIGHT][INPUT_WIDTH][INPUT_DEPTH];
float output[INPUT_HEIGHT][INPUT_WIDTH][LAYER_NUM_FILTERS];
float layer_weights[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[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_CONVY; jj++) // spatial convolution
                  for (int ii=0; ii<LAYER_CONVX; ii+)  // spatial convolution
                     output[j][i][f] += layer_weights[f][jj][ii][kk] * input[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)

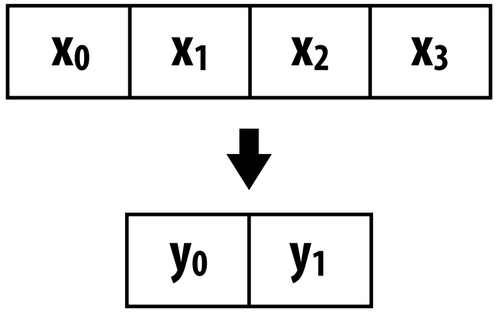
Avoid cost of materializing input matrix.
Avoids footprint O(N) footprint increase by avoiding materializing input matrix
In theory loads O(N) times less data (potentially higher arithmetic intensity… but matrix mult is typically compute-bound)
But must roll your own highly optimized implementation of complicated loop nest.

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)

1D 3-tap total cost:
4 multiplies
8 additions
(4 to compute m's + 4 to reduce final result)

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

Reducing network footprint

- Large storage cost for model parameters
  - AlexNet model: ~200 MB
  - VGG-16 model: ~500 MB
  - This doesn't even account for intermediates during evaluation

- Footprint: cumbersome to store, download, etc.
  - 500 MB app downloads make users unhappy!

- Consider energy cost of 1B parameter network
  - Running on input stream at 20 Hz
  - 640 pJ per 32-bit DRAM access
  - (20 x 1B x 640pJ) = 12.8W for DRAM access

    (more than power budget of any modern smartphone)

Compressing a network

[Han ICLR16]

Step 1: prune low-weight links (iteratively retrain network, then prune)

- Over 90% of weights can be removed without significant loss of accuracy
- Store weight matrices in compressed sparse row (CSR) format

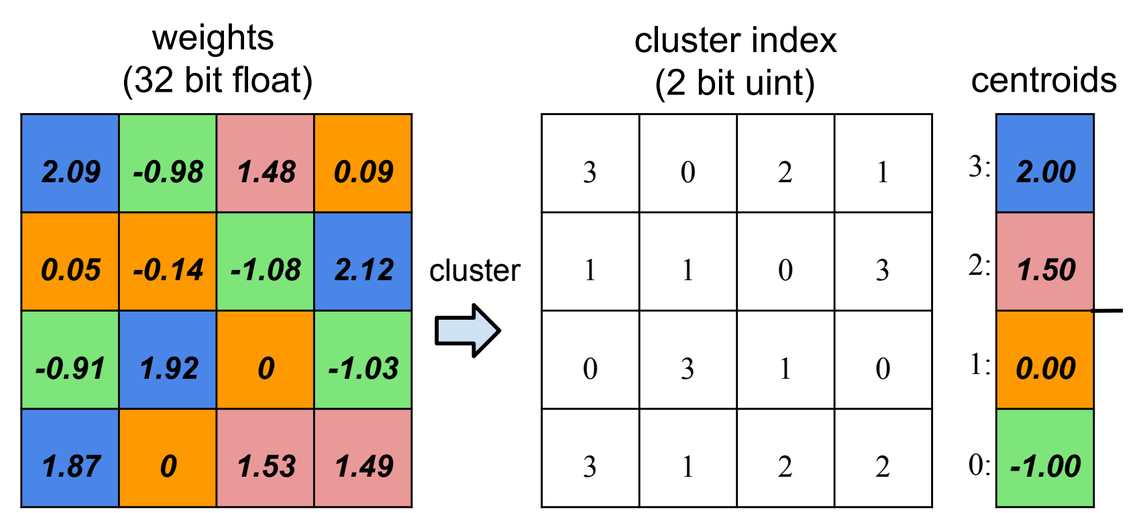
```
Indicies    1    4   9  ...
Value     1.8  0.5  2.1
```

0	1.8	0	0	0.5	0	0	0	0	1.1	...

Step 2: weight sharing: make surviving connects share a small set of weights

- Cluster weights via k-means clustering (irregular ("learned") quantization)
- Compress weights by only storing cluster index (lg(k) bits)
- Retrain network to improve quality of cluster centroids

Step 3: Huffman encode quantized weights and CSR indices



VGG-16 compression

[Han ICLR16]

Substantial savings 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)

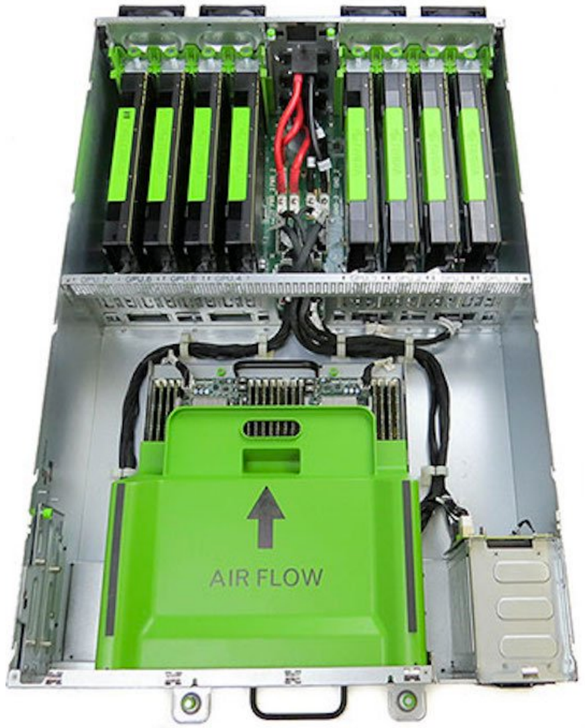
H = Huffman encode

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**×

Deep neural networks on GPUs

- **Today, best performing DNN implementations target GPUs**
  - High arithmetic intensity computations (computational characteristics similar to dense matrix-matrix multiplication)
  - Benefit from flop-rich architectures
  - Highly-optimized library of kernels exist for GPUs (cuDNN)
    - Most CPU-based implementations use basic matrix-multiplication-based formulation (good implementations could run faster!)



Facebook's Big Sur

Emerging architectures for deep learning?

- **NVIDIA Pascal (upcoming GPU)**
  - Adds double-throughput 16-bit floating point ops
  - Feature that is already common on mobile GPUs

- **Intel Xeon Phi (Knights Landing)**
  - Potential competitor for NVIDIA GPUs

- **FPGAs, ASICs?**
  - Not new: FPGA solutions have been explored for years
  - Significant amount of ongoing industry and academic research

Programming frameworks for deep learning

- Heavyweight processing (low-level kernels) carried out by target-optimized libraries (NVIDIA cuDNN, Intel MKL)

- Popular frameworks use these kernel libraries
  - Caffe, Torch, Theano, TensorFlow, MxNet

- DNN application development = constructing novel network topologies
  - Programming by constructing networks
  - Significant interest in new ways to express network construction

Summary: efficiently evaluating convnets

- **Computational structure**
  - Convlayers: high arithmetic intensity, significant portion of cost of evaluating a network
  - Similar data access patterns to dense-matrix multiplication (exploiting temporal reuse is key)
  - But straight reduction to matrix-matrix multiplication is often sub-optimal
  - Work-efficient techniques for convolutional layers (FFT-based, Wingrad convolutions)
- **Large numbers of parameters: significant interest in reducing size of networks for both training and evaluation**
  - Pruning: remove least important network links
  - Quantization: low-precision parameter representations often suffice
- **Many ongoing studies of specialized hardware architectures for efficient evaluation**
  - Future CPUs/GPUs, ASICs, FPGS, …
  - Specialization will be important to achieving "always on" applications