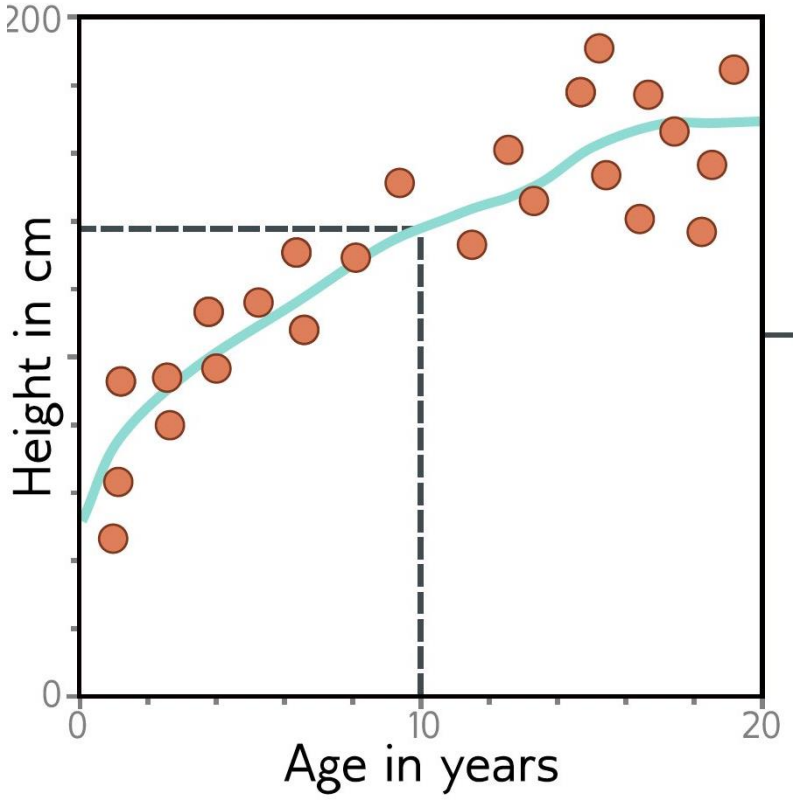
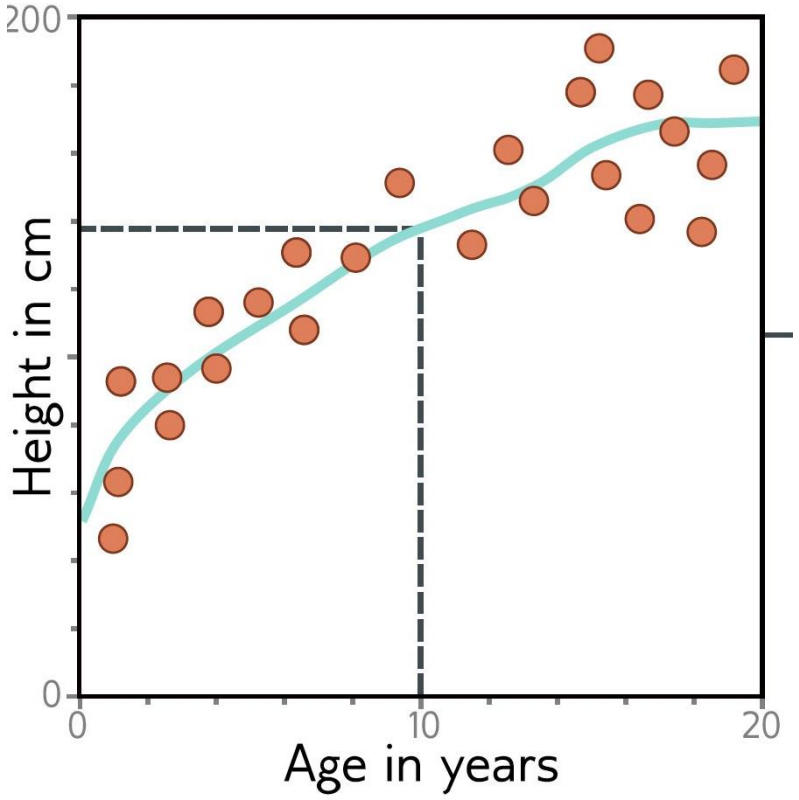
Machine Learning & Neural Networks – 7PAM2021

Supervised Learning in a Nutshell



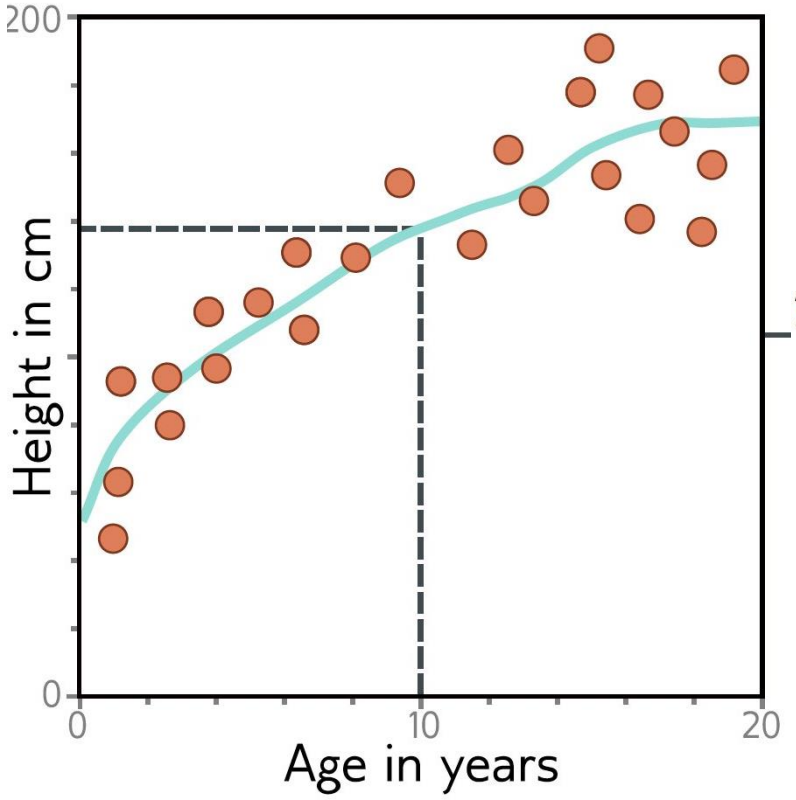
$$\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$$

Supervised Learning in a Nutshell



$$y = \mathrm{f}[x, \boldsymbol{\phi}]$$
$$= \phi_0 + \phi_1 x$$

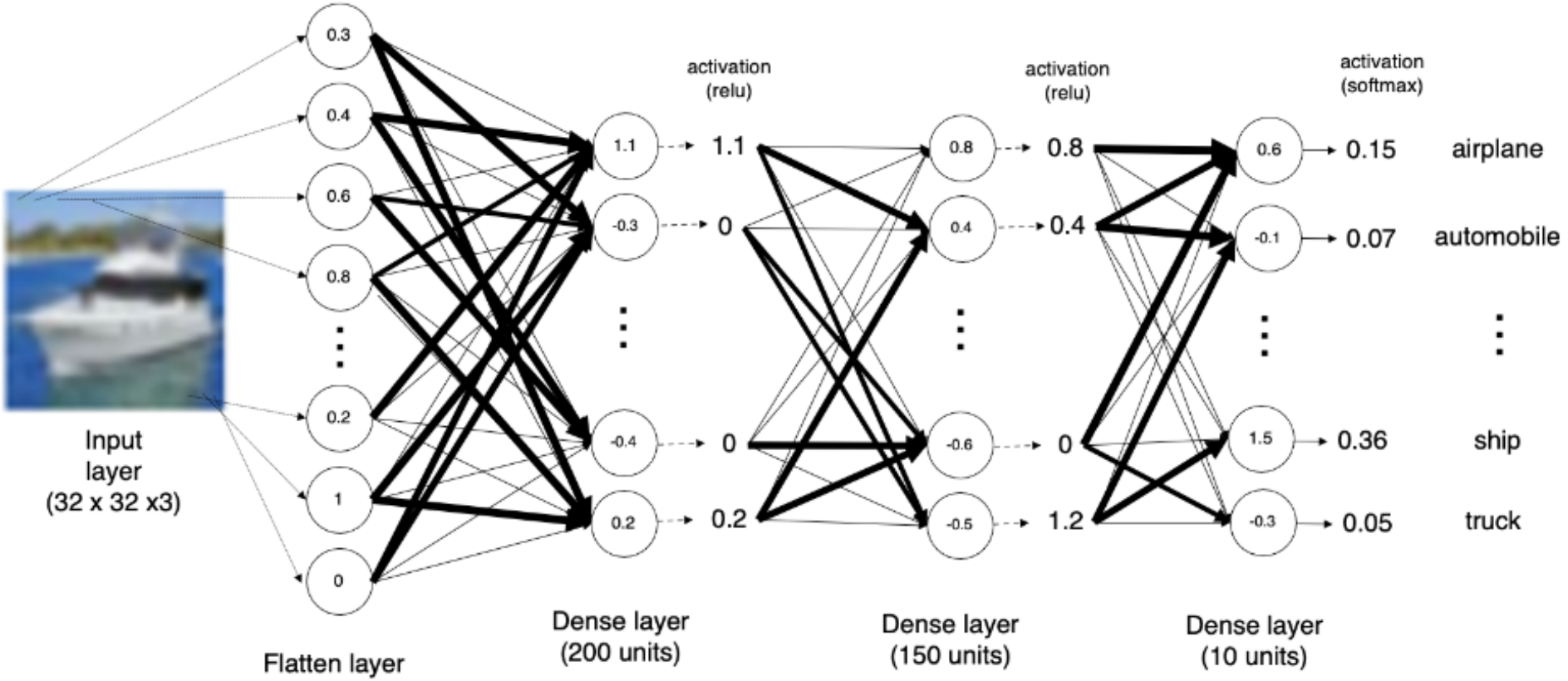
Supervised Learning in a Nutshell



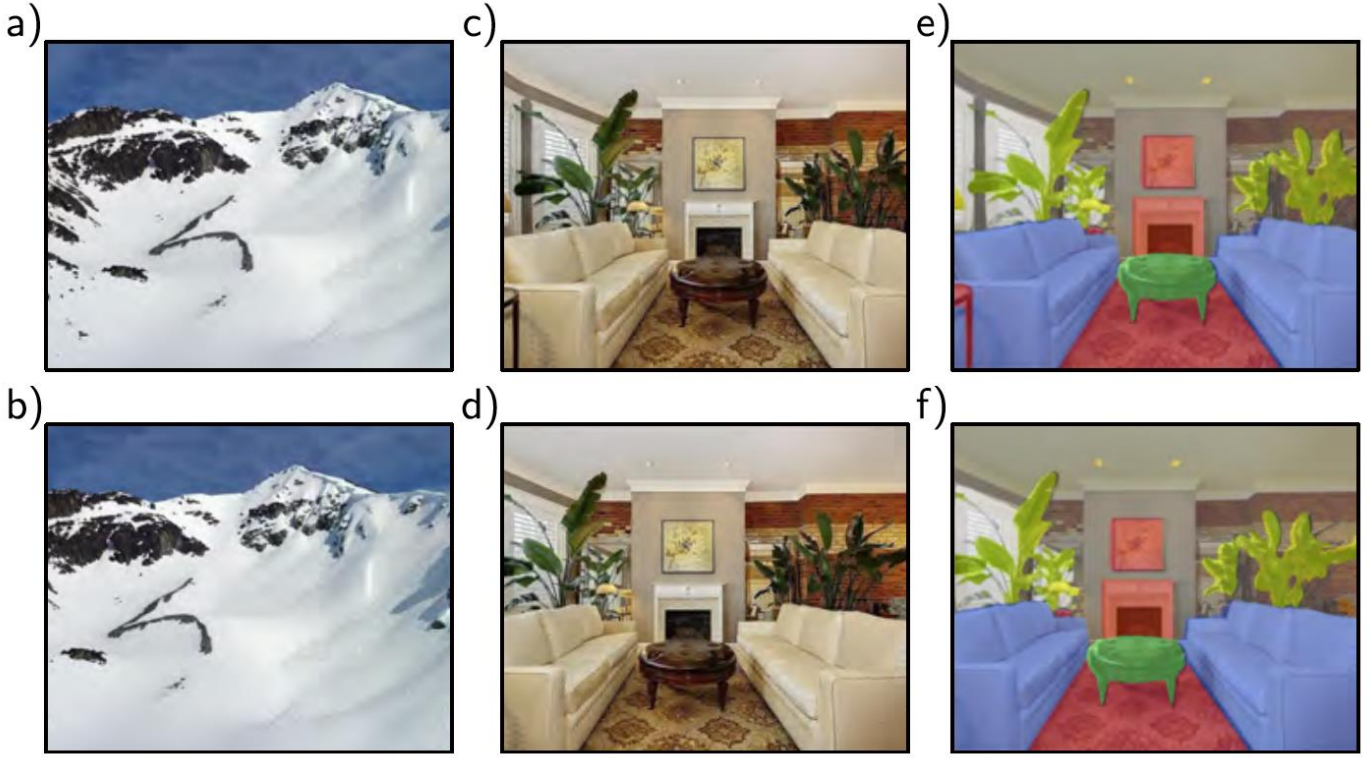
$$
\begin{aligned}
y &= \mathrm{f}[x, \boldsymbol{\phi}] \\
&= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x].
\end{aligned}
$$

Why do we need CNNs?

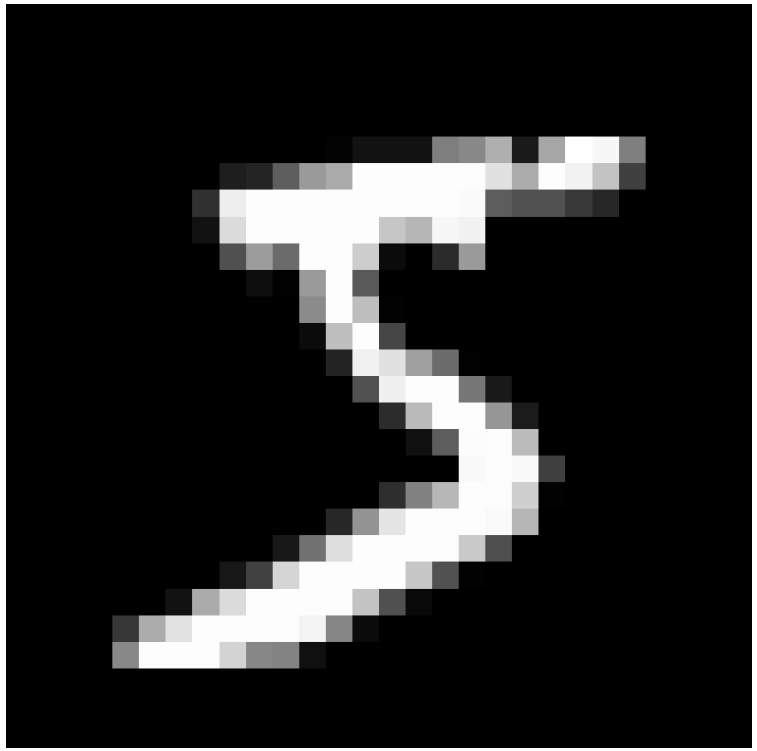
The architecture of MLPs is fundamentally limited in processing image data due to their fully connected nature, which does not take advantage of the spatial and hierarchical structure of images.

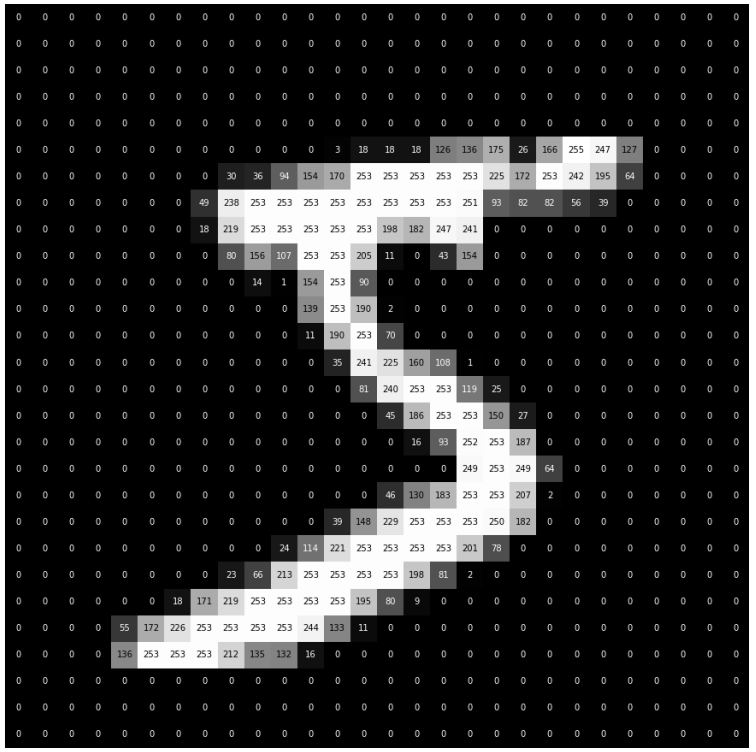


Invariance and equivariance

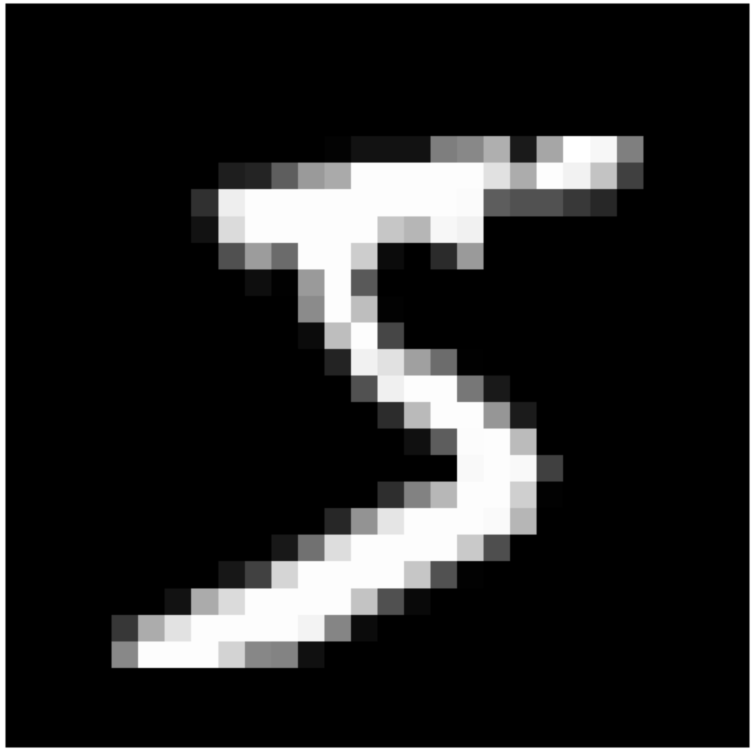


$$\mathbf{f}[\mathbf{t}[\mathbf{x}]] = \mathbf{f}[\mathbf{x}]$$





What do computers see?



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255	247	127	0	0	0	0
0	0	0	0	0	0	0	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242	195	64	0	0	0	0
0	0	0	0	0	0	0	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56	39	0	0	0	0	0
0	0	0	0	0	0	0	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	14	1	154	253	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	139	253	190	2	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	11	190	253	70	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	35	241	225	160	108	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	93	252	253	187	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	249	253	249	64	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	18	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	55	172	226	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	136	253	253	253	212	135	132	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

What do computers see?

Image Classification

One of the most common forms of supervised learning is identifying the contents of an image

This is a task that humans are VERY good at—in most cases—but is harder than it looks for a computer

Predictions of labels are often expressed as probabilities—95% chance the image is a cat

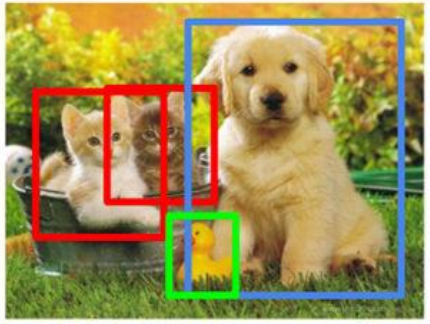
Cat? Or Dog?

Object Detection

Classification

CAT

Object Detection



CAT, DOG, DUCK

Like image classification, object detection attempts to predict the contents of an image, but now with multiple objects, not just one

The models are trained to predict the “Bounding Boxes” of objects in the image, and apply labels for the contents of each box

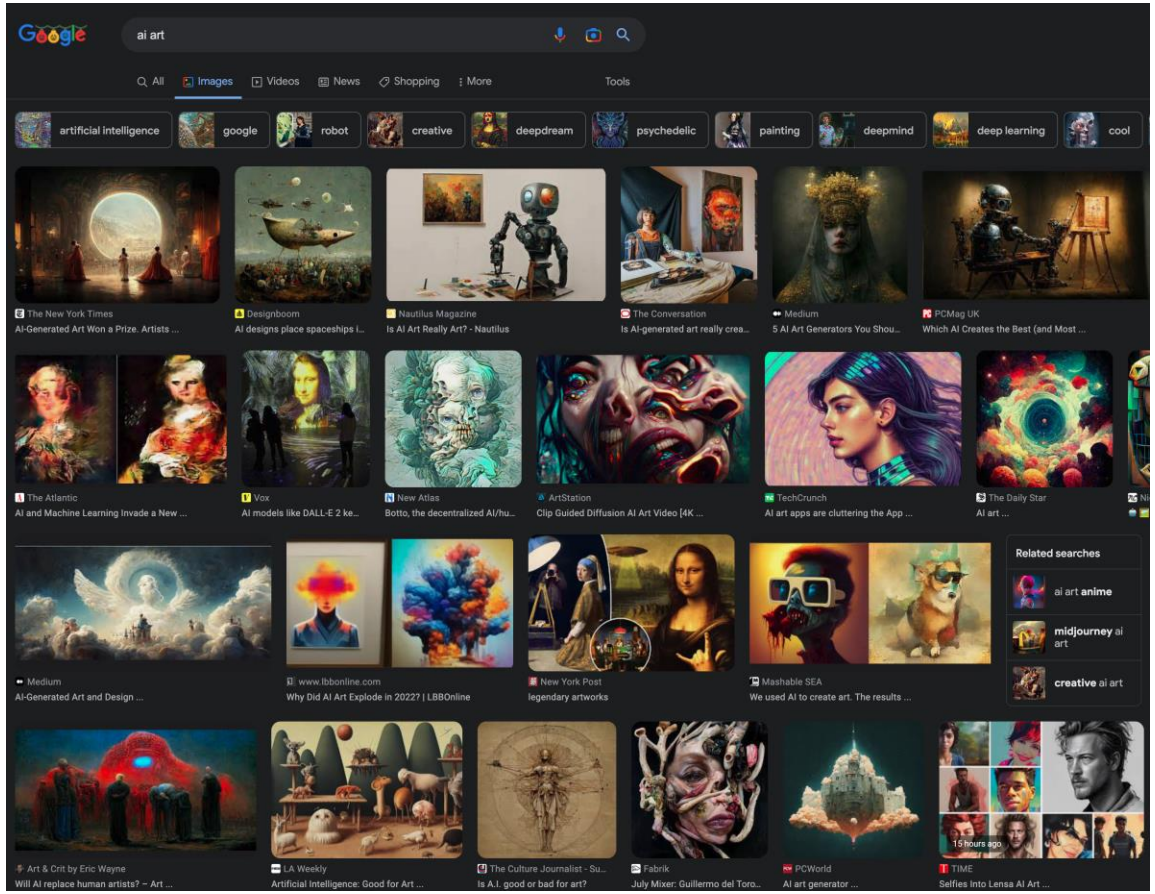


Image Generation

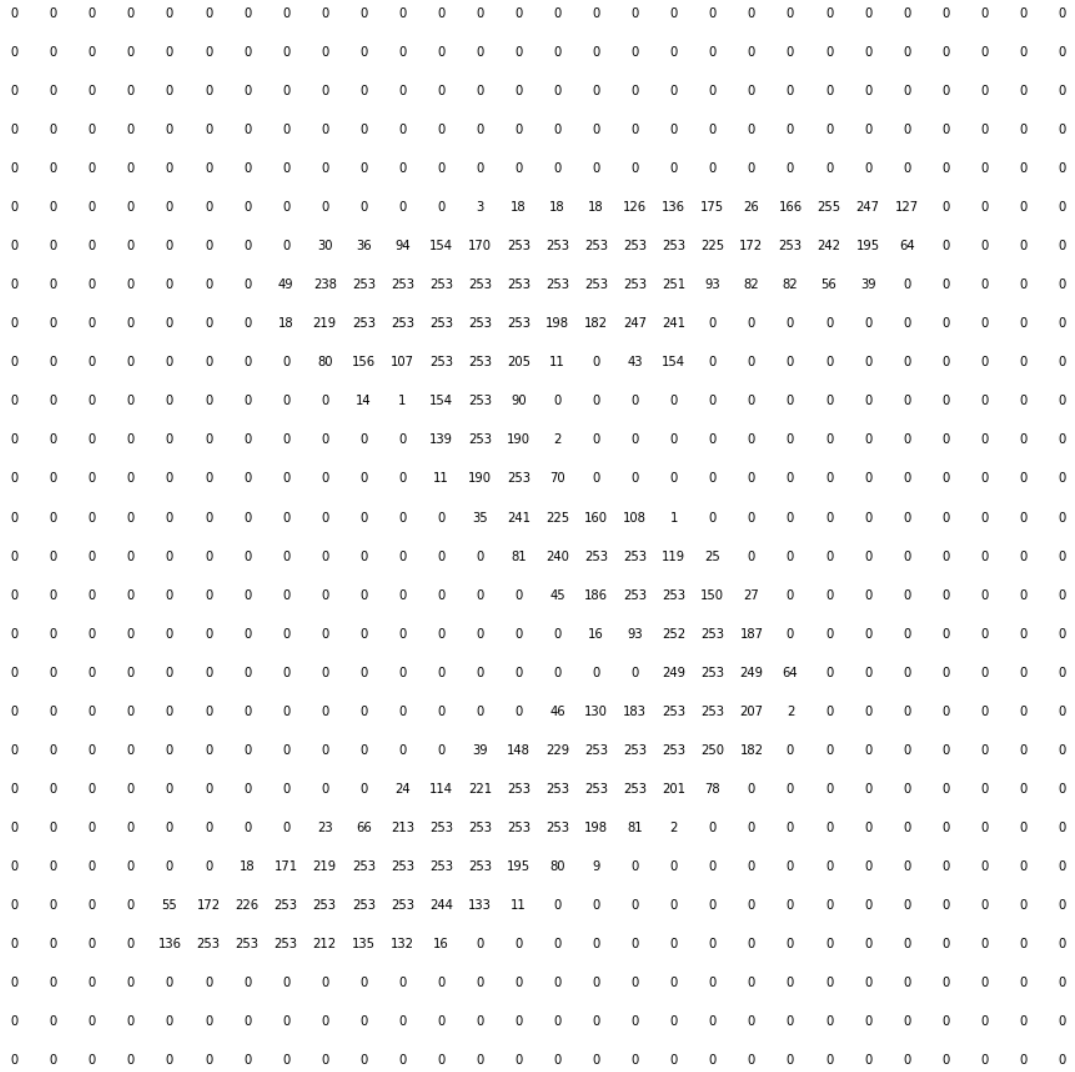
Increasingly popular in the last few years, training models to create images, either from scratch or with a prompt, has many applications.

Raise of Stable Diffusion has led to an explosion of “AI Art”, but other models include VAEs and Generalised Adversarial Nets (GANs)

Can be used for image restoration and scene reconstruction.

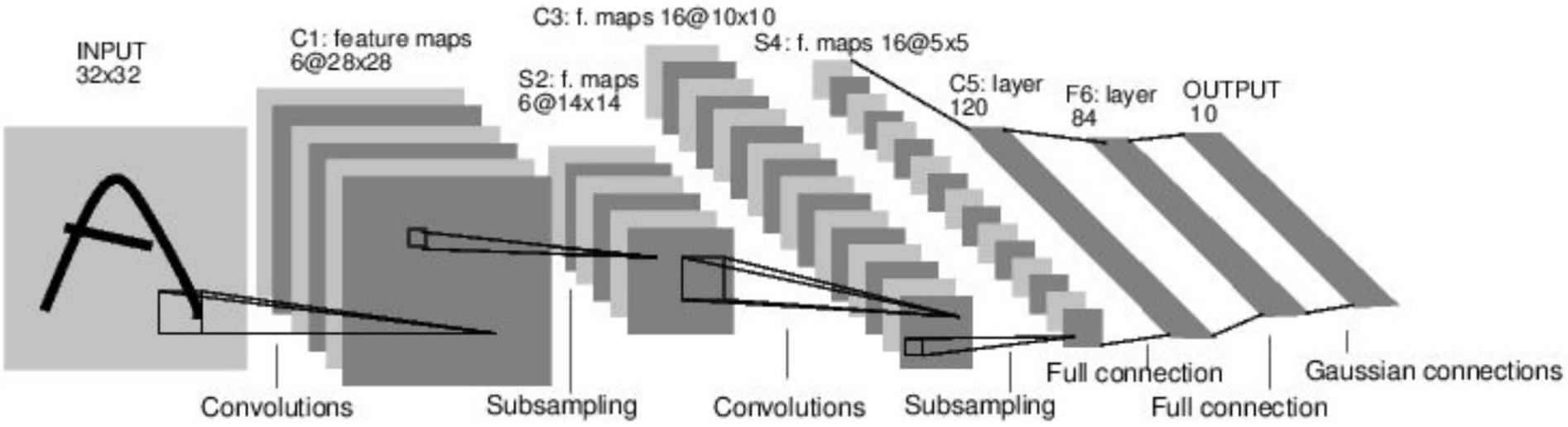
Summary: Computer Vision

- Computer vision covers a wide range of tasks and fields of research.
- At the core is understanding what computers "see"—matrices of numbers.
- But how do we go from these matrices to models?

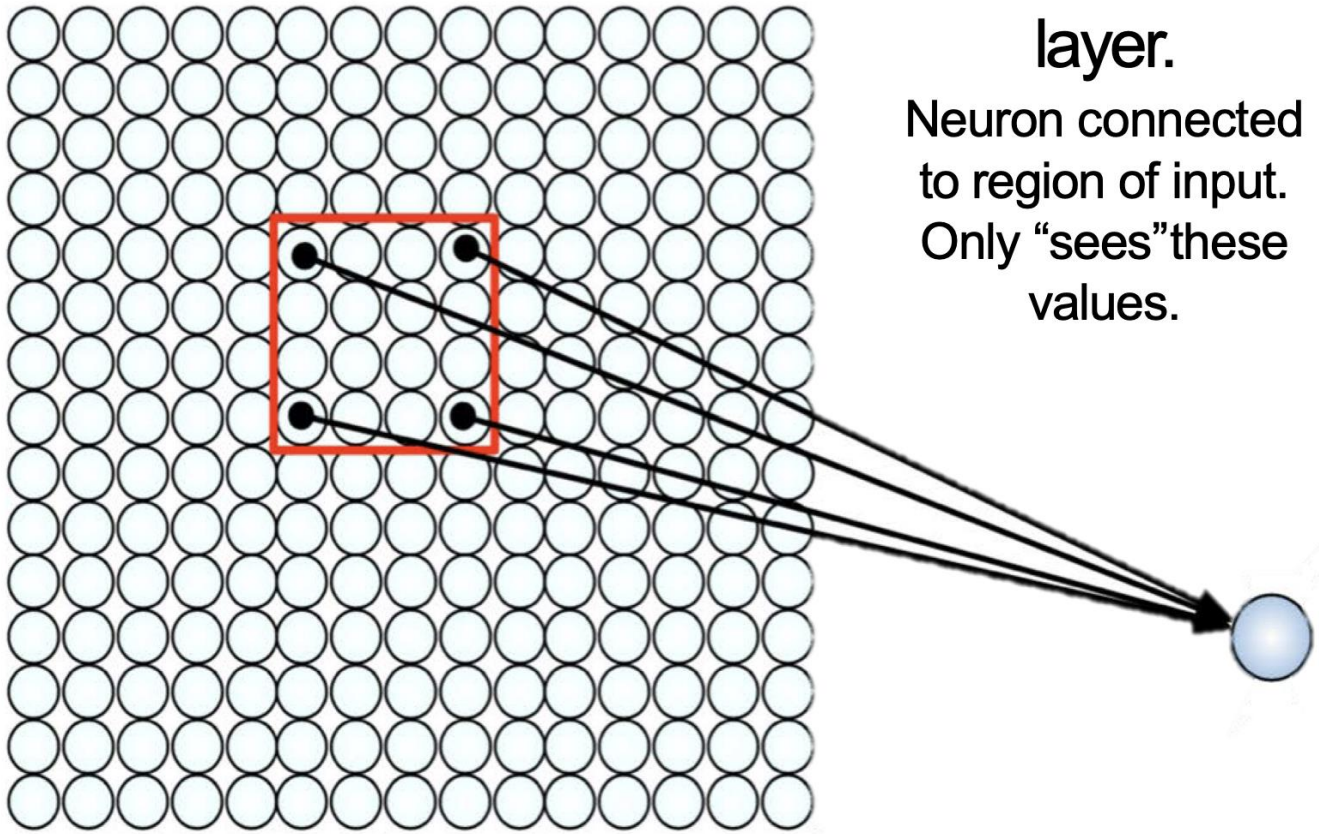


Convolutional Layers

Convolutional Neural Networks



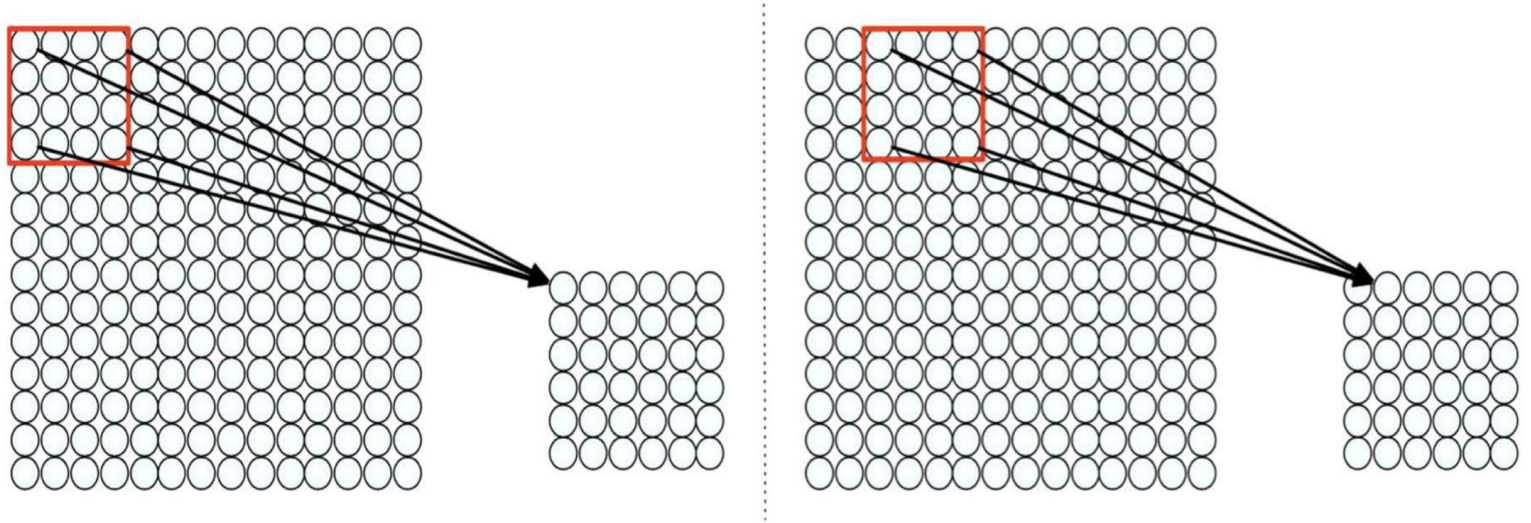
Input: 2D image. Array of pixel values



Idea: connect patches of input to neurons in hidden layer.

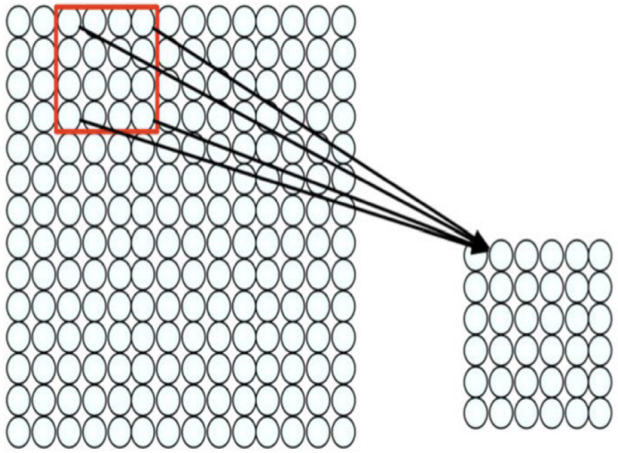
Neuron connected to region of input. Only “sees”these values.

Using Spatial Structure



Connect patch in input layer to a single neuron in subsequent layer.
Use a sliding window to define connections.
How can we **weight** *the patch to detect particular features?*

Feature Extraction with Convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

This "patchy" operation is **convolution**

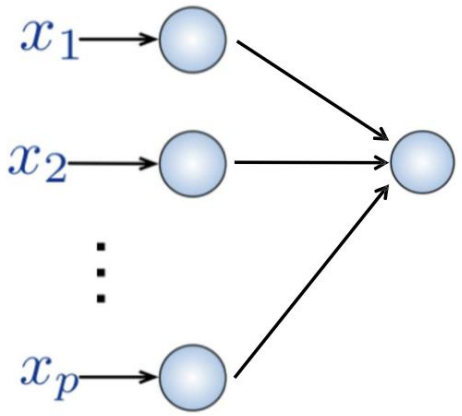
1) Apply a set of weights – a filter – to extract **local features**

2) Use **multiple filters** to extract different features

3) Spatially share parameters of each filter

Fully Connected Neural Network

Input:
- 2D image
- Vector of pixel values

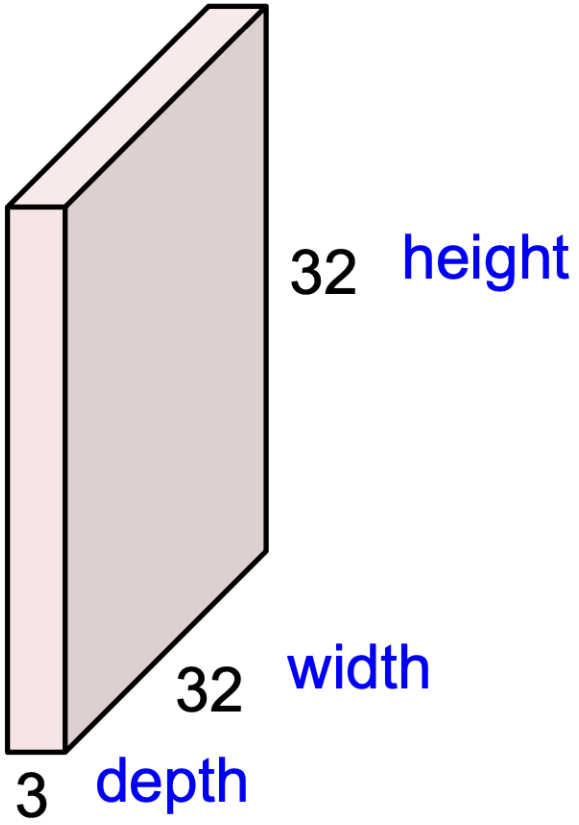


Fully Connected:
- Each neuron in hidden layer connected to all neurons in input layer
- No spatial information
- Many, many parameters

Key idea: Use **spatial structure** in input to inform architecture of the network

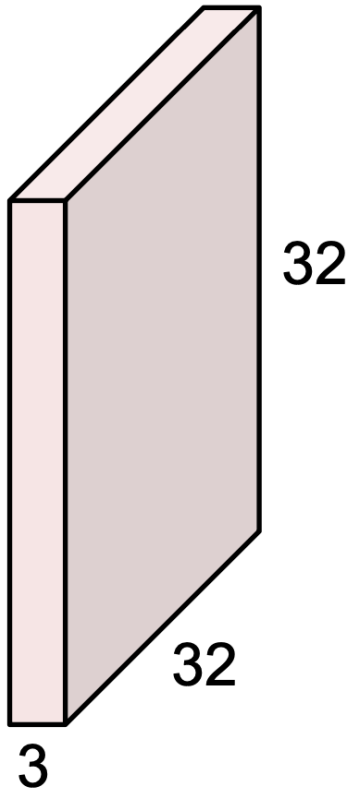
Convolution Layer

32x32x3 image

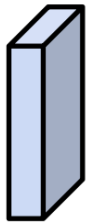


Convolution Layer

32x32x3 image



5x5x3 filter

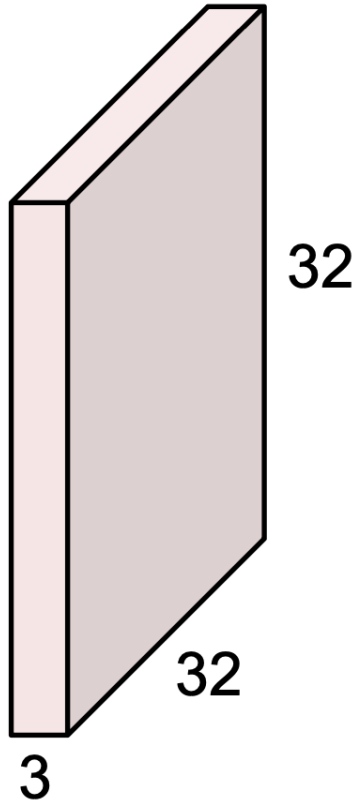


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

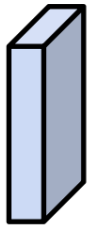
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

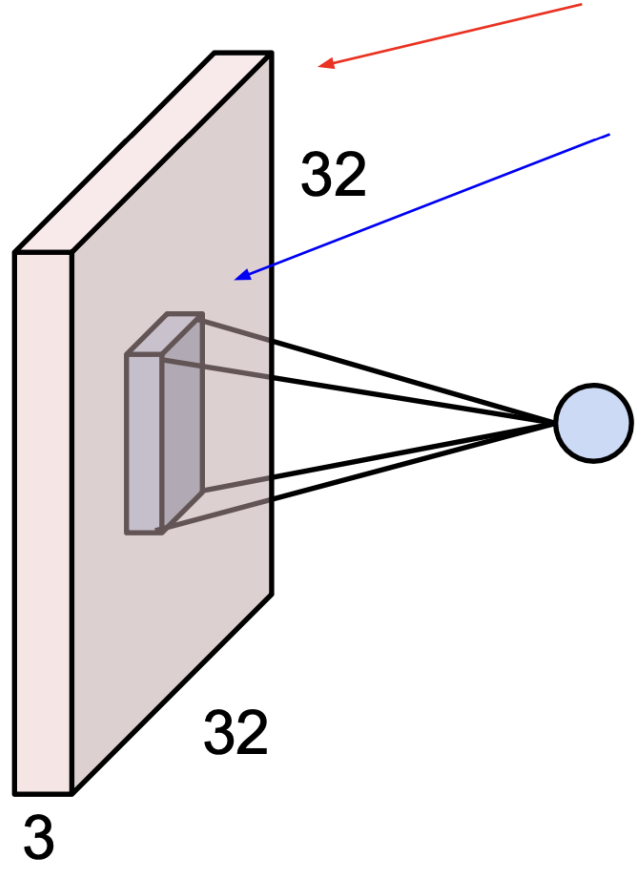


5x5x3 filter



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Convolution Layer



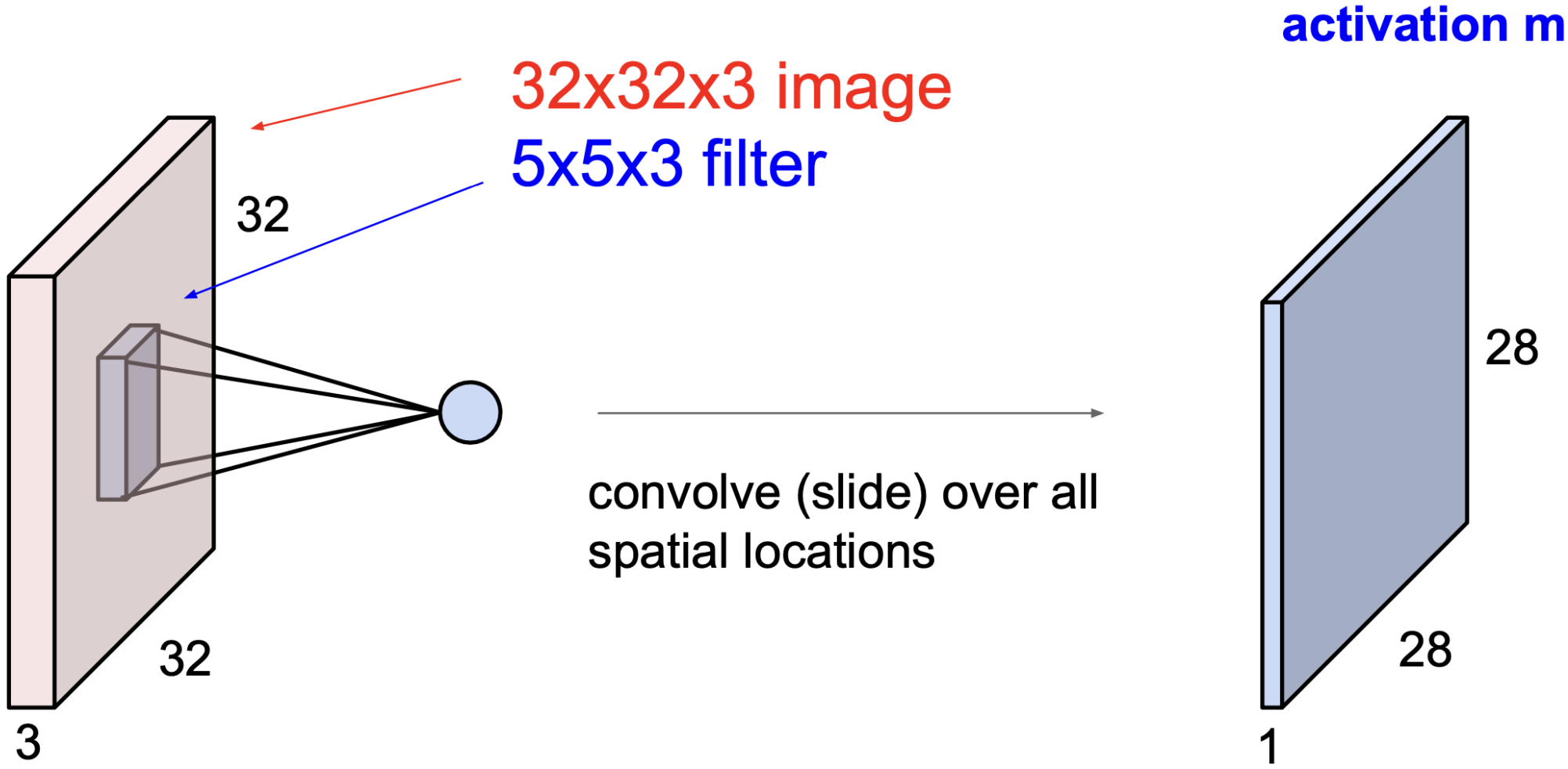
32x32x3 image

5x5x3 filter w

1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

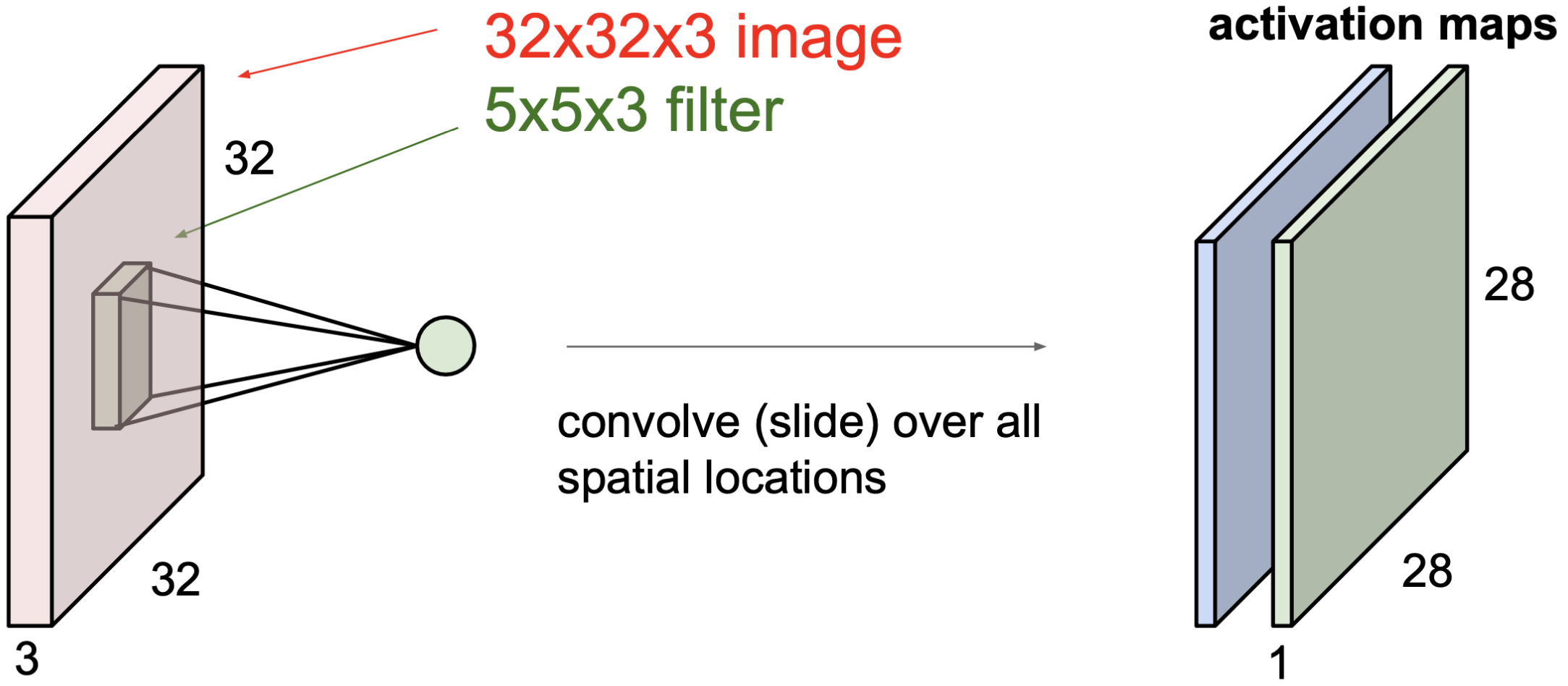
$$w^T x + b$$

Convolution Layer

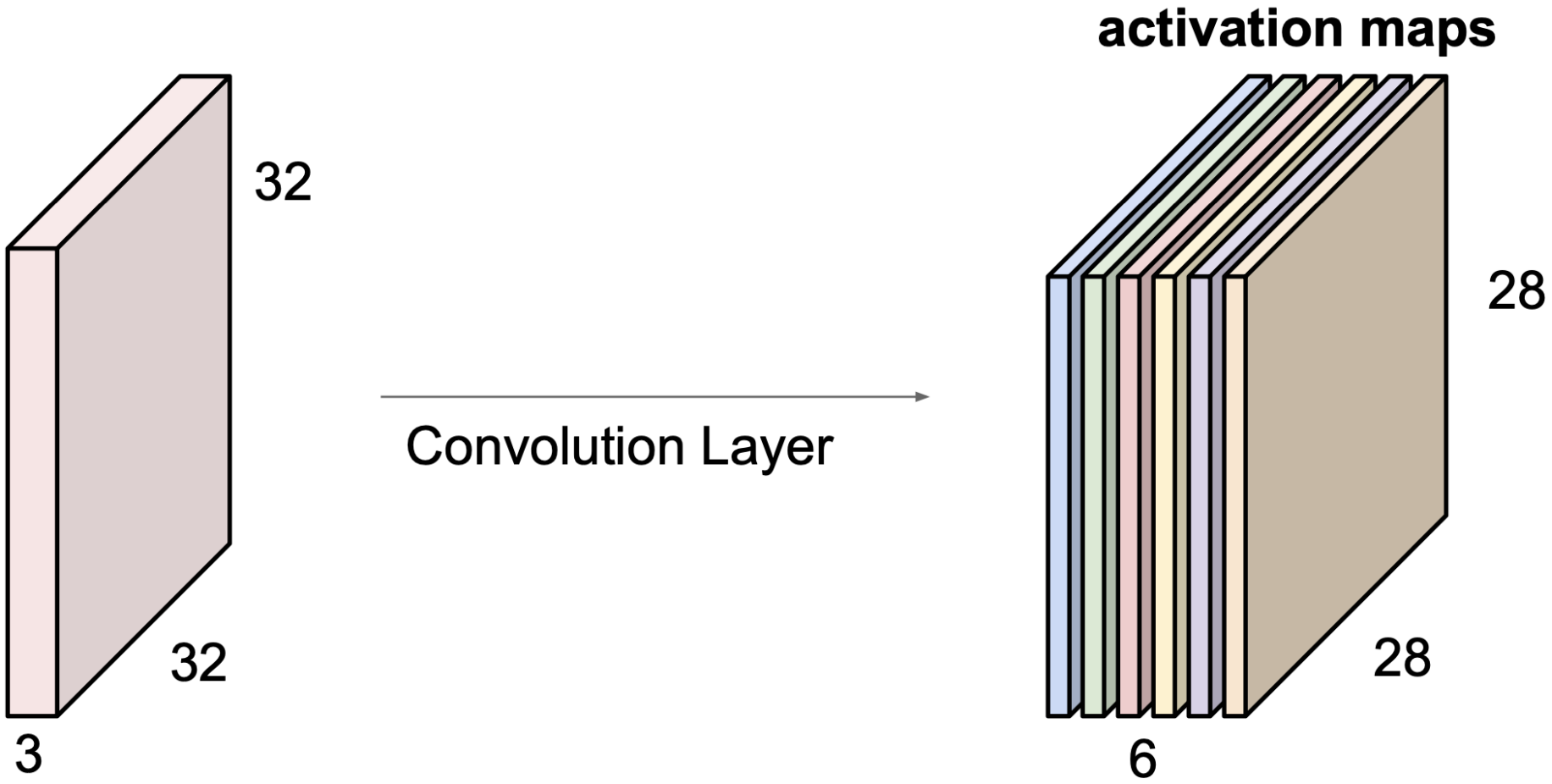


Convolution Layer

consider a second, green filter

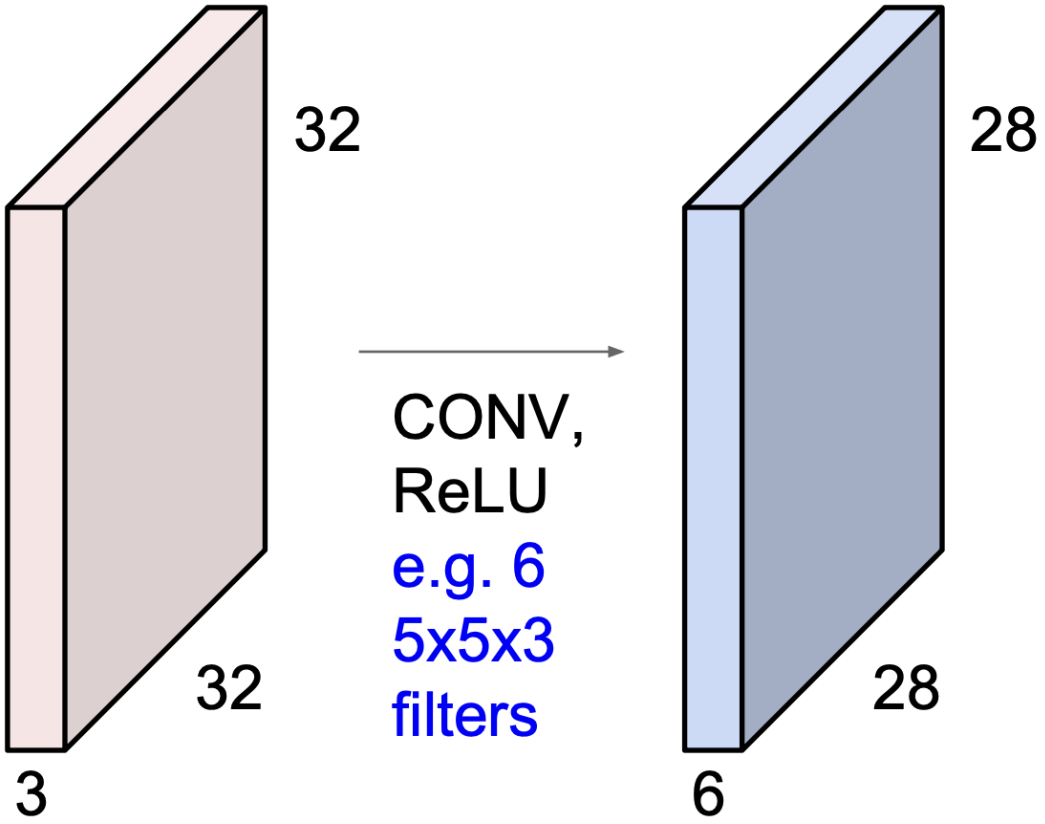


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

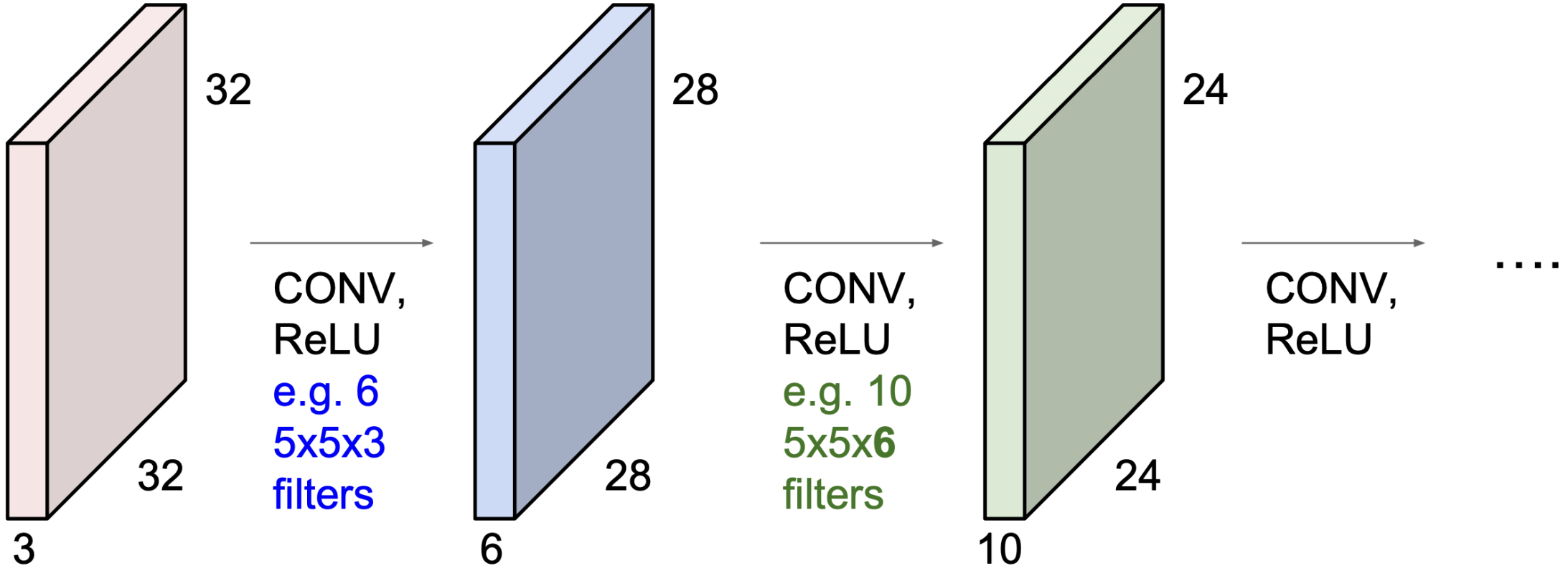


We stack these up to get a "new image" of size 28x28x6!

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



- **Stride** refers to the number of pixels by which the filter/kernel is moved across the input image or matrix when performing the convolution operation. A stride of 1 means the filter moves one pixel at a time. A larger stride results in downsampling the input and leads to a smaller output dimension.

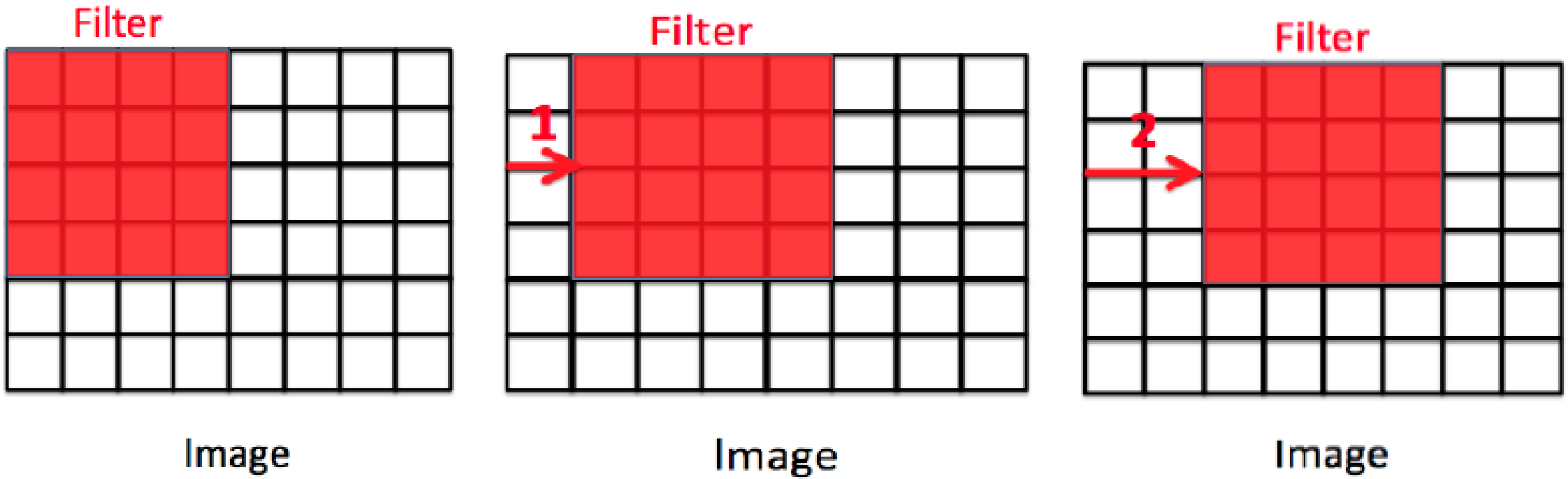
- **Padding** is a technique used to preserve the spatial dimensions of the input volume such that the spatial size of the output volume is the same as the input volume when using a convolutional layer. Padding adds a certain number of rows and columns to the input image/matrix, typically with zeros (zero-padding). This allows the convolution operation to be applied to the bordering elements of the input.

- A **kernel** or **filter** in the context of CNNs is a small matrix used to apply effects such as blurring, sharpening, edge detection, etc. This kernel is convolved over the input matrix or image to produce a feature map indicating the presence of specific features or patterns in the input.

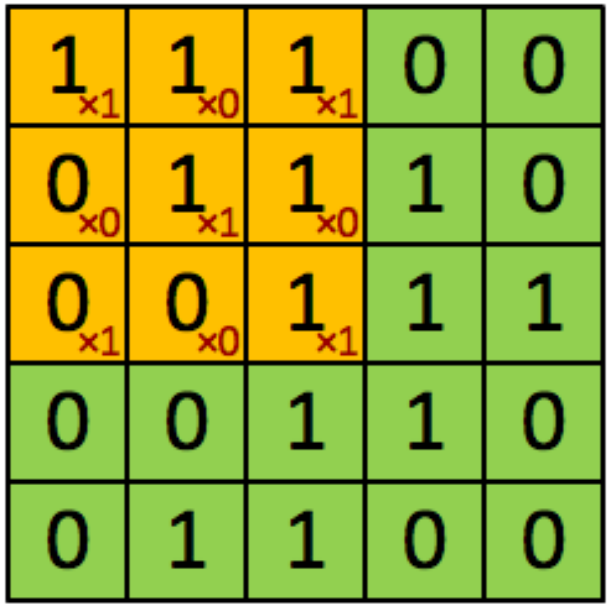
- A **feature map** is the output generated by applying a filter to the input. It represents the responses of the filter throughout the spatial extent of the input. Essentially, it is a mapped output of where certain features are detected in the input.

- **Convolution** is the operation of adding each element of the image to its local neighbors, weighted by the kernel. This operation blends the pixels within a local region of the image to produce a single average output of that region.

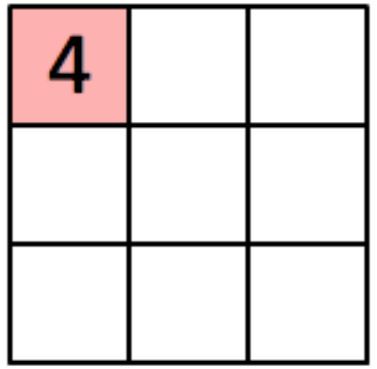
- **Pooling** is a technique used to reduce the dimensions of the feature maps. It reduces the computational complexity and the number of parameters. The most common pooling operation is max pooling, which selects the maximum element from the region of the feature map covered by the filter. There is also average pooling, which calculates the average of the elements in that region.


Filter
Image
Filter
1
Image
Filter
2
Image

Convolution operation



Image



Convolved Feature

- The kernel, a yellow matrix, slides horizontally and then vertically across the original green matrix (image) to detect features by calculating the dot product sum of image and kernel values, outputting a new matrix.
- This kernel, with initially random values, is a trainable parameter.

Producing Feature Maps

Original

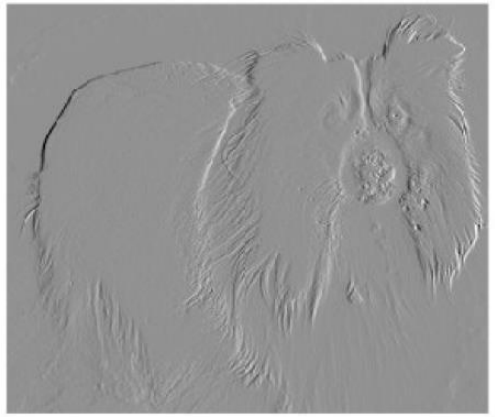
Sharpen

Edge Detect

"Strong" Edge Detect

A simple pattern: Edges
How can we detect edges with a kernel?

Input

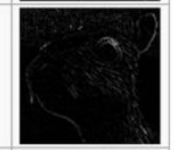


Output

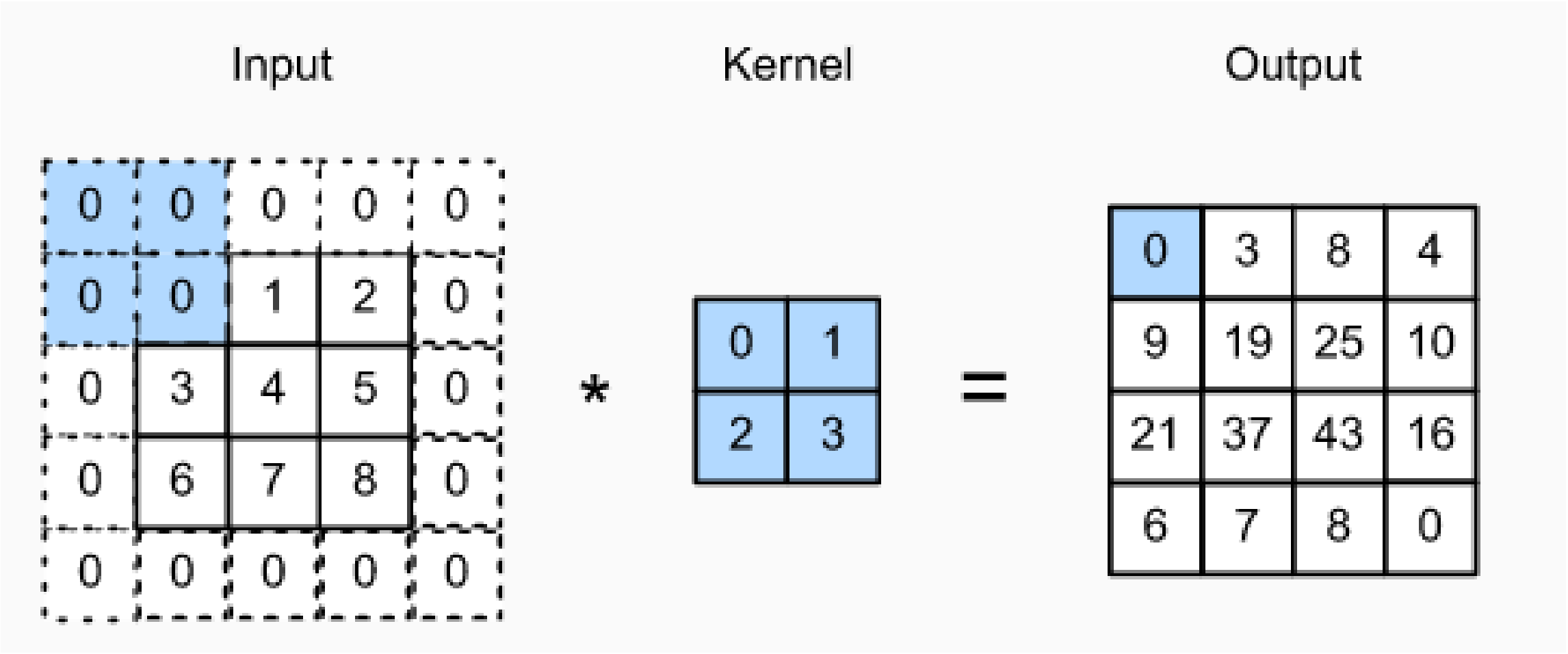
1	-1

Filter

Simple Kernels / Filters

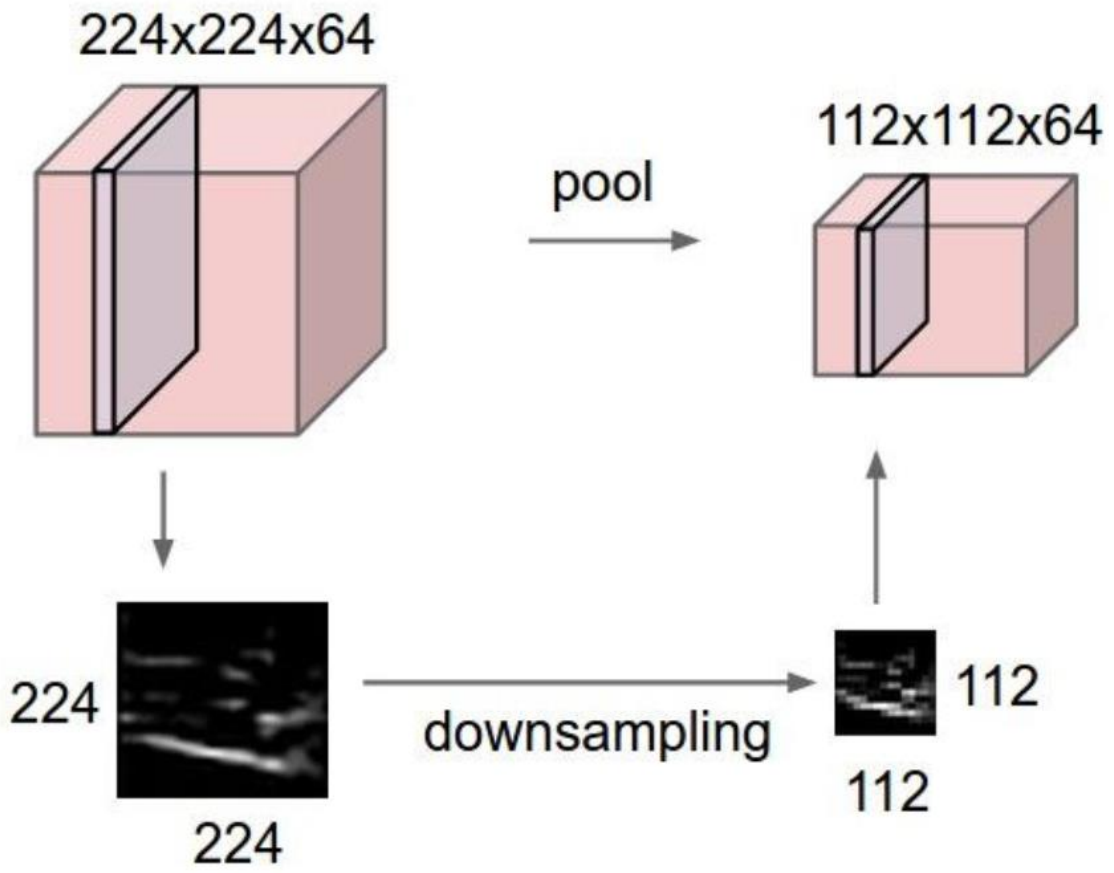
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Padding


Input
Kernel
Output
0 0 0 0 0
0 0 1 2 0
0 3 4 5 0
0 6 7 8 0
0 0 0 0 0
*
0 1
2 3
=
0 3 8 4
9 19 25 10
21 37 43 16
6 7 8 0

Pooling layer

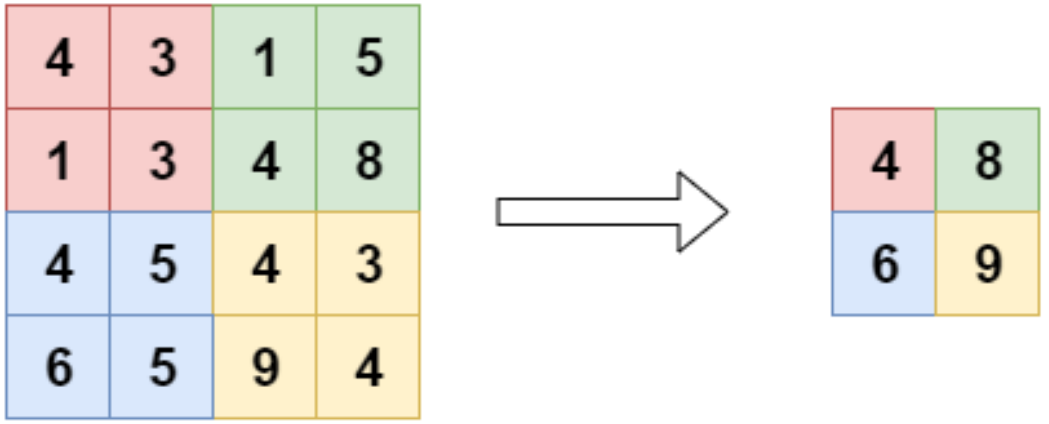
- makes the representations smaller and more manageable
- operates over each activation map independently:



Max Pooling - Selects the brighter pixels from the image.

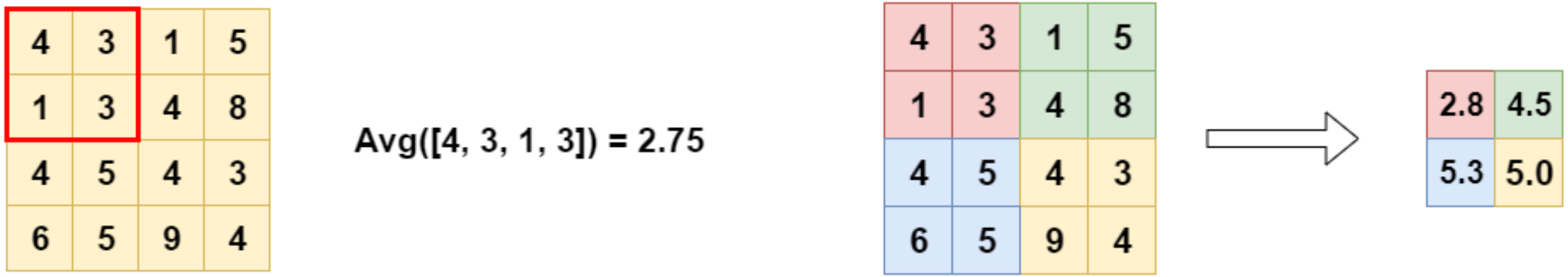
4	3	1	5
1	3	4	8
4	5	4	3
6	5	9	4

Max([4, 3, 1, 3]) = 4



A pooling layer is another building block of a CNN. Pooling Its function is to progressively reduce the spatial size of the representation to reduce the network complexity and computational cost.

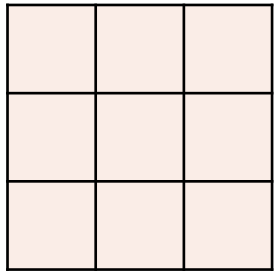
Average Pooling - retains much information about the “less important” elements of a block, or pool.



A pooling layer is another building block of a CNN. Pooling Its function is to progressively reduce the spatial size of the representation to reduce the network complexity and computational cost.

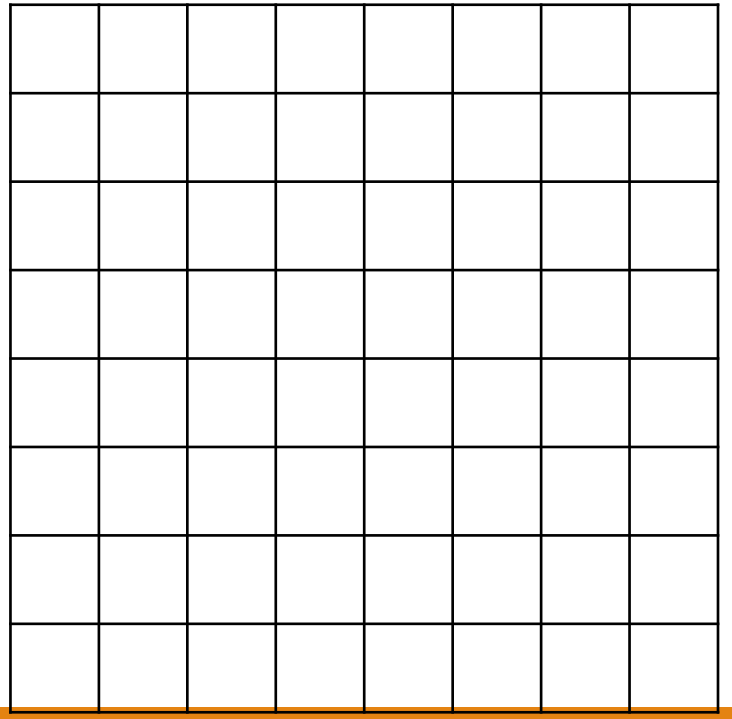
Convolutional Layers

Input Shape
8x8

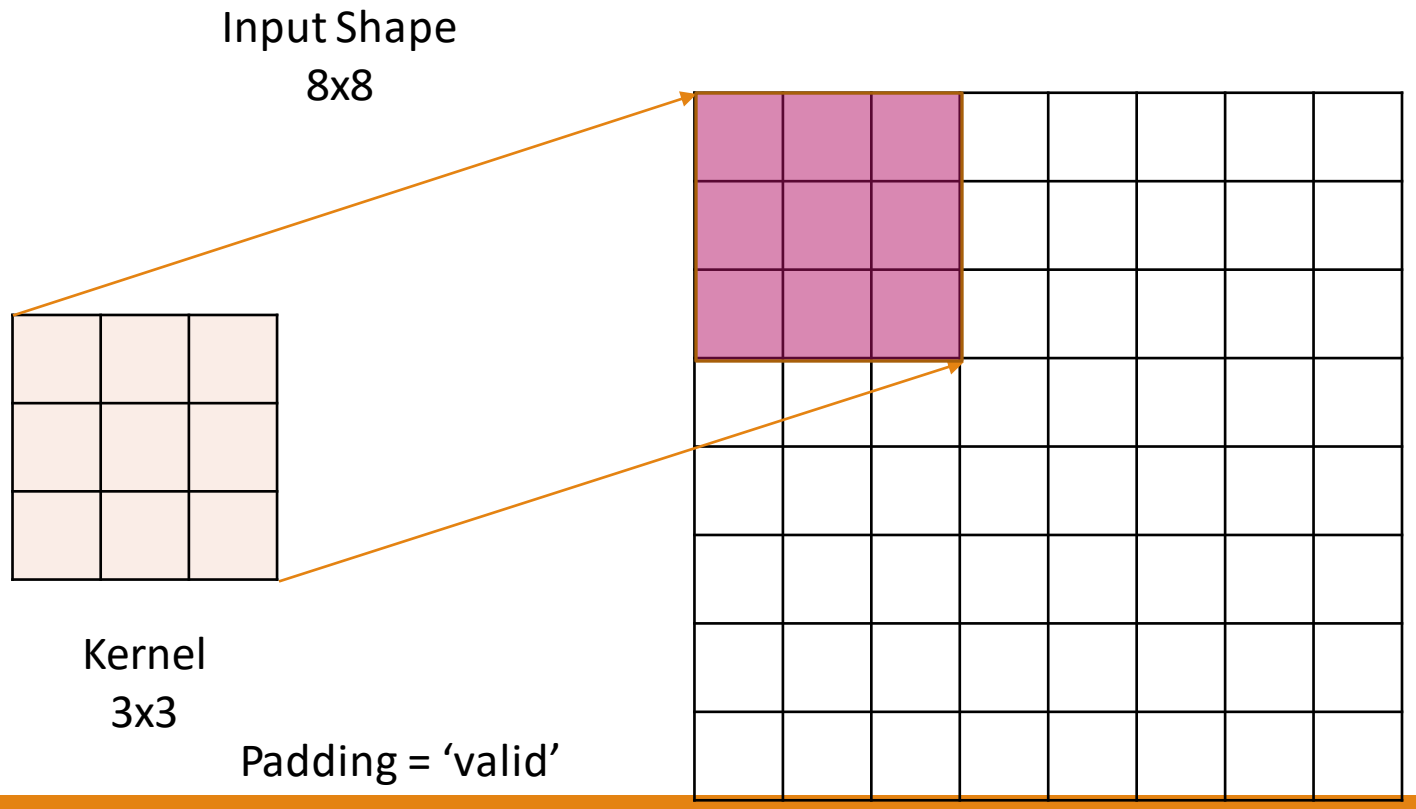


Kernel
3x3

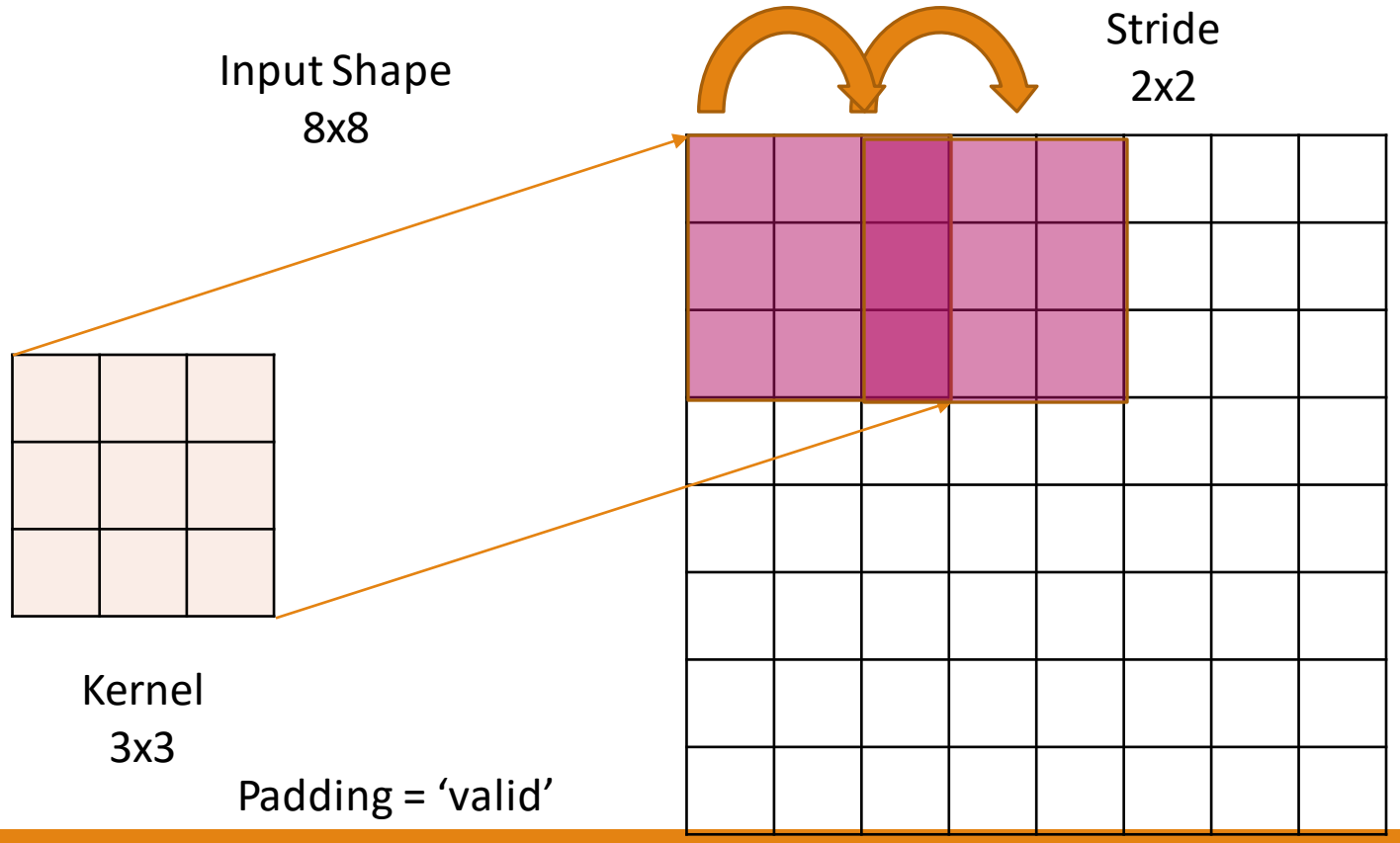
Padding = ‘valid’



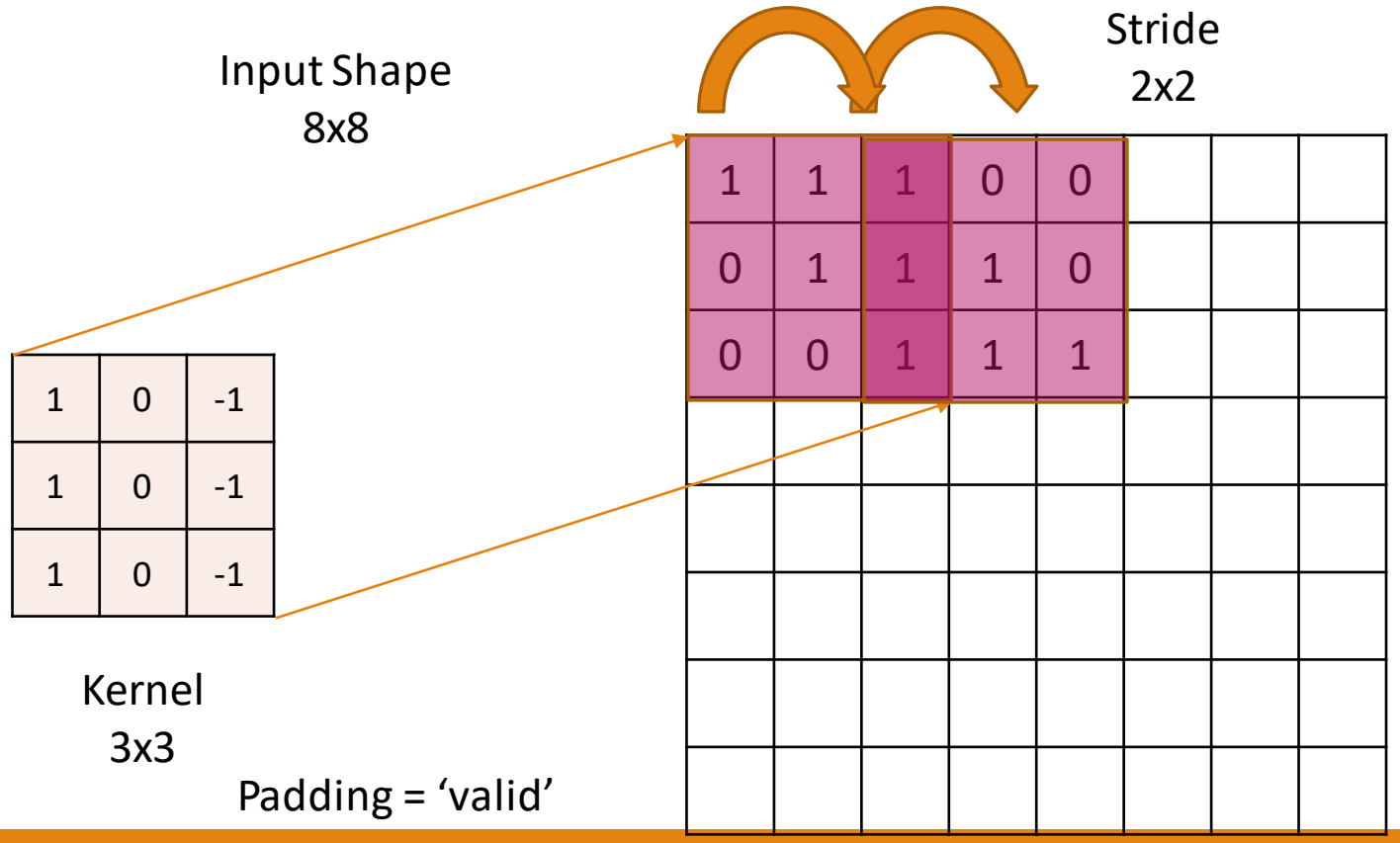
Convolutional Layers



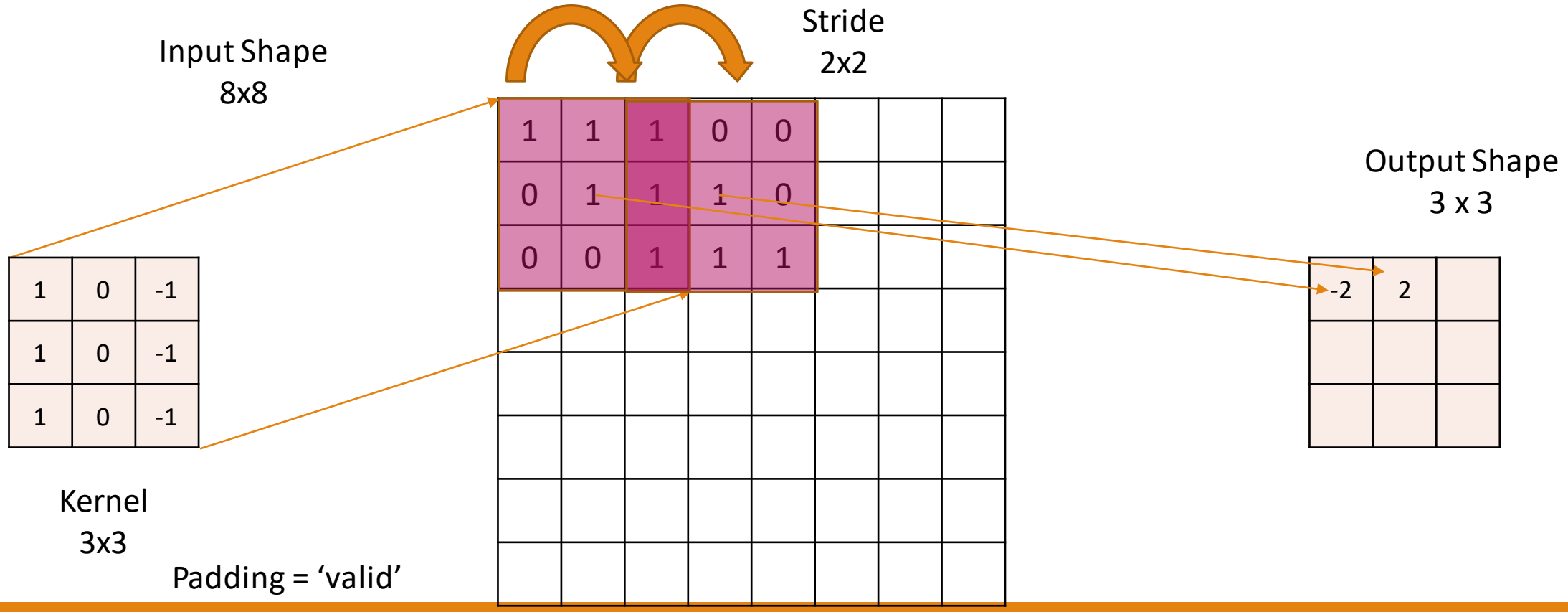
Convolutional Layers



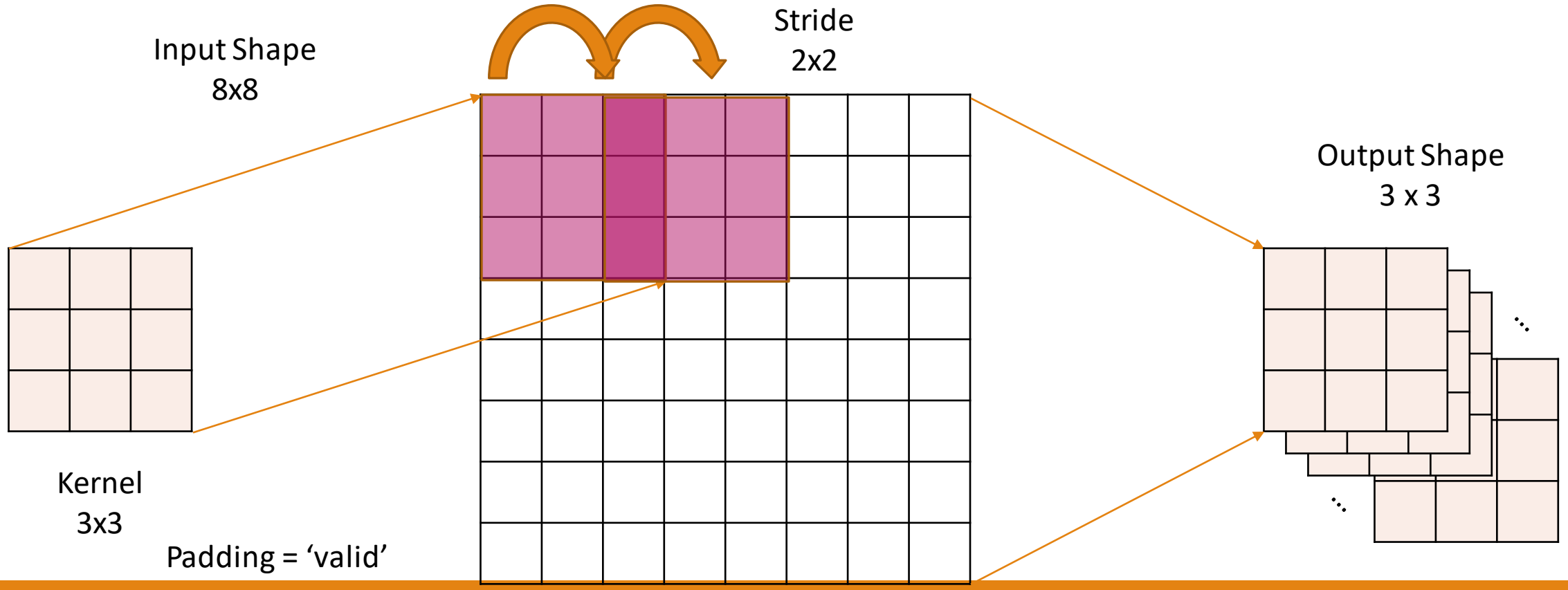
Convolutional Layers

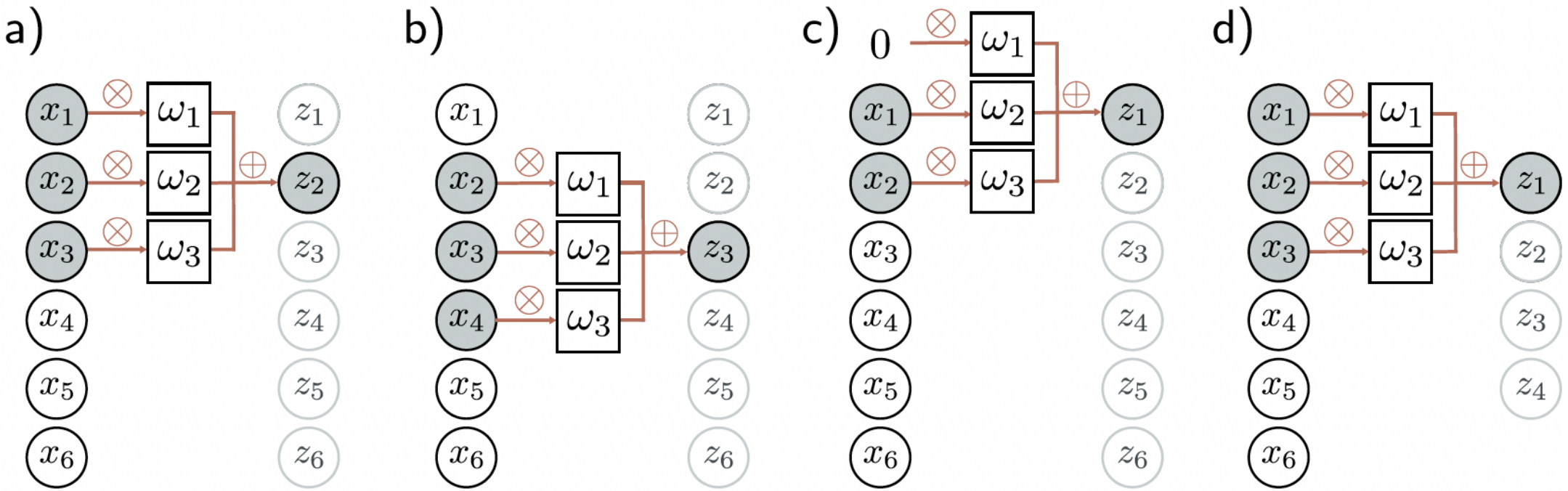


Convolutional Layers

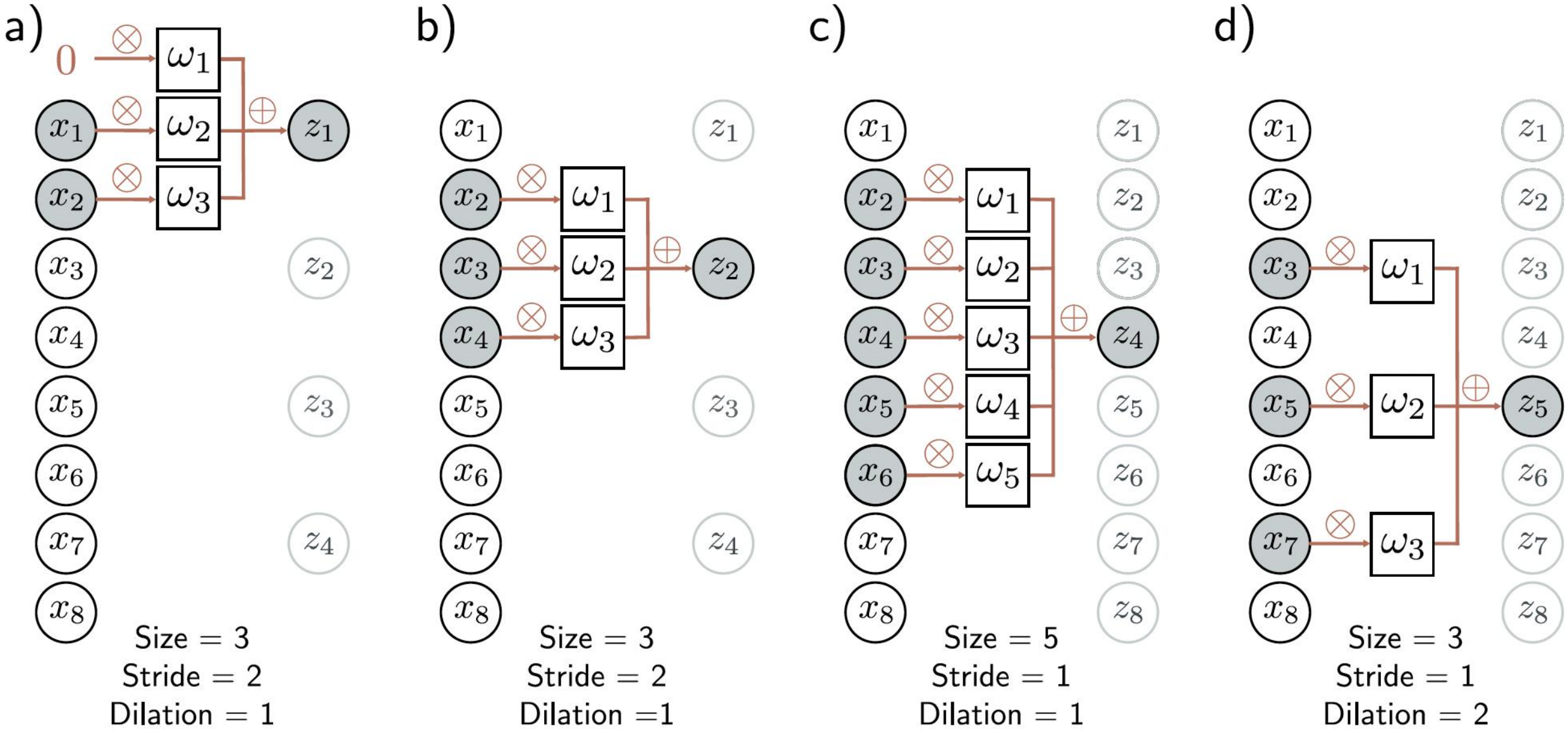


Convolutional Layers





$$z_i = \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1},$$


a)
b)
c)
d)
Size = 3
Stride = 2
Dilation = 1
Size = 3
Stride = 2
Dilation =1
Size = 5
Stride = 1
Dilation = 1
Size = 3
Stride = 1
Dilation = 2

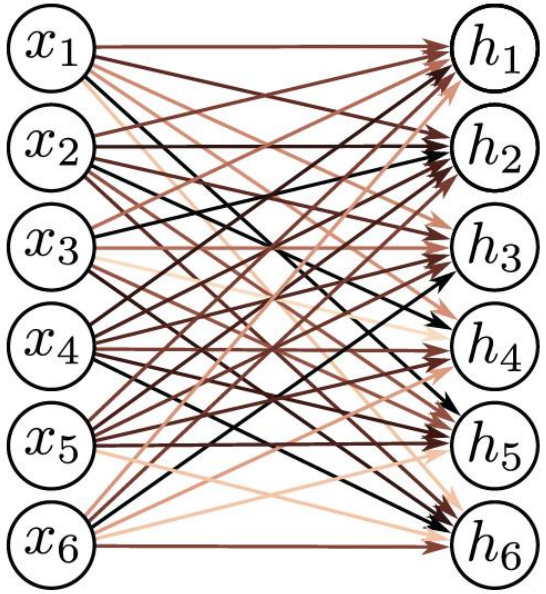
Convolutional layers

$$
\begin{aligned}
h_i &= \mathrm{a}\left[\beta + \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}\right] \\
&= \mathrm{a}\left[\beta + \sum_{j=1}^{3} \omega_j x_{i+j-2}\right],
\end{aligned}
$$

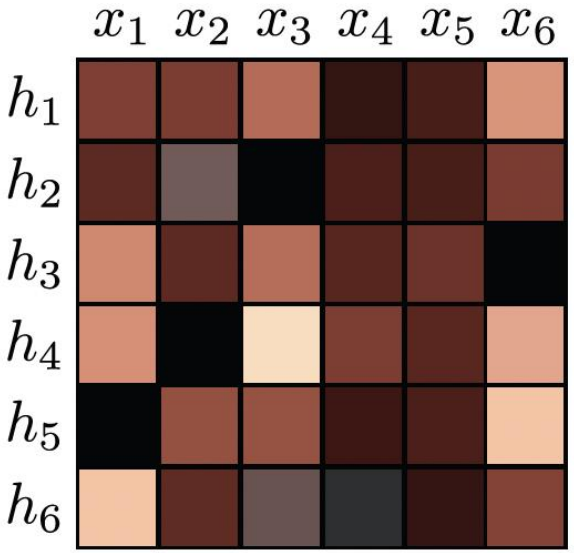
fully connected layer

$$
h_i = \mathrm{a}\left[\beta_i + \sum_{j=1}^{D} \omega_{ij} x_j\right].
$$

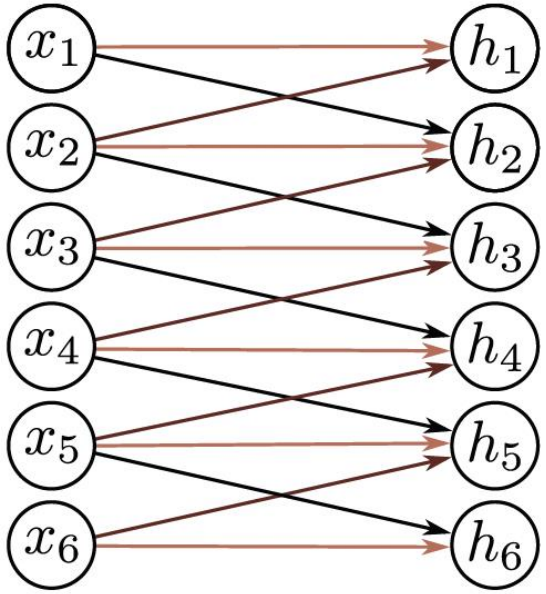
a)


x_1 x_2 x_3 x_4 x_5 x_6
h_1 h_2 h_3 h_4 h_5 h_6

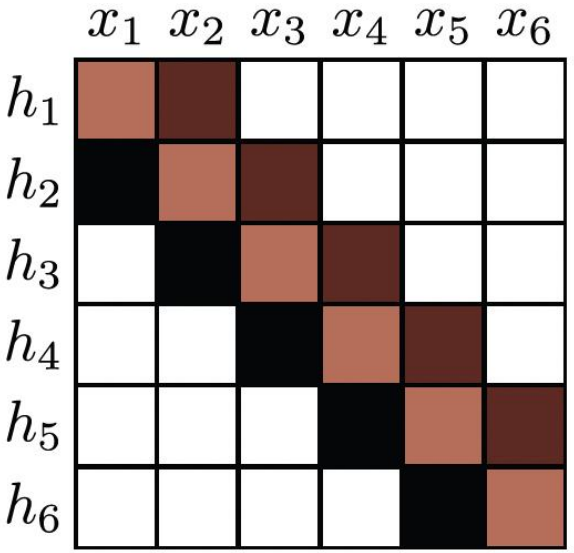
b)


x_1 x_2 x_3 x_4 x_5 x_6
h_1 h_2 h_3 h_4 h_5 h_6

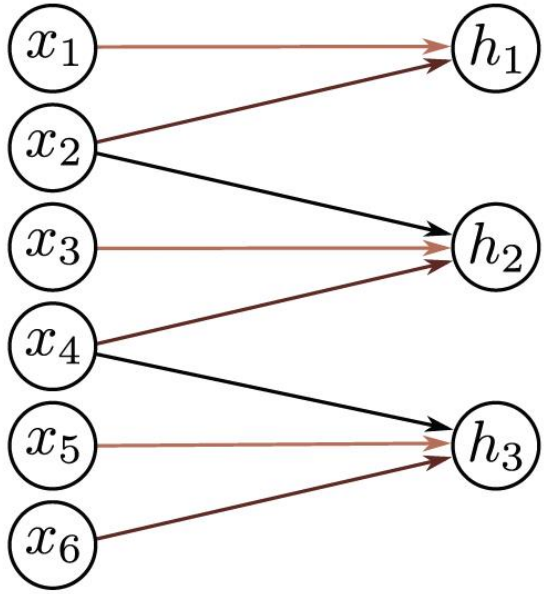
c)


x_1 x_2 x_3 x_4 x_5 x_6
h_1 h_2 h_3 h_4 h_5 h_6

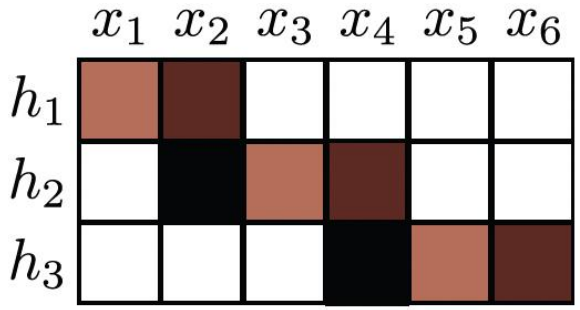
d)

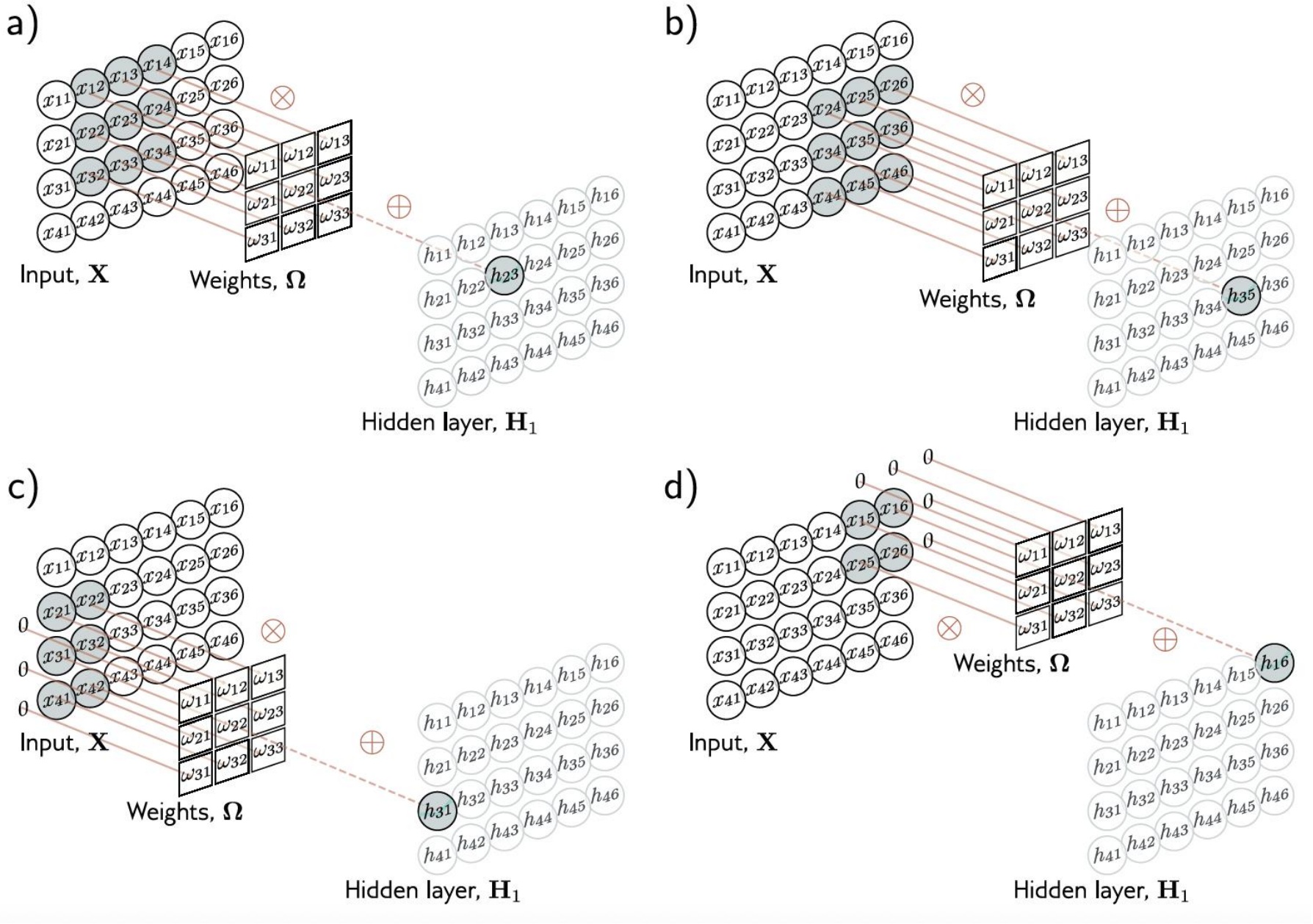

x_1 x_2 x_3 x_4 x_5 x_6
h_1 h_2 h_3 h_4 h_5 h_6

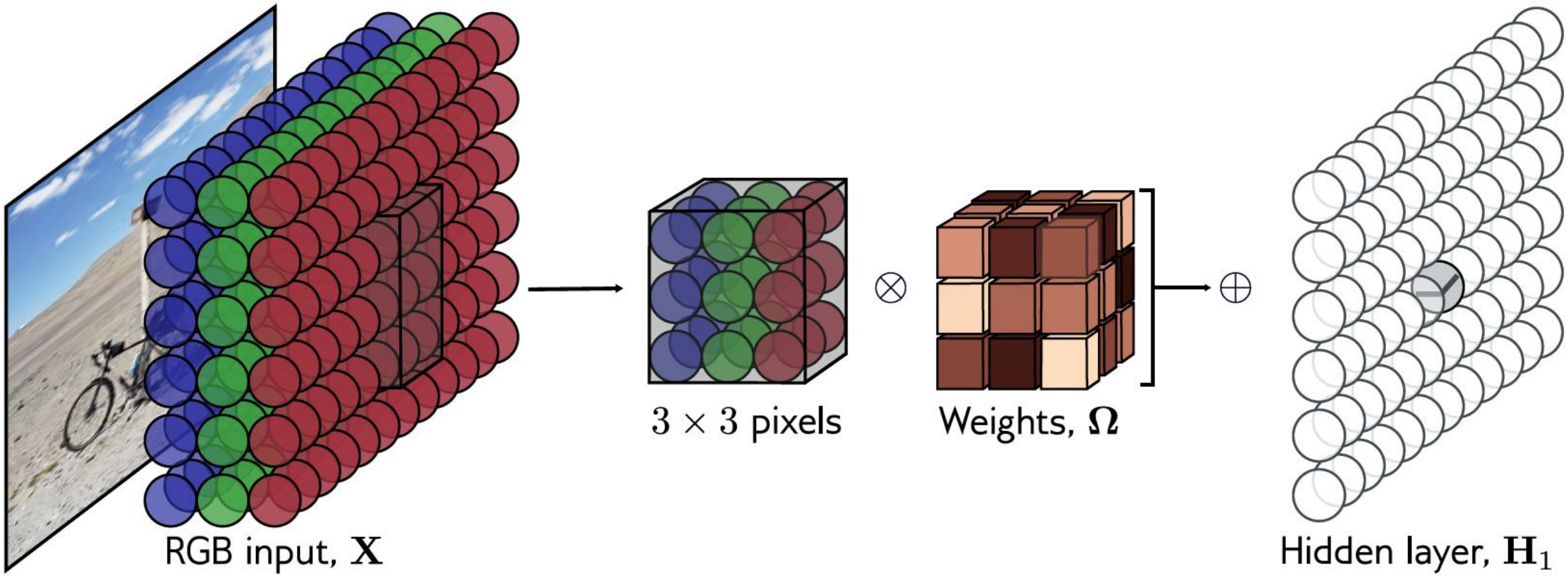
e)

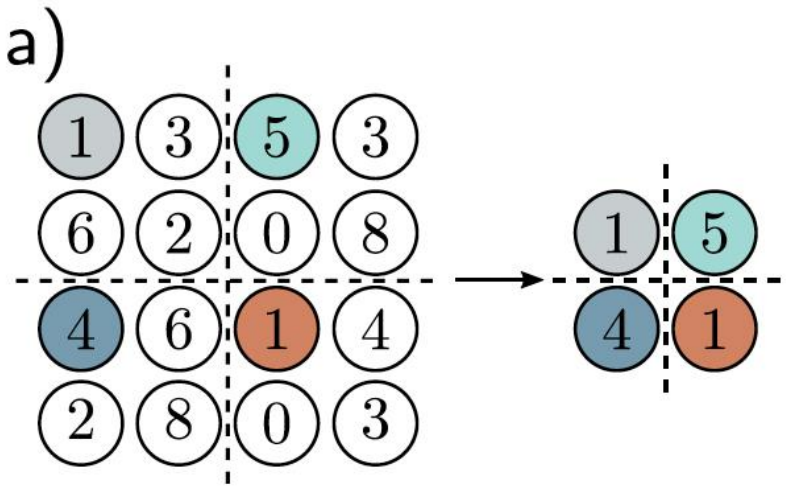

x_1 x_2 x_3 x_4 x_5 x_6
h_1 h_2 h_3

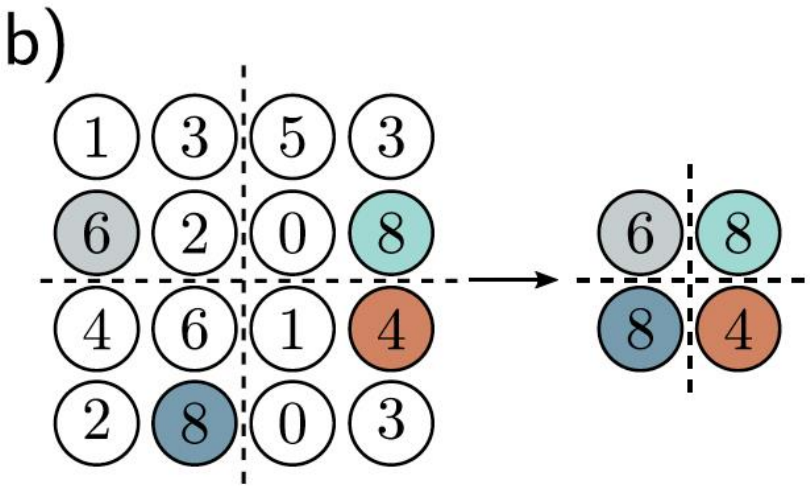
f)

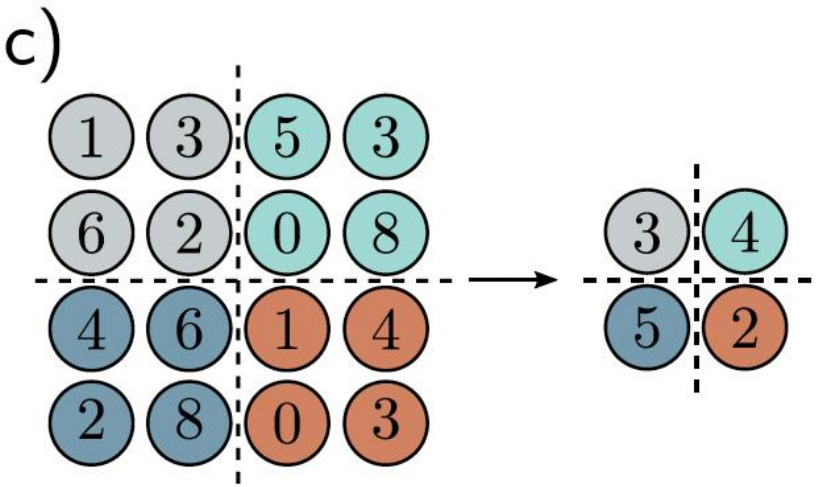

x_1 x_2 x_3 x_4 x_5 x_6
h_1 h_2 h_3

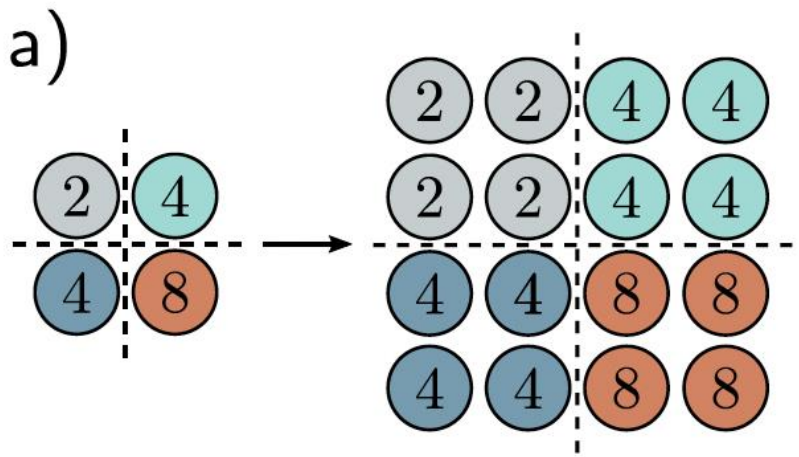

a)
Input, $\mathbf{X}$
Weights, $\boldsymbol{\Omega}$
Hidden layer, $\mathbf{H}_1$
b)
Input, $\mathbf{X}$
Weights, $\boldsymbol{\Omega}$
Hidden layer, $\mathbf{H}_1$
c)
Input, $\mathbf{X}$
Weights, $\boldsymbol{\Omega}$
Hidden layer, $\mathbf{H}_1$
d)
Input, $\mathbf{X}$
Weights, $\boldsymbol{\Omega}$
Hidden layer, $\mathbf{H}_1$

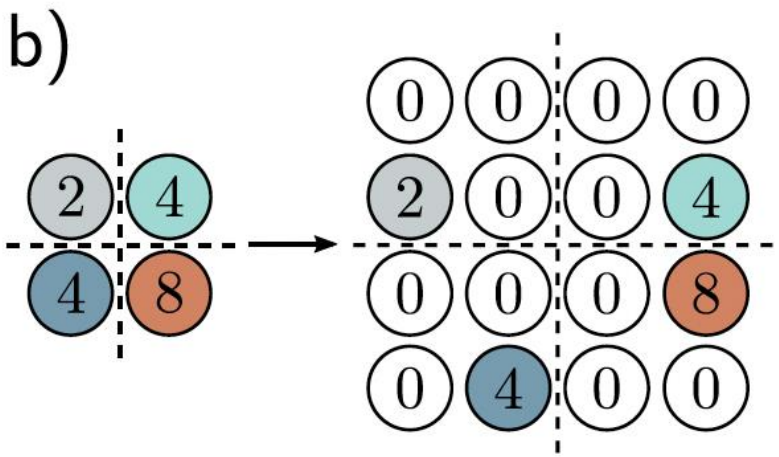

RGB input, $\mathbf{X}$
3×3 pixels
Weights, $\mathbf{\Omega}$
Hidden layer, $\mathbf{H}_1$

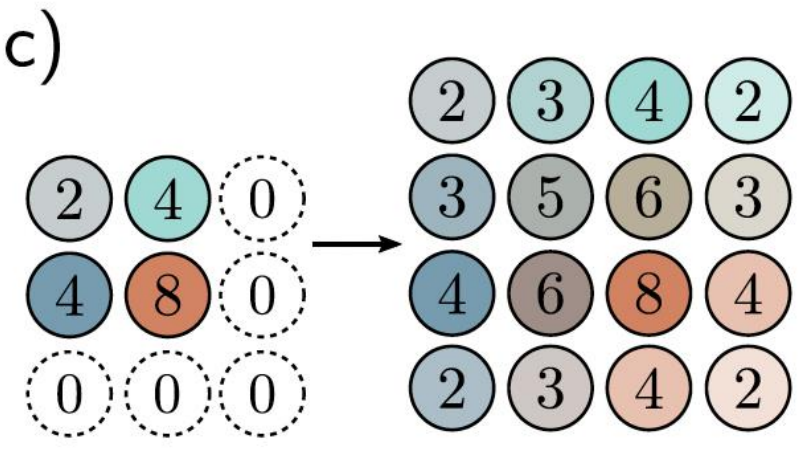

a)
1 3 5 3
6 2 0 8
4 6 1 4
2 8 0 3
1 5
4 1

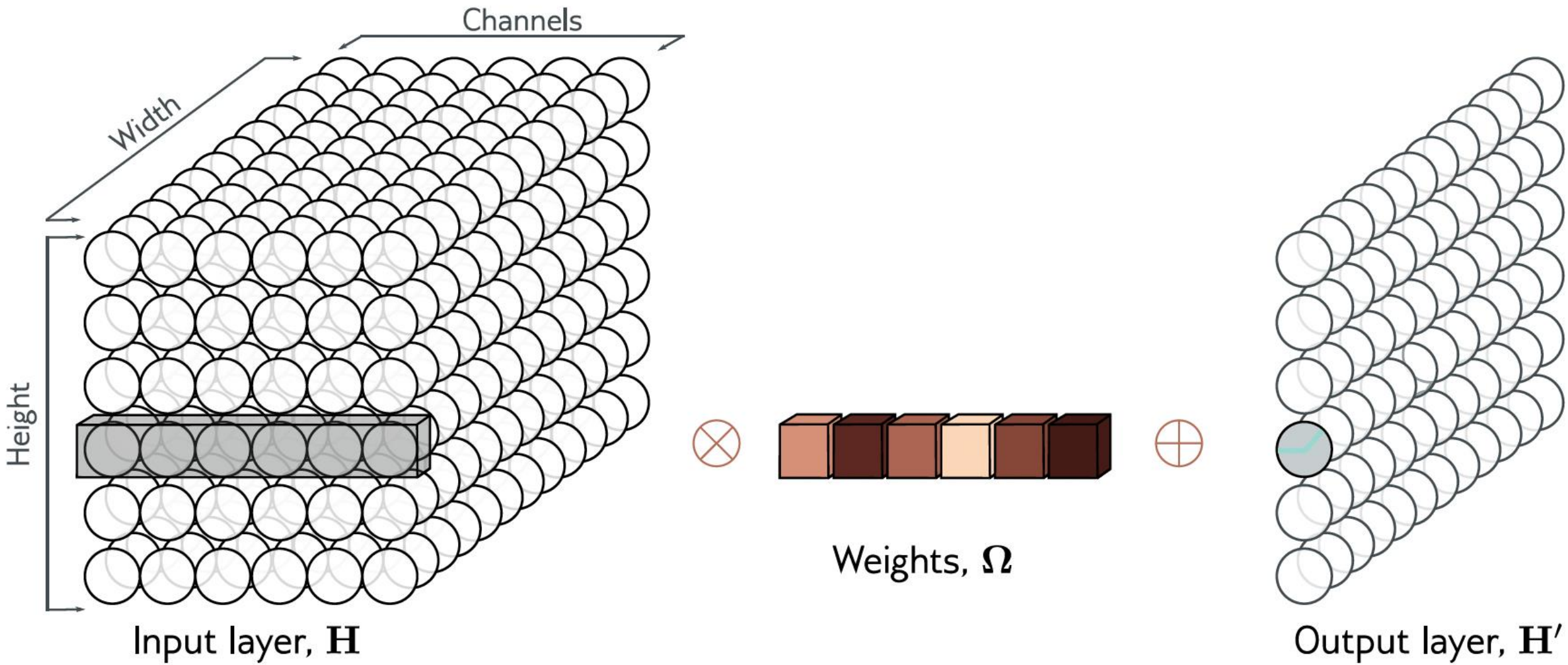

b)
1 3 5 3
6 2 0 8
4 6 1 4
2 8 0 3
6 8
8 4

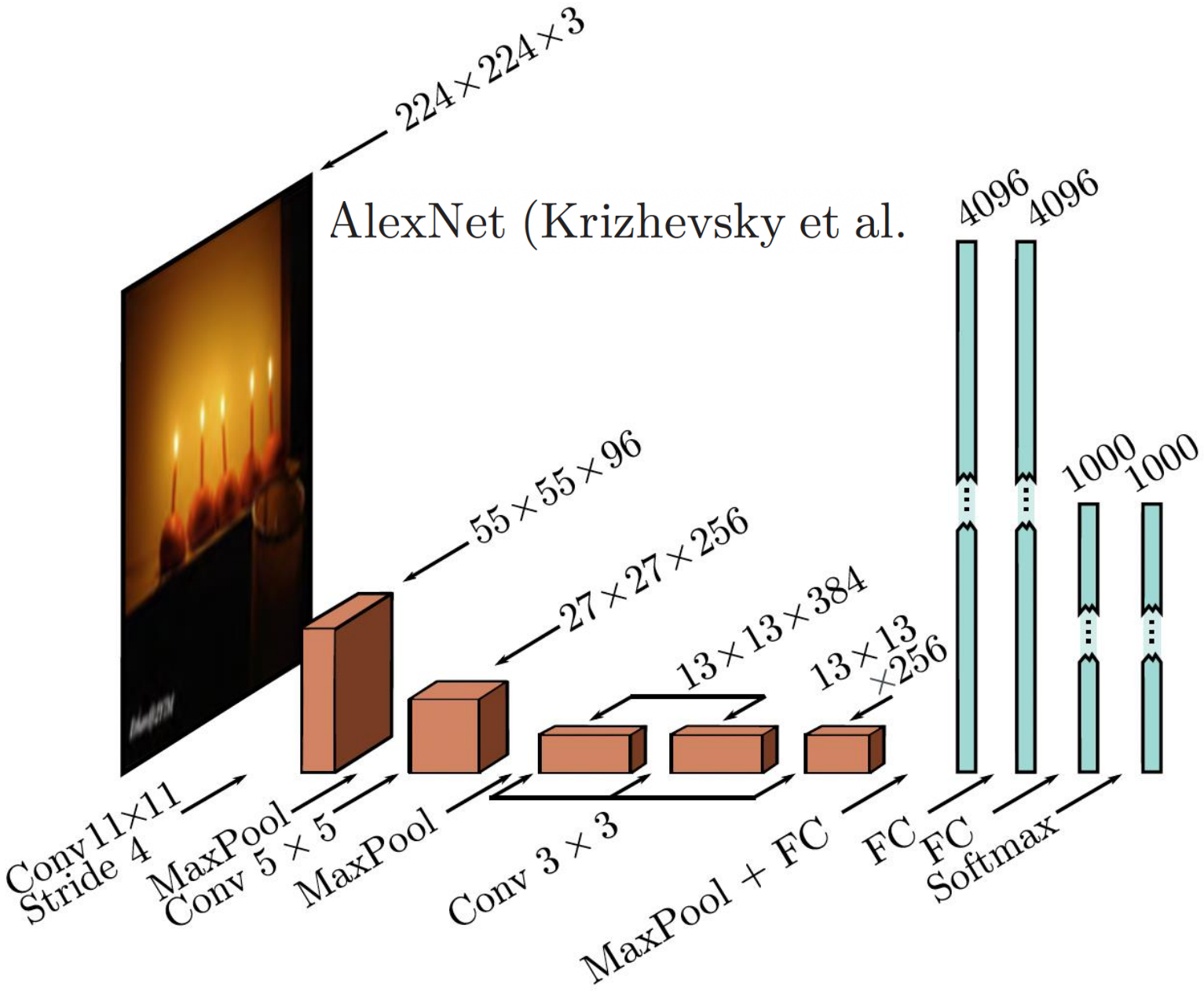

c)
1 3 5 3
6 2 0 8
4 6 1 4
2 8 0 3
3 4
5 2


a)
2 4
4 8
2 2 4 4
2 2 4 4
4 4 8 8
4 4 8 8


b)
2 4
4 8
0 0 0 0
2 0 0 4
0 0 0 8
0 4 0 0

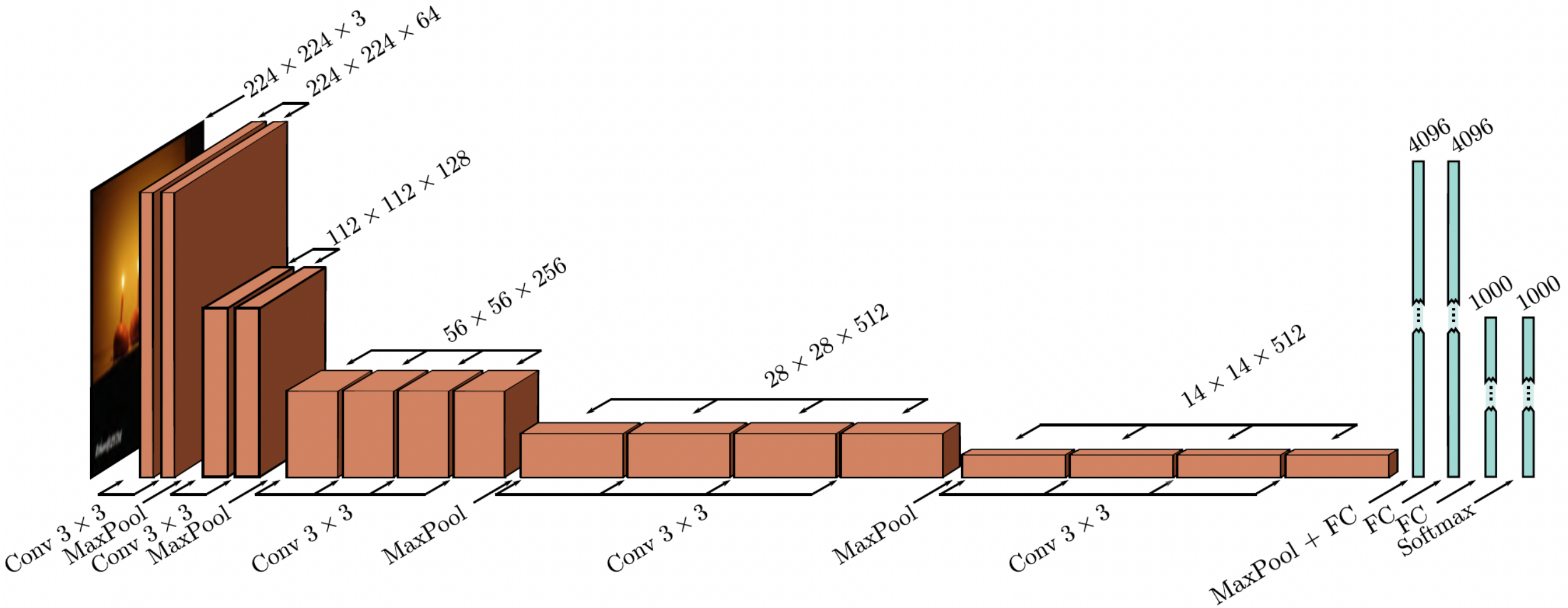

c)
2 4 0
4 8 0
0 0 0
2 3 4 2
3 5 6 3
4 6 8 4
2 3 4 2

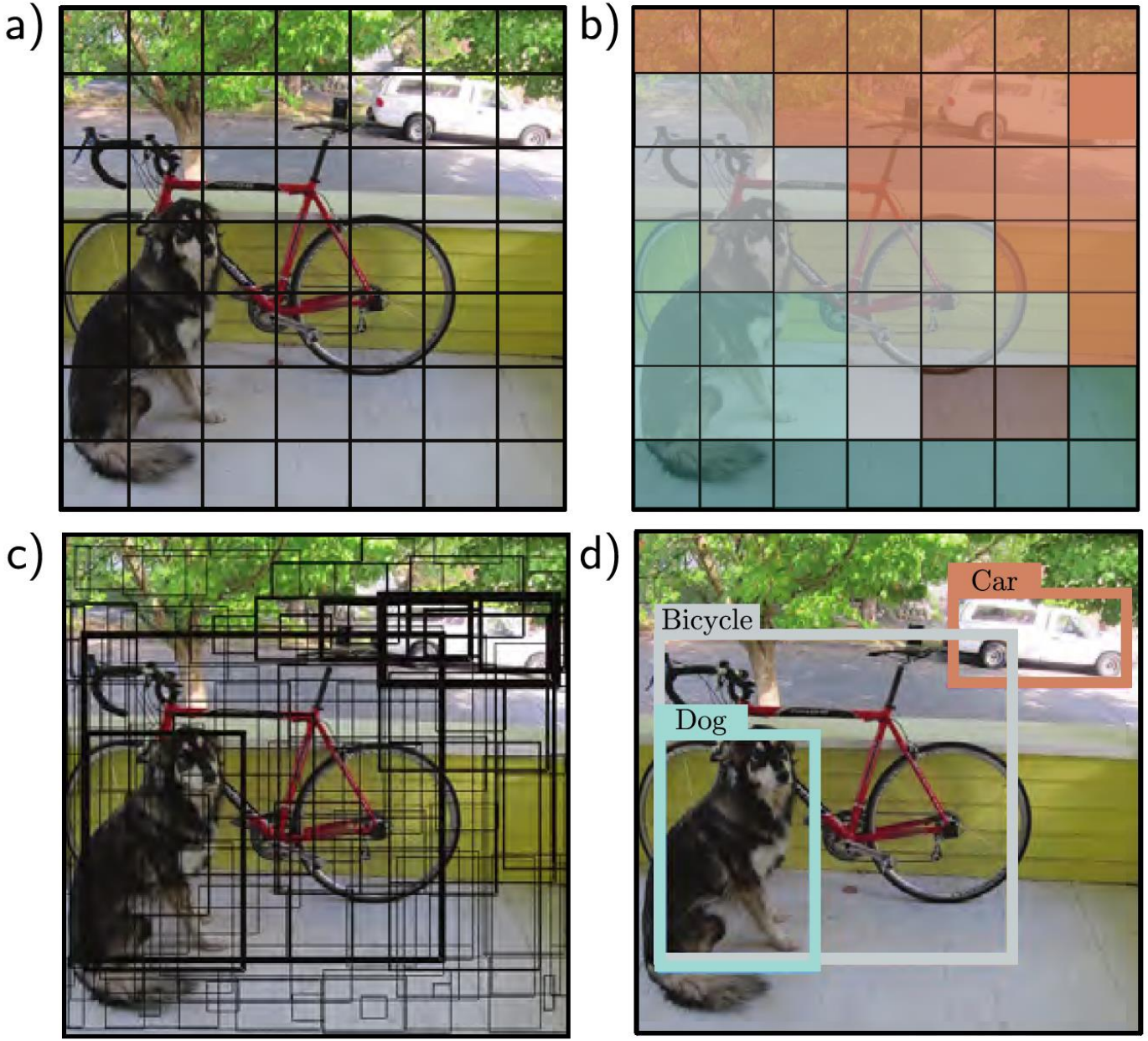

Channels
Width
Height
Input layer, $\mathbf{H}$
Weights, $\mathbf{\Omega}$
Output layer, $\mathbf{H}'$


AlexNet (Krizhevsky et al.
224×224×3
55×55×96
27×27×256
13×13×384
13×13
×256
4096
4096
1000
1000
Conv11×11
Stride 4
MaxPool
Conv 5 × 5
MaxPool
Conv 3 × 3
MaxPool + FC
FC
FC
Softmax


Texture
Distinct color
Distinct shape

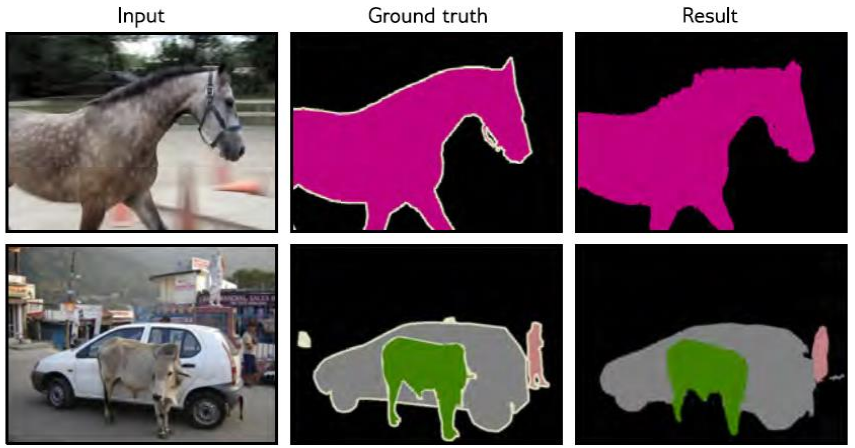
VGG network (Simonyan & Zisserman, 2014)

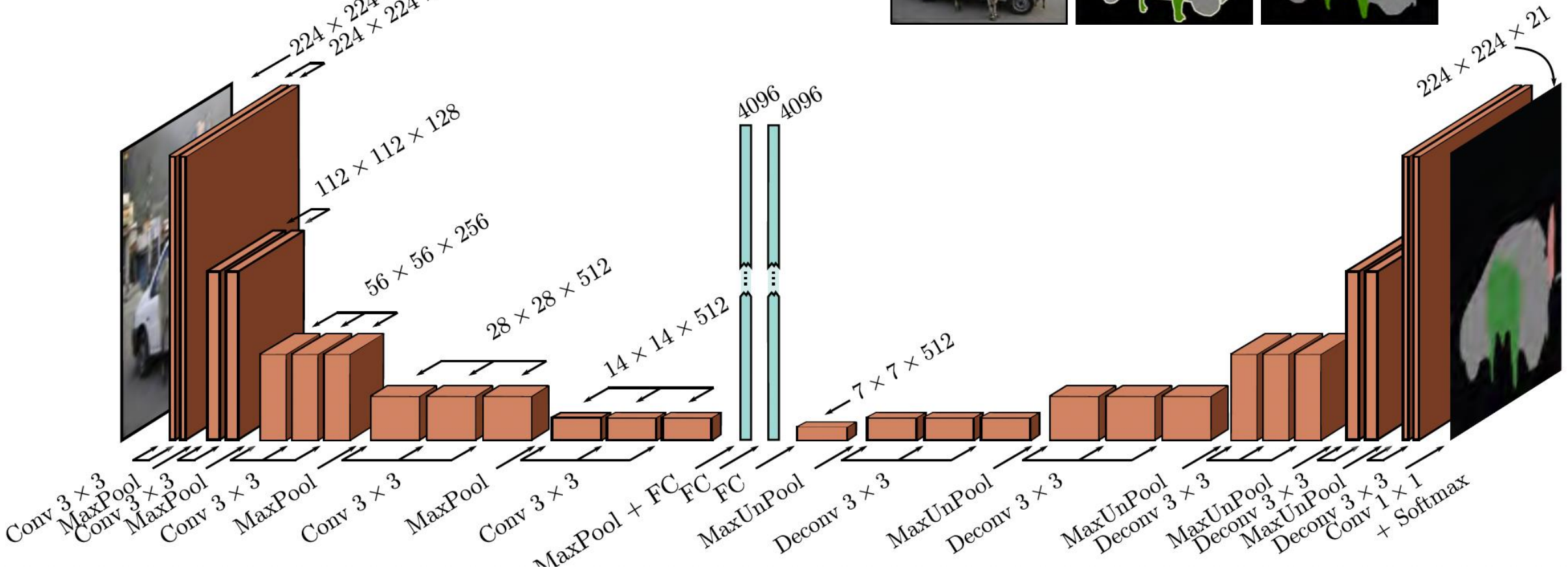


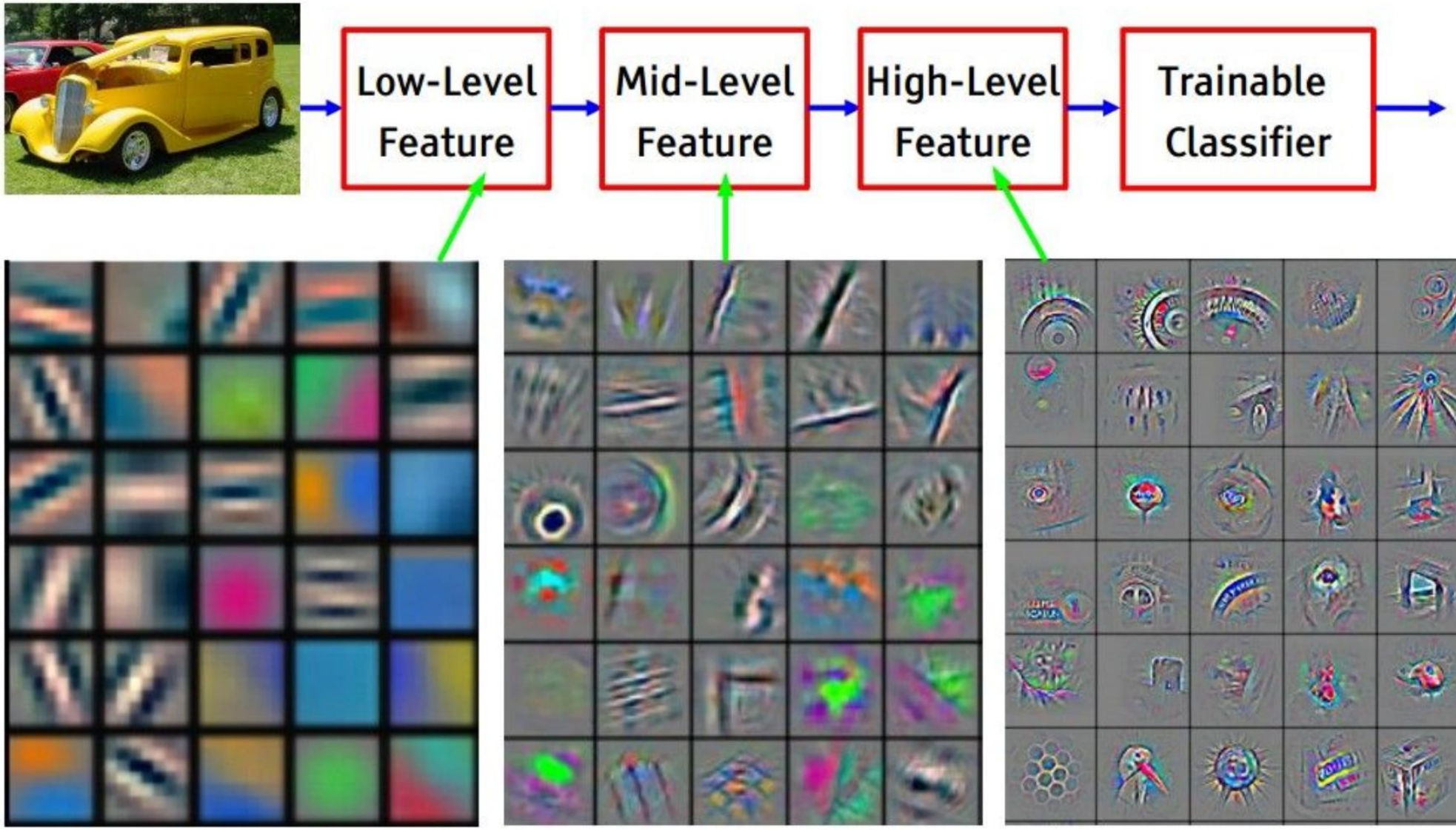


YOLO object detection

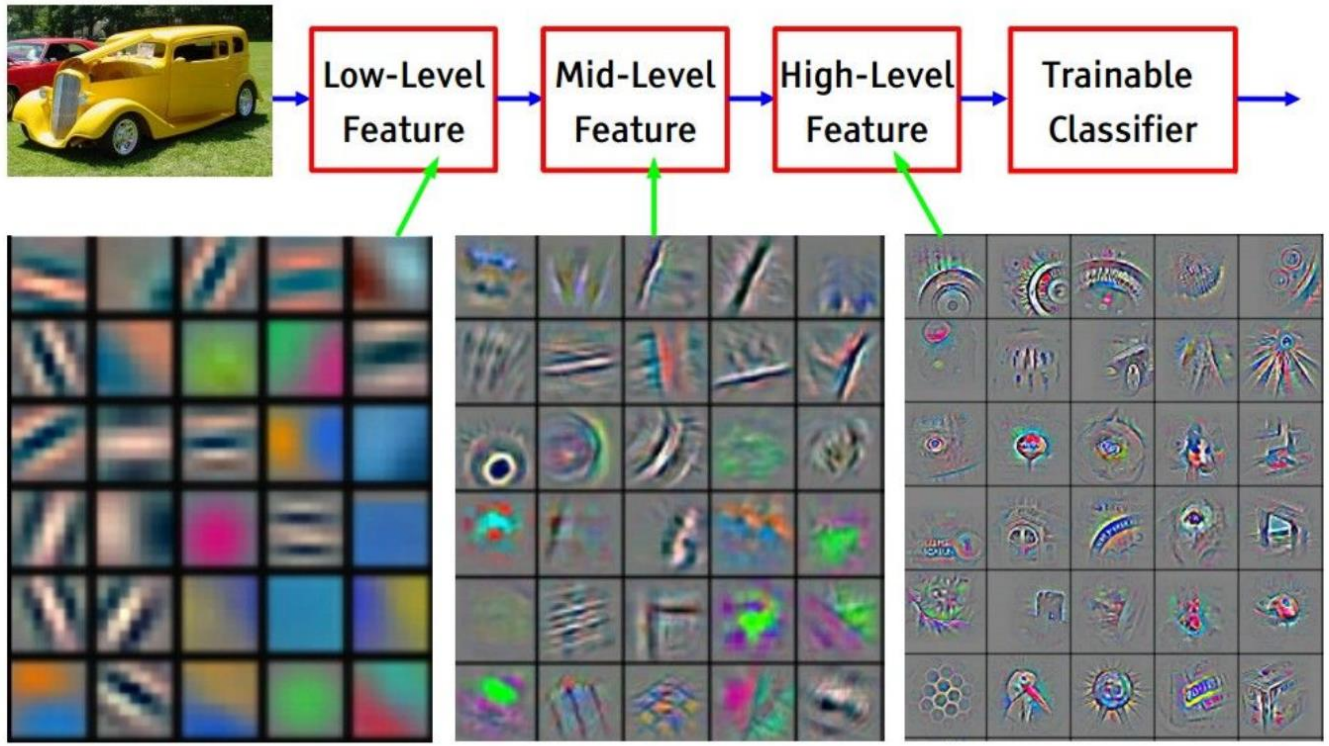
Semantic segmentation



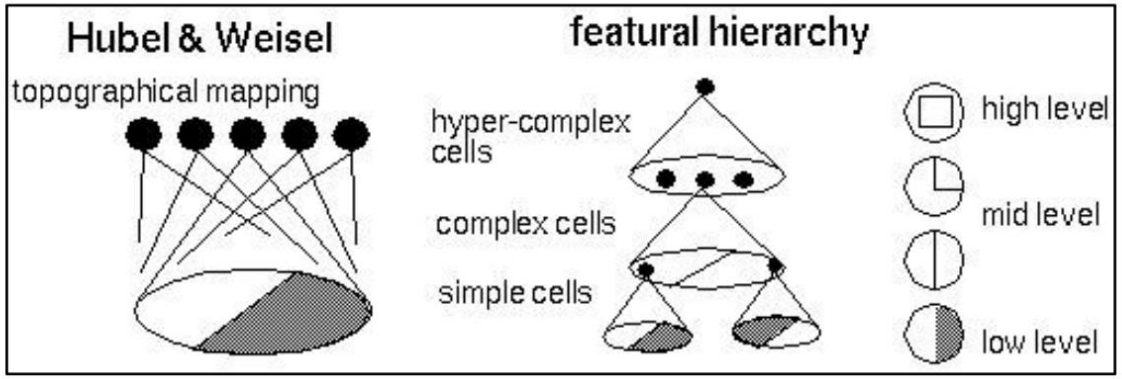




Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

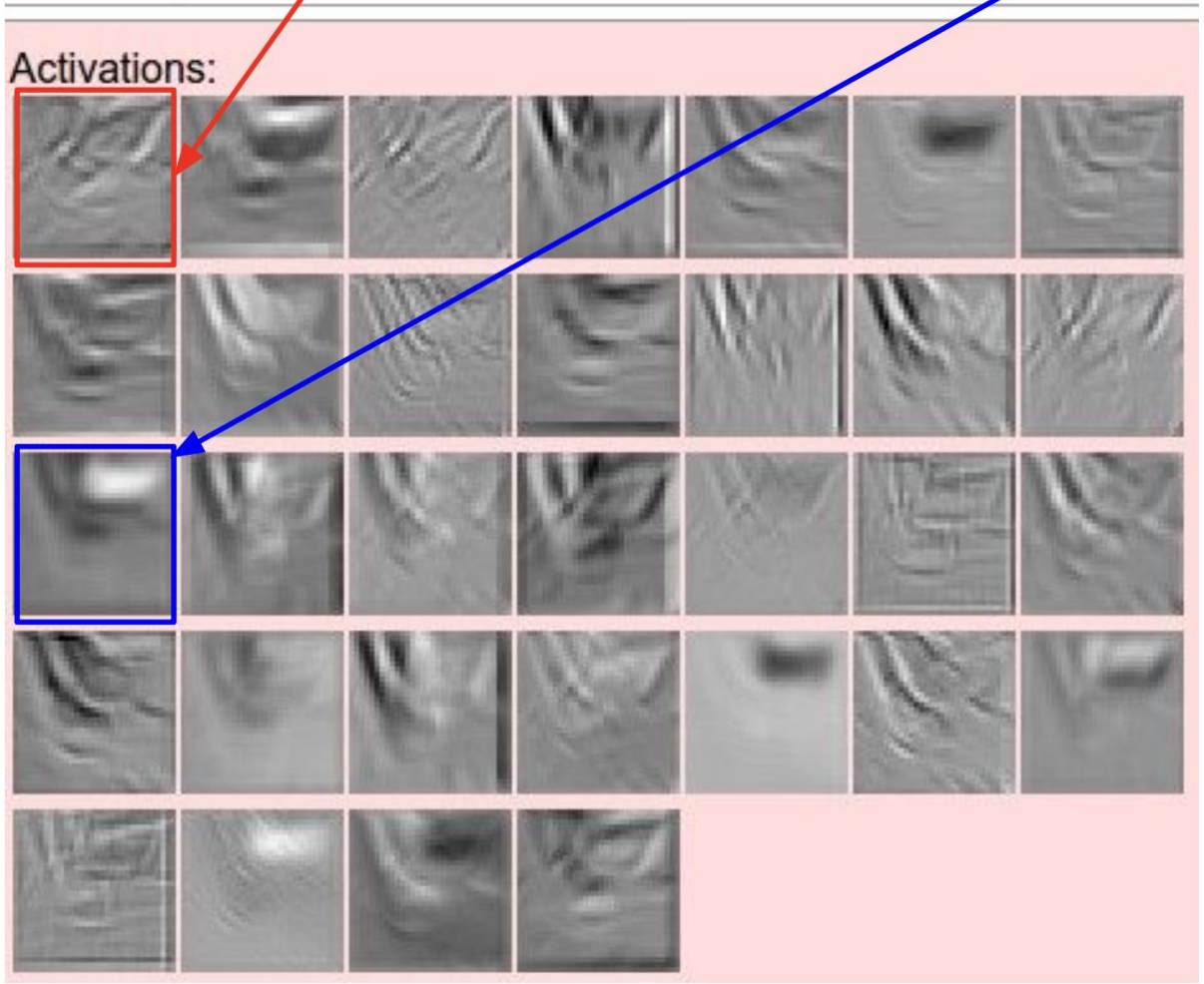


Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]



[From recent Yann LeCun slides]

one filter =>
one activation map

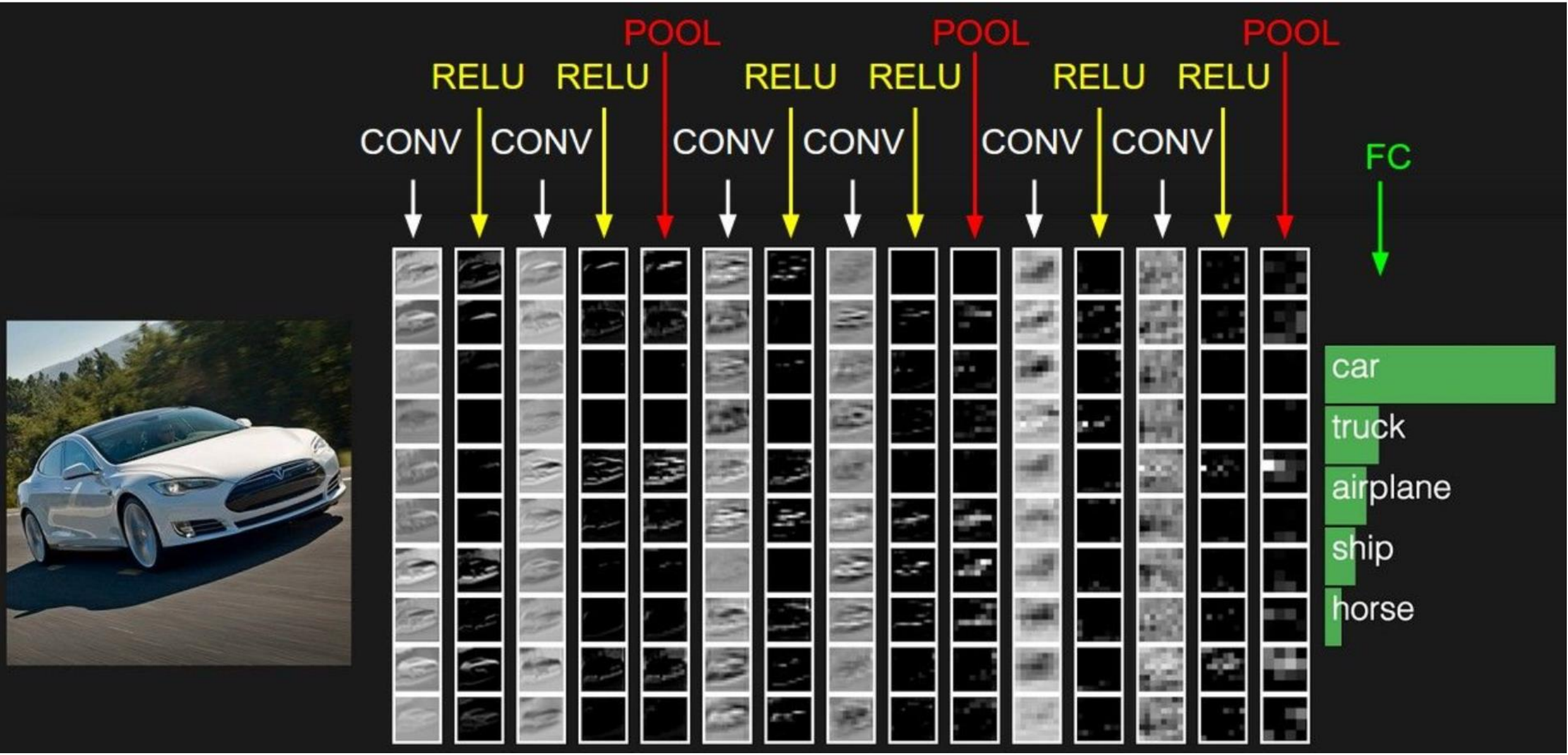


example 5x5 filters
(32 total)

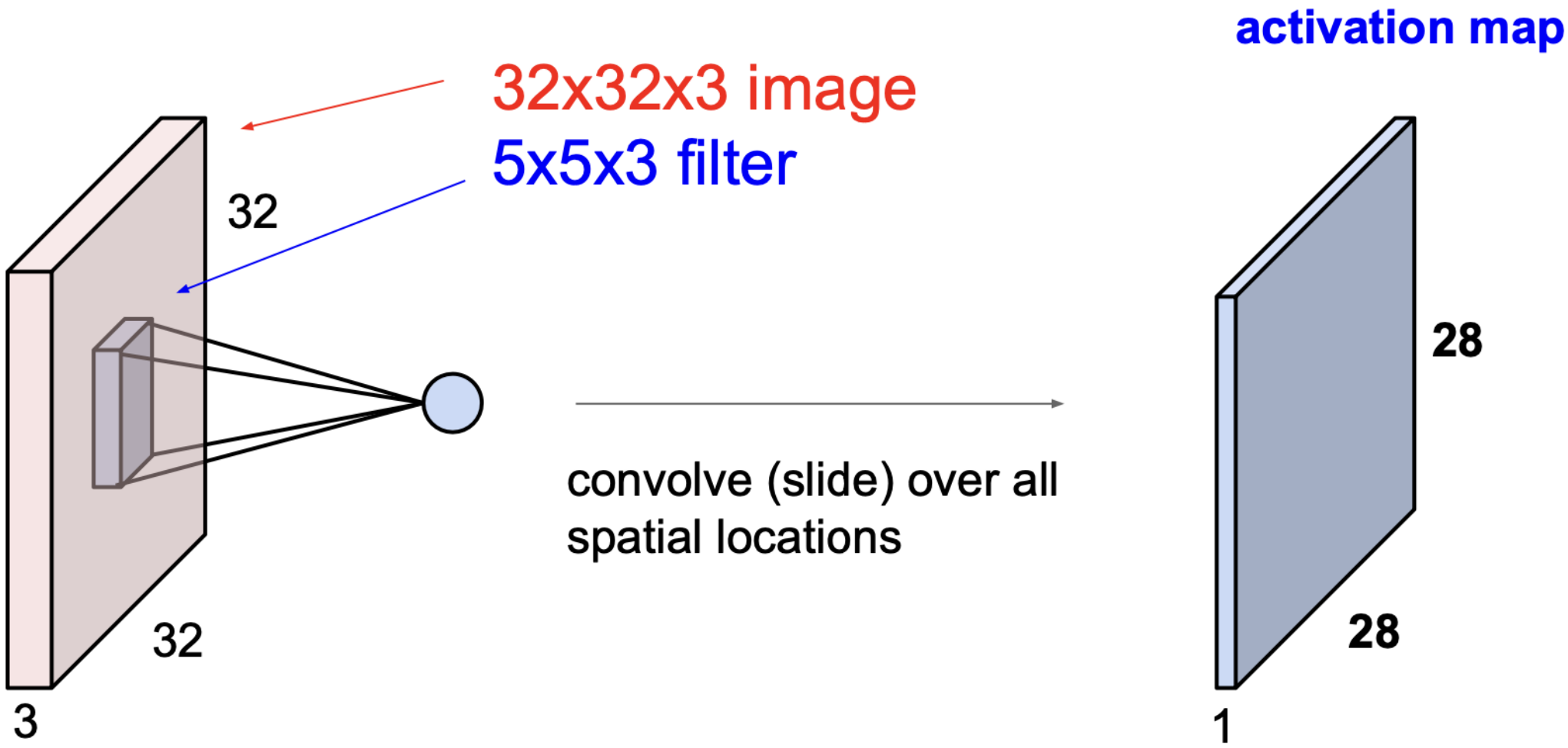
We call the layer convolutional because it is related to convolution of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

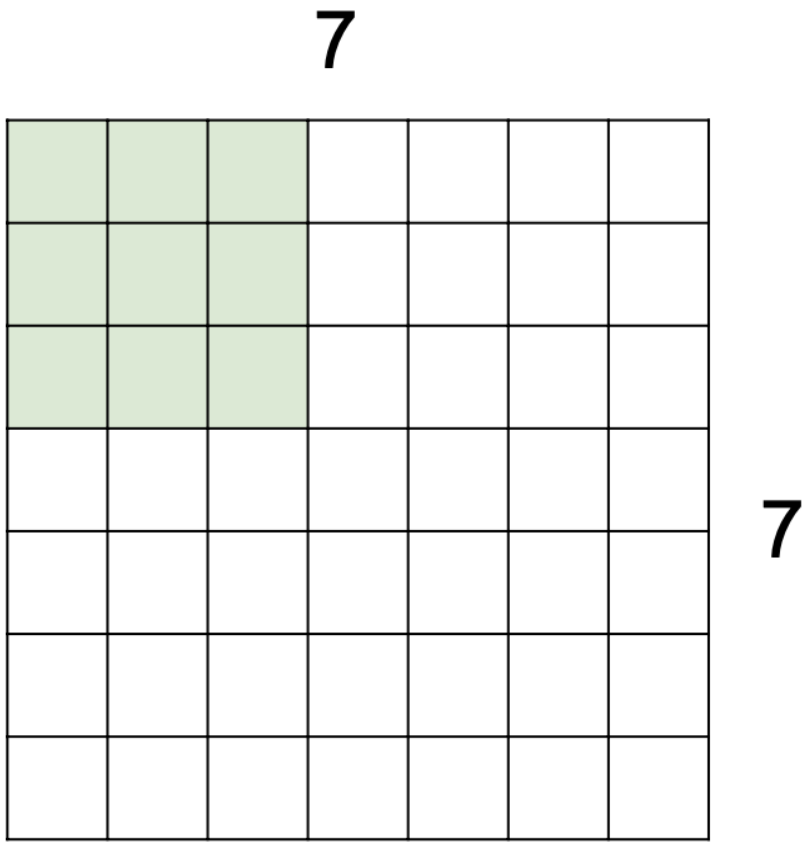
elementwise multiplication and sum of a filter and the signal (image)


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

A closer look at spatial dimensions:

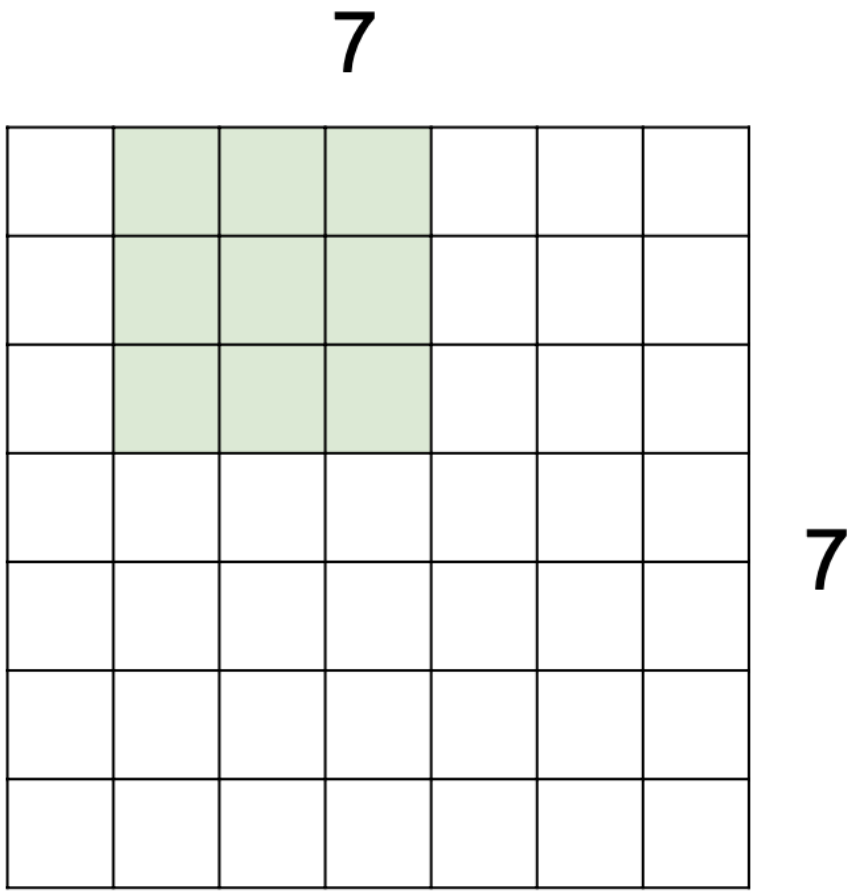


A closer look at spatial dimensions:



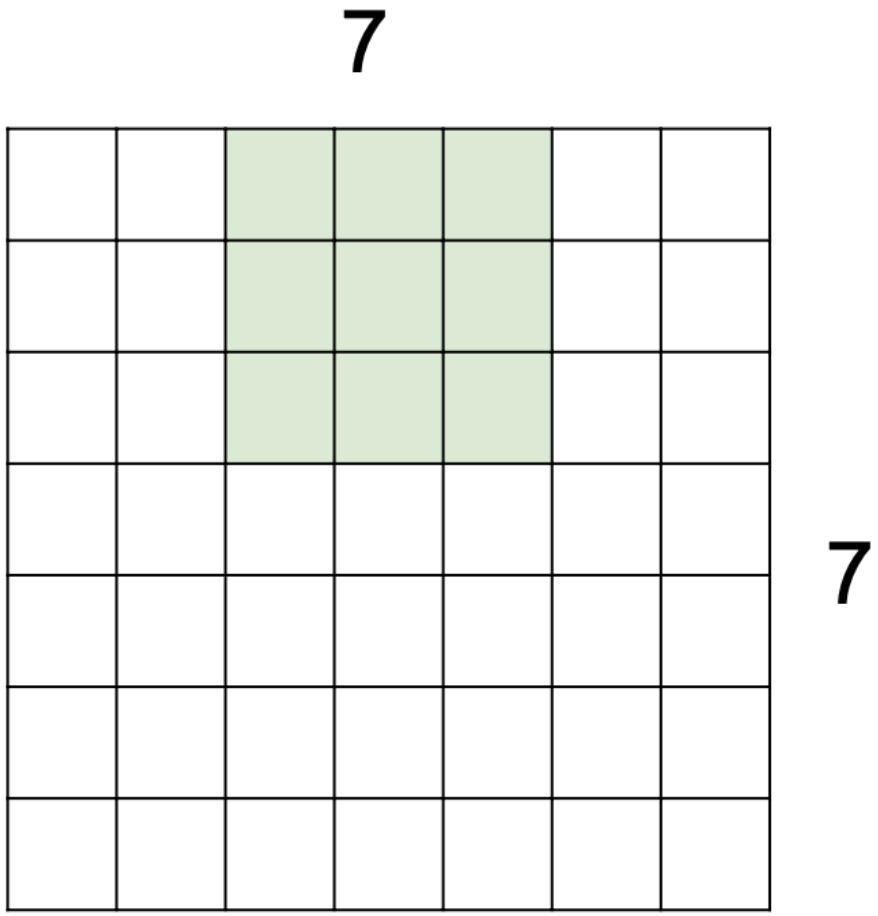
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



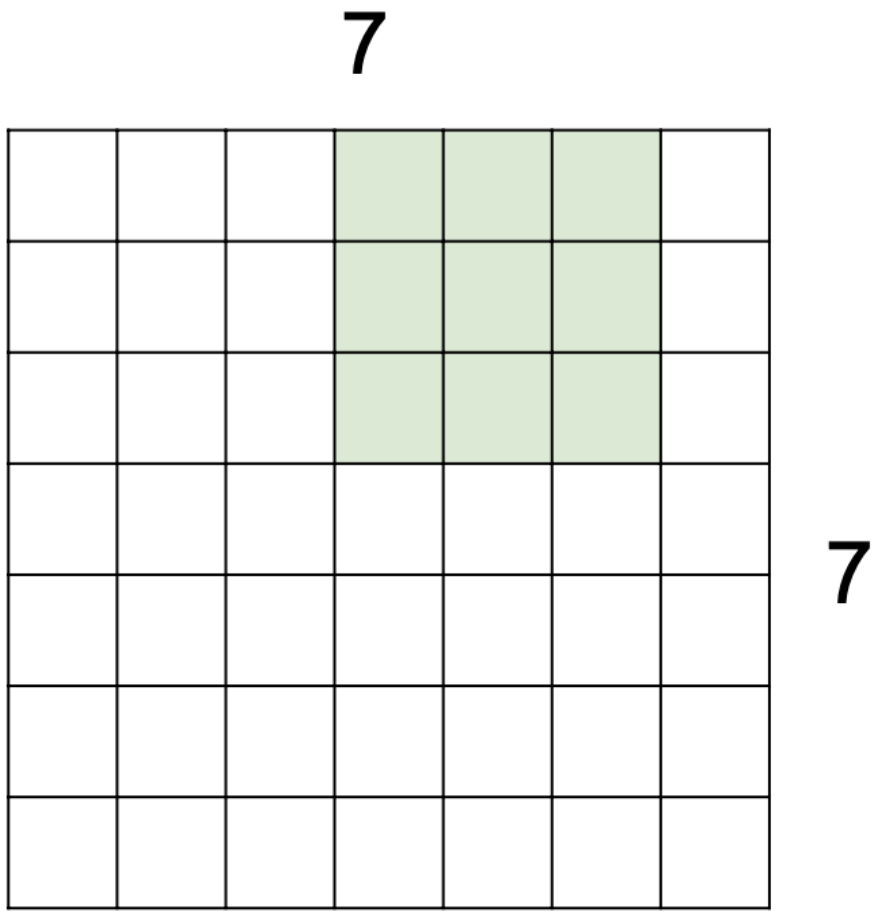
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



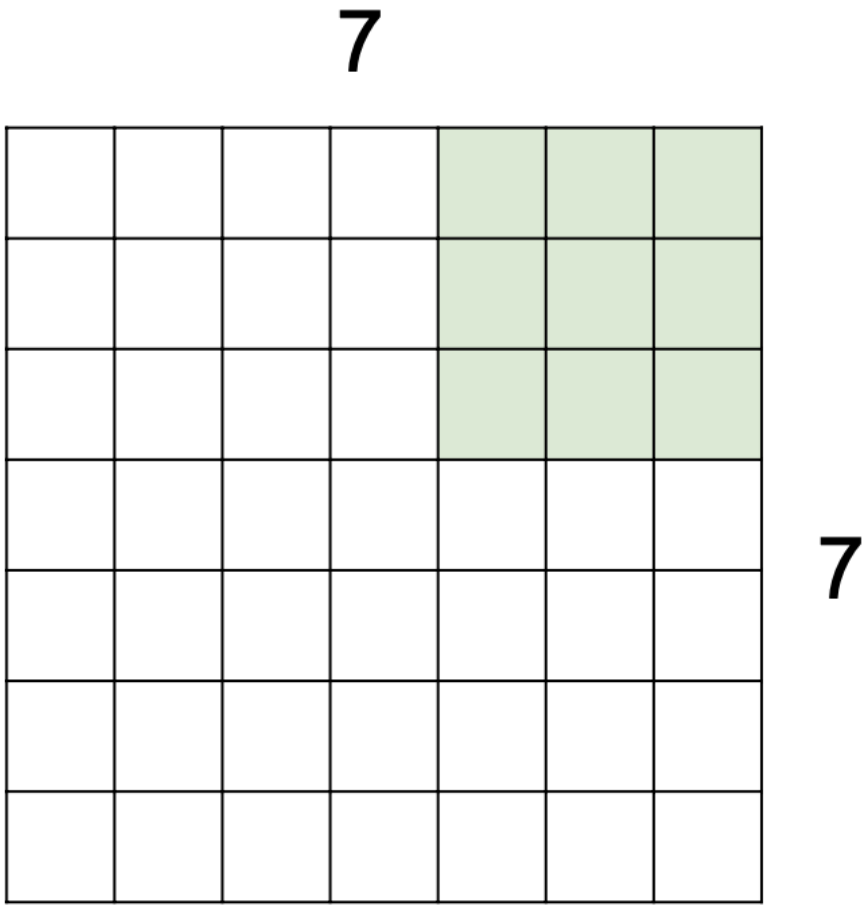
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

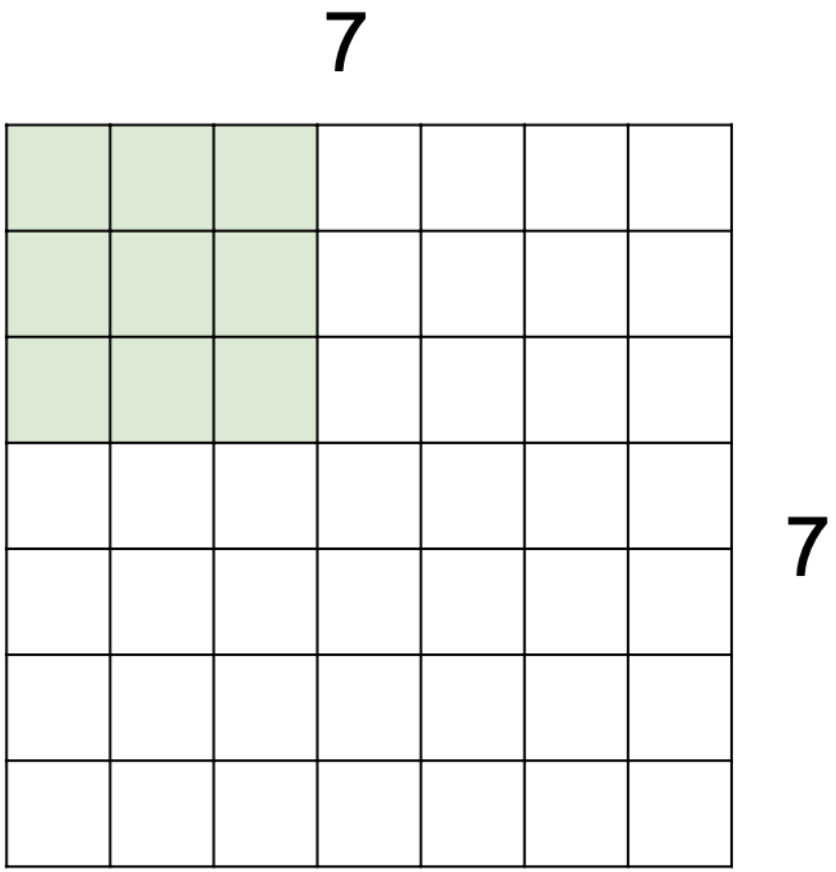
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

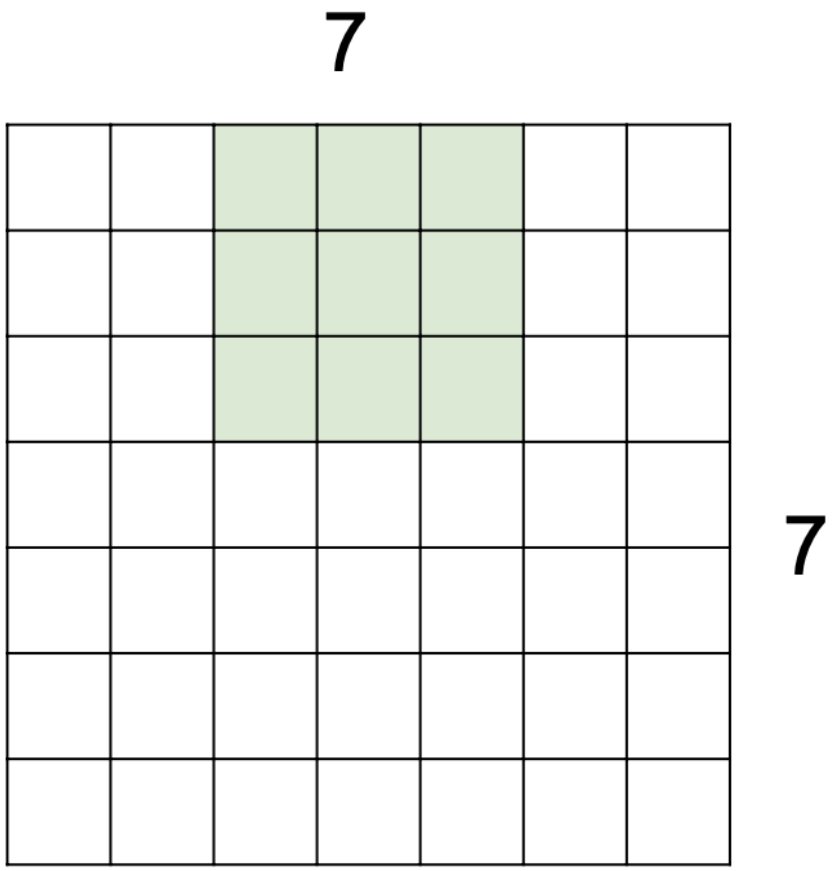
=> 5x5 output

A closer look at spatial dimensions:



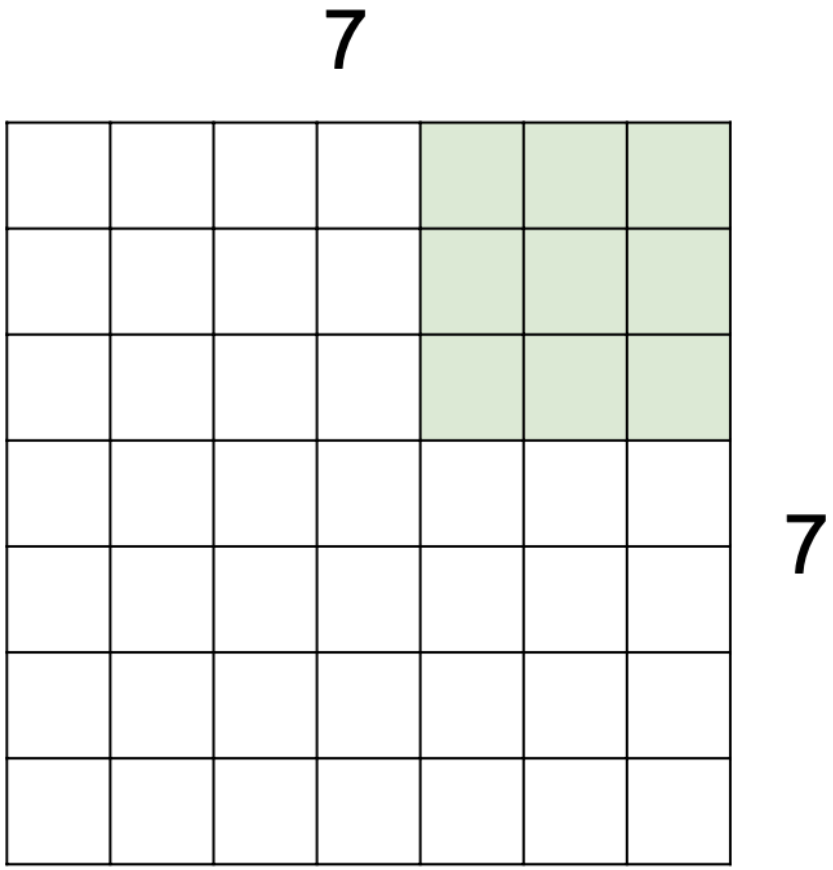
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



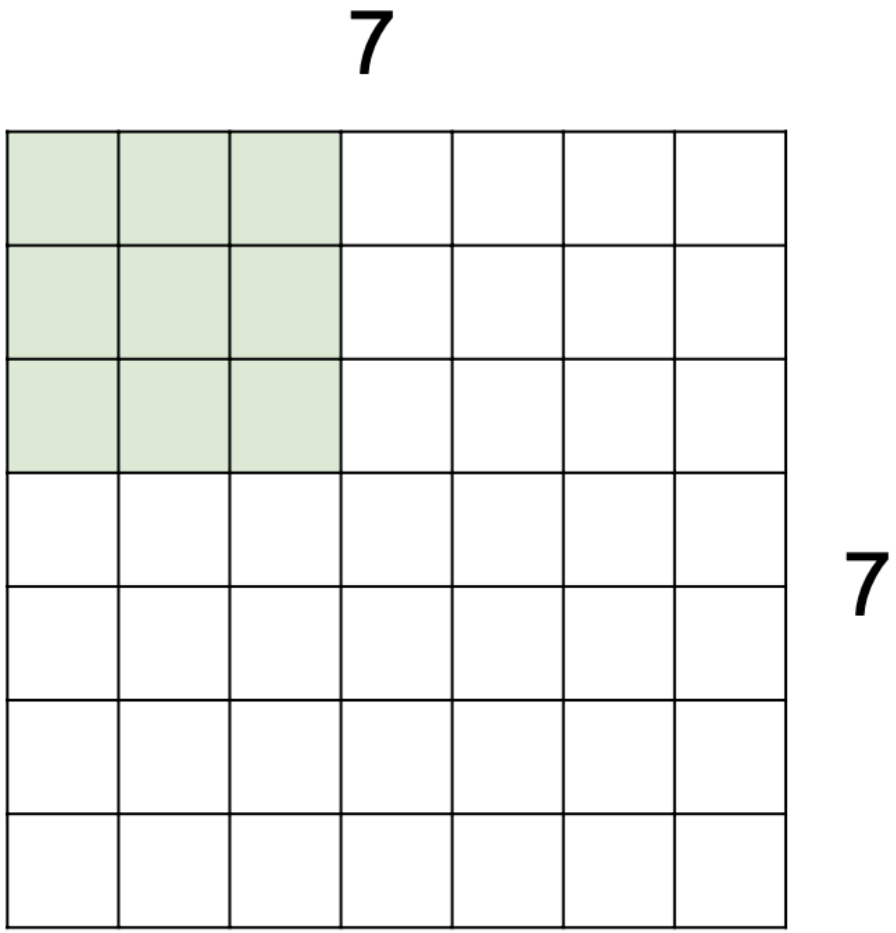
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



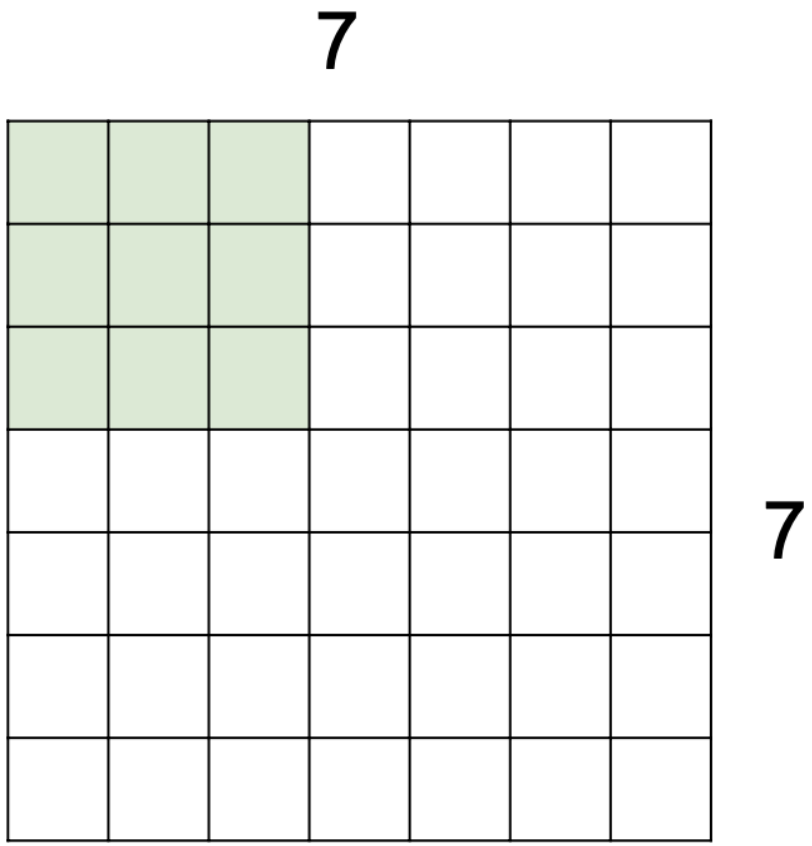
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



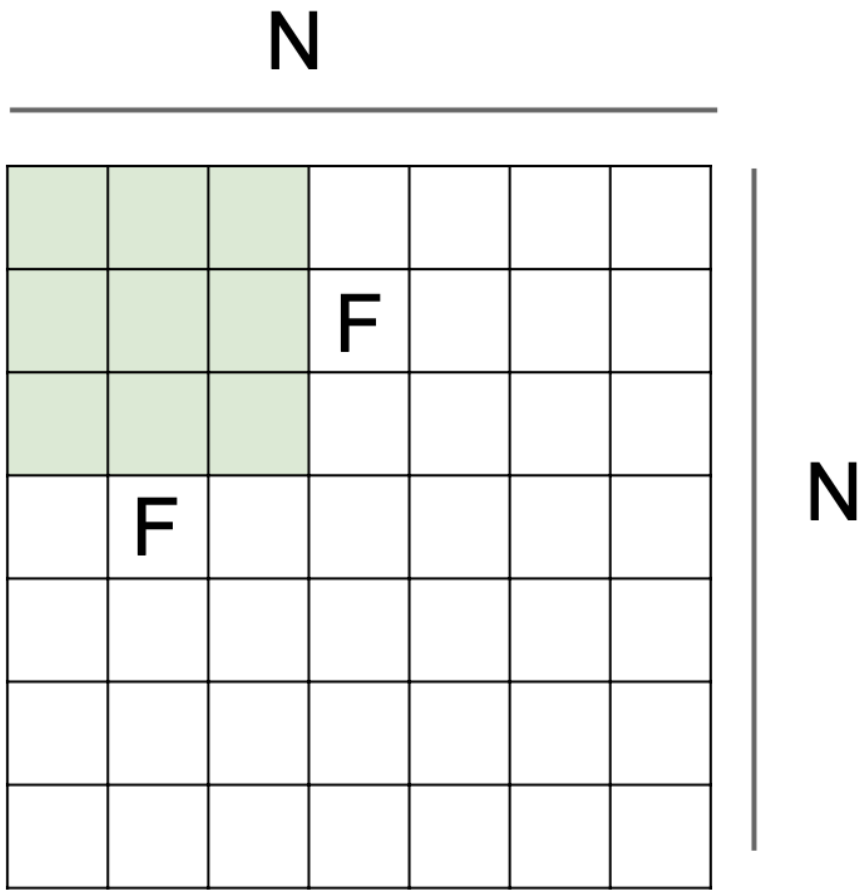
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

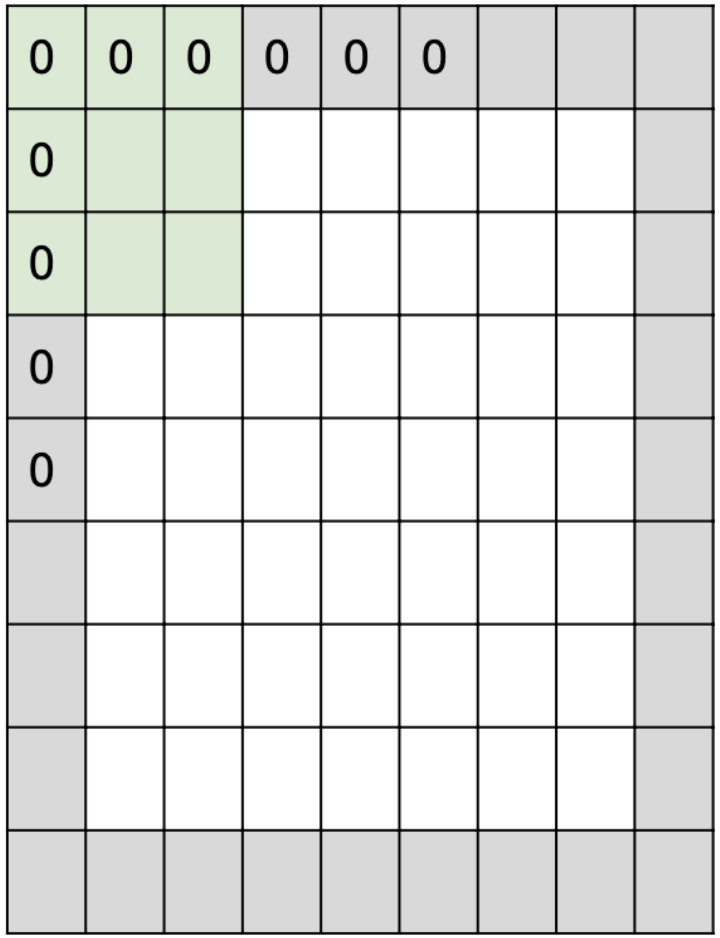
doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

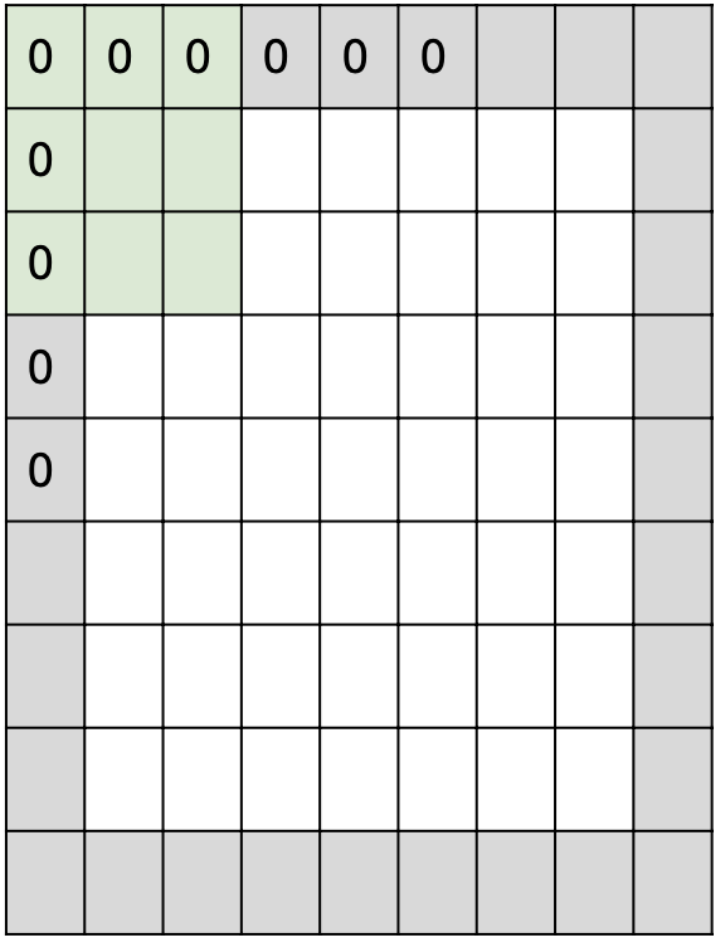
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

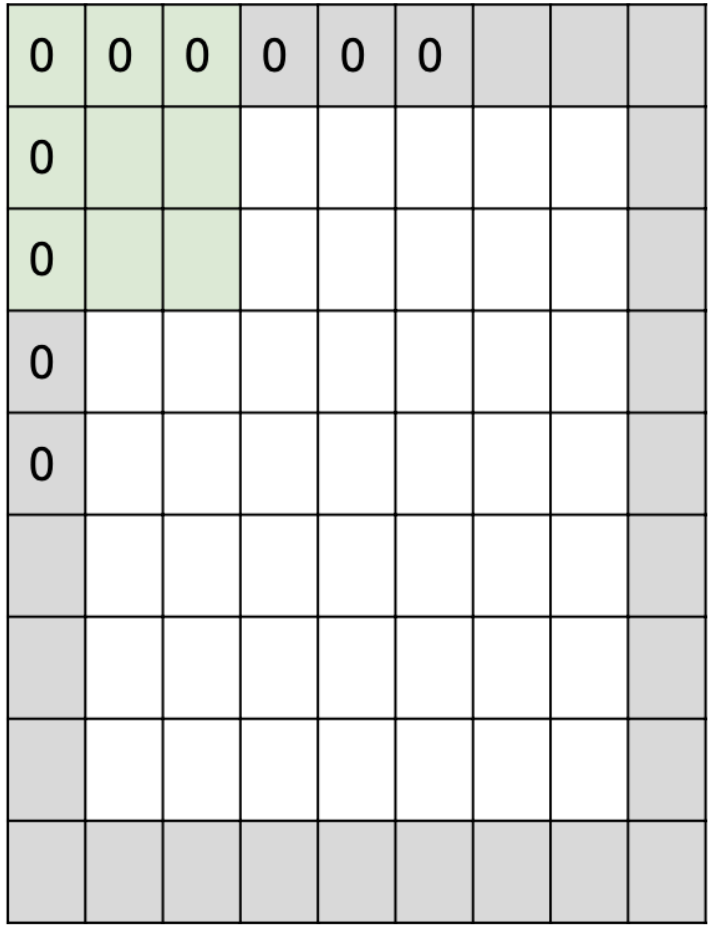
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

In practice: Common to zero pad the border

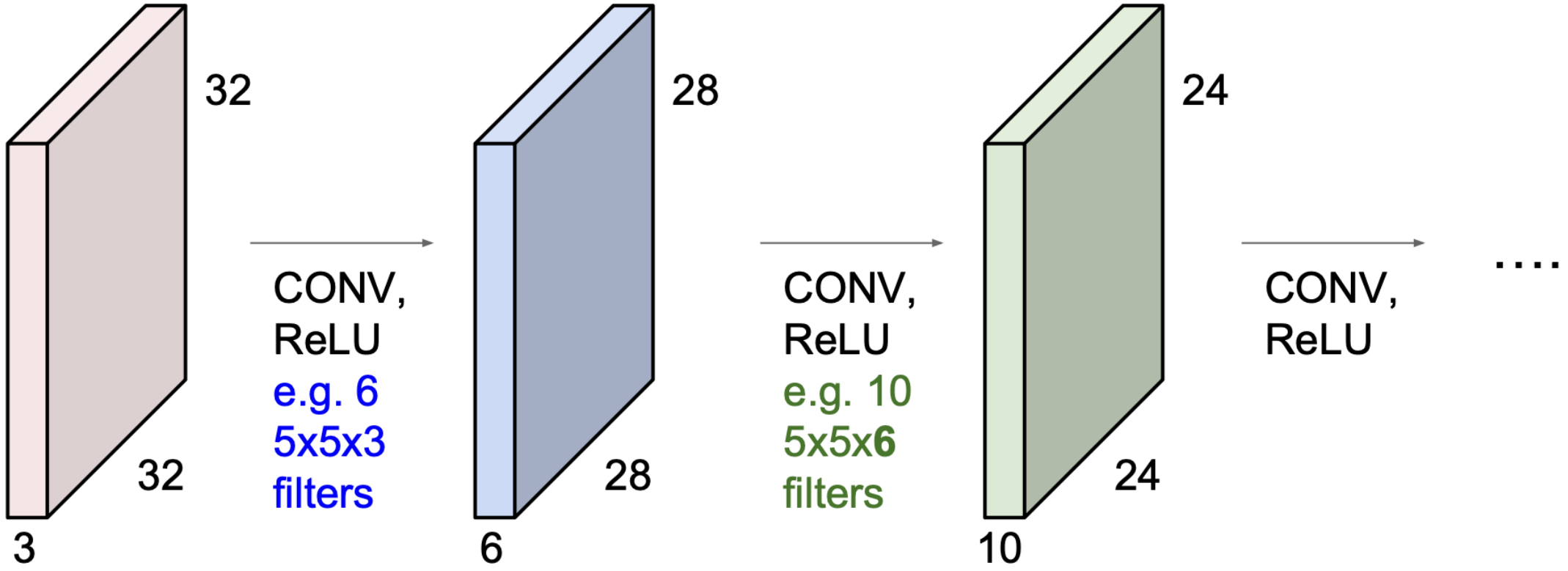


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

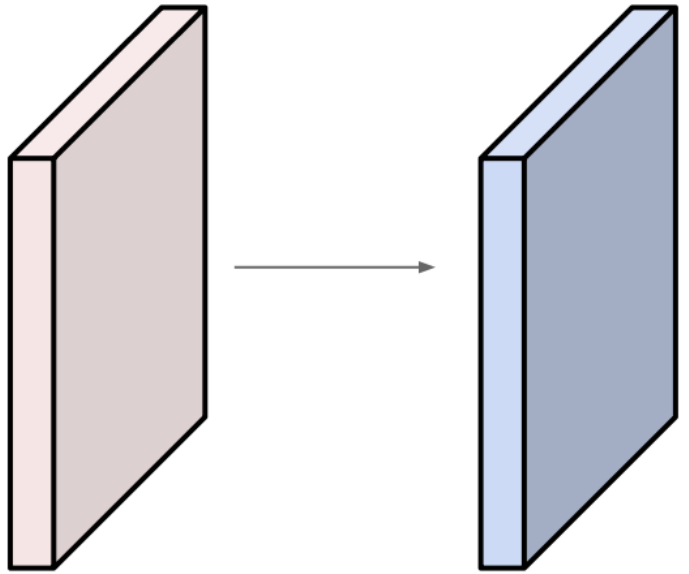
7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Remember back to…

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.



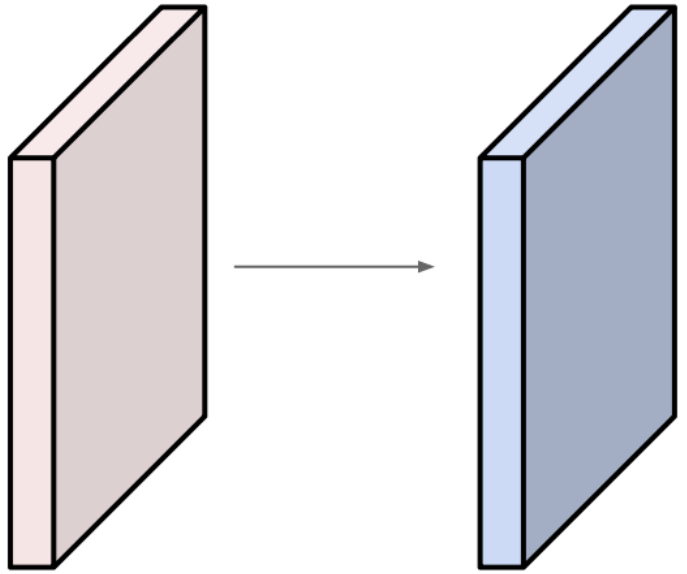
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?

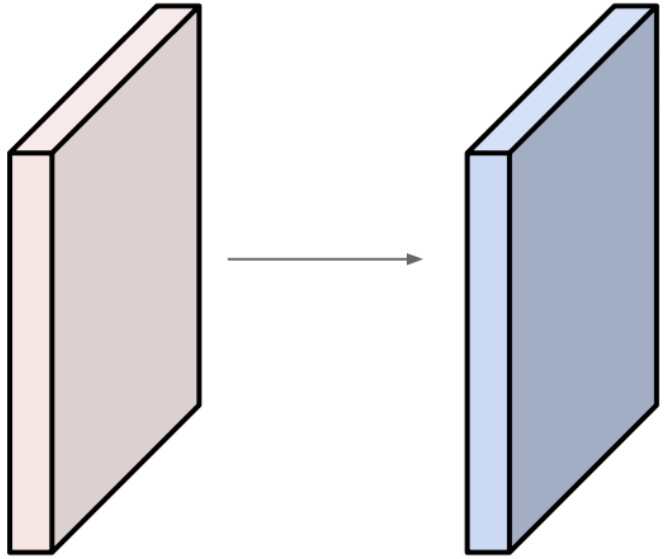
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

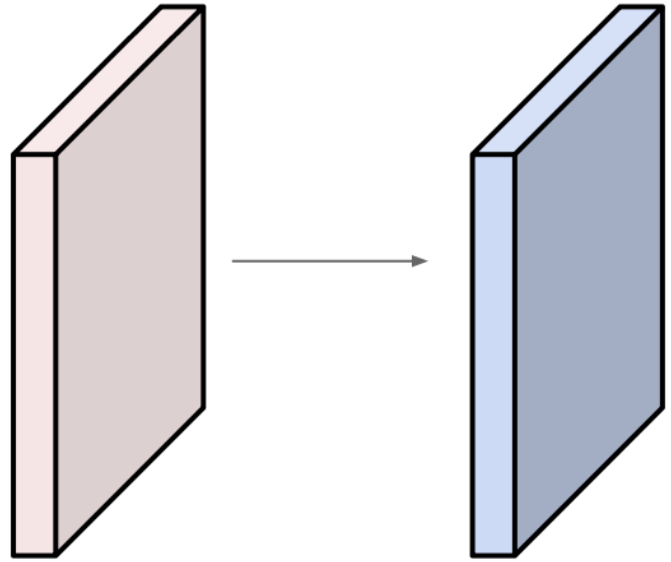
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Common settings:

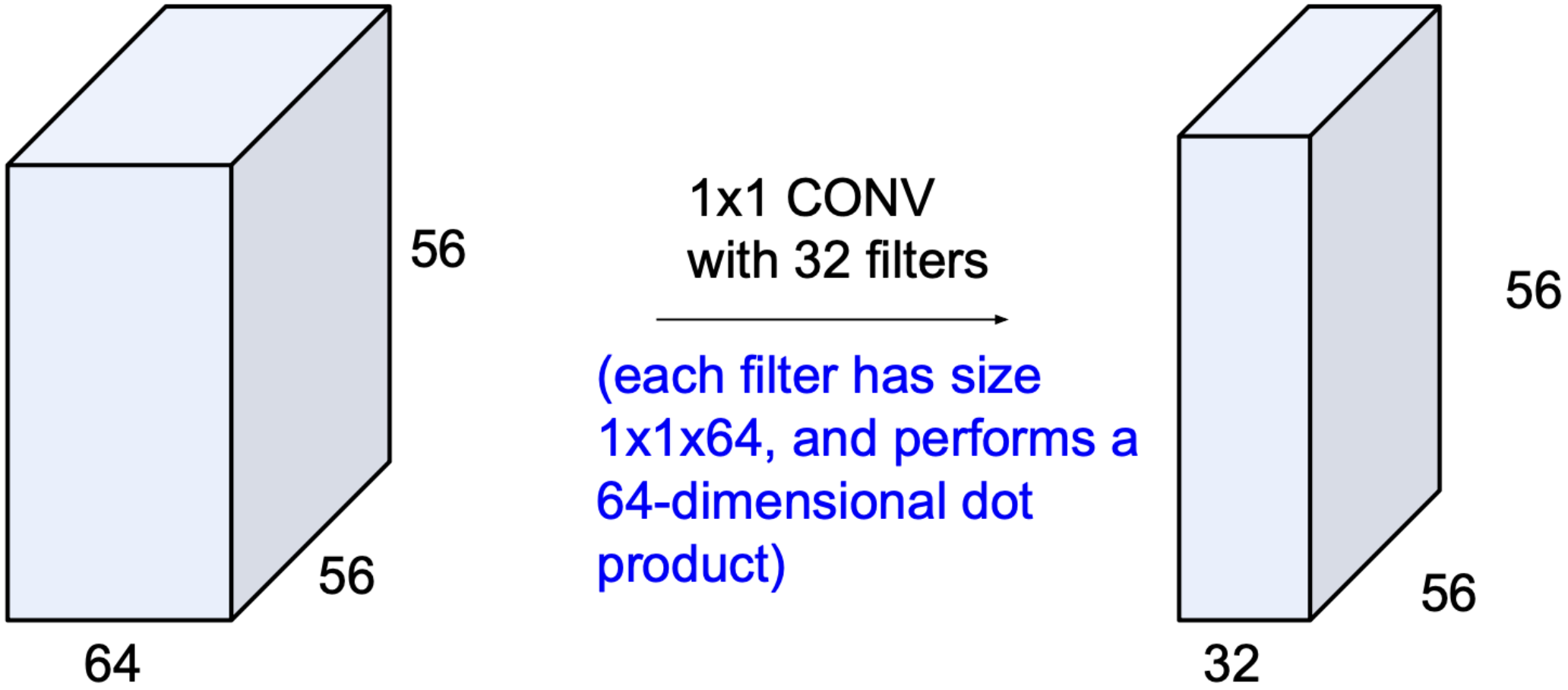
K = (powers of 2, e.g. 32, 64, 128, 512)

- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

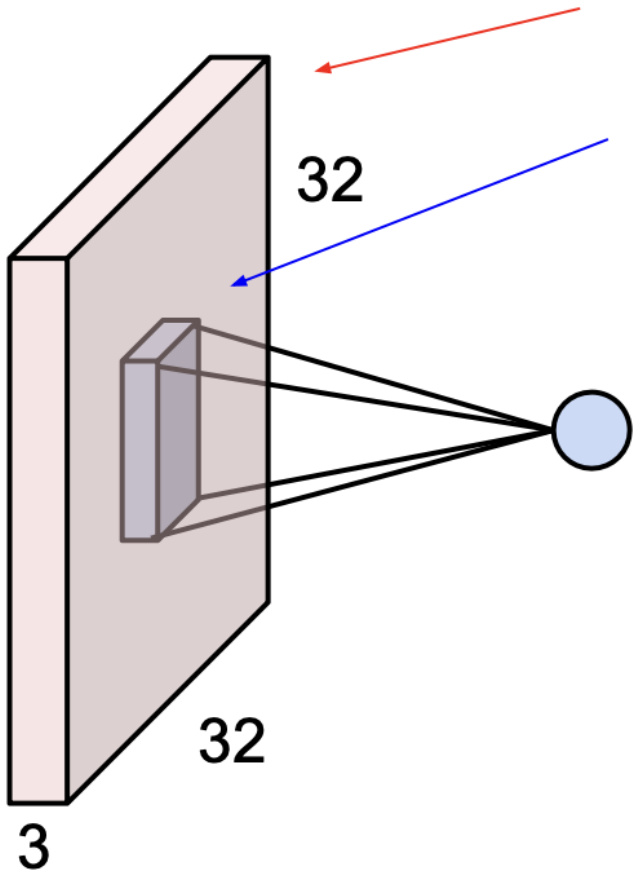
Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

(btw, 1x1 convolution layers make perfect sense)



The brain/neuron view of CONV Layer

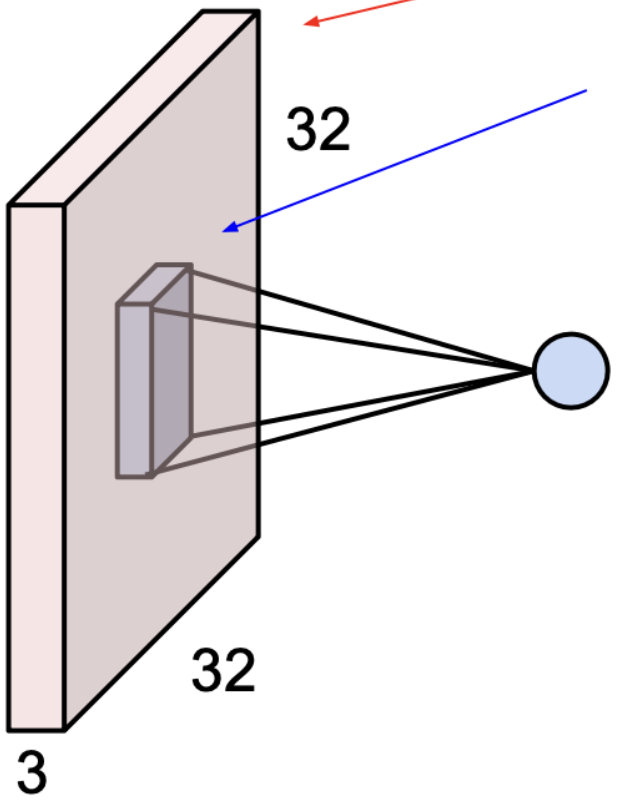


32x32x3 image

5x5x3 filter

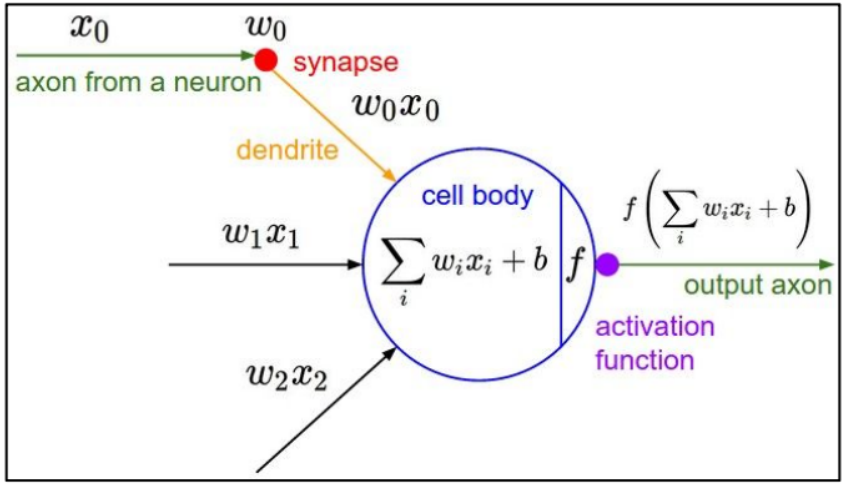
1 number:
the result of taking a dot product between the filter and this part of the image
(i.e. 5*5*3 = 75-dimensional dot product)

The brain/neuron view of CONV Layer



32x32x3 image

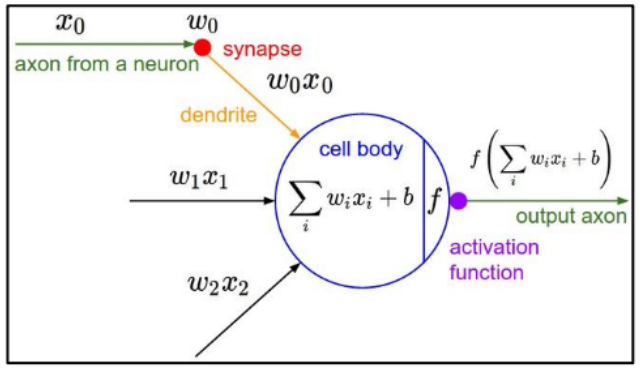
5x5x3 filter

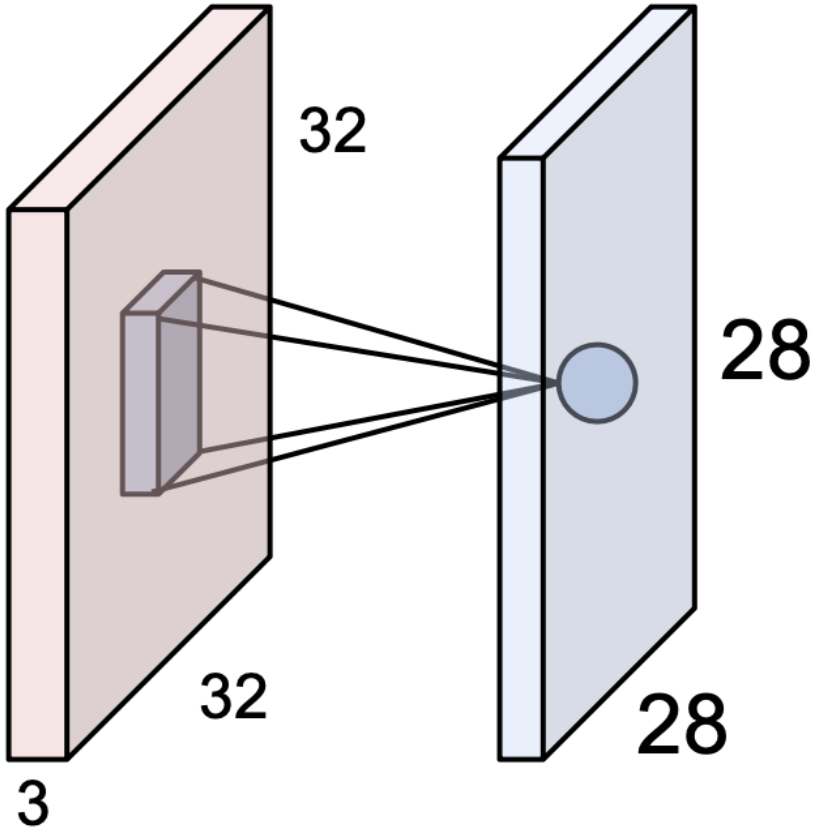


It's just a neuron with local connectivity...

1 number:
the result of taking a dot product between the filter and this part of the image
(i.e. 5*5*3 = 75-dimensional dot product)

The brain/neuron view of CONV Layer



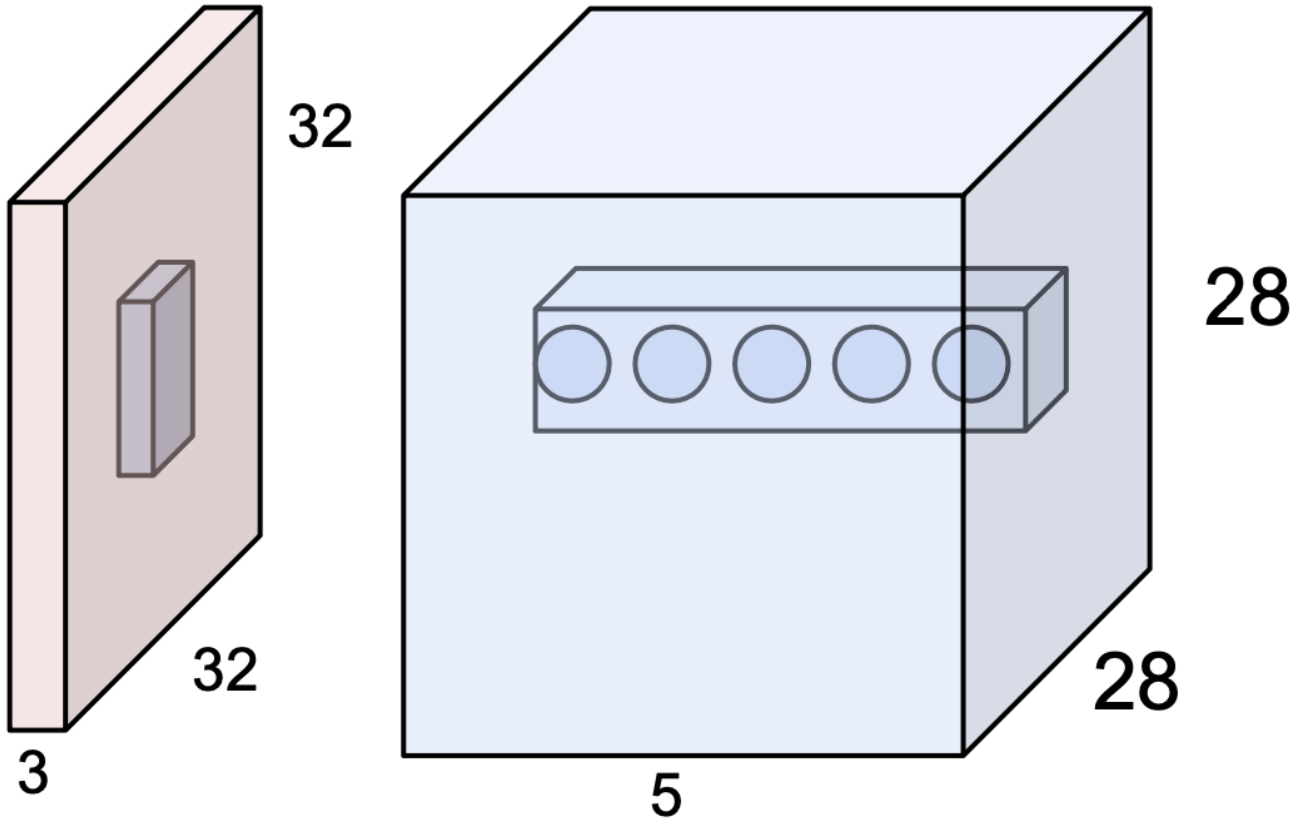


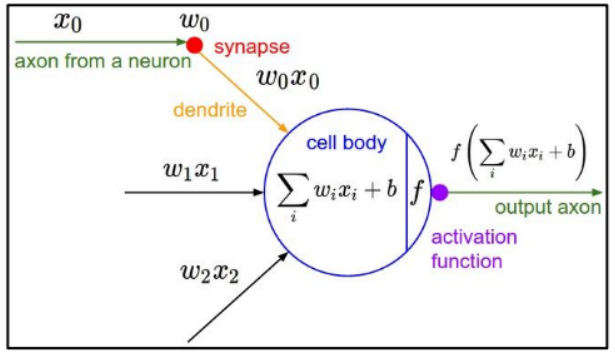
An activation map is a 28x28 sheet of neuron outputs:

1. Each is connected to a small region in the input
2. All of them share parameters

"5x5 filter" -> "5x5 receptive field for each neuron"

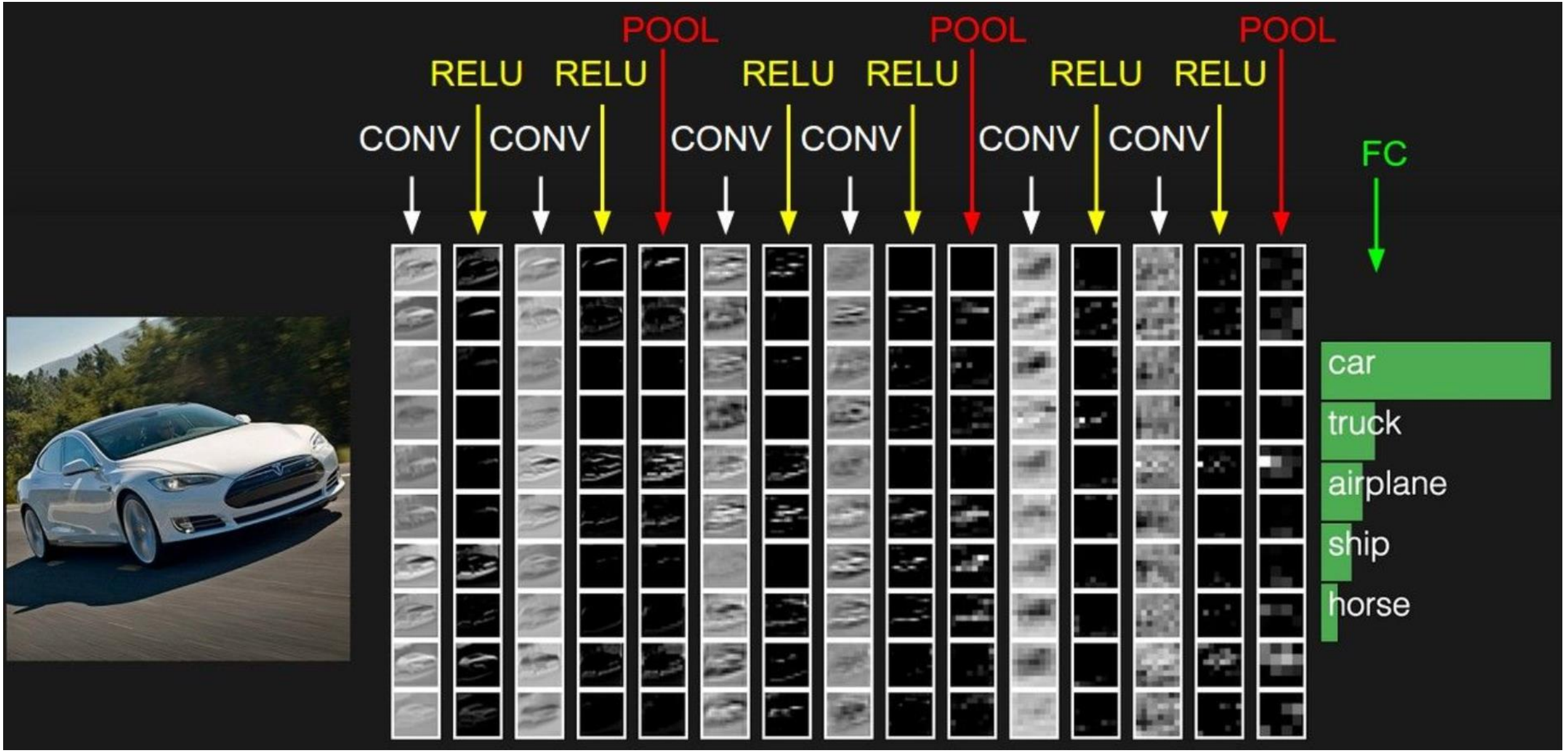
The brain/neuron view of CONV Layer





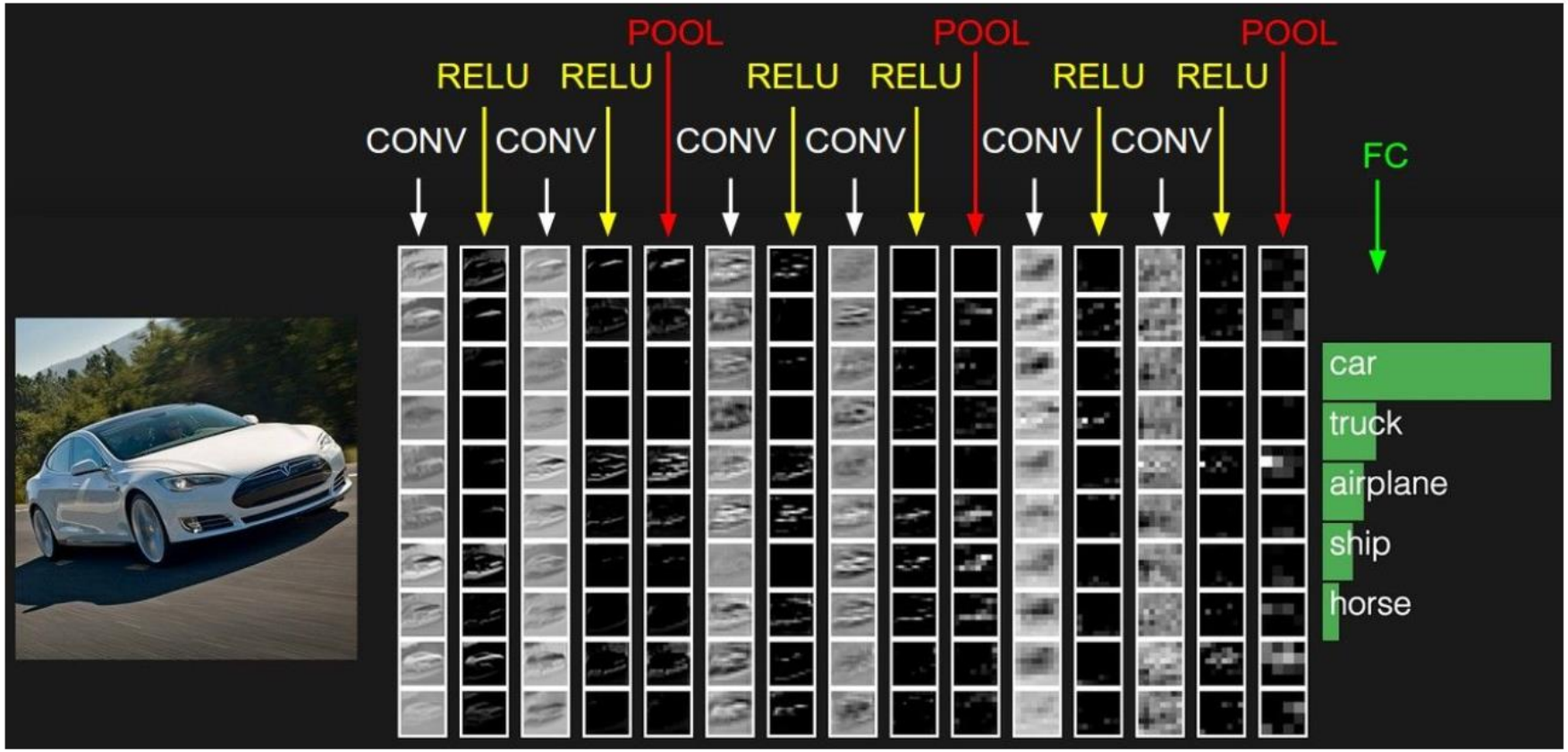
E.g. with 5 filters,
CONV layer consists of neurons arranged in a 3D grid (28x28x5)

There will be 5 different neurons all looking at the same region in the input volume


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

Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



LeNet-5

- *Gradient Based Learning Applied To Document Recognition - Y. Lecun, L. Bottou, Y. Bengio, P. Haffner; 1998*
- Helped establish how we use CNNs today
- Replaced manual feature extraction

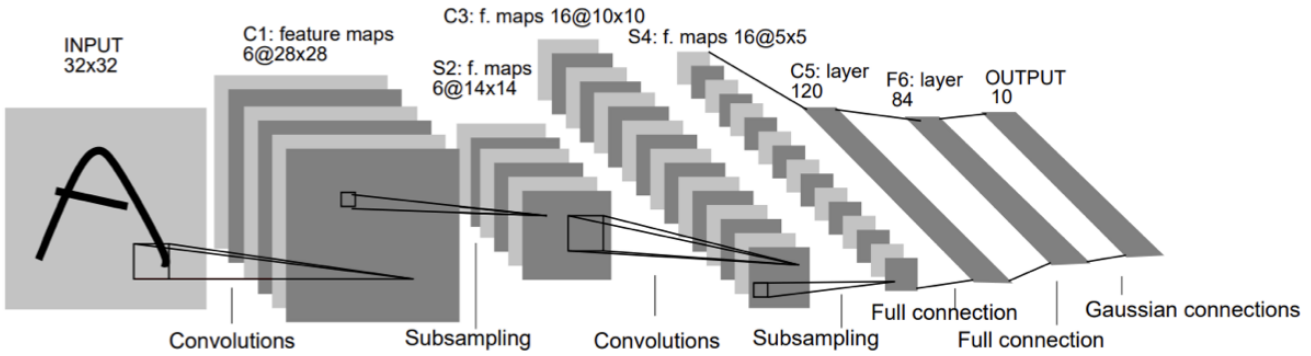
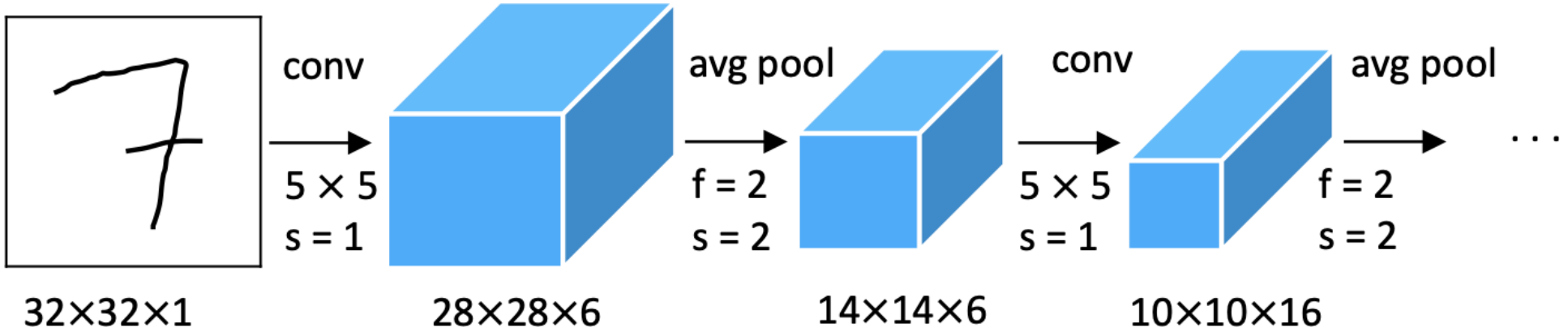
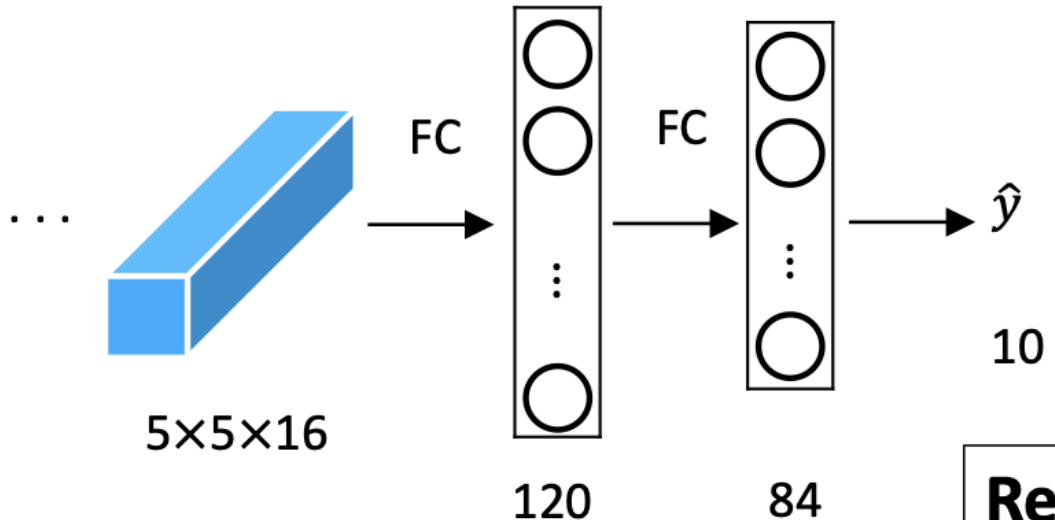


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

[LeCun et al., 1998]

LeNet-5





Reminder:

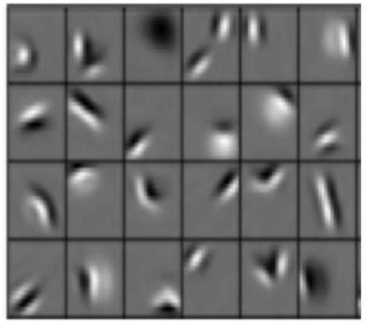
Key idea:

learn hierarchy of features

directly from the data

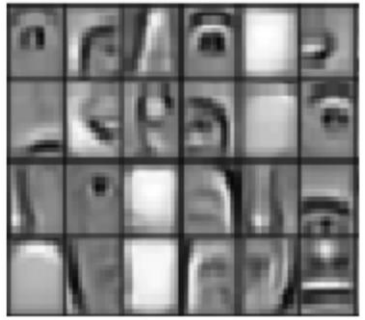
(rather than hand-engineering them)

Low level features



Edges, dark spots

Mid level features



Eyes, ears, nose

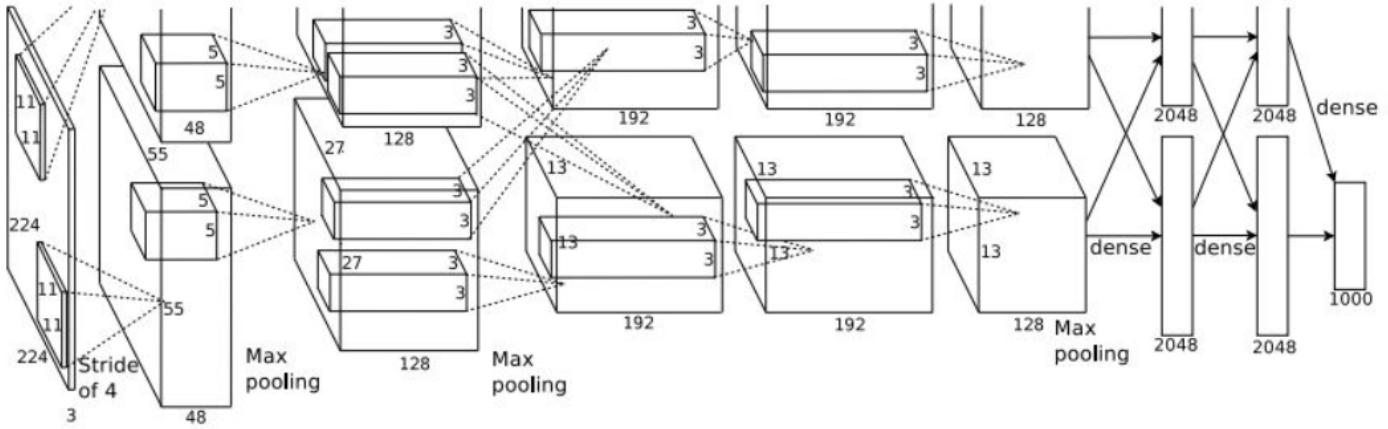
High level features

Facial structure

Lee+ *ICML* 2009

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%

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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

best model

11.2% top 5 error in ILSVRC 2013
->
7.3% top 5 error

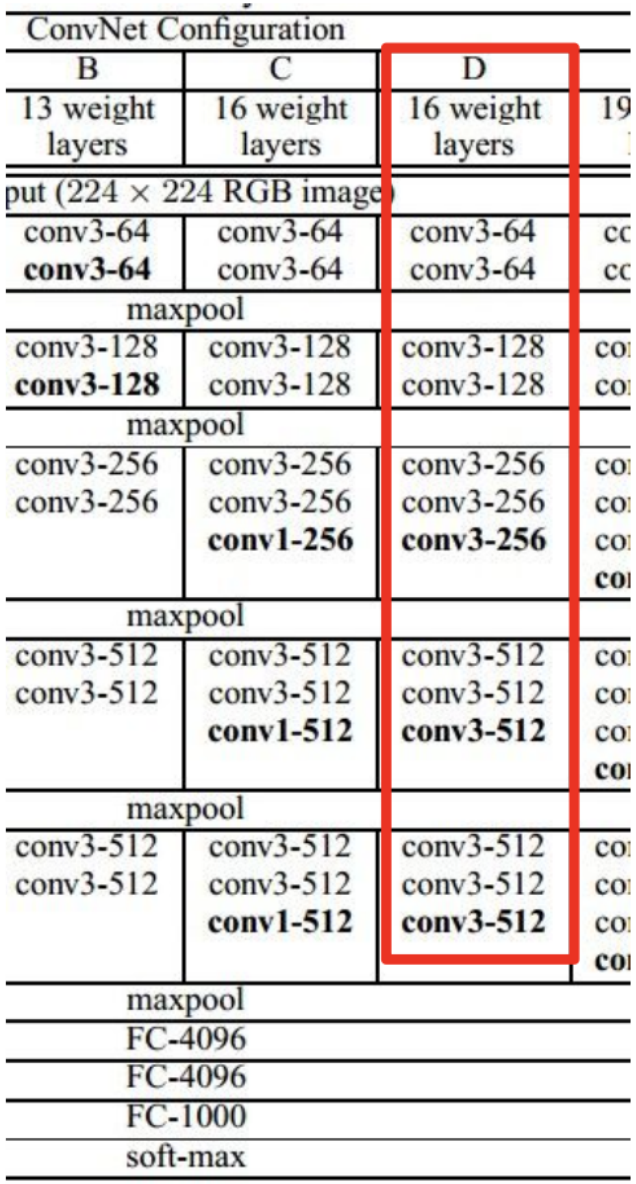
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 **conv1-256**	conv3-256 **conv3-256**	conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

(not counting biases)

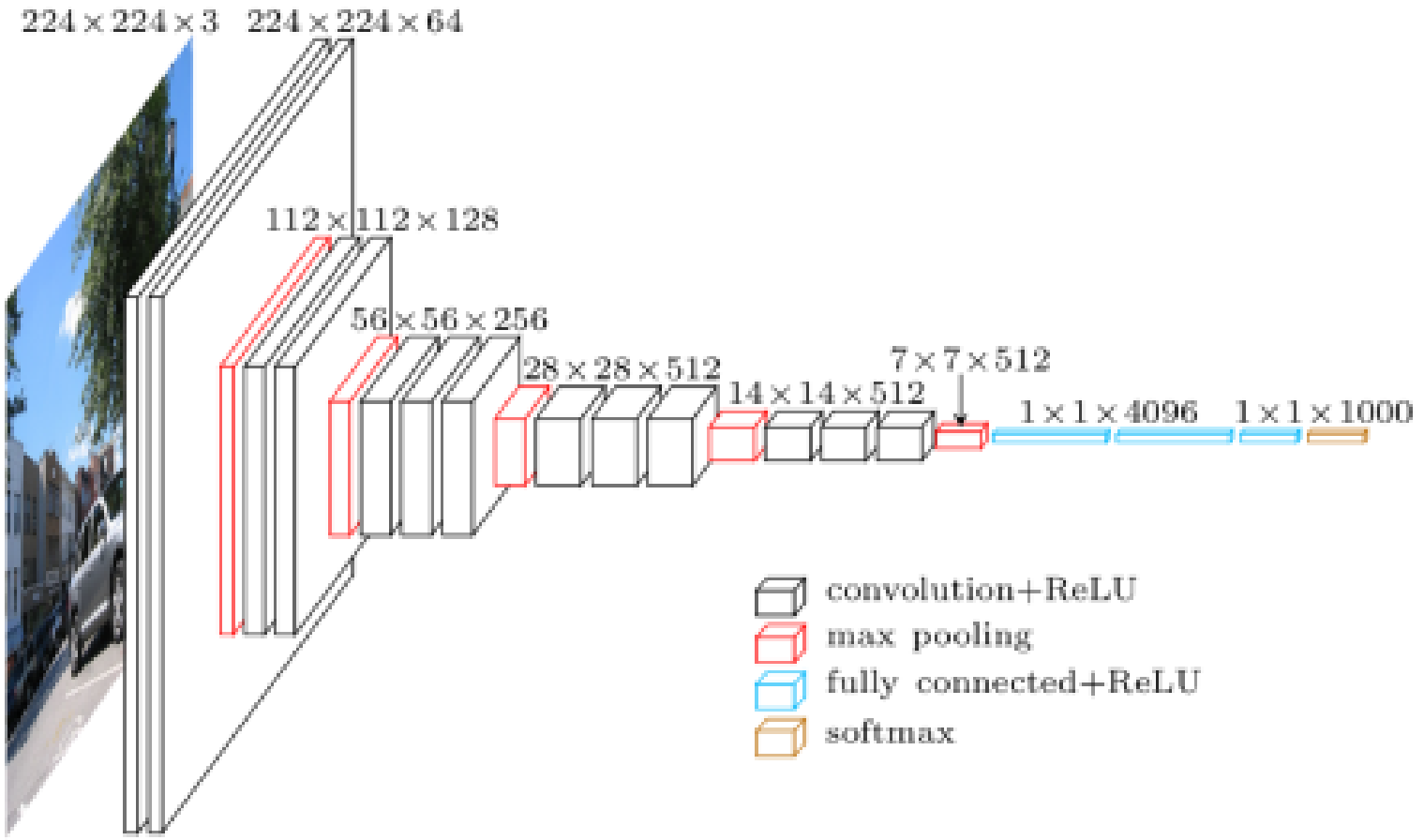
INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M 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



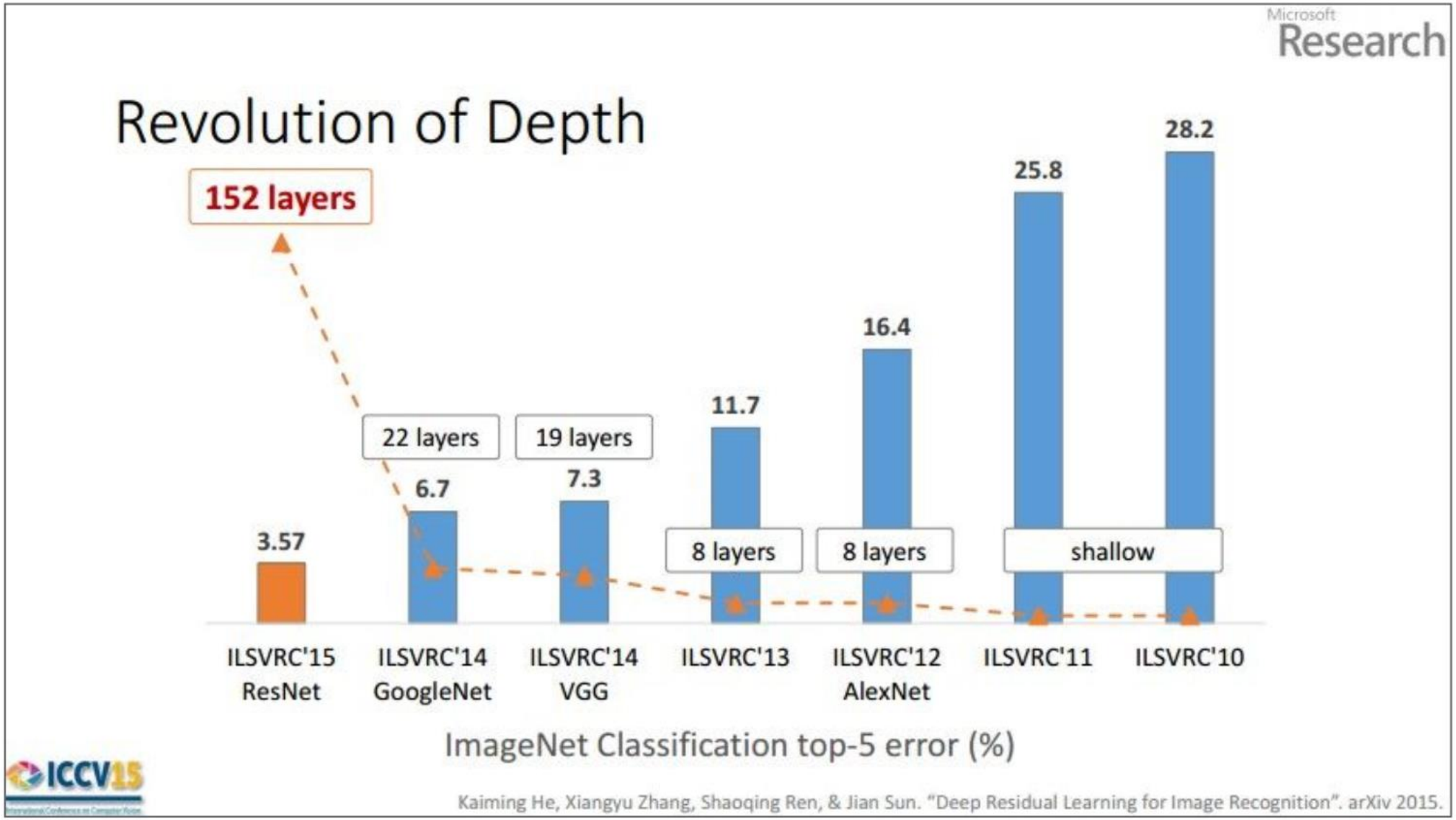
ConvNet Configuration			
B	C	D	
13 weight layers	16 weight layers	16 weight layers	19
put (224 × 224 RGB image)			
conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	cc cc
maxpool			
conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	co co
maxpool			
conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	co co co **co**
maxpool			
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	co co co **co**
maxpool			
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	co co co **co**
maxpool			
FC-4096			
FC-4096			
FC-1000			
soft-max			

VGGNet16

- Replaces layers with large kernels with stacked $(3,3)$ kernels. This uses fewer parameters but produces the same receptive field.
- Uses pooling layers instead of strides, again reducing the number of parameters.
- Even with these savings, VGGNet16 is a BIG model—almost 600MB.

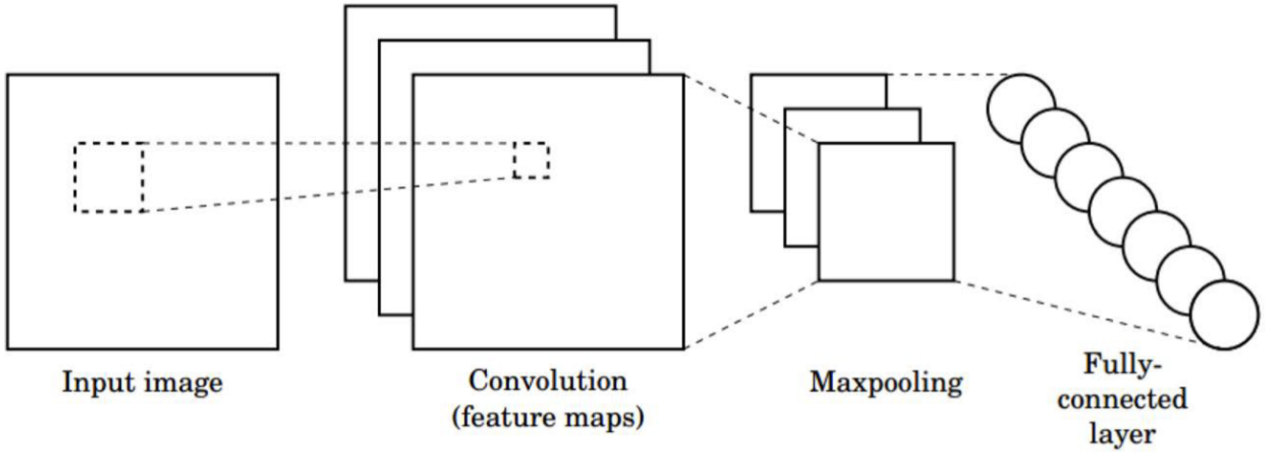


https://arxiv.org/abs/1409.1556v6



(slide from Kaiming He's recent presentation)

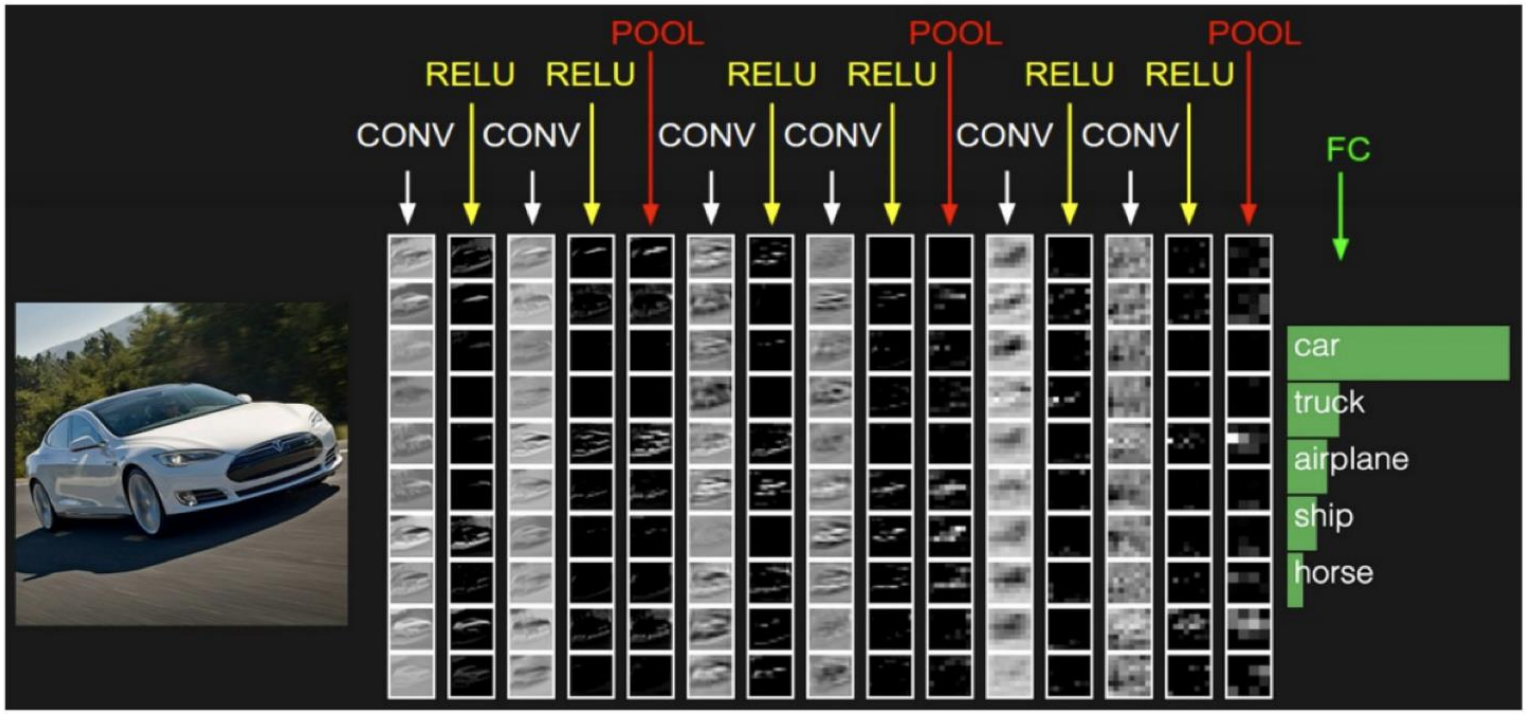
CNNs for Classification



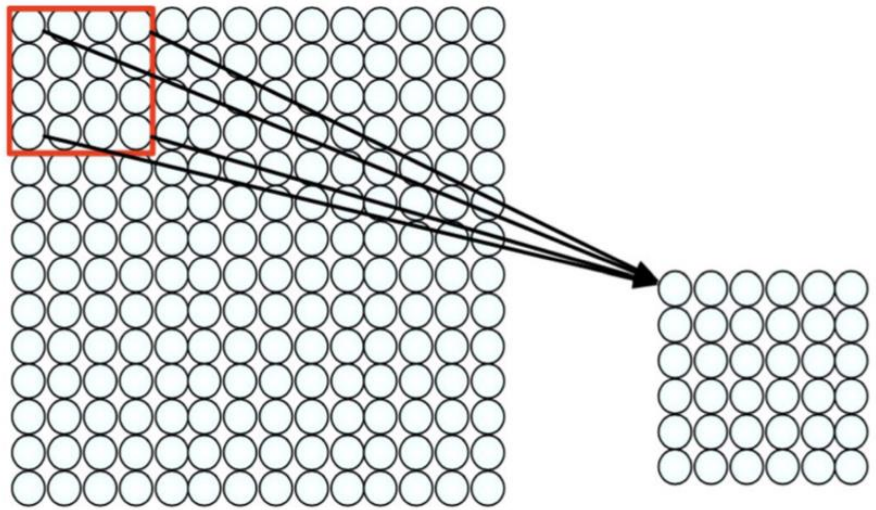
1. **Convolution**: Apply filters to generate feature maps.
2. **Non-linearity**: Often ReLU.
3. **Pooling**: Downsampling operation on each feature map.

Train model with image data.
Learn weights of filters in convolutional layers.

Example – Six convolutional layers



Convolutional Layers: Local Connectivity



`tf.keras.layers.Conv2D`

For a neuron in hidden layer:

- Take inputs from patch
- Compute weighted sum
- Apply bias

4x4 filter: matrix of weights w_{ij}

$$\sum_{i=1}^{4}\sum_{j=1}^{4} w_{ij}\, x_{i+p,j+q} + b$$

for neuron (p,q) in hidden layer

1) applying a window of weights
2) computing linear combinations
3) activating with non-linear function

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                        input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

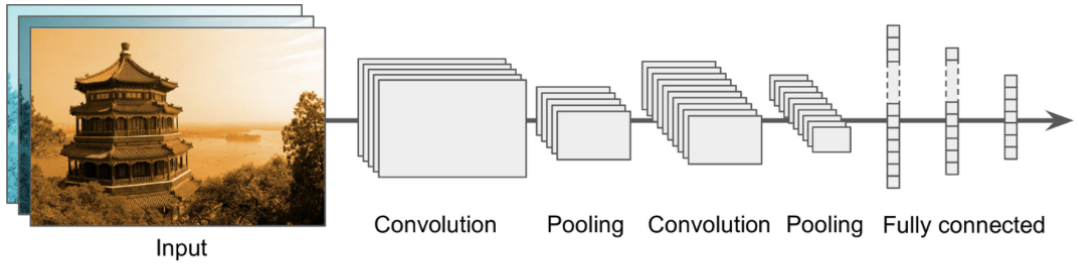
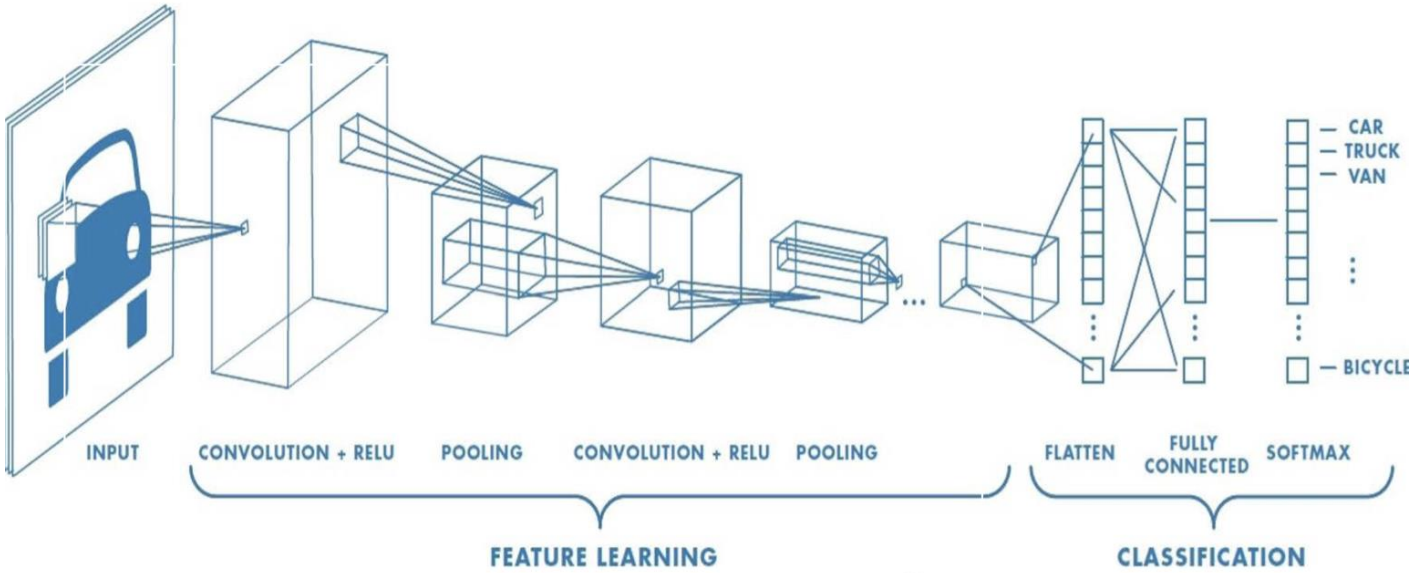


Figure 14-11. Typical CNN architecture

CNNs for Classification: Class Probabilities

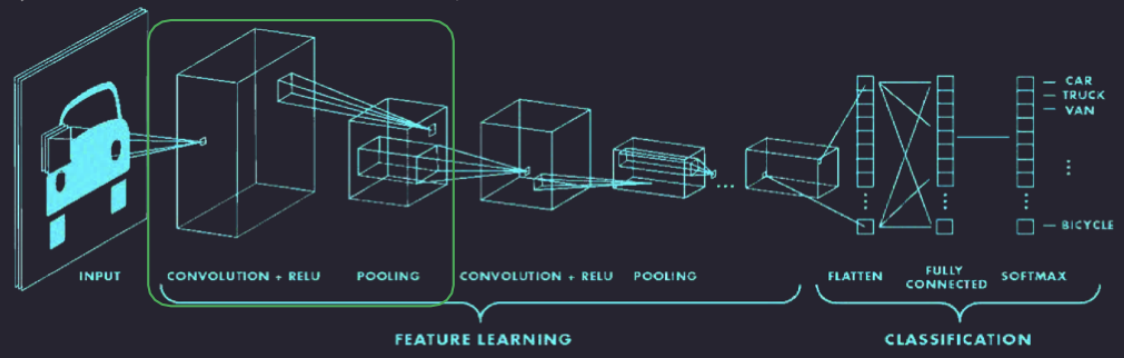


$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as **probability** of image belonging to a particular class

Putting it all together

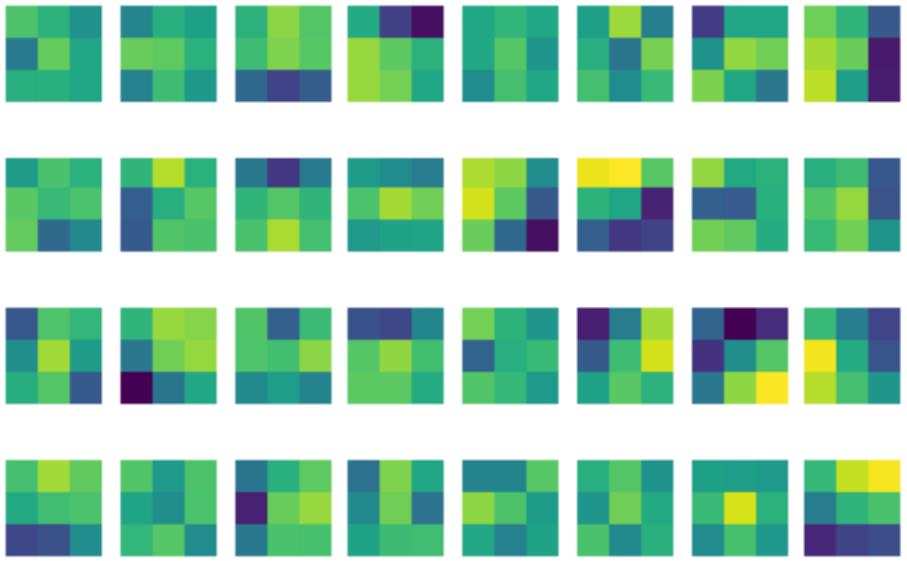
```
import tensorflow as tf

def generate_model():
    model = tf.keras.Sequential([
        # first convolutional layer
        tf.keras.layers.Conv2D(32, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

        # second convolutional layer
        tf.keras.layers.Conv2D(64, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

      # fully connected classifier
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(1024, activation='relu'),
      tf.keras.layers.Dense(10, activation='softmax')
      # 10 outputs

    ])
    return model
```



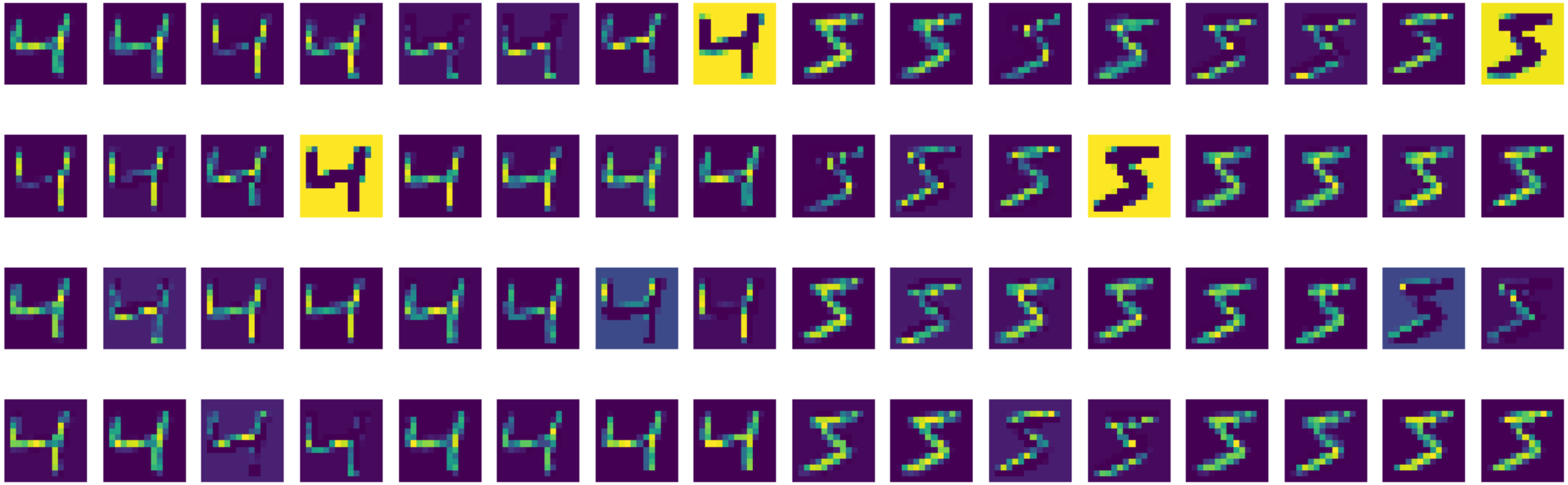
Convolutional Filters

The filters in a convolutional layer are learned from a random initialisation

The filters job is to pick out details in the image, so they learn to do things like edge detection, mapping transitions between colours, identifying areas of high or low contrast



Convolutional Filters



Filter Shapes

The shapes of filters can be somewhat misleading—if we think about a layer with 32 filters and a (3,3) kernel, we might think that it has a shape of (3,3,32), like an image.

But it all depends on the input—each filter has the same number of channels as the input tensor

So the first Conv Layer might have as input a (28,28,1) image, and 32 (3,3) filters. So the shape of the filters is:

$$(3,3,1,32)$$

But the second layer, which has 64 filters, will have shapes like this:

$$(3,3,32,64)$$

Each of the 64 filters, has the shape $(3,3,32)$—one kernel for each input channel

Kernel sizes

Consider the number of parameters that are needed for a convolutional layer with (k, k) kernels, c_{in} input channels and c_{out} output channels:

$$n = k^2 * c_{in} * c_{out}$$

So a layer with $(5,5)$ kernels has $\frac{25}{9}$ more parameters than a layer with $(3,3)$ kernels

Kernel sizes

Consider the number of parameters that are needed for a convolutional layer with (k, k) kernels, c_{in} input channels and c_{out} output channels:

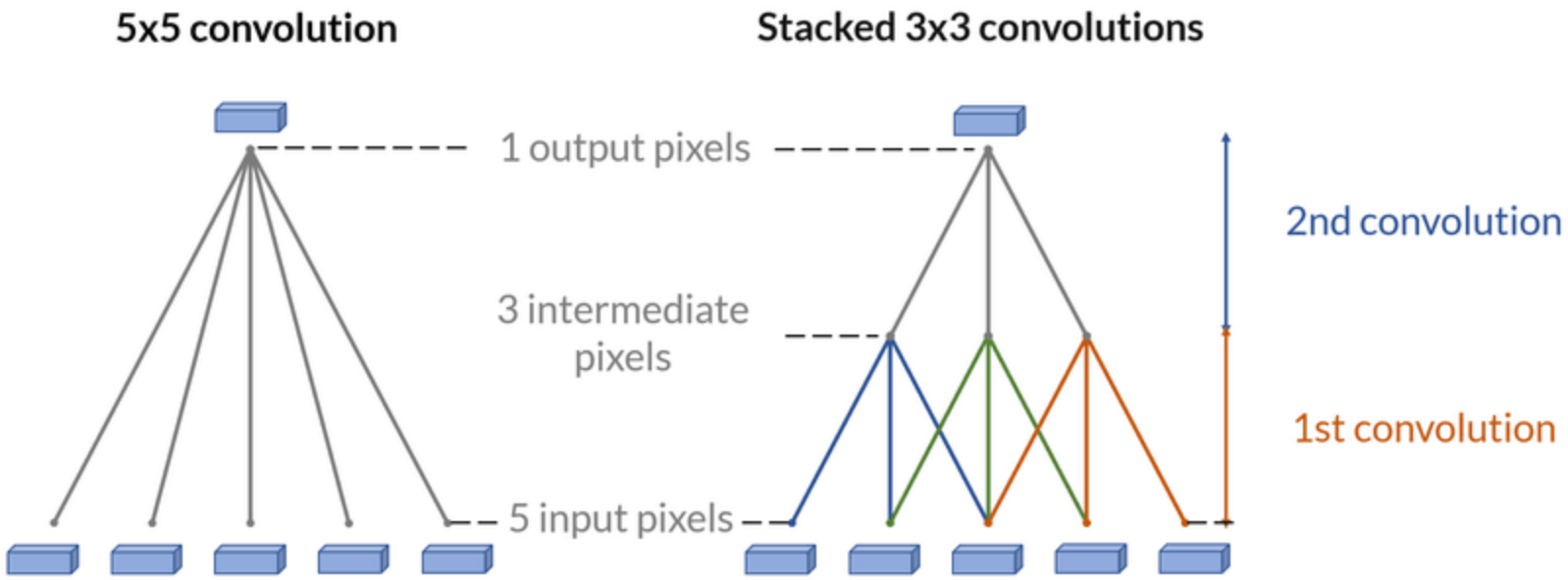
$$n = k^2 * c_{in} * c_{out}$$

So a layer with $(5,5)$ kernels has $\frac{25}{9}$ more parameters than a layer with $(3,3)$ kernels

If we consider the outputs of these layers, the $(5,5)$ reduces 5 input pixels to 1 output pixel—reducing the size by a factor of 5. The $(3,3)$ kernels reduce by a factor of 3—so we get 2 or 3 output pixels, depending on padding.

But we can stack the $(3,3)$ layers to get the same effect as a $(5,5)$ layer, with less parameters…

Kernel Sizes

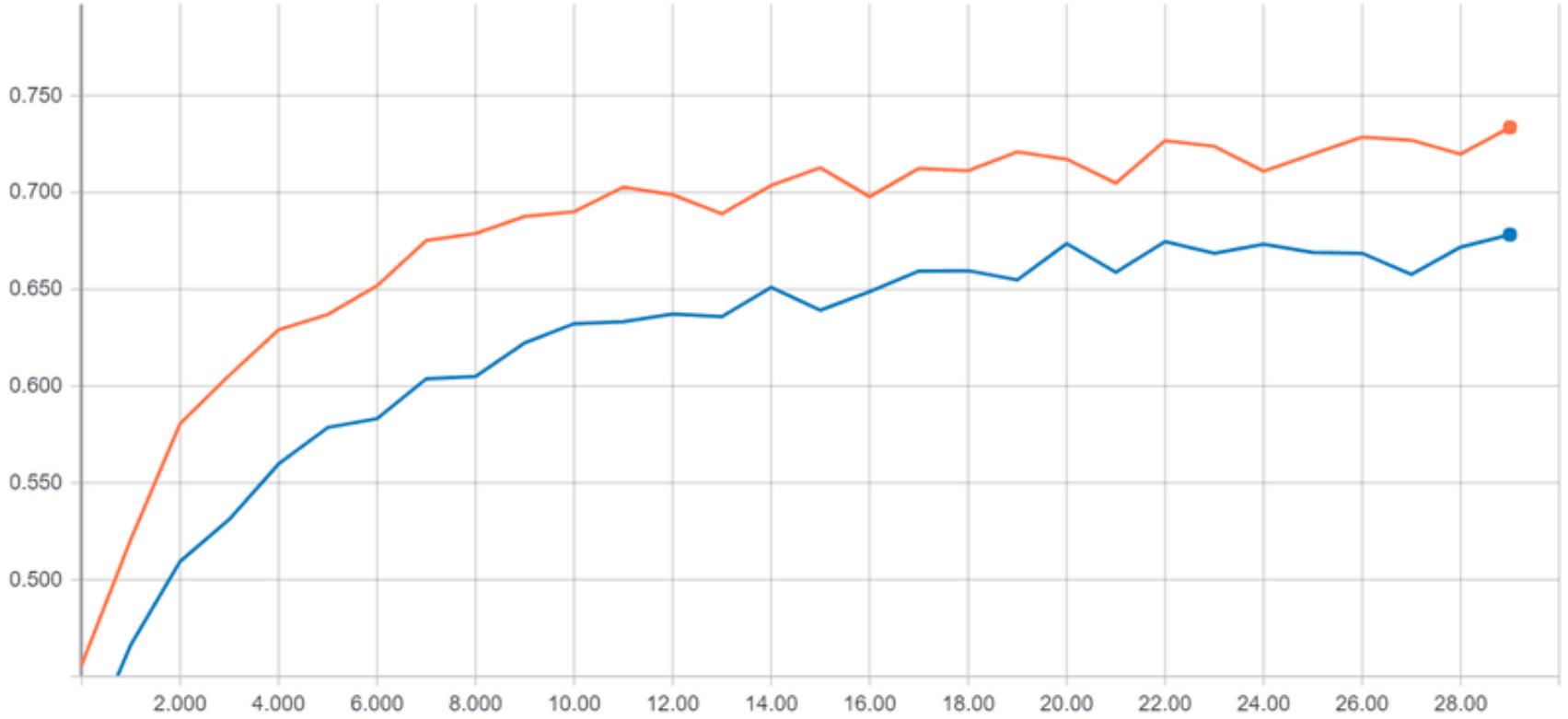


$n = 5^2 * c_{in} * c_{out}$

$n = 2 * 3^2 * c_{in} * c_{out}$

https://www.sicara.fr/blog-technique/2019-10-31-convolutional-layer-convolution-kernel

Smaller kernels are lighter… and better



Validation Accuracy on a 3x3-based Convnet (orange) and the equivalent 5x5-based Convnet (blue)
https://www.sicara.fr/blog-technique/2019-10-31-convolutional-layer-convolution-kernel

Using stacked layers with smaller kernels uses fewer parameters in total, but often yields better results.

Key development in CNN Architectures…

Strides vs Pooling

STRIDED CONVOLUTIONS

Downsamples the image at the same time as performing the convolution.

Useful when you have large input images and want to downscale them quickly.

Larger Kernels can produce the same receptive fields as multiple layers with small kernels.

POOLING

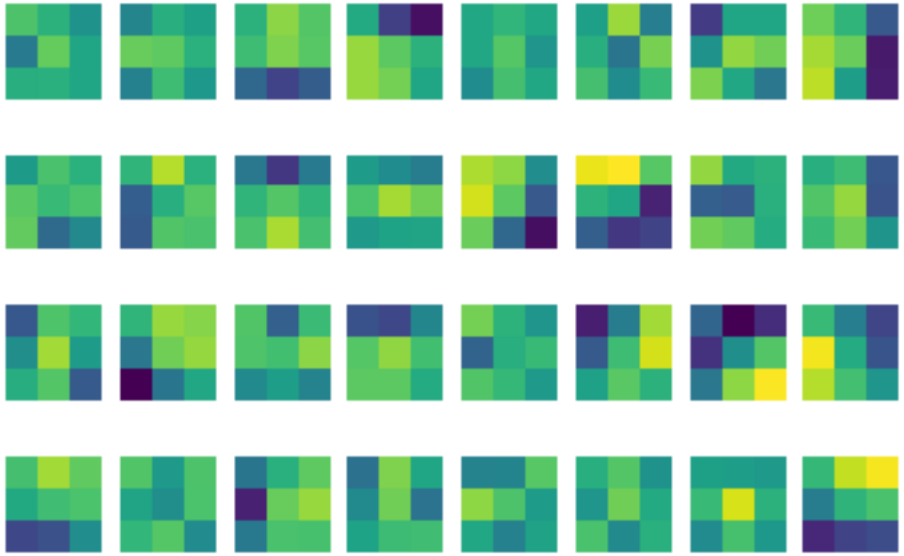
Uses max pooling, average pooling, and other techniques to reduce the size of an input image.

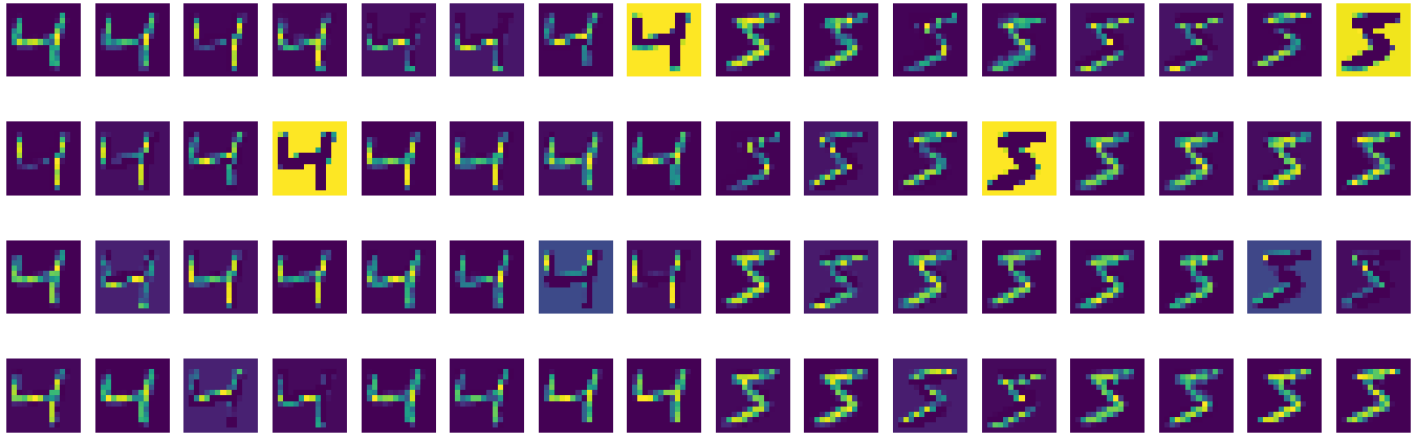
Computationally cheap, both in terms of speed and parameters (none for a pooling layer).

No difficulties with propagating gradients.

Summary: Convolutional Layers

- Convolutional layers are a computers eye. They learn filters which highlight and enhance different features in an image—like edges
- Designing CNNs is tricky business, we have to consider the number of filters, the kernel size, and strides vs pooling





VGGNet

ILSVRC 2014

In 2014, there was a major shift in CNN design, as seen in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in that year.

Up to this point, the winners of this challenge had large kernels in their initial layers—$(11,11)$ in the case of AlexNet in 2012.

As we have seen, using smaller kernel sizes can lead to lighter models and better results, and this was shown by the 2nd place model in ILSVRC 2014.

VGGNet used only $(3,3)$ kernels, while most other models had $(7,7)$ and $(5,5)$ kernels, at least in their initial layers.

AlexNet: https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

Depth in VGGNet

There are multiple versions of VGGNet, with a varying number of layers. The most popular have 16 and 19 layers.

Ideally, adding more layers would lead to a better model—because we would have more parameters with which to learn with.

So why not use a VGGNet50 or VGGNet100?

Depth in VGGNet

There are multiple versions of VGGNet, with a varying number of layers. The most popular have 16 and 19 layers.

Ideally, adding more layers would lead to a better model—because we would have more parameters with which to learn with.

So why not use a VGGNet50 or VGGNet100?

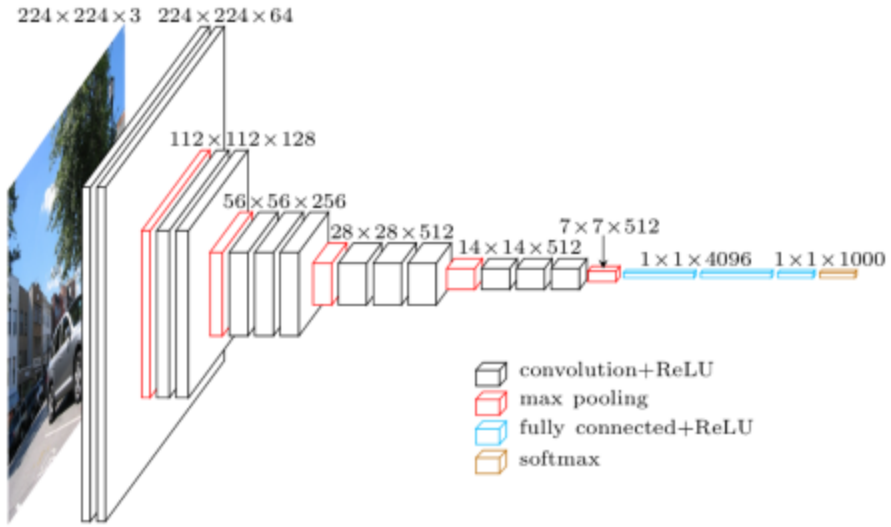
If we keep adding layers to the network, we run into problems with backpropagation. Namely, the Vanishing Gradient Problem.

The ReLU activations in VGGNet can see suffer from vanishing gradients, especially in very deep networks.

With too many layers, it becomes impossible to train early convolutions.

Summary: VGGNet

- VGGNet, created by the Visual Geometry Group, consists of blocks of convolutional layers, each with a max pooling layer.
- The network uses small kernel sizes to reduce the number of parameters, while maintaining the receptive field.
- VGGNet suffers in terms of depth, more layers do not markedly improve the network and can make it perform worse.



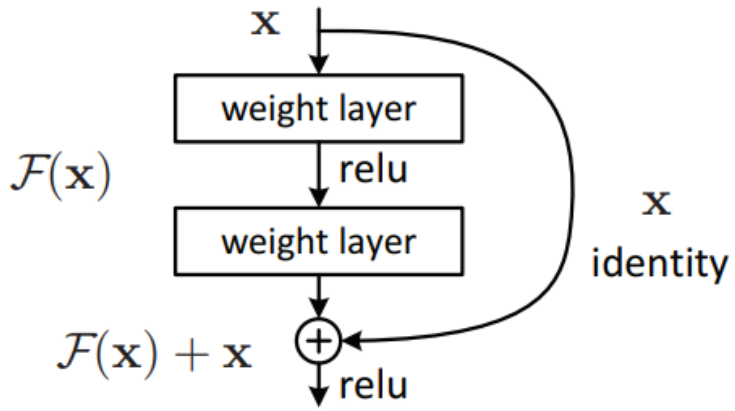
Deep Residual Networks

ResNet – ILSVRC 2015

After VGGNet took the computer vision field by storm, it was quickly unseated as the top dog by ResNet—Deep Residual Networks.

ResNet sets out to tackle the key problem with earlier CNNs, the difficulty in adding deeper layers to the network, without the gradients vanishing.

It does this by learning a Residual Function in its layers.

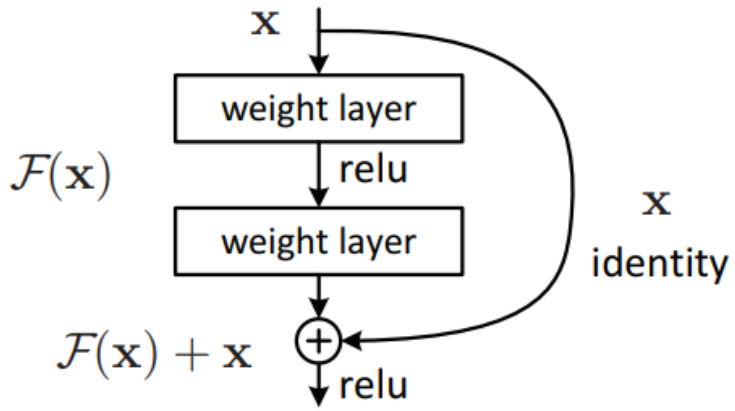


Skip Connections

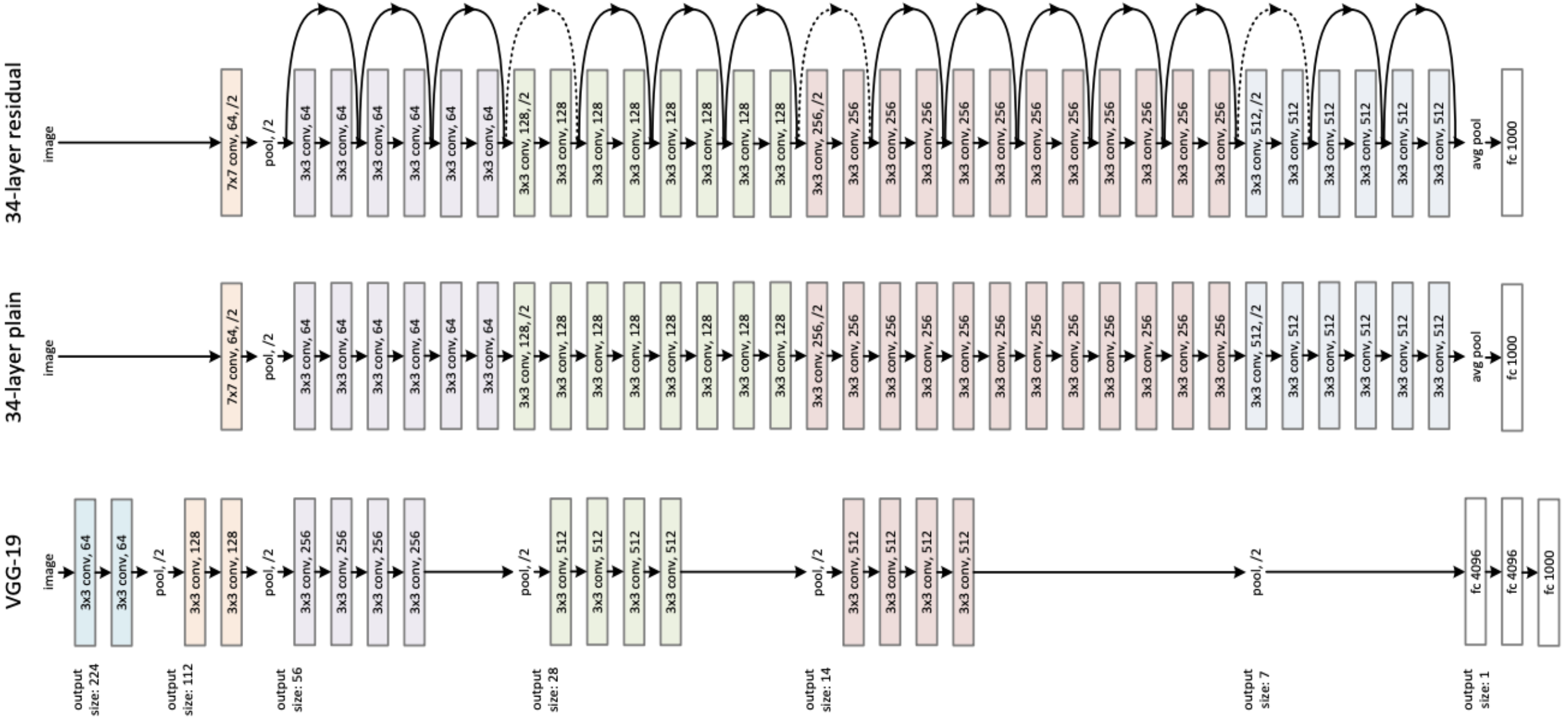
The key tool in ResNets Architecture is the skip connection.

Instead of learning the function $H(x)$ to map the input to the receptive field, it learns the **residual function**, $F(x) = H(x) - x$, with two stacked convolutional layers.

The input, x, is then passed through an "identity" layer, and added to $F(x)$.



ResNet: https://arxiv.org/abs/1512.03385



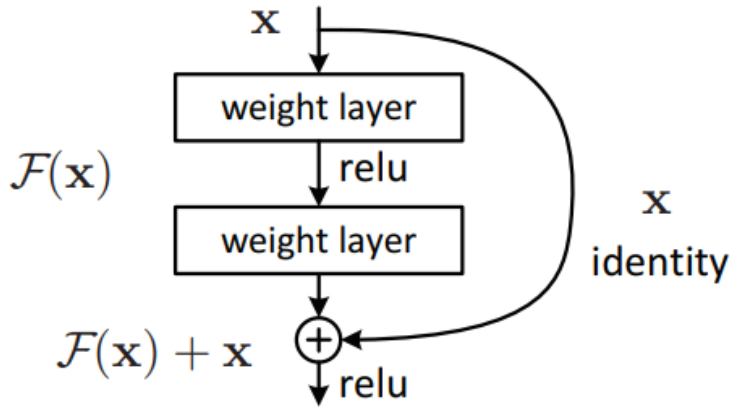
Architecture of ResNet-34 (top), a plain 34-layer network (middle), and VGGNet19 (bottom).
Solid skip connections are identity mappings, dotted lines change the dimensions.

Back Propagation in ResNet

The skip connections act as a shortcut for the gradients of the loss function.

By skipping over the convolutions in the deeper layers of the network, the early layers become much easier to train, because the effect of the vanishing gradients is greatly reduced.

ResNet also has significantly fewer parameters than VGGNet (~3.6 Billion FLOPs vs. ~19.6 Billion FLOPs).



Summary: Deep Residual Networks

- ResNet utilises skip connections to learn residual mapping between layers of the network.
- The skip connections allow the network to be significantly deeper than previous models, due to the reduced vanishing gradients.
- ResNet gains in performance to much greater depths than previous models, with models from ResNet-34 and ResNet-50, to ResNet-100 and ResNet-152. All with fewer parameters than VGGNet19

model	top-1 err.	top-5 err.
VGG-16 [41]	28.07	9.33
GoogLeNet [44]	-	9.15
PReLU-net [13]	24.27	7.38
plain-34	28.54	10.02
ResNet-34 A	25.03	7.76
ResNet-34 B	24.52	7.46
ResNet-34 C	24.19	7.40
ResNet-50	22.85	6.71
ResNet-101	21.75	6.05
ResNet-152	**21.43**	**5.71**

Table 3. Error rates (%, **10-crop** testing) on ImageNet validation. VGG-16 is based on our test. ResNet-50/101/152 are of option B that only uses projections for increasing dimensions.

Questions?