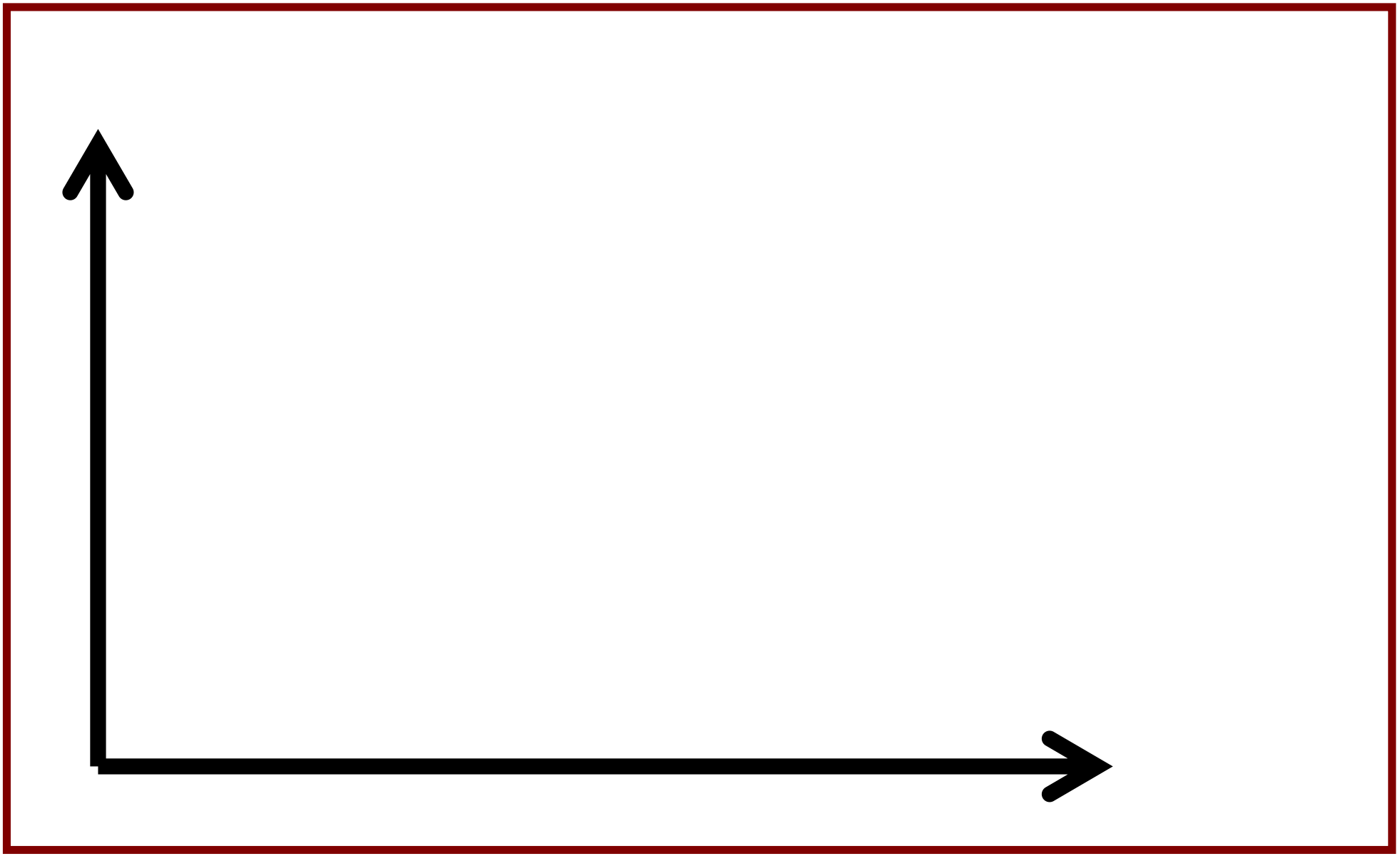
深度学习

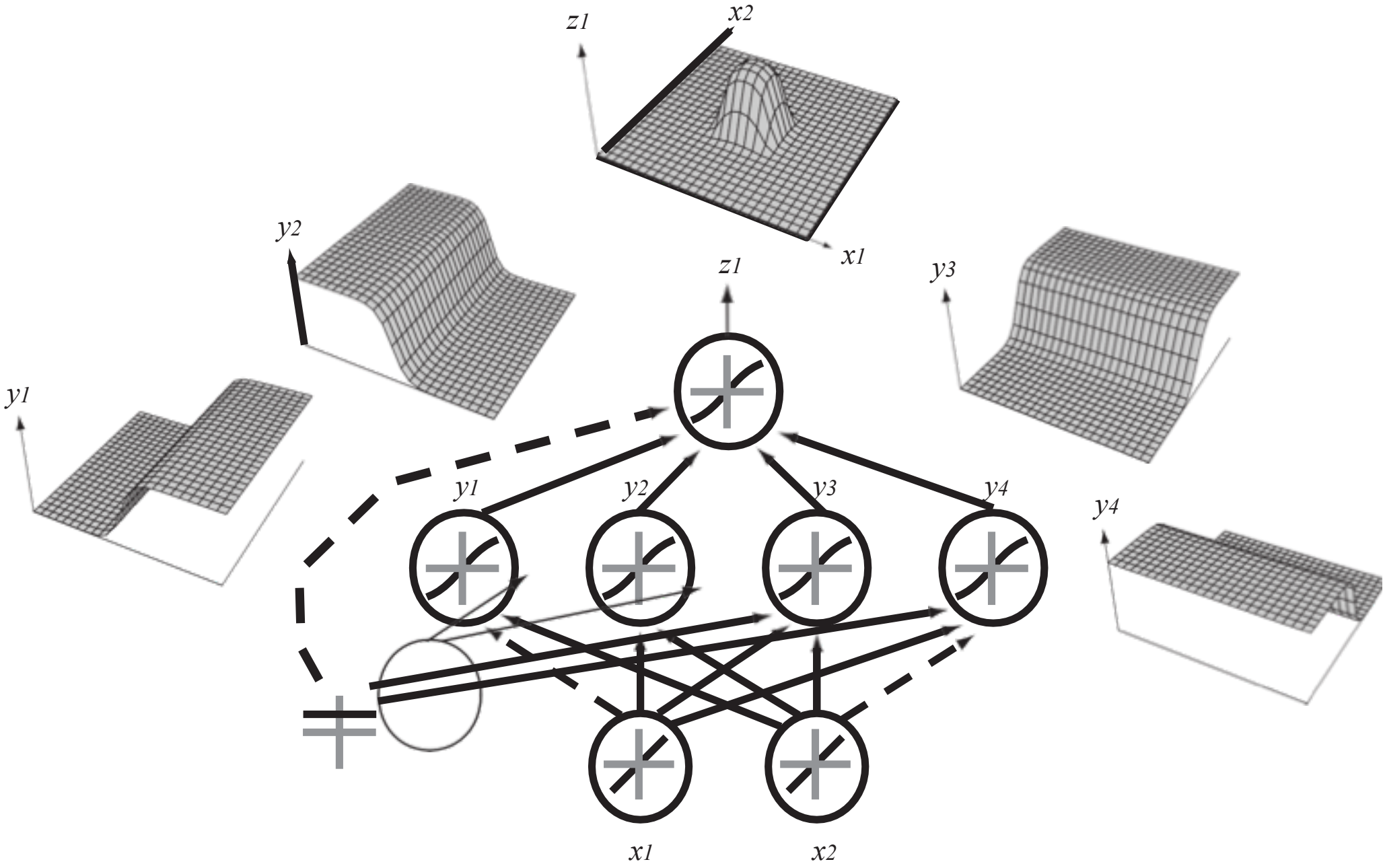
第五讲

王胤

Recap: Limits of Perceptrons (XOR)

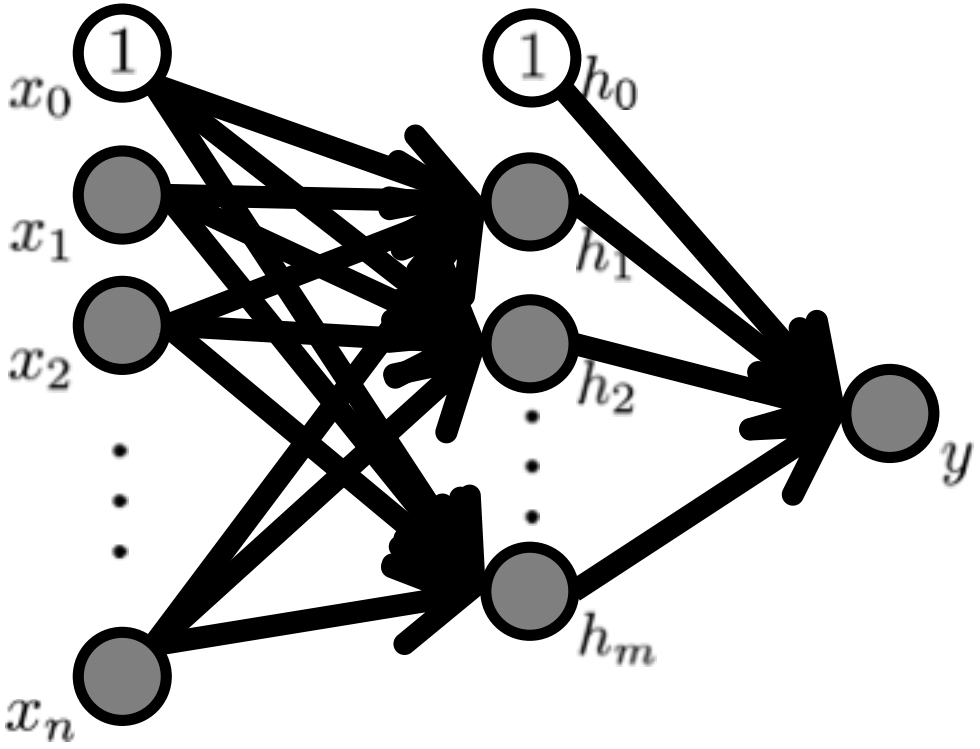


Recap: Nonlinear Activation



$$\Delta w_j = \alpha h_j (t - y) f'(net)$$

$$= \alpha h_j \delta^y$$

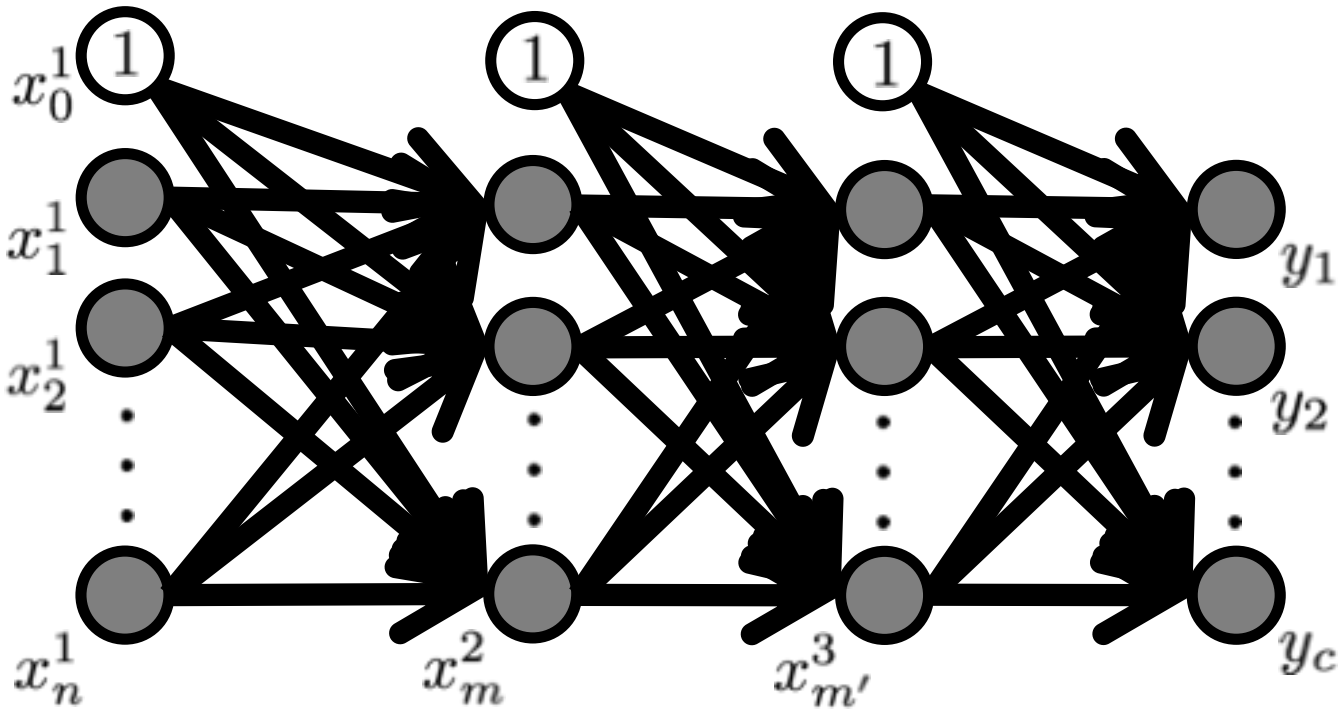


$$\Delta w_{ij} = \alpha x_i w_j \delta^y f'(net_j)$$

$$= \alpha x_i \delta_j^h$$

Recap: Putting it all together (forward pass)

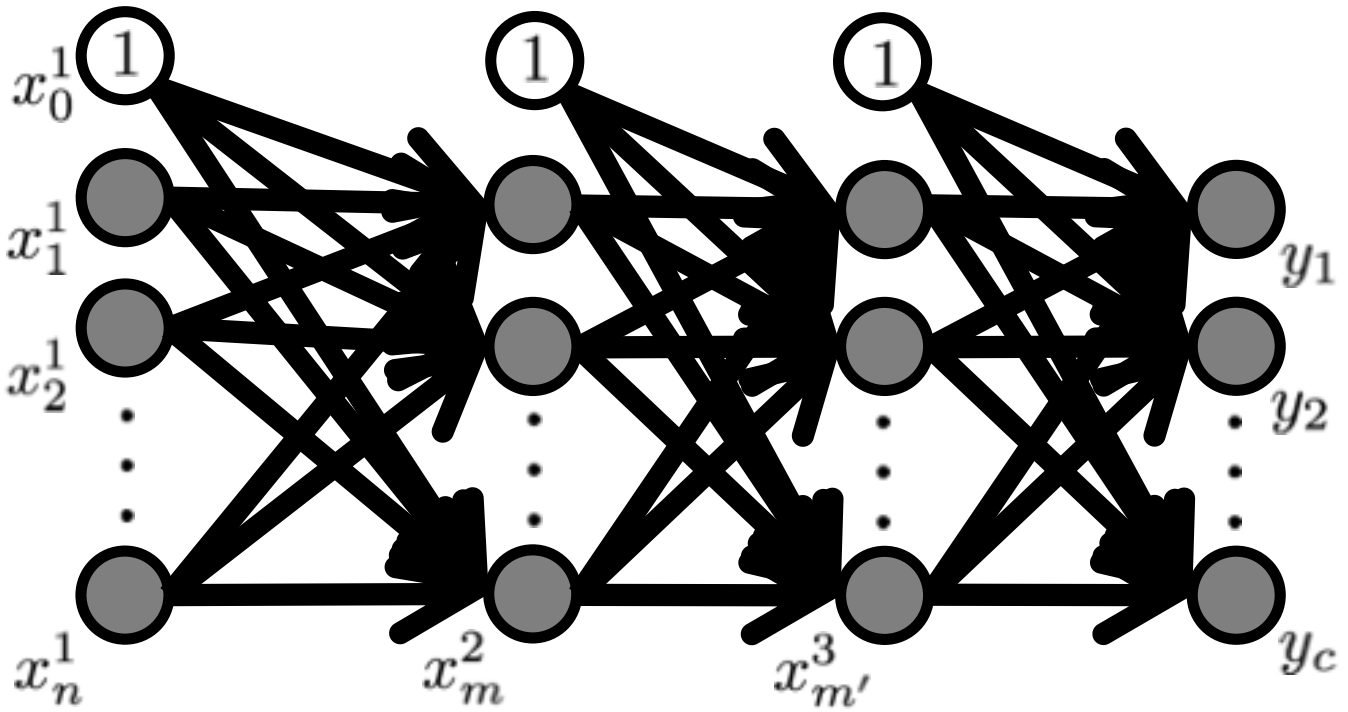
$$x_k^{i+1} = f(\sum_{j=0} w_{jk}^i x_j^i) \qquad y_k = f(\sum_{j=0} w_{jk}^3 x_j^3)$$



Recap: Putting it all together (backward pass)

$$\delta_k^y = (t_k - y_k) f'(net_k^y) \quad \Delta w_{jk}^y = \alpha x_j^3 \delta_k^y$$

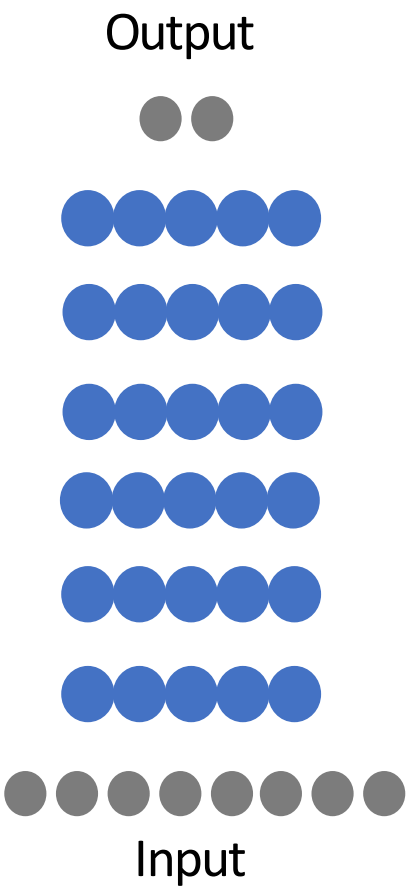
$$\delta_j^{i-1} = f'(net_j^{i-1}) \sum_k w_{jk}^i \delta_k^i \quad \Delta w_{jk}^i = \alpha x_j^i \delta_k^i$$



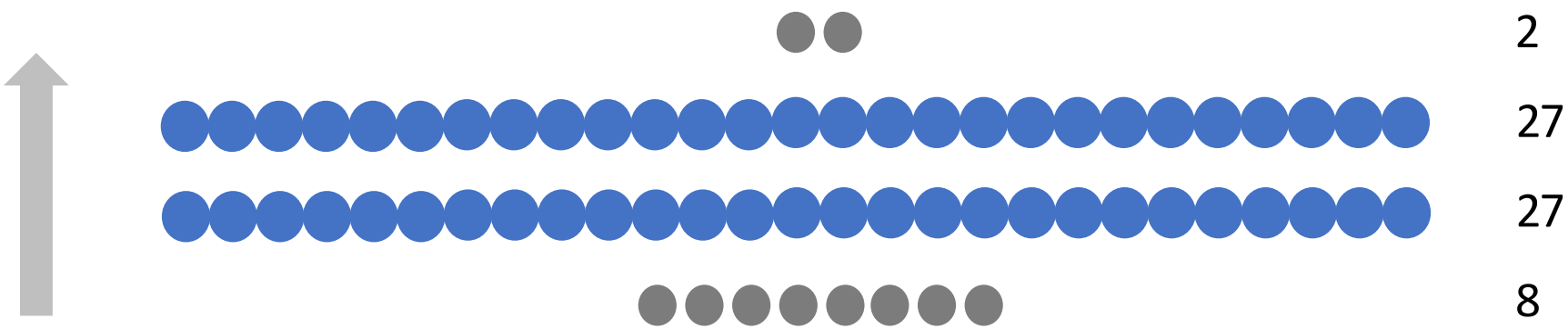
Recap: Training Tips

- How many hidden units?
- Train/validation/test
- Training data size
- Deep vs. shallow networks
- Learning rate
- Batch vs. incremental updates
- Learning rate schedule
- Relu vs. Sigmoid

Weight initialization

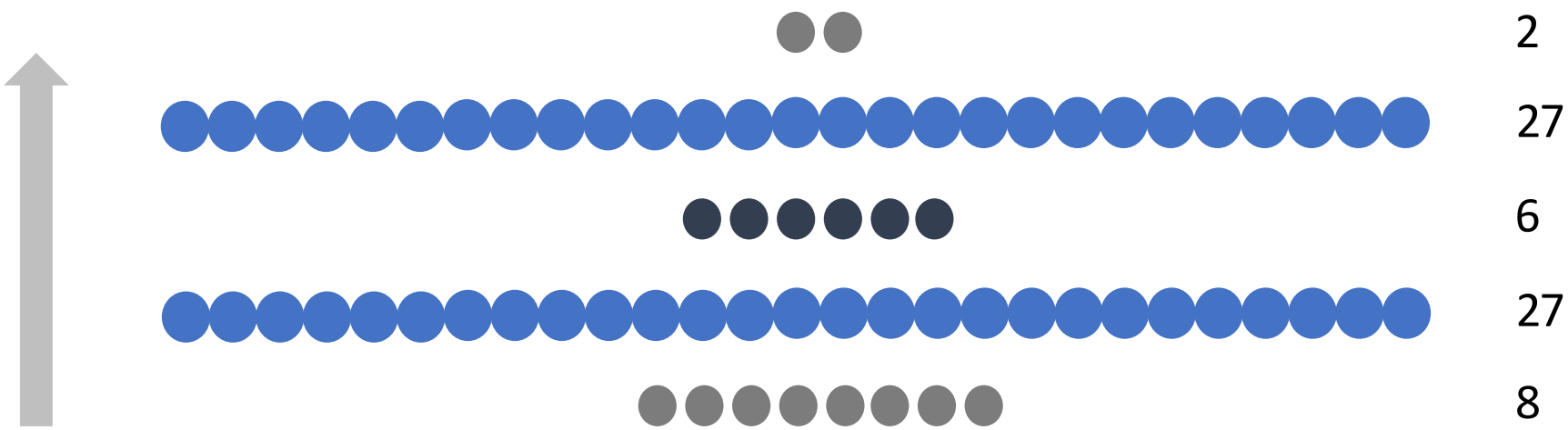


Network compression



Number of parameters:

Network compression



Number of parameters:

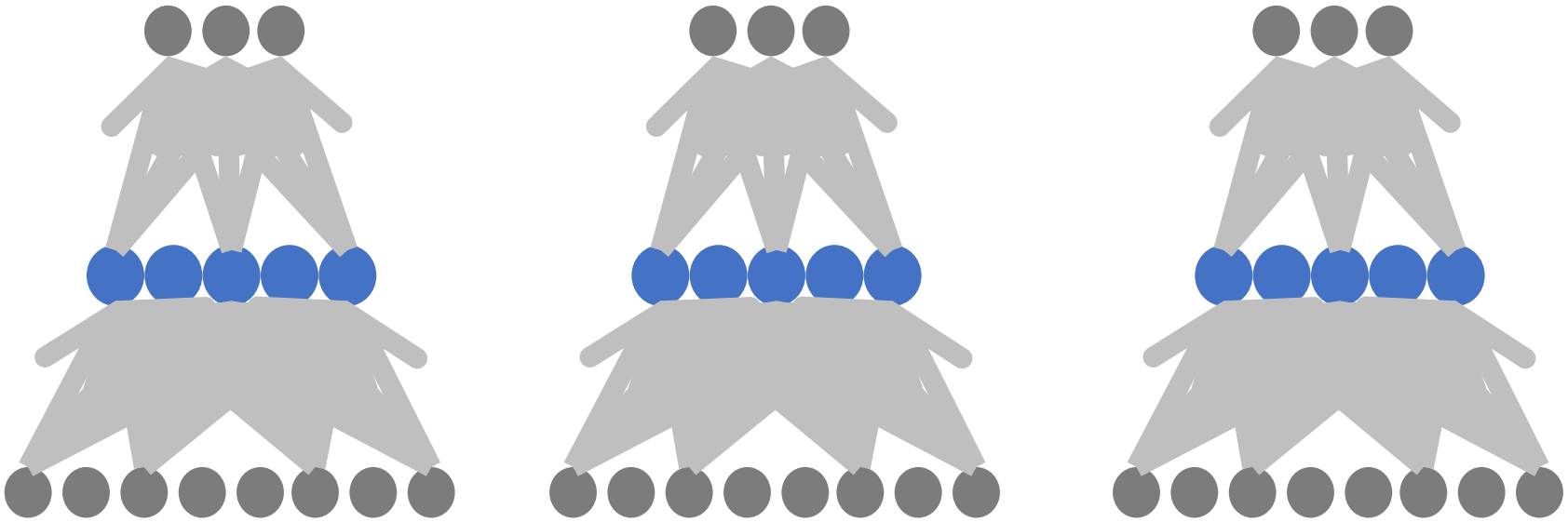
Weight regularization

L1 and L2 regularization of the weights

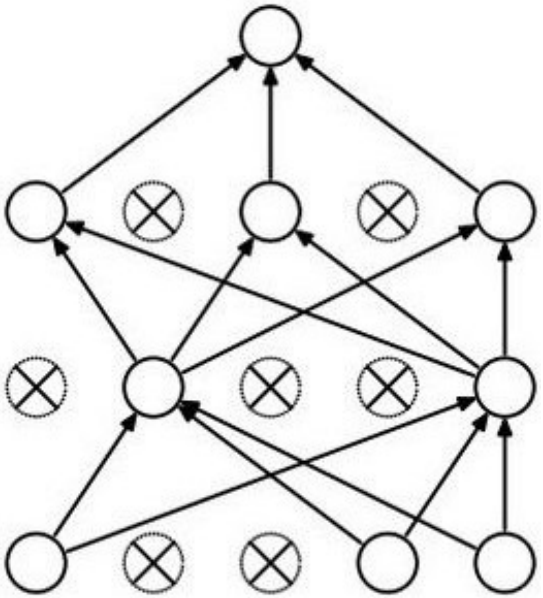
L1: $\min \sum_i |w_i|$ L2: $\min \sum_i w_i^2$

Derivatives

Dropout



Dropout



Forces the network to have a redundant representation.

Expanding the dataset

Image: CÉSAR LAURENT

Batch normalization

- Normalize the activations of the units over each mini-batch
- Allows use of faster learning rate, higher accuracy.

$$x_i' = \frac{x_i - \bar{x}_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

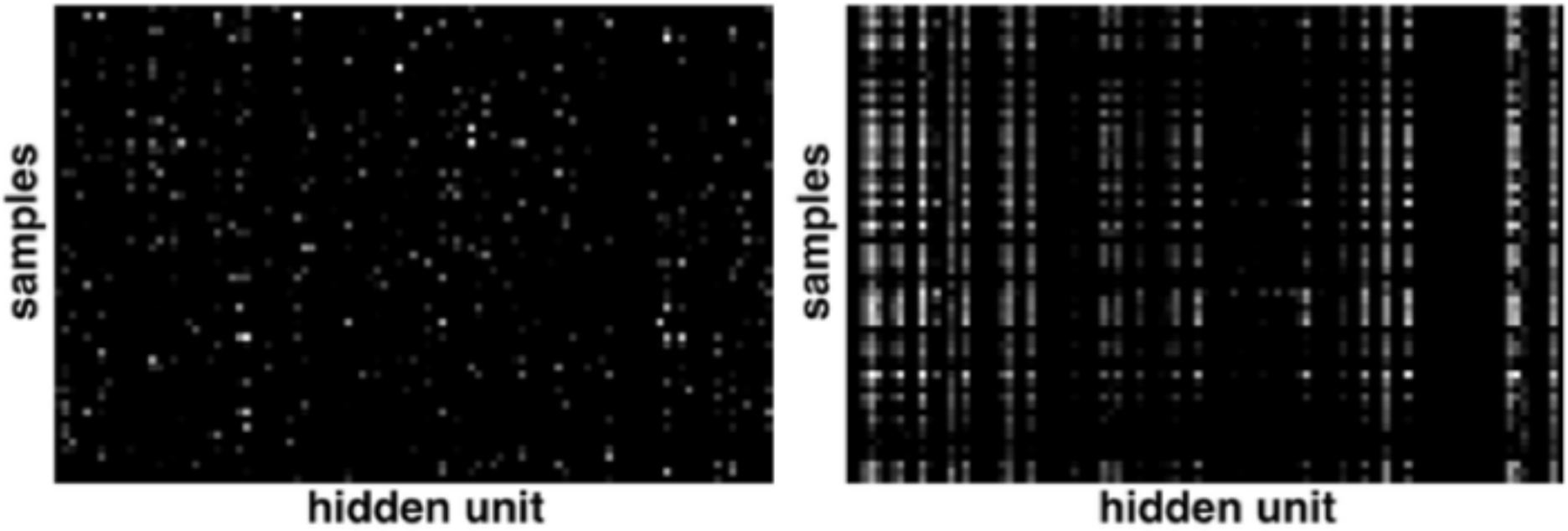
"internal covariate shift"

Training odds and ends

- Training can take weeks for larger datasets, even with GPUs
- Guaranteed to find a local minimum.
- Local minimum is usually pretty good, especially when using a larger number of units with regularization.
- Always examine the loss during training (train and validation).

Debugging training

- Inspect hidden units

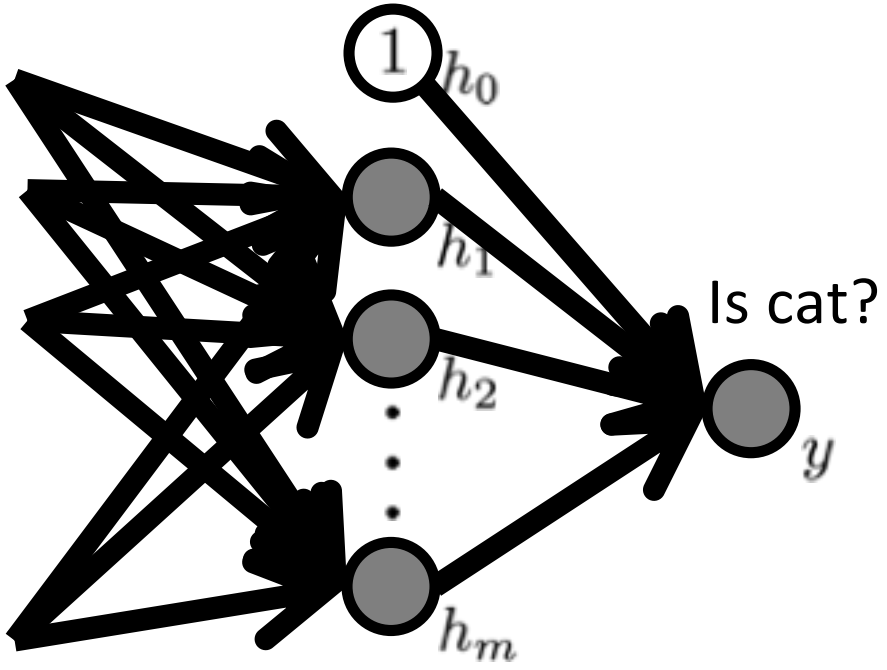


Slide: M. Ranzato

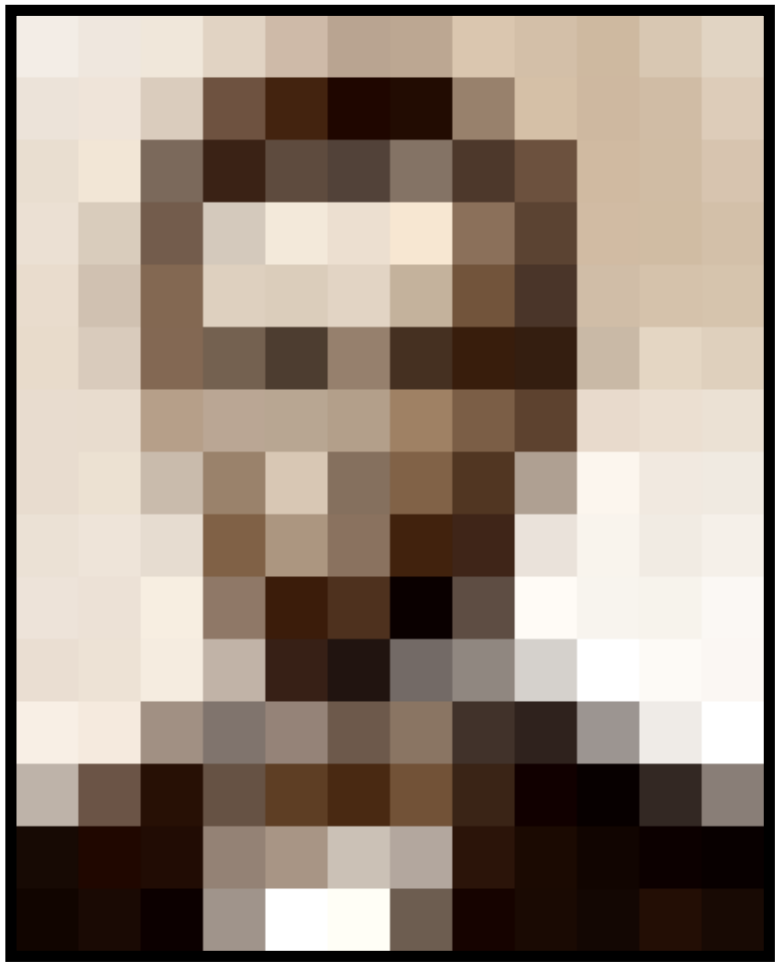
Debugging Gradient Descent

- Gradient checking

Images



What do computers see?

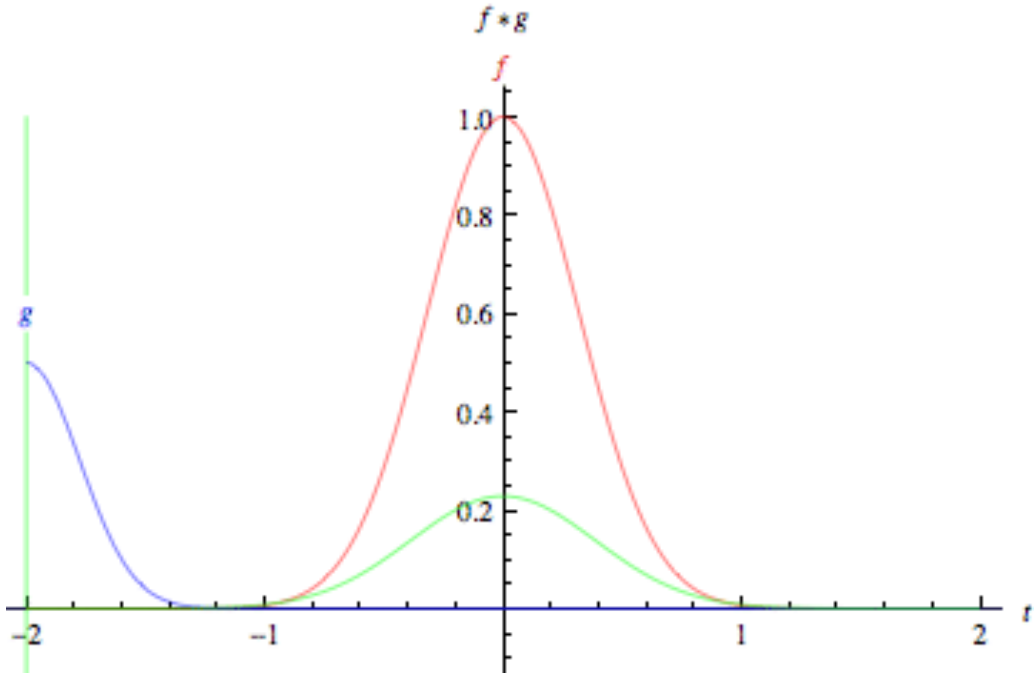


Images can be viewed as a 2D function.

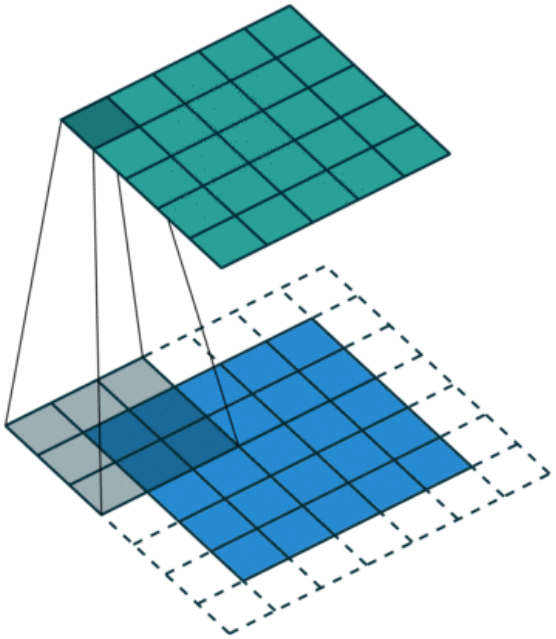
Classifying images

320x240 pixels = 76,800

1D Convolution



Filtering (2D Convolution)



Linear filtering =

Applying a local function to the image using a sum of weighted neighboring pixels.

Such as blurring:

0	0	0	0	0	0
0	0	0	0	0	0
0	0	200	0	0	0
0	0	0	0	0	0
0	0	0	100	0	0
0	0	0	0	0	0

Input image

*

0	0	0	0	0
0	0.11	0.11	0.11	0
0	0.11	0.11	0.11	0
0	0.11	0.11	0.11	0
0	0	0	0	0

Kernel/Weights

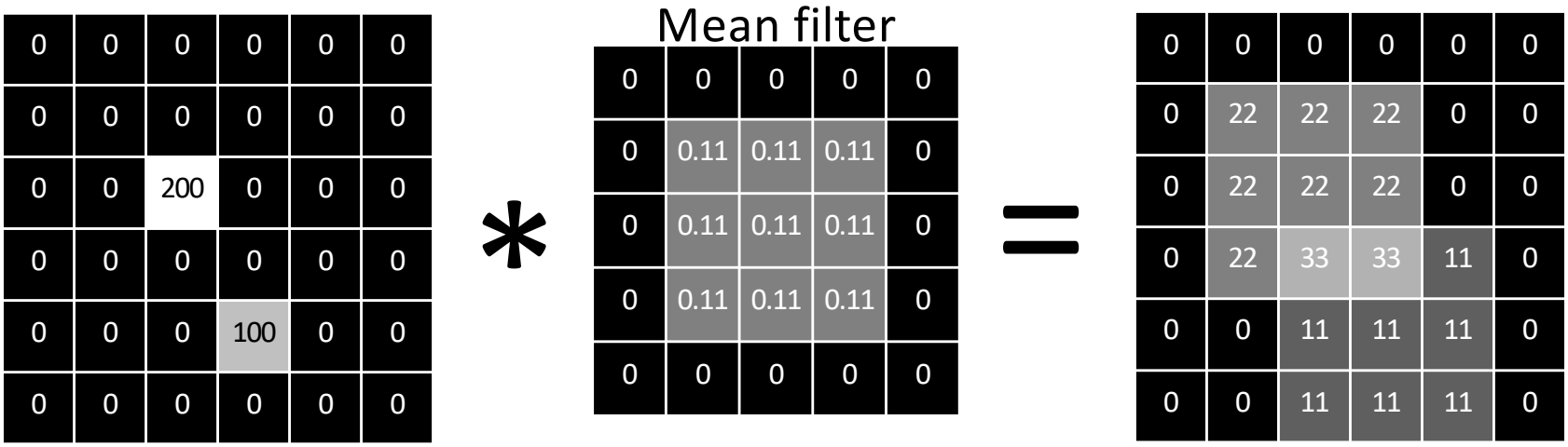
=

0	0	0	0	0	0
0	22	22	22	0	0
0	22	22	22	0	0
0	22	33	33	11	0
0	0	11	11	11	0
0	0	11	11	11	0

Output image

Filtering

$$g(x,y) = \sum_{x'} \sum_{y'} f(x+x', y+y') h(x', y')$$



Input image f Filter h Output image g

Filtering

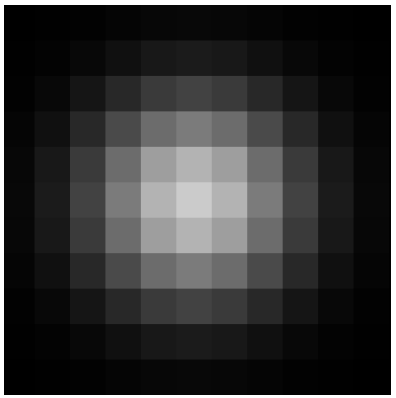
```
float filter[radius][radius];          // filter to apply
float inImage[height][width];          // input image
float outImage[height][width];         // outputimage


for(r=0;r<height;r++) {
        for(c=0;c<width;c++) {
                sum = 0;
                for(rd=-radius;rd<=radius;rd++)
                        for(cd=-radius;cd<=radius;cd++)
                                sum += inImage[r + rd][c + cd] *
                                        filter[rd + radius][cd + radius];
                outImage[r][c] = sum;
        }
}
```

What does filters do?

Gaussian blur

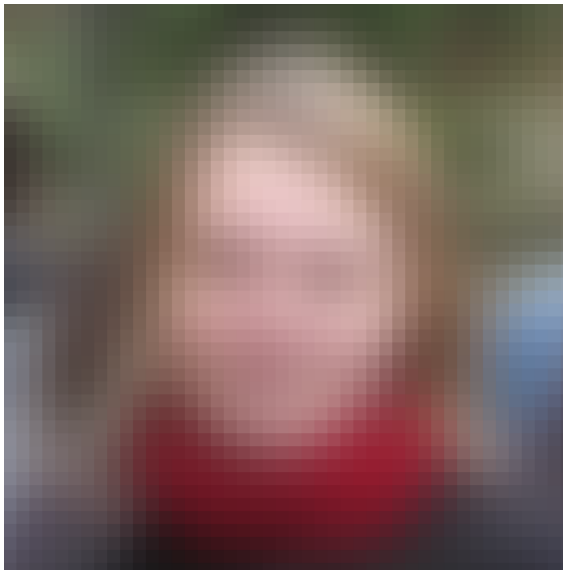
Input image f

$*$



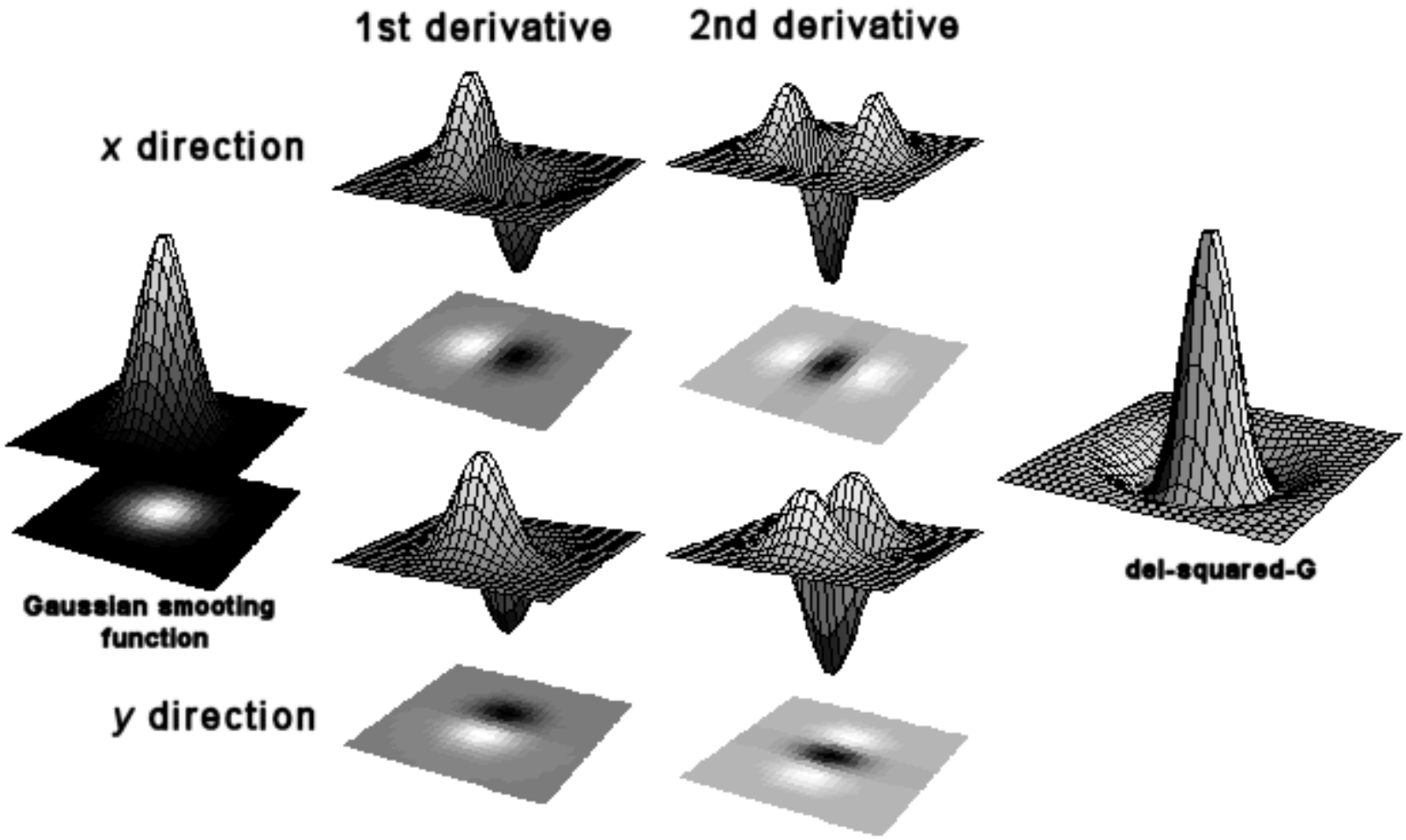
Filter h

$=$



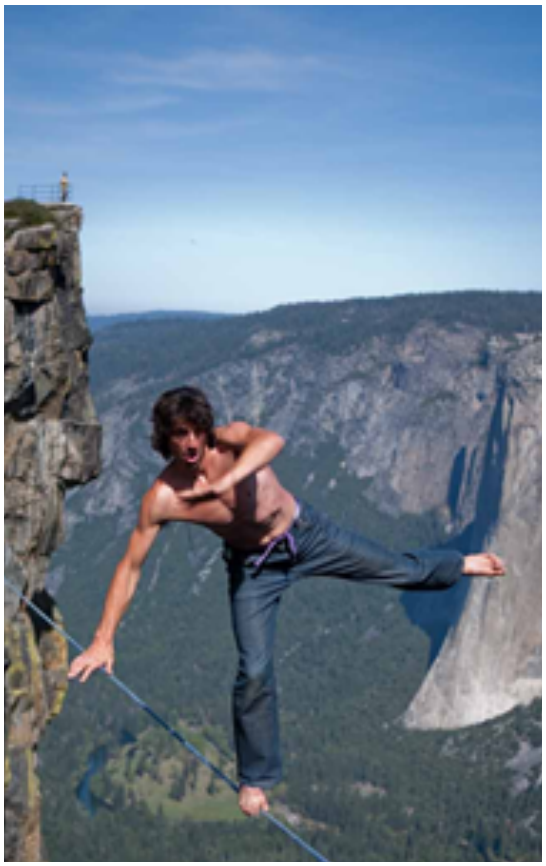
Output image g

First and second derivatives

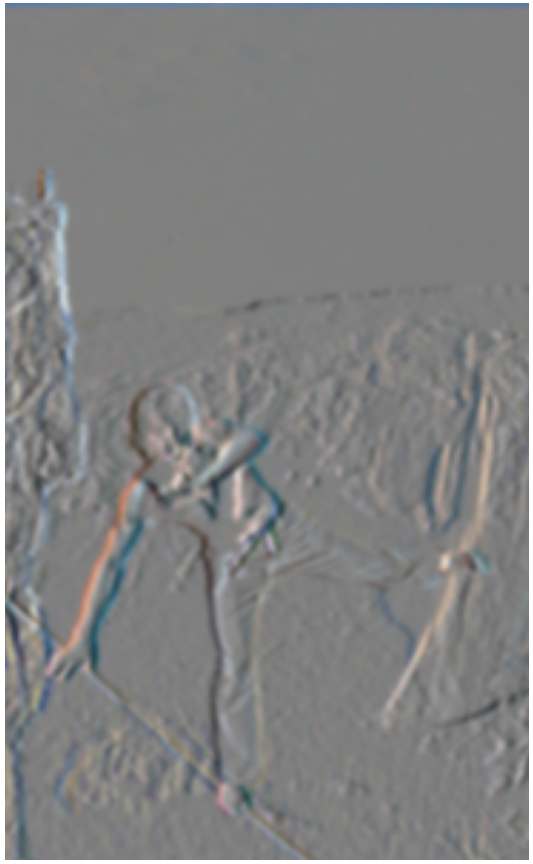


First and second derivatives

What are these good for?

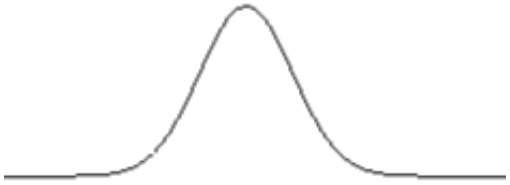


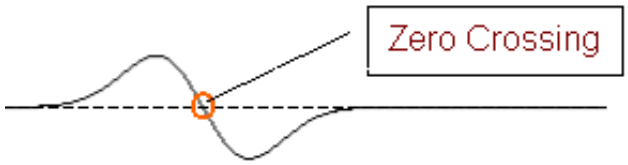
Original



First Derivative x

Second Derivative x, y

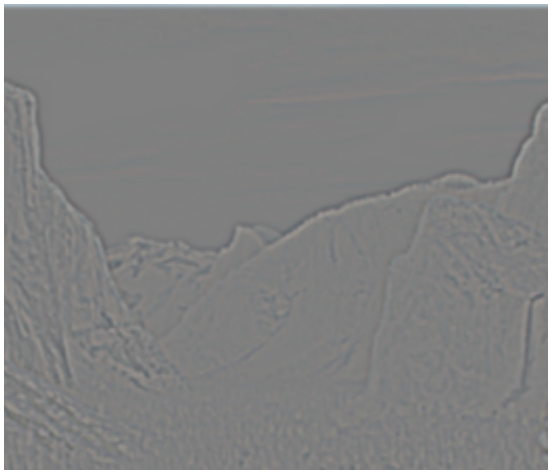




Just for fun…

$$Sharpen(x,y) = f(x,y) - \alpha(f * \nabla^2 \mathcal{G}_\sigma(x,y))$$

Original



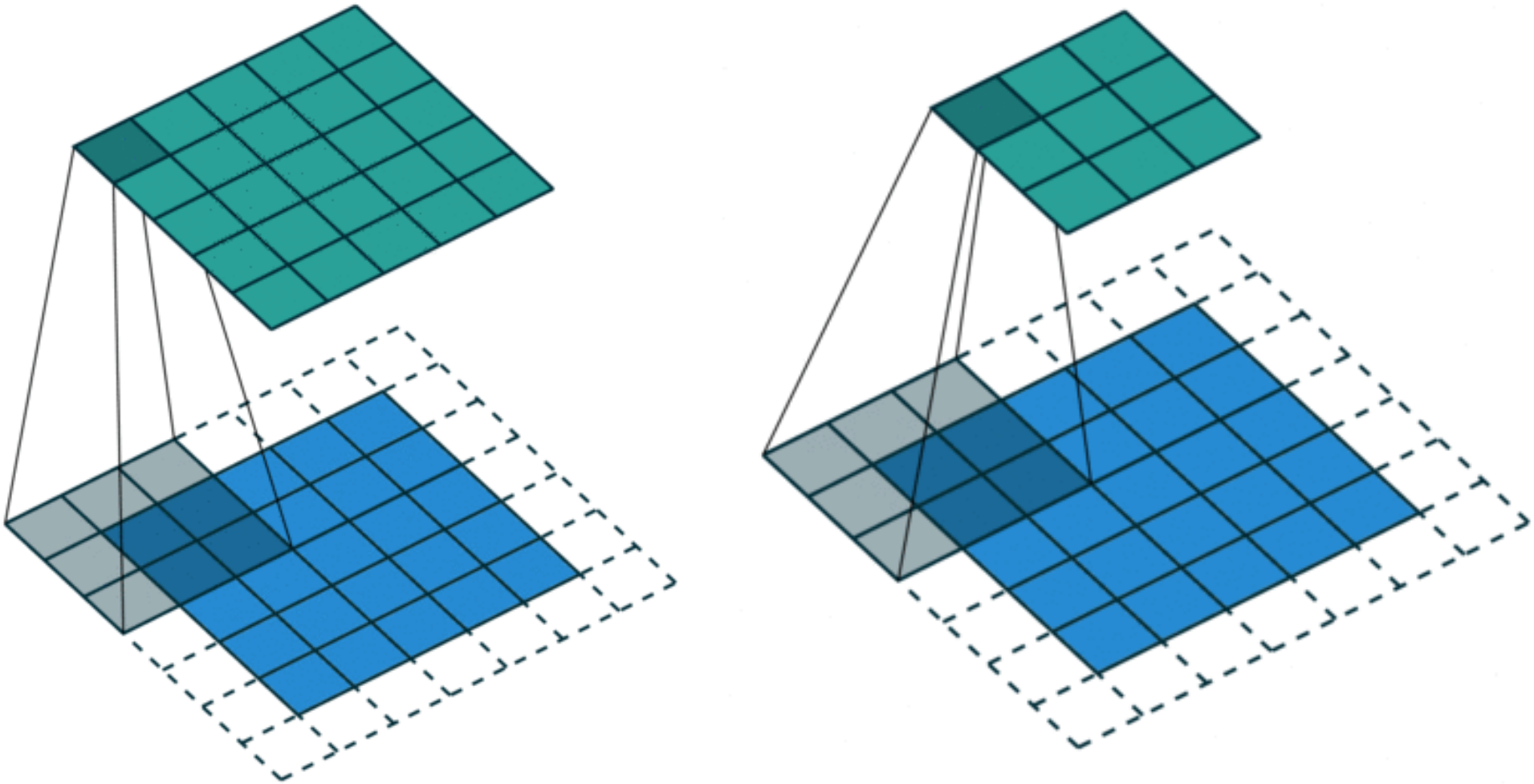
Second Derivative

Sharpened

Back to neural networks…

Why local features?

Strided Convolution



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	0	0	1	0	2	0
0	1	0	2	0	1	0
0	1	0	2	2	0	0
0	2	0	0	2	0	0
0	2	1	2	2	0	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	2	1	2	1	1	0
0	2	1	2	0	1	0
0	0	2	1	0	1	0
0	1	2	2	2	2	0
0	0	1	2	0	1	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	2	1	1	2	0	0
0	1	0	0	1	0	0
0	0	1	0	0	0	0
0	1	0	2	1	0	0
0	2	2	1	1	1	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

-1	0	1
0	0	1
1	-1	1

w0[:,:,1]

-1	0	1
1	-1	1
0	1	0

w0[:,:,2]

-1	1	1
1	1	0
0	-1	0

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	1	-1
0	-1	0
0	-1	1

w1[:,:,1]

-1	0	0
1	-1	0
1	-1	0

w1[:,:,2]

-1	1	-1
0	-1	-1
1	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

2	3	3
3	7	3
8	10	-3

o[:,:,1]

-8	-8	-3
-3	1	0
-3	-8	-5

toggle movement

Convolution (3D)

```
float weights[out_filters][in_filters][radius][radius];          // weights
float input[in_filters][height][width];                          // input
float output[out_filters][height][width];                        // output


for(r=0;r<height;r++) {
    for(c=0;c<width;c++) {
        for(i=0;i<out_filters;i++) {
            sum = 0;
            for(j=0;j<in_filters;j++)
                for(rd=-radius;rd<=radius;rd++)
                    for(cd=-radius;cd<=radius;cd++)
                        sum += input[j][r + rd][c + cd] * weights[i][j][rd + radius][cd + radius];
                        output[i][r][c] = activation_function(sum);
        }
    }
}
```

Convolution as Matrix Multiplication (1D Example)

We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & x & 0 & 0 & 0 \\ 0 & x & y & x & 0 & 0 \\ 0 & 0 & x & y & x & 0 \\ 0 & 0 & 0 & x & y & x \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ ax + by + cz \\ bx + cy + dz \\ cx + dy \end{bmatrix}$$

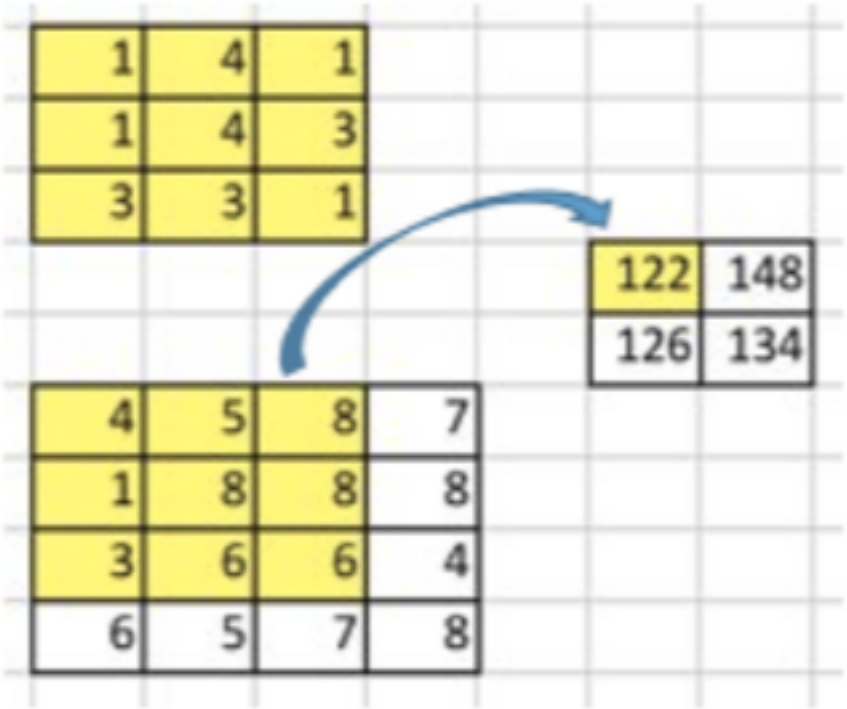
Example: 1D conv, kernel size=3, stride=1, padding=1

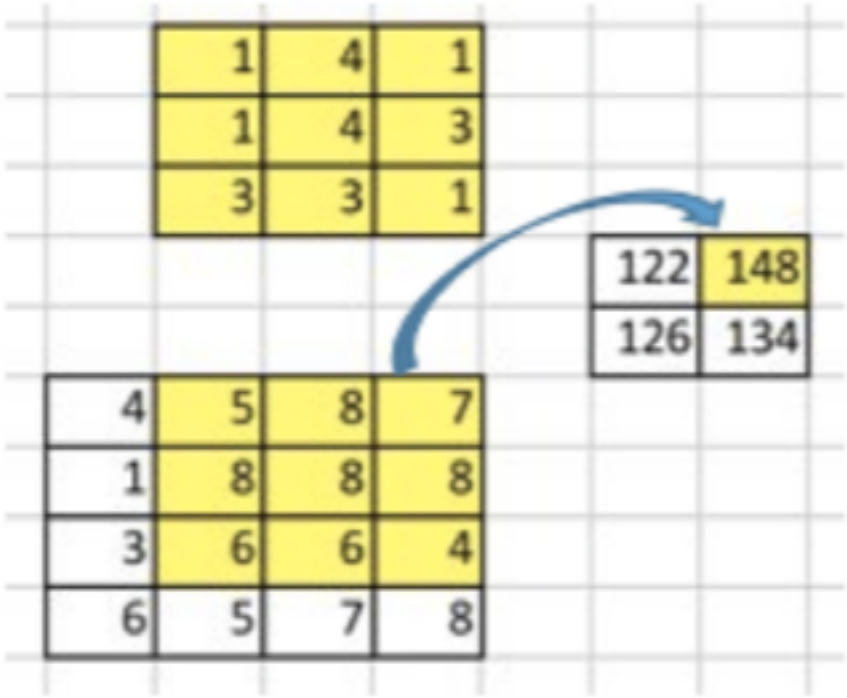
We can express convolution in terms of a matrix multiplication

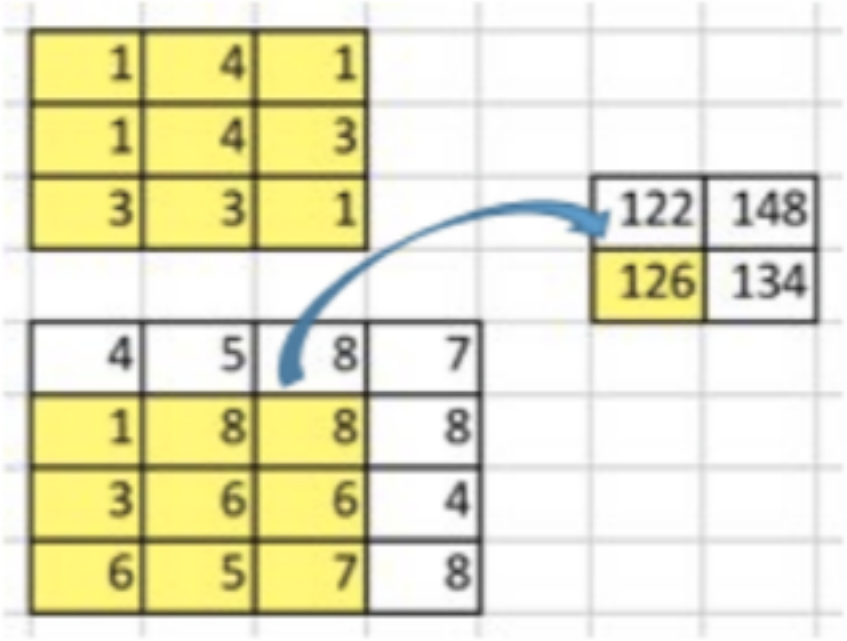
$$\vec{x} * \vec{a} = X\vec{a}$$

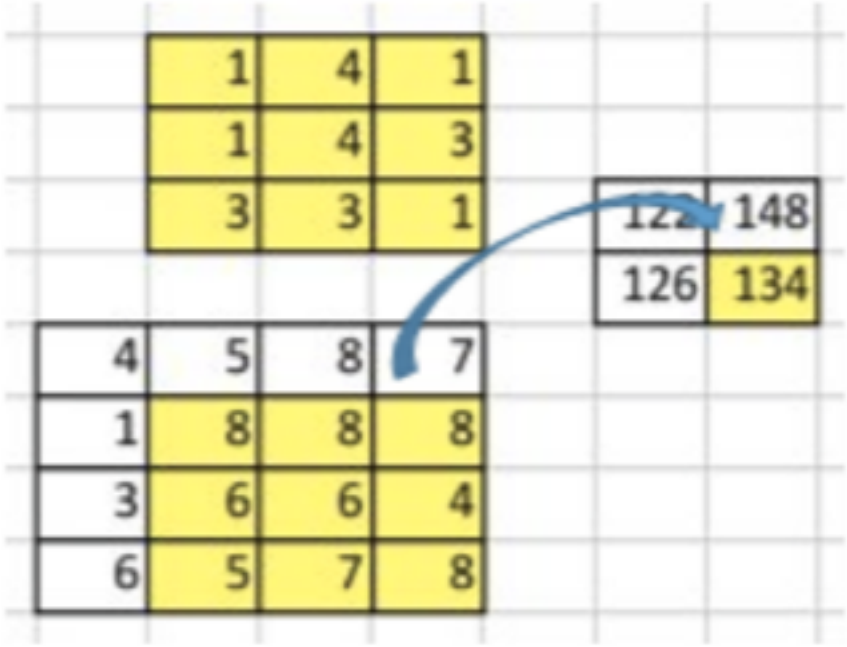
$$\begin{bmatrix} x & y & x & 0 & 0 & 0 \\ 0 & 0 & x & y & x & 0 \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ bx + cy + dz \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=2, padding=1

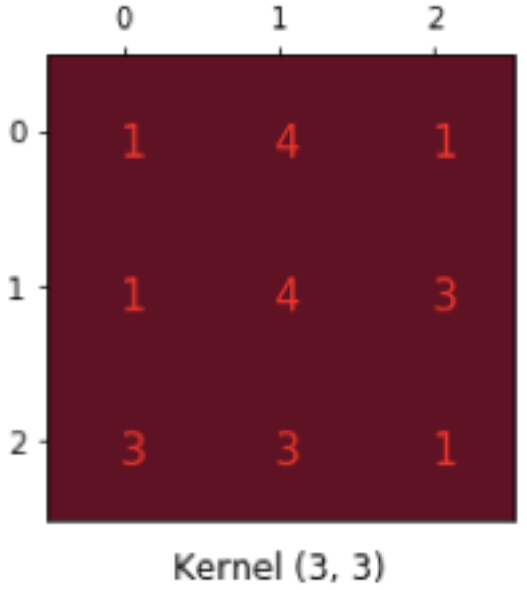

1 4 1
1 4 3
3 3 1
122 148
126 134
4 5 8 7
1 8 8 8
3 6 6 4
6 5 7 8


1 4 1
1 4 3
3 3 1
122 148
126 134
4 5 8 7
1 8 8 8
3 6 6 4
6 5 7 8

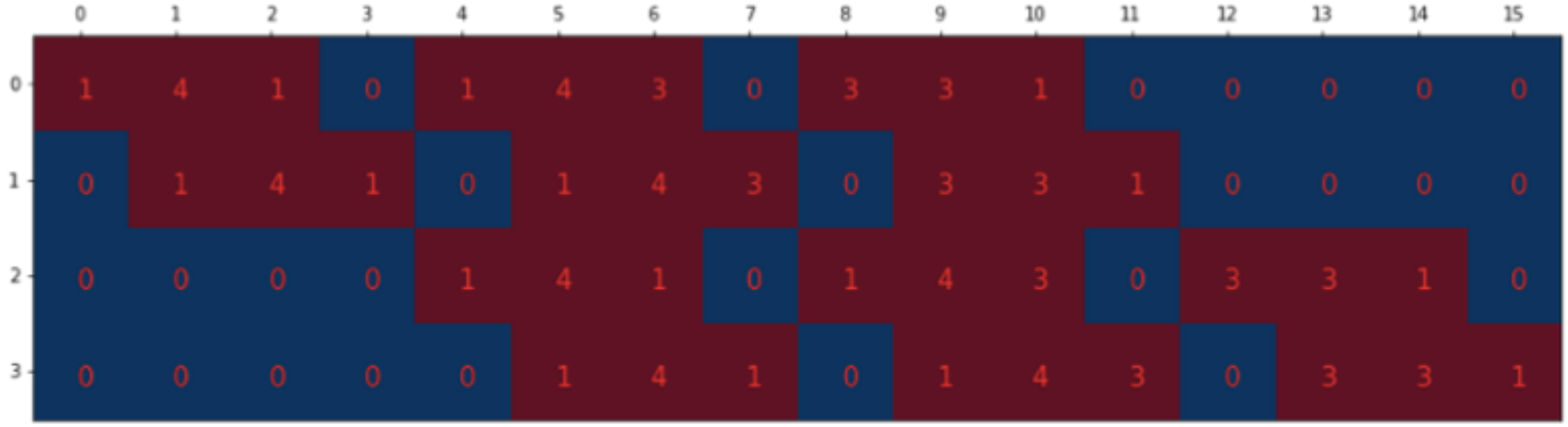

1 4 1
1 4 3
3 3 1
122 148
126 134
4 5 8 7
1 8 8 8
3 6 6 4
6 5 7 8


1 4 1
1 4 3
3 3 1
122 148
126 134
4 5 8 7
1 8 8 8
3 6 6 4
6 5 7 8

Convolution as Matrix Operation

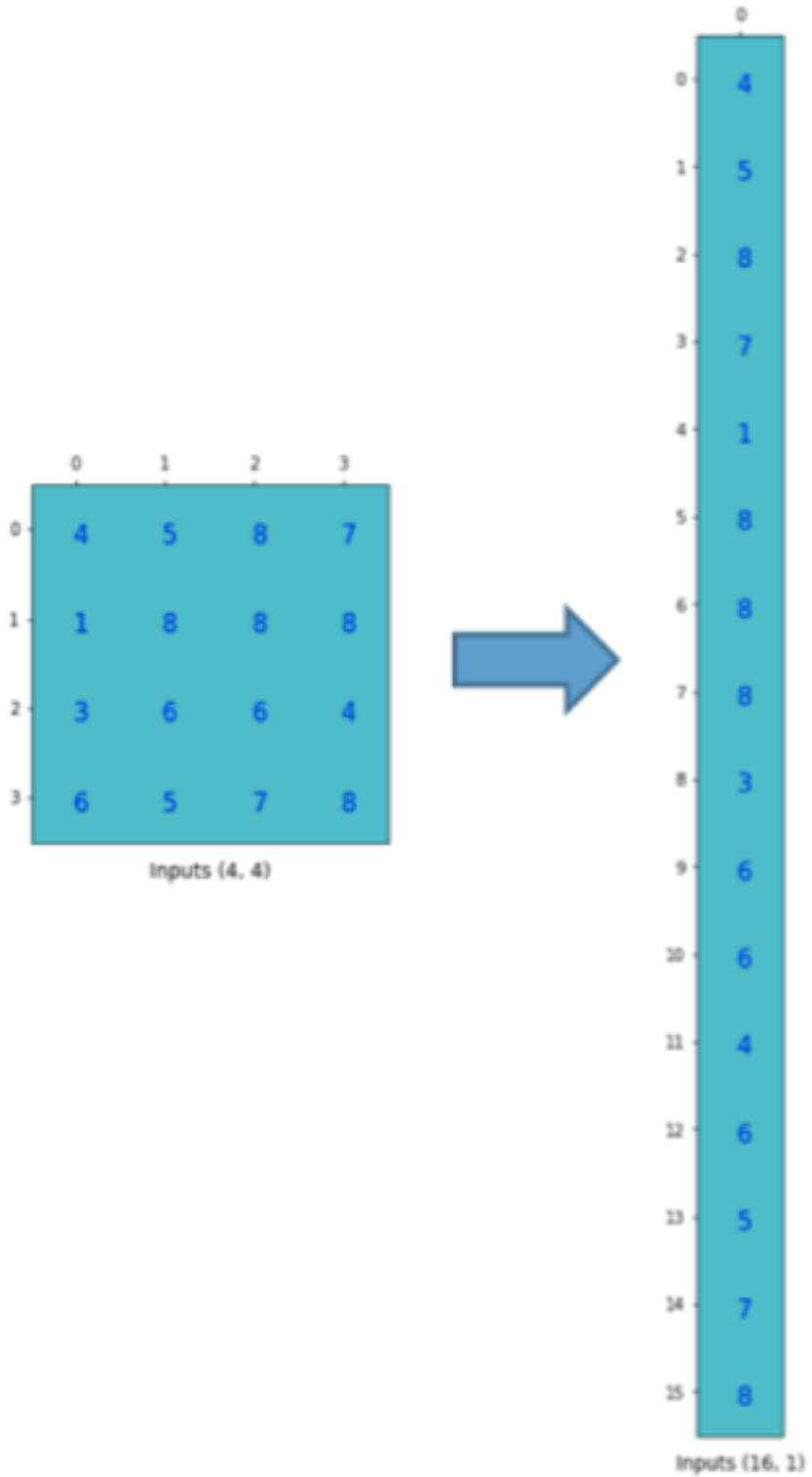


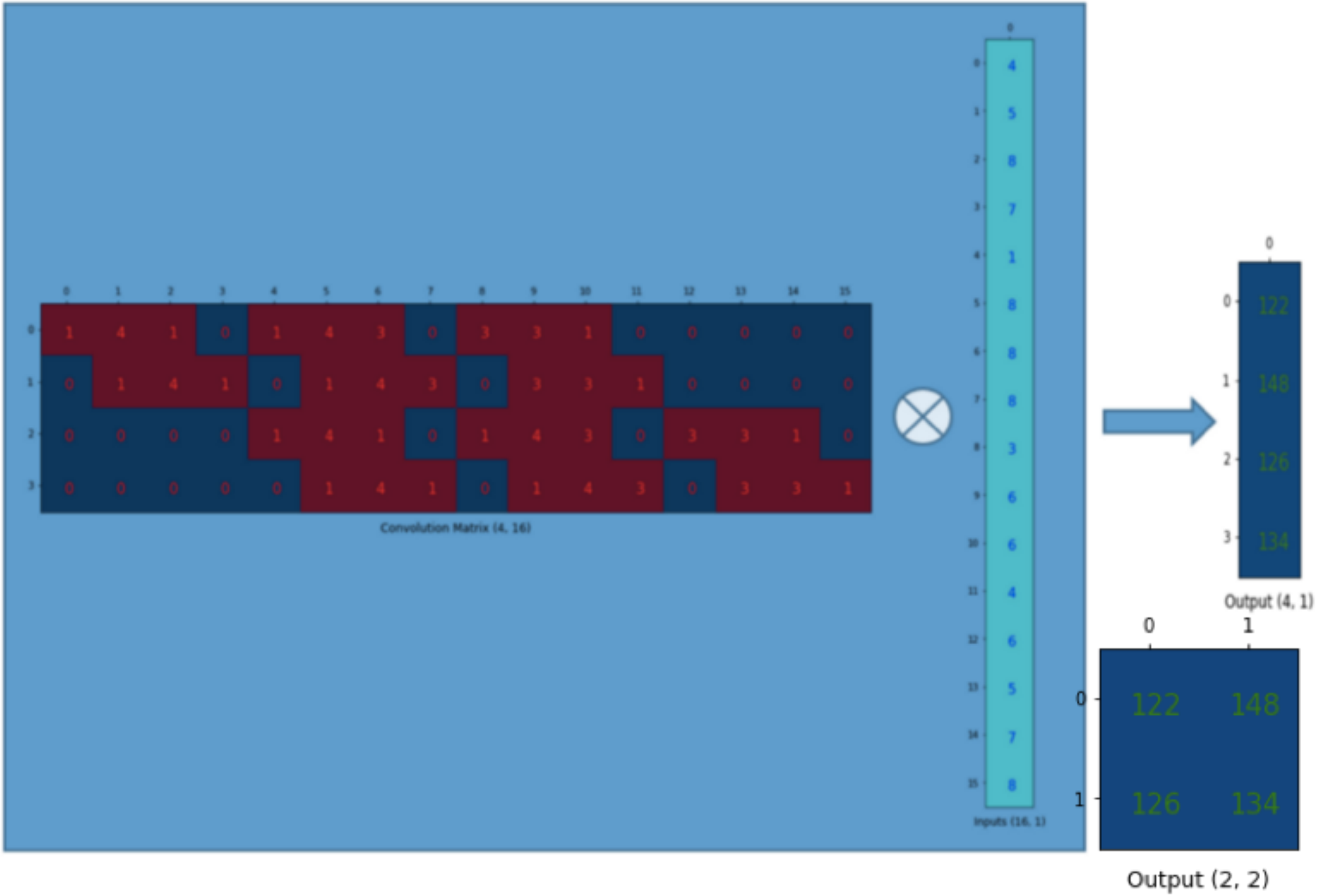
Kernel (3, 3)



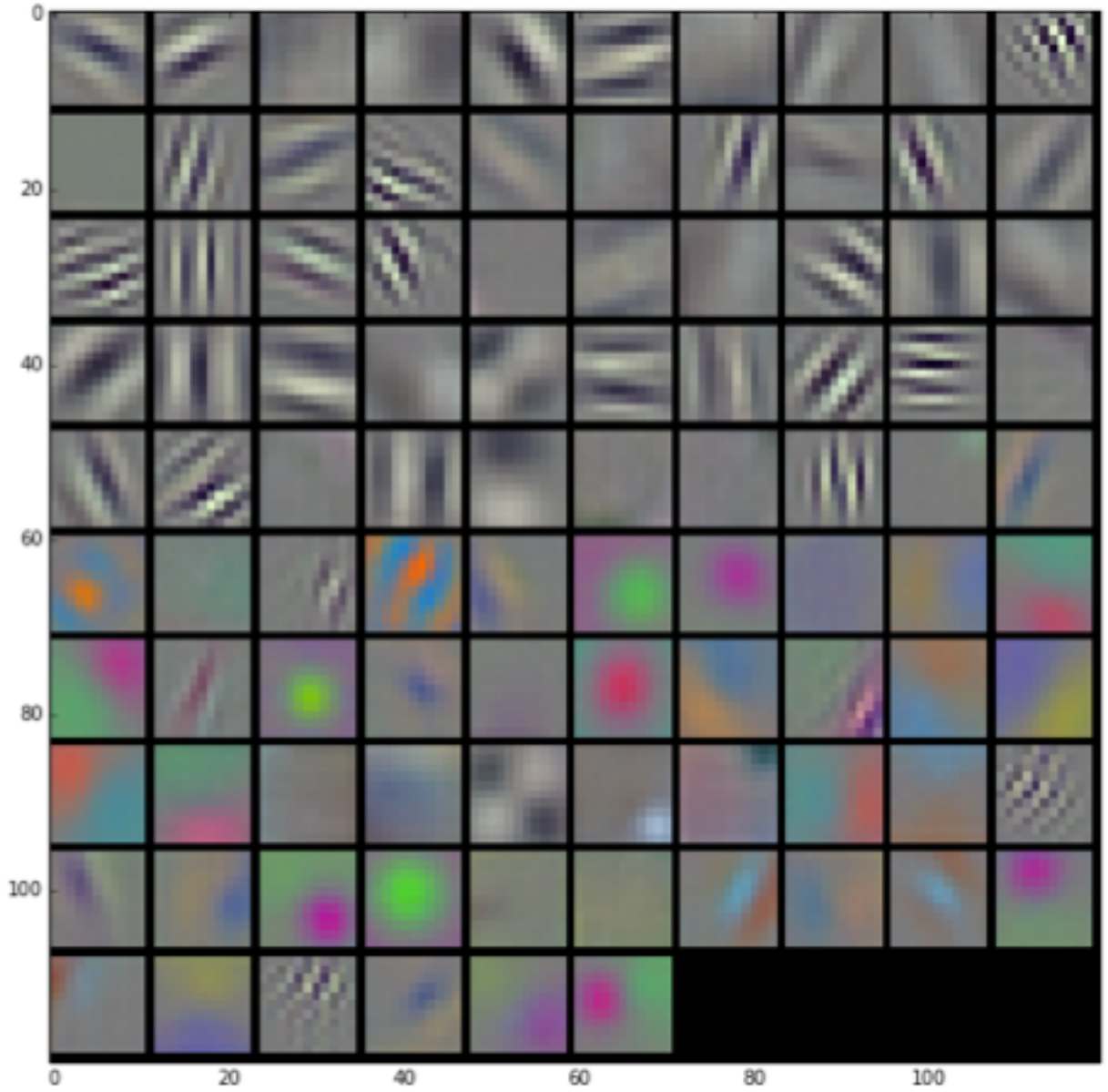
Convolution Matrix (4, 16)

Flattened Image



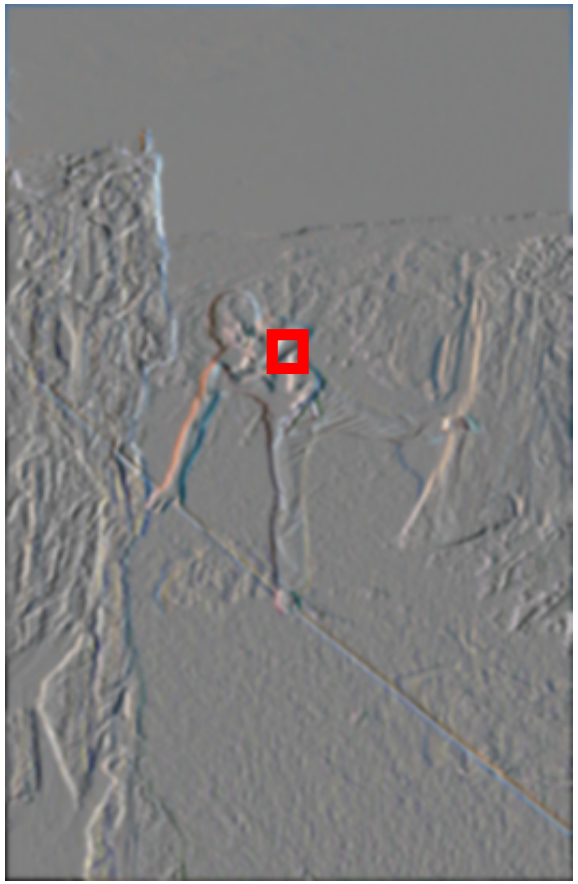

Convolution Matrix (4, 16)
Inputs (16, 1)
Output (4, 1)
Output (2, 2)

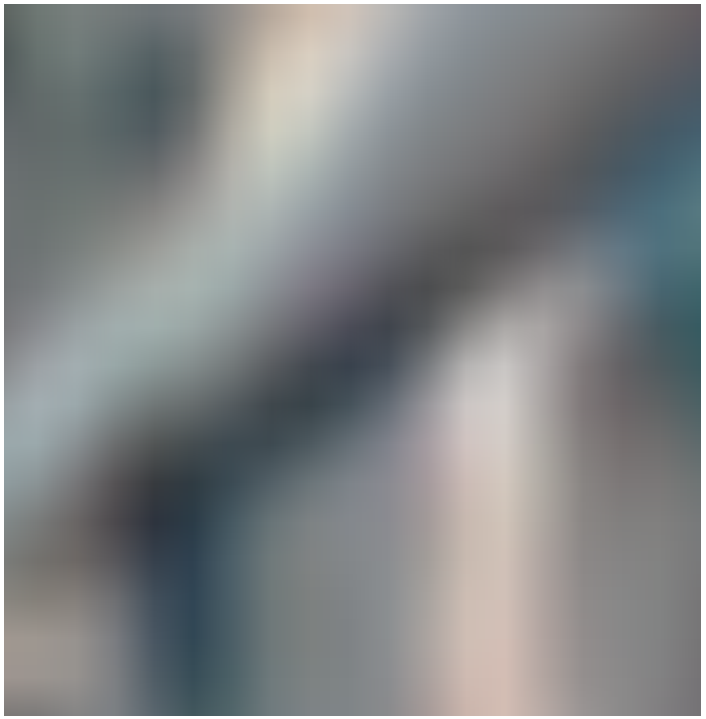
Single layer of convolution



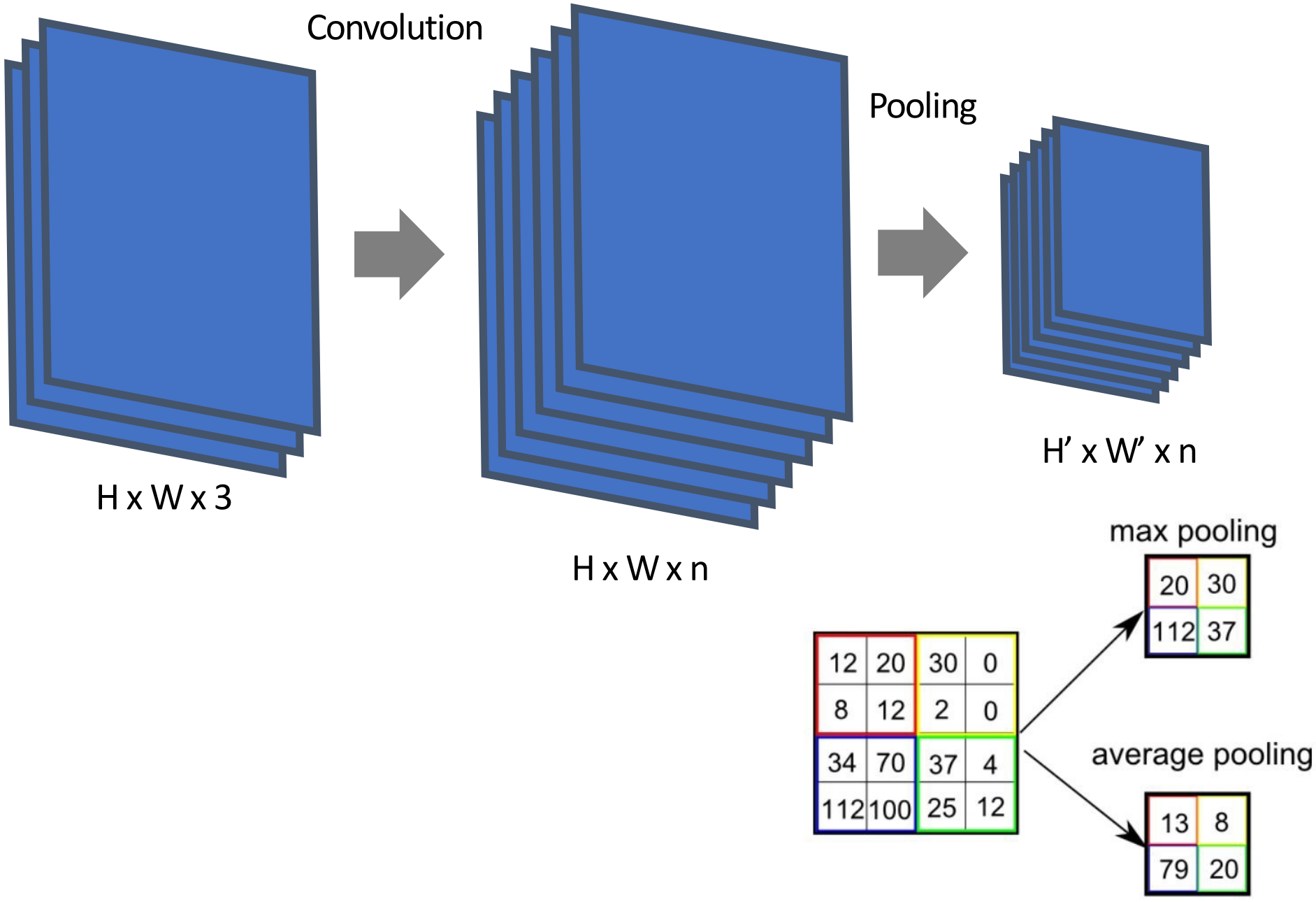
Pooling

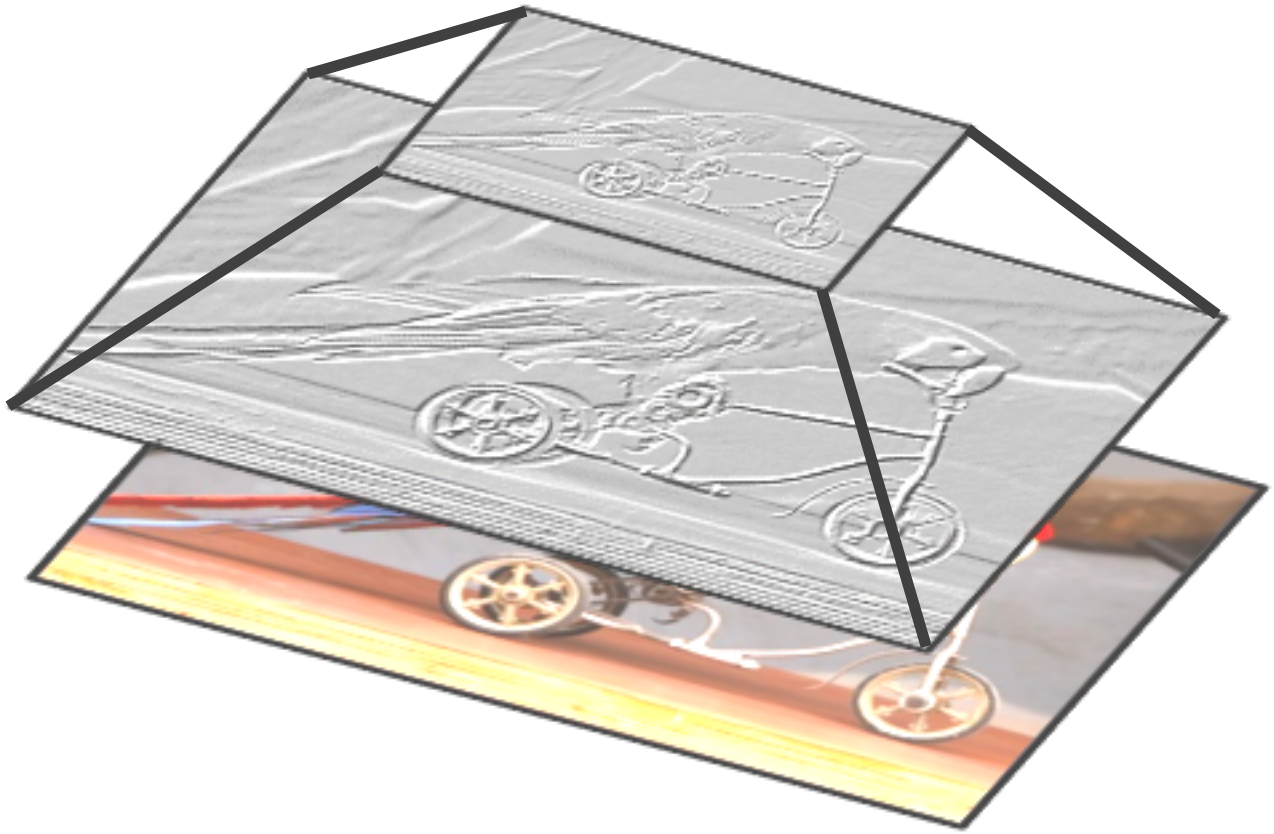
Neighboring values are highly redundant





Pooling


Convolution
Pooling
H x W x 3
H x W x n
H' x W' x n
max pooling
average pooling
12 20 30 0
8 12 2 0
34 70 37 4
112 100 25 12
20 30
112 37
13 8
79 20



Pooling

Convolution

Image

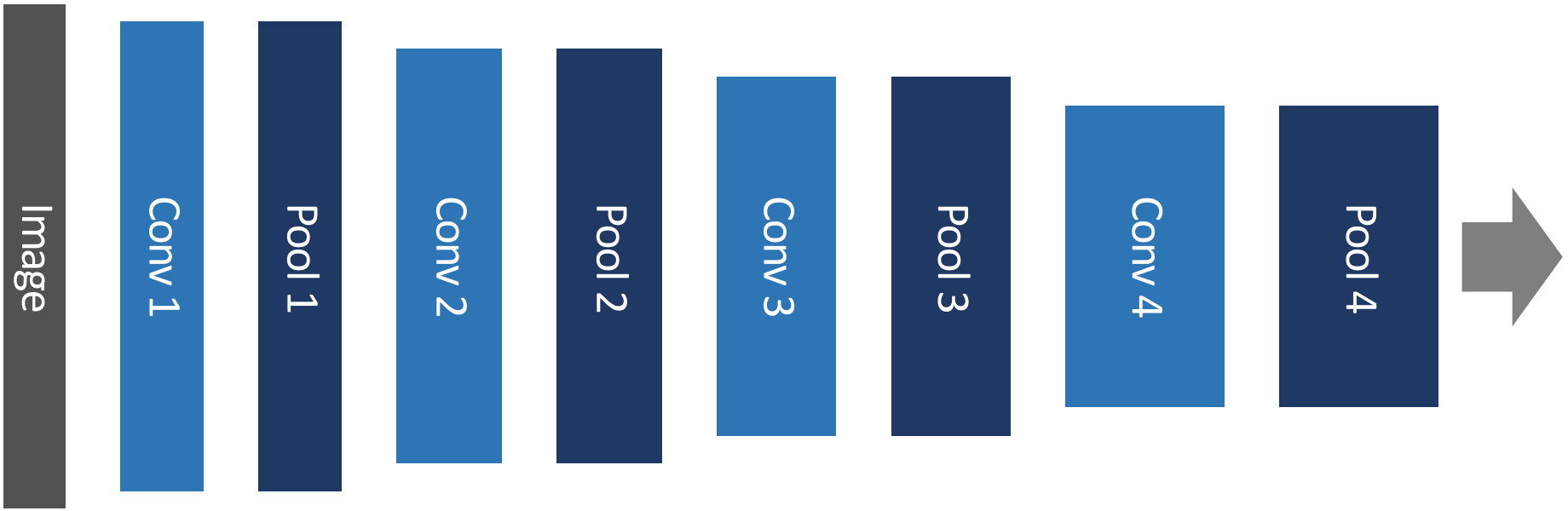
Pooling

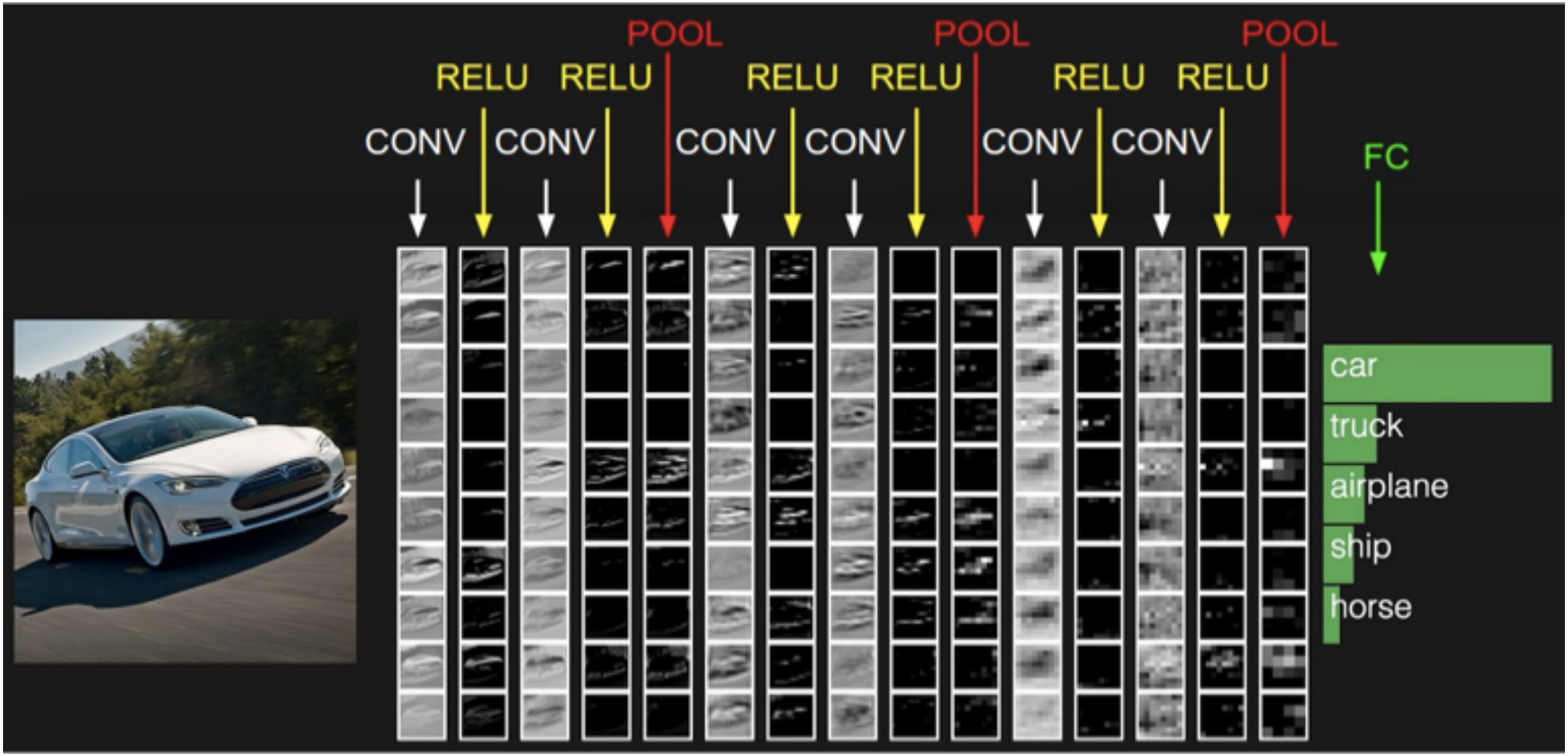
- Invariance to small transformations
- Larger receptive field

Pooling

- Typically use Max or Sum.
- Downsample by 2x
- Filters are 3x3 Why?

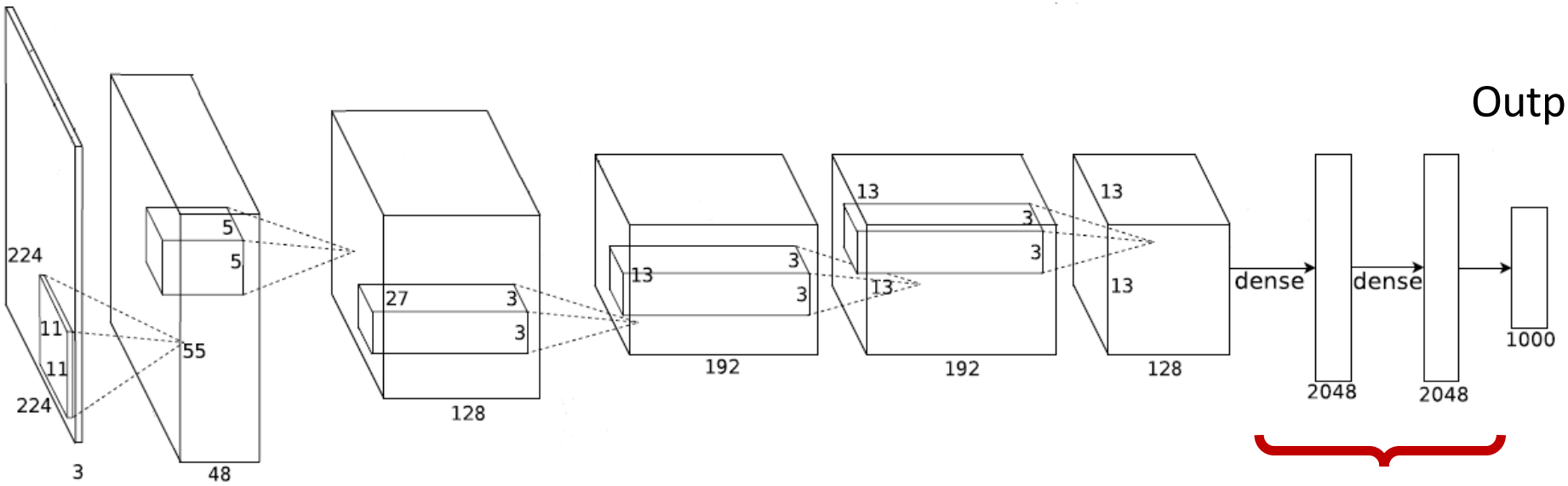
Deeper networks




POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

AlexNet

Image

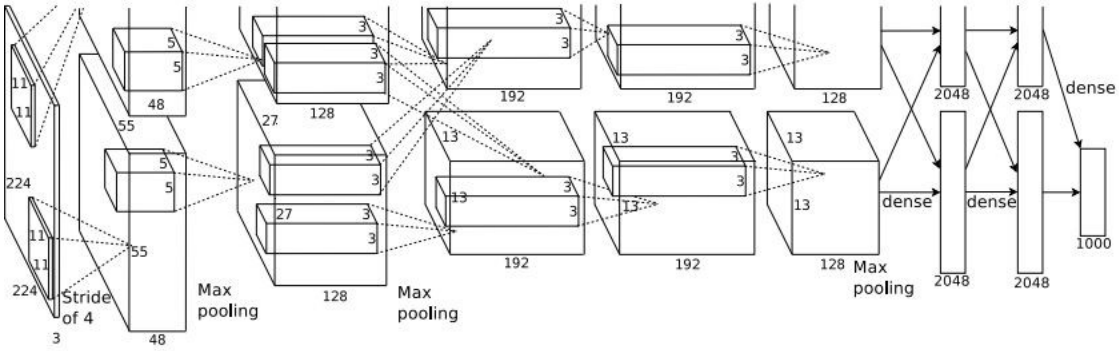


90% parameters

Imagenet Classification with Deep Convolutional Neural Networks, Krizhevsky, Sutskever, and Hinton, NIPS 2012

Case Study: AlexNet

[Krizhevsky et al. 2012]

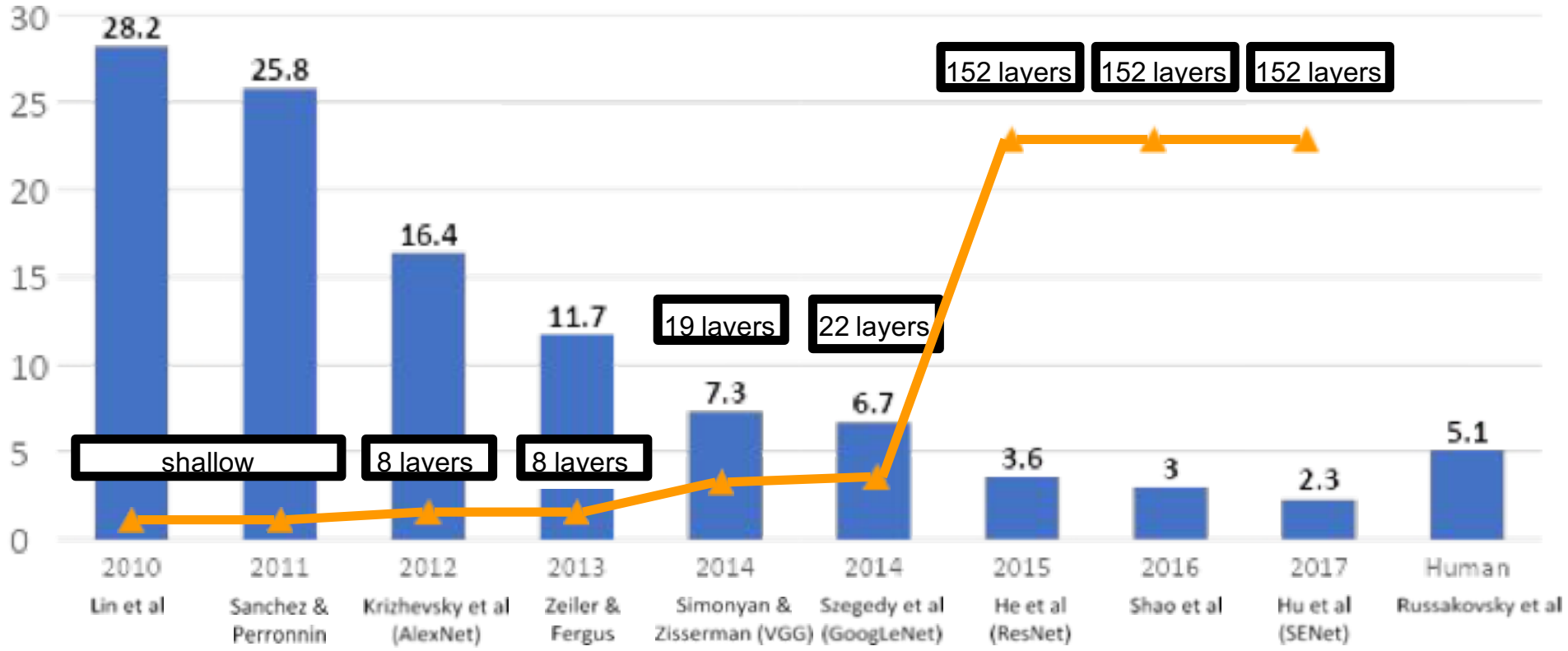


Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:

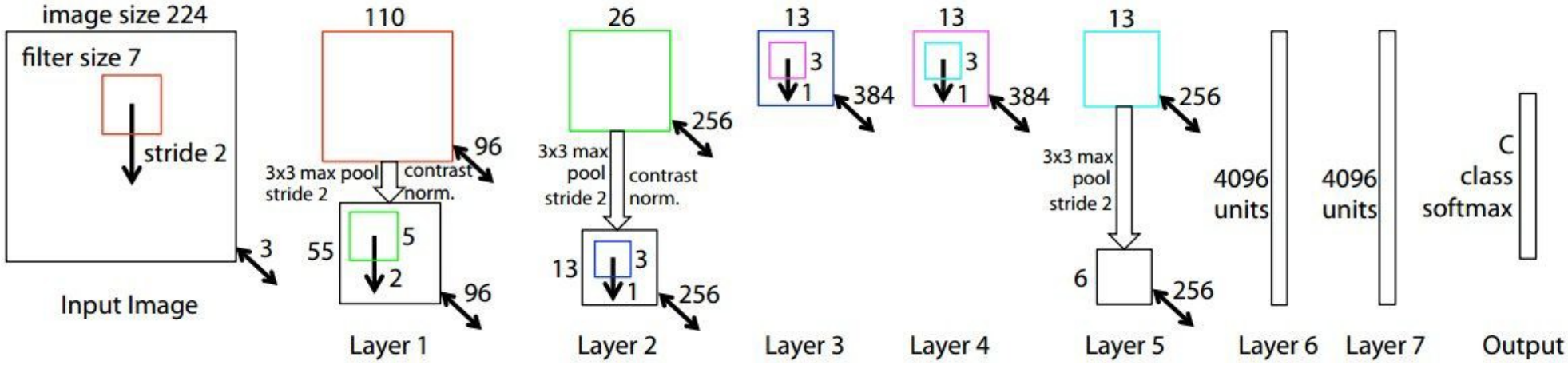
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ZFNet

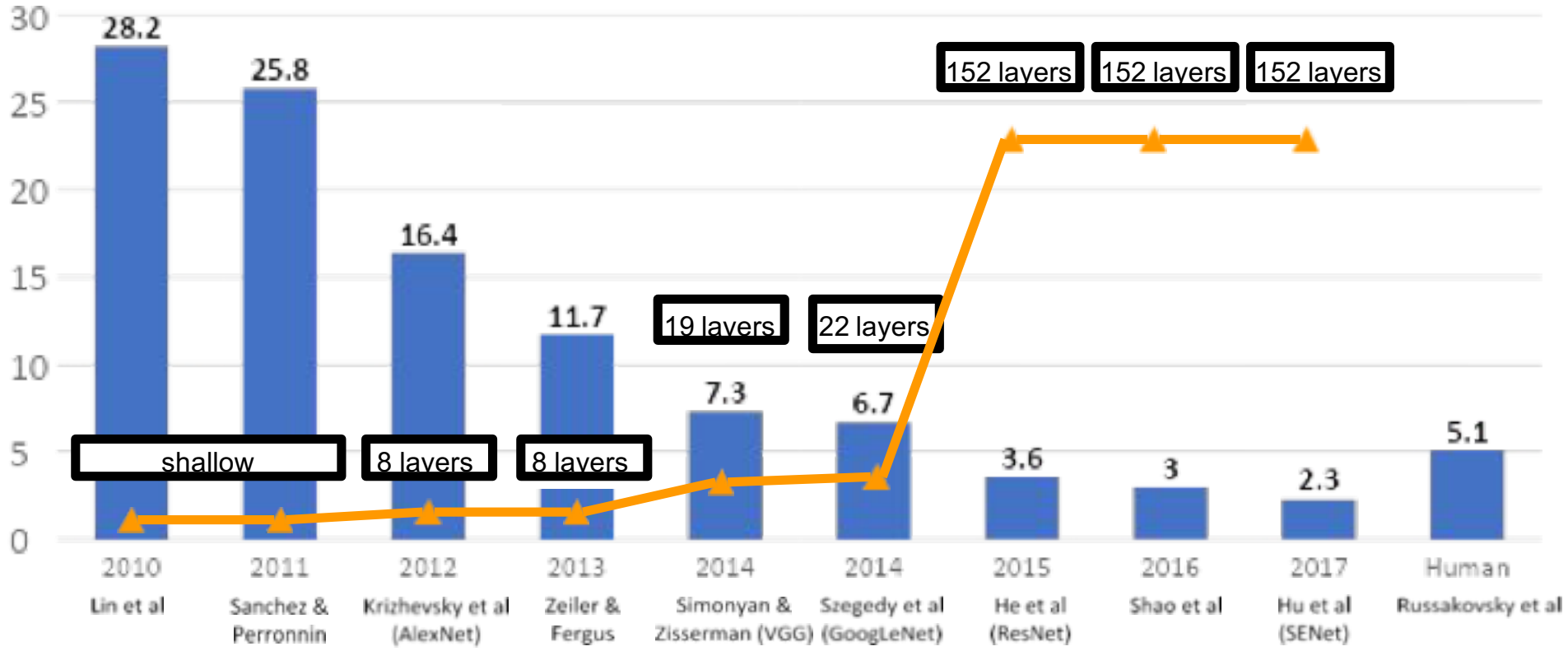
[Zeiler and Fergus, 2013]



AlexNet but:
CONV1: change from (11x11 stride 4) to (7x7 stride 2)
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 16.4% -> 11.7%

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: VGGNet

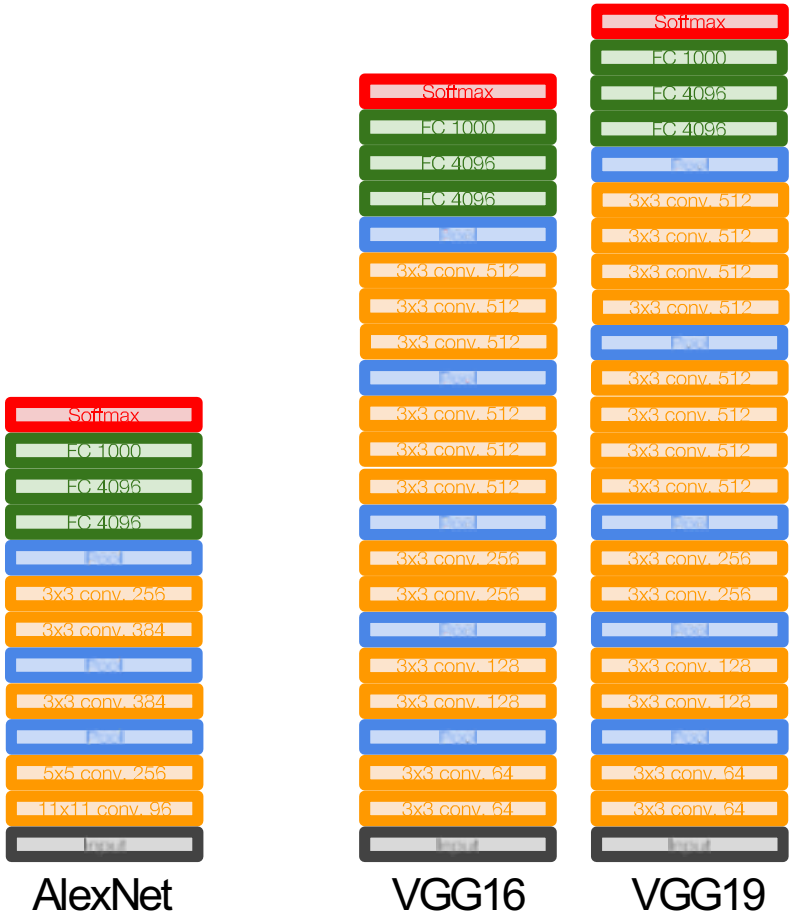
[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13
(ZFNet)
-> 7.3% top 5 error in ILSVRC'14



INPUT: [224x224x3] memory: 224*224*3=150K params: 0

CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864

POOL2: [112x112x64] memory: 112*112*64=800K params: 0

CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456

POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912

CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824

POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648

CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296

POOL2: [14x14x512] memory: 14*14*512=100K params: 0

CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0

FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216

FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 96MB / image (for a forward pass)
TOTAL params: 138M parameters

VGG16


Q&A