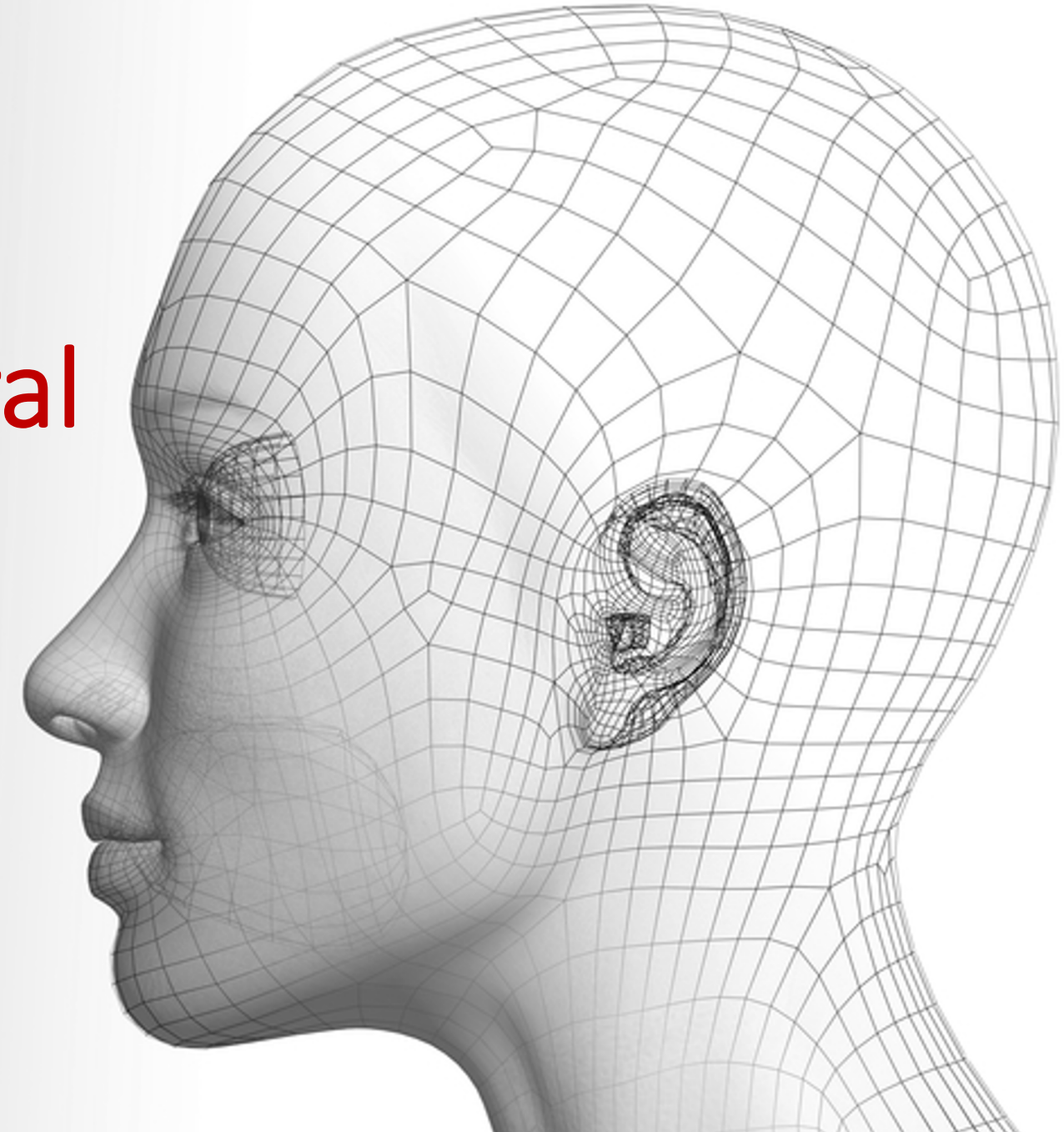
Convolutional Neural Networks

Chen-Change Loy

吕健勤

http://www.ntu.edu.sg/home/ccloy/

https://twitter.com/ccloy



Instructors

Loy Chen Change

2010 — **PhD**

Queen Mary University of London

2011 — **Post-doctoral Fellow**

Queen Mary University of London
Vision Semantics

2013 — **Research Assistant Professor**

The Chinese University of Hong Kong

2018 — **Nanyang Associate Professor**

Nanyang Technological University

Adjunct Associate Professor

The Chinese University of Hong Kong

Visiting Scholar

Chinese Academy of Sciences

Research areas

- Image restoration and processing
- Image editing and manipulation
- Detection, segmentation and recognition
- Deep learning
- Face anaylsis

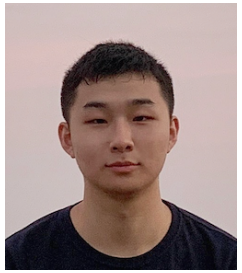
Email: ccloy@ntu.edu.sg
Office: By email appointment

Instructors (Second-half)

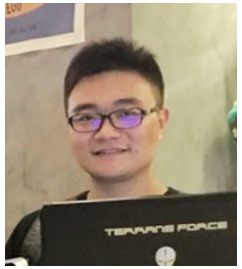
Assoc. Professor Loy Chen Change
E-mail: ccloy@ntu.edu.sg
Office: N4-02c-108

TA: **Mr. Kelvin Chan**
Email: CHAN0899@e.ntu.edu.sg
Office: Multimedia and Interactive Computing Lab, N4-B1C-17
Office hours: Friday 3:00 – 4:00pm

TA: **Mr. Zhang Wenwei**
Email: WENWEI001@e.ntu.edu.sg
Office: Parallel and Distributed Computing Lab, N4-B2a-02
Office hours: Friday 3:00 – 4:00pm



TA: **Mr. Zhou Shangchen**
Email: S200094@e.ntu.edu.sg
Office: Multimedia and Interactive Computing Lab, N4-B1C-17
Office hours: Friday 3:00 – 4:00pm



Schedule

- Week 8 – Convolutional Neural Networks (CNN)
- Week 9 – Recurrent Neural Networks (RNN)
- Week 10 – Gated Recurrent Neural Networks
- Week 11 – Autoencoders
- Week 12 – Generative Adversarial Networks (GAN)
- Week 13 – Selected Topics

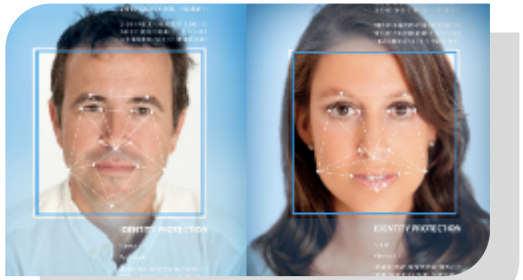
Outline

- Applications and success
- Basic components in CNN
- CNN architectures
- Training basics
- Optimizers

Applications and Success

Deep learning enables AI breakthroughs

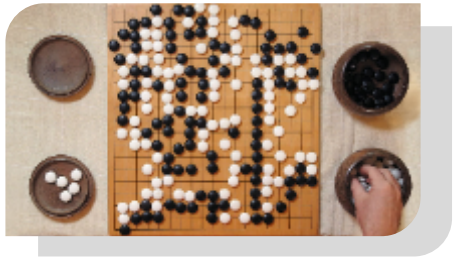
Facial Recognition



Voice Recognition

Image Recognition

Game Playing



Deep Learning

Enabling machines to acquire knowledge and skills inducted from massive data

Natural Language Processing

Autonomous Driving

Robo-advisor, Finance Trading Bot

Identity Authentication

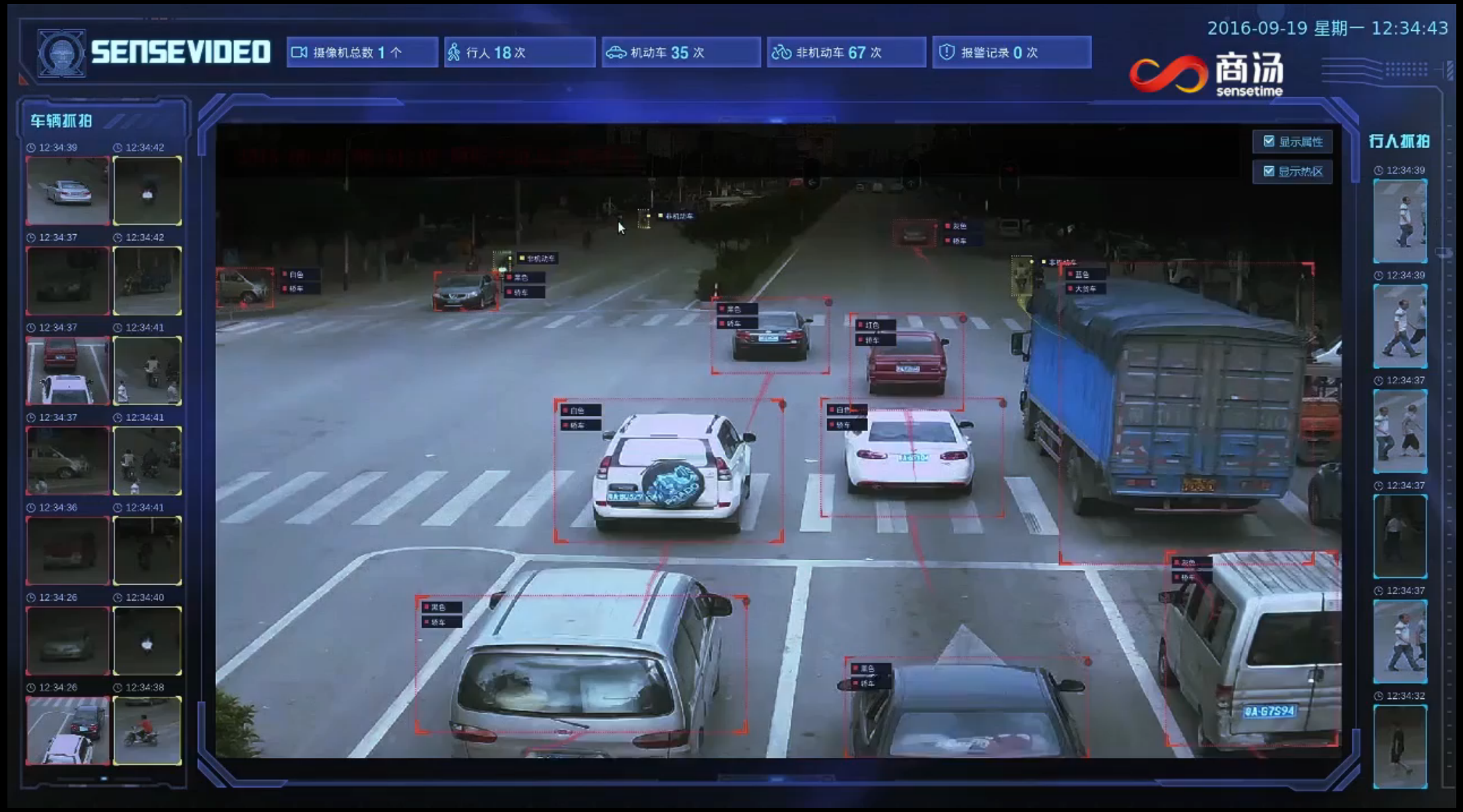
人像

身份证

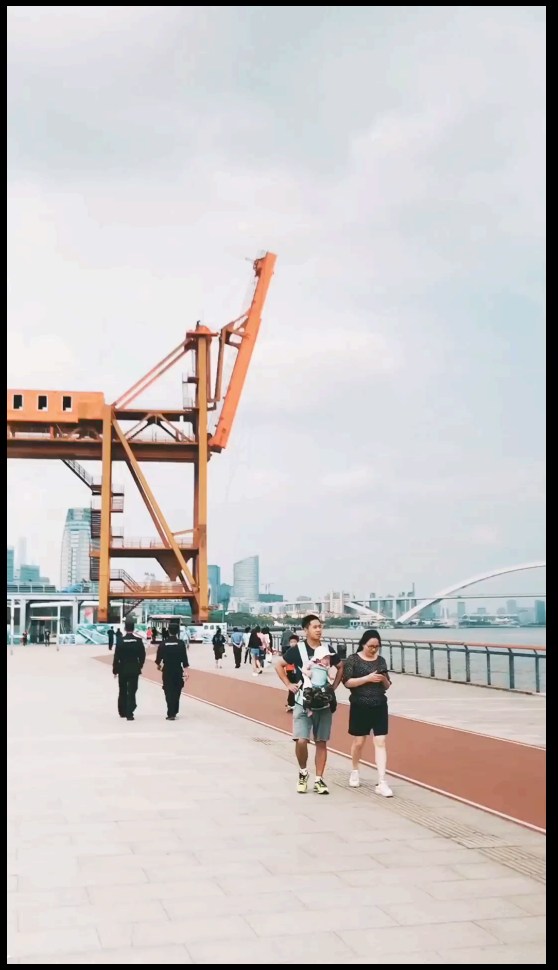
By using face recognition technology of SenseTime, more than **400 million** people registered and completed facial identity authentication in 2016


商汤
sensetime
Monitor
History
Task
SenseFace
Target
Statistic
Tools
Apr 3 2016
11:56:16
Task List
Control Task 1
local-video-1
surveillance task1
surveillance task2
surveillance task3
surveillance task4
surveillance task5
surveillance task6
surveillance task7
surveillance task8
surveillance task9
surveillance task10
surveillance task11
surveillance task12
surveillance task13
surveillance task14
surveillance task15
surveillance task16
surveillance task17
surveillance task18
08-13-2015 16:12:52
禁止跨越
Captured
Today: 1019
Month: 1019
More
11:56:17
11:56:16
11:56:15
11:56:08
11:56:05
11:55:57
11:55:47
11:55:47
11:55:47
11:55:47
11:55:46
11:55:46
Target
Target Number: 223
More
2016-04-03 11:55:39
local-video-1
Captured
92.6%
Target
chenzhaoj...
2016-04-03 11:54:56
local-video-1
Captured
91.7%
Target
suzhekun
2016-04-03 11:53:51
local-video-1
Captured
93.9%
Target
chenzhaoj...

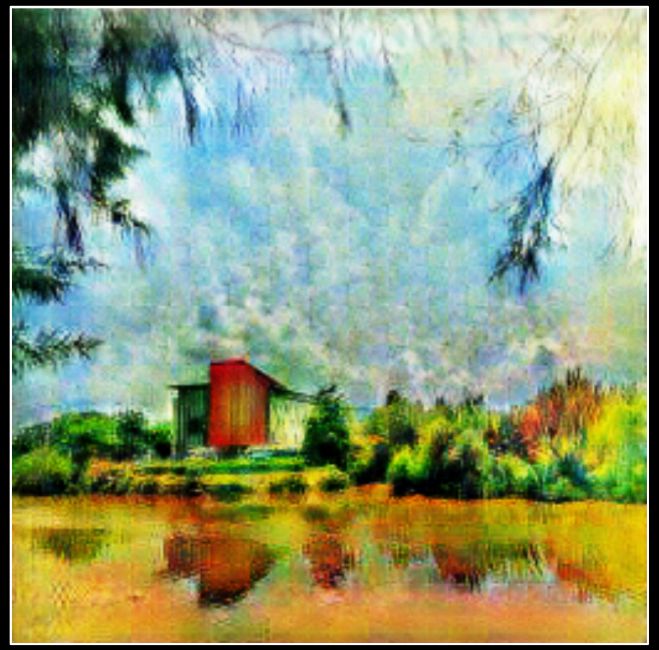
Safe City


SENSEVIDEO
摄像机总数 1 个
行人 18 次
机动车 35 次
非机动车 67 次
报警记录 0 次
2016-09-19 星期一 12:34:43
商汤
sensetime
车辆抓拍
行人抓拍
显示属性
显示热区

Entertainment

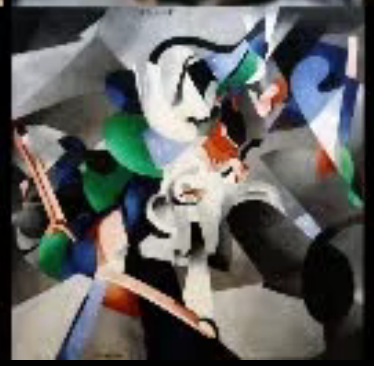


Horse Chestnut Tree in Blossom
Vincent van Gogh, May 1887



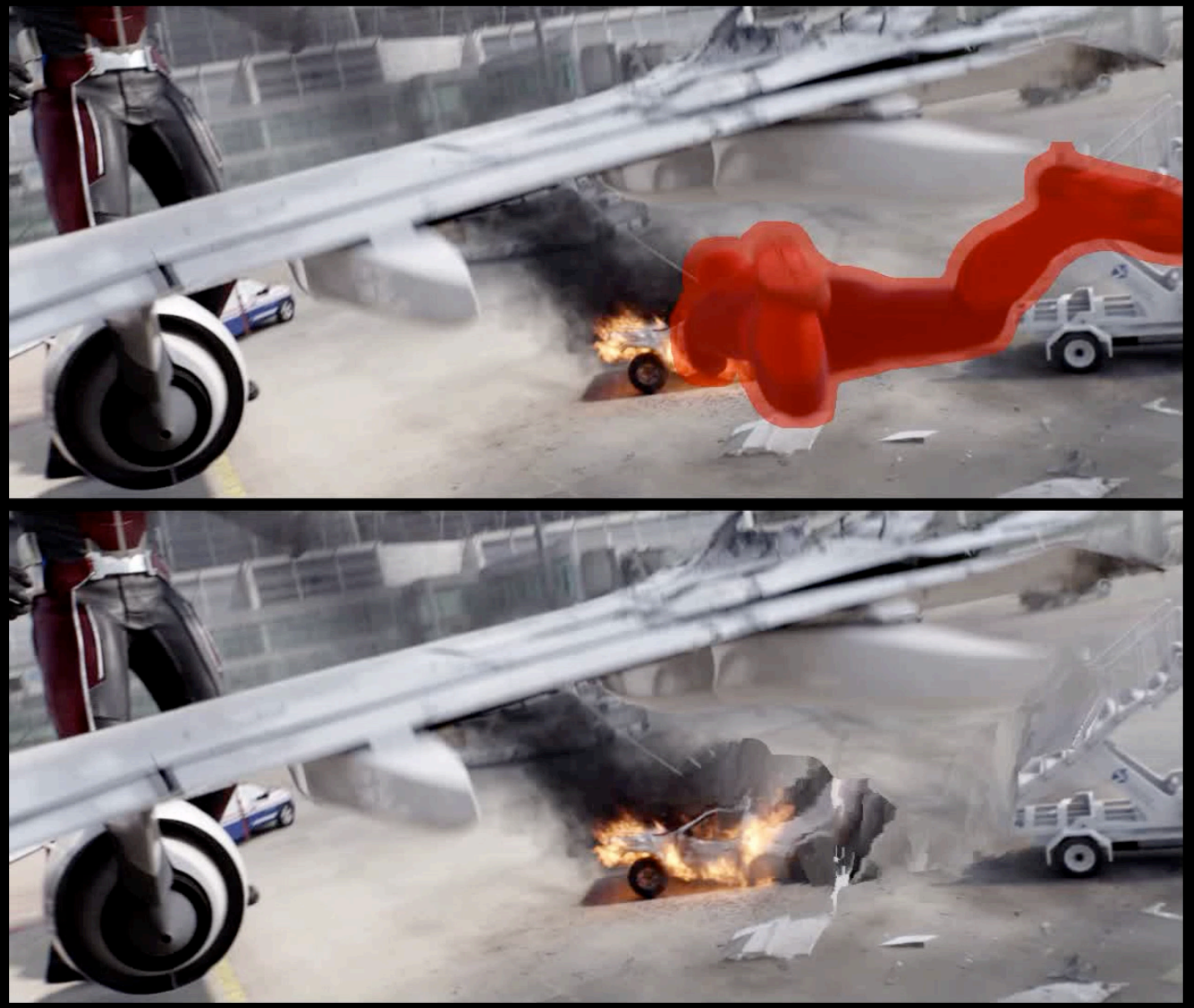
Garden of the Asylum
Vincent van Gogh, December 1889

Entertainment

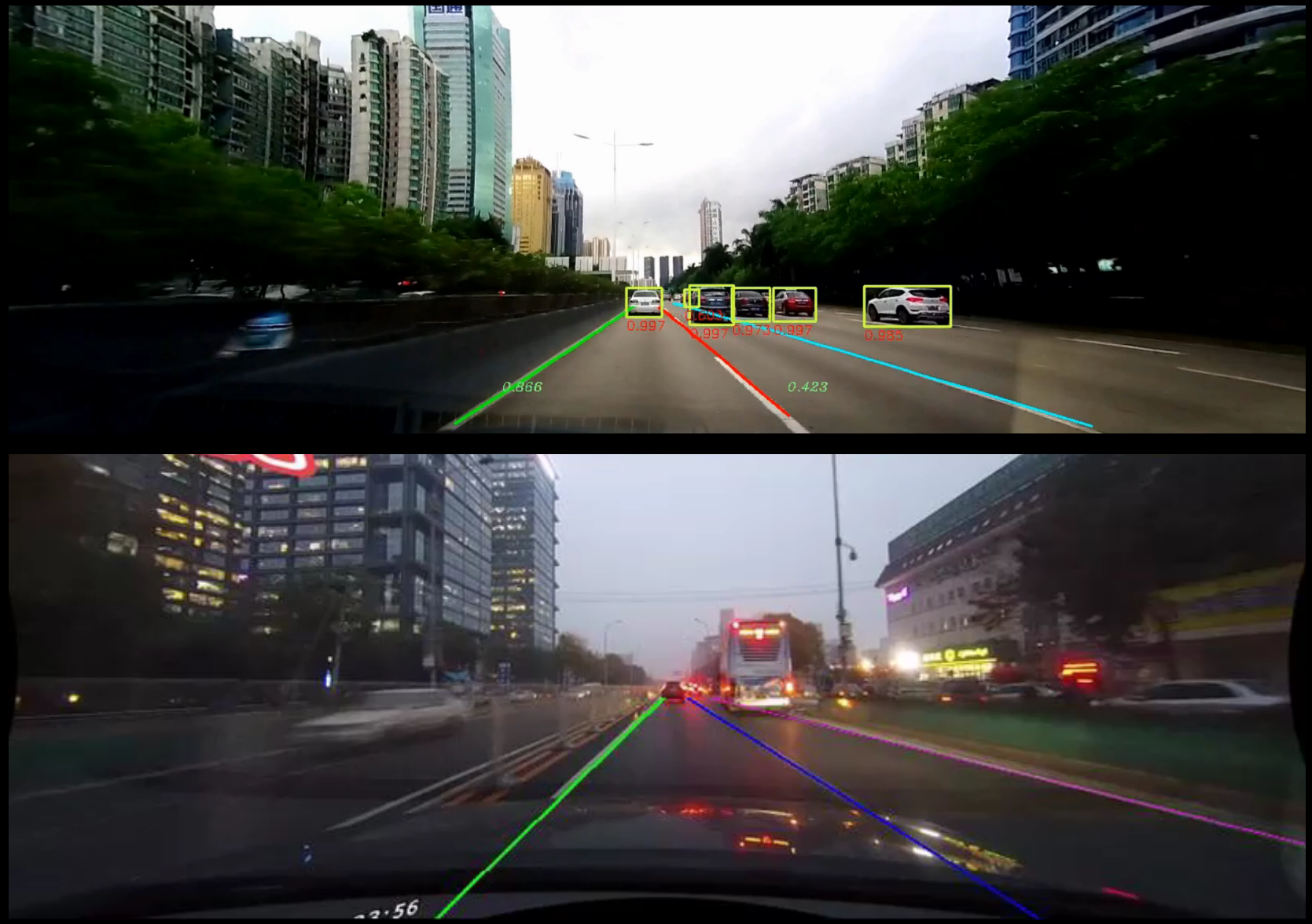


Entertainment

Image and Video Editing



Autonomous driving

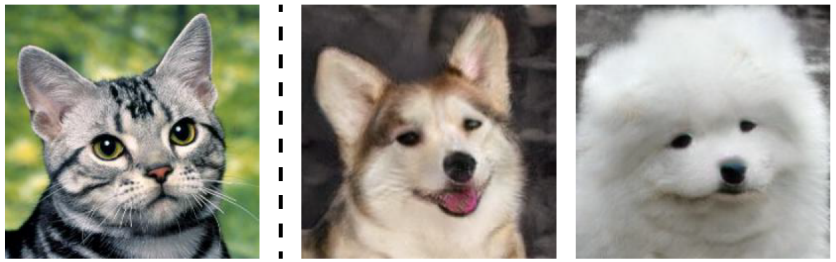


Results of StyleGAN

Credit: Tero Karras et al., A Style-Based Generator Architecture for Generative Adversarial Networks, CVPR 2019

Almost Everything can be Faked!

CycleGAN
[Zhu et al., ICCV 2017]



MUNIT
[Huang et al., ECCV 2018]

StarGAN
[Choi et al., CVPR 2018]

Outline

- Applications and success
- Basic components in CNN
- CNN architectures
- Training basics
- Optimizers

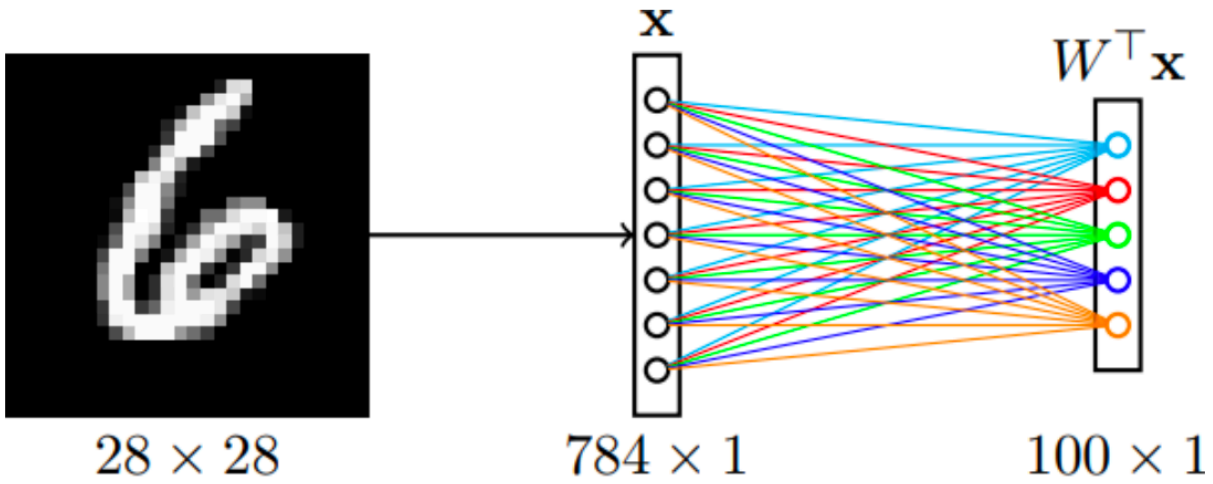
Basic Components in CNN

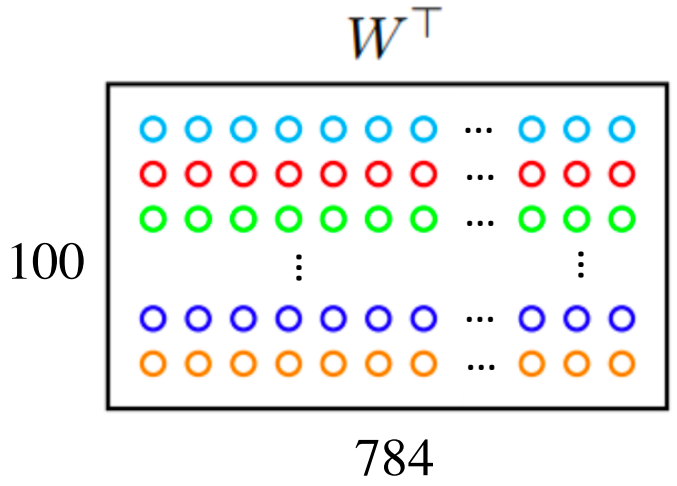
Credits

- CS 230: Deep Learning
  - https://stanford.edu/~shervine/teaching/cs-230.html

- CS231n: Convolutional Neural Networks for Visual Recognition
  - http://cs231n.stanford.edu/syllabus.html

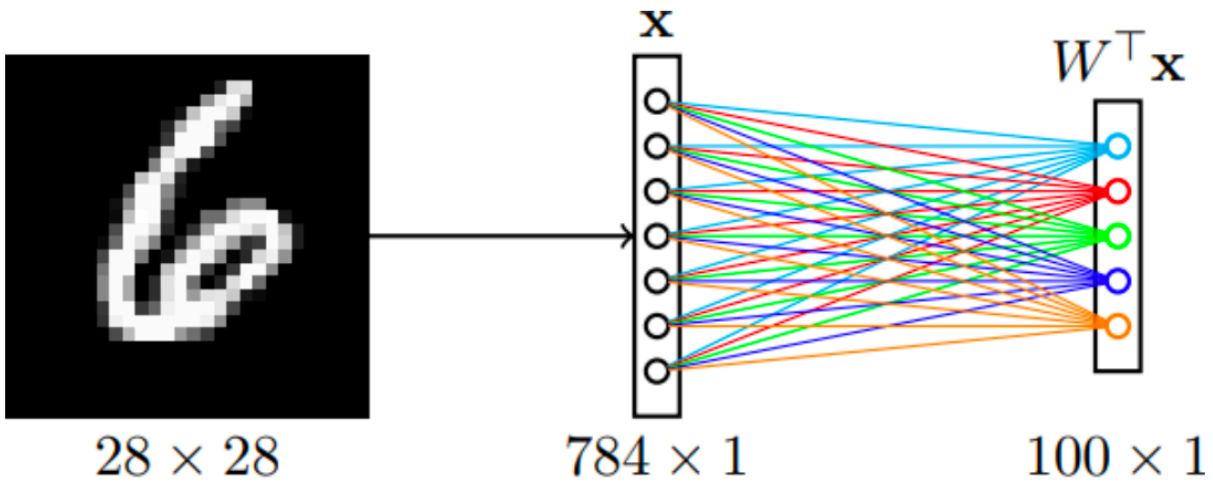
Motivation

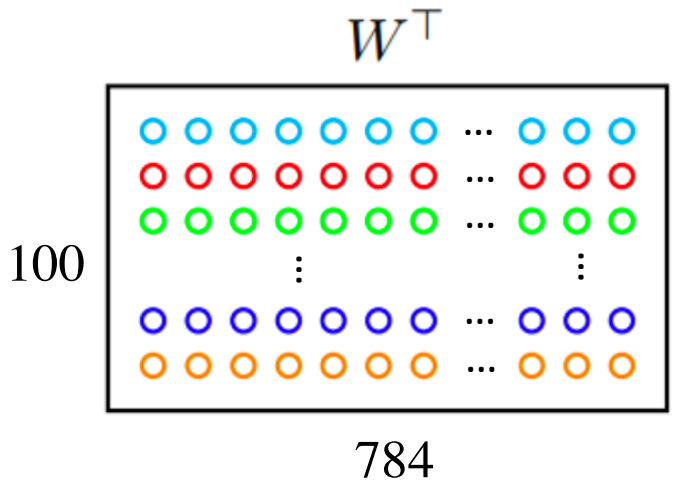




Let us consider the first layer of a MLP taking images as input. What are the problems with this architecture?

Motivation





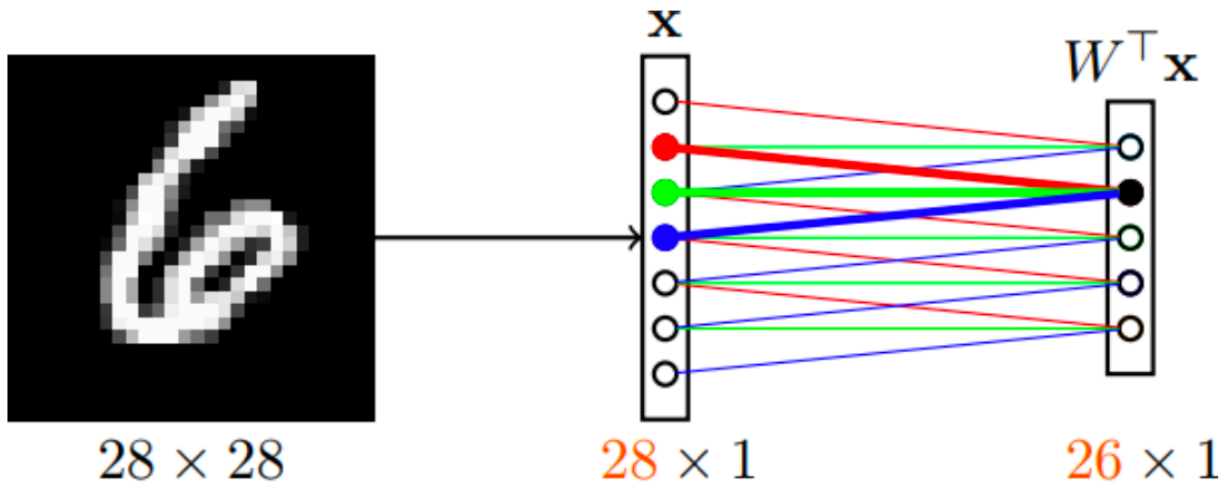
Issues

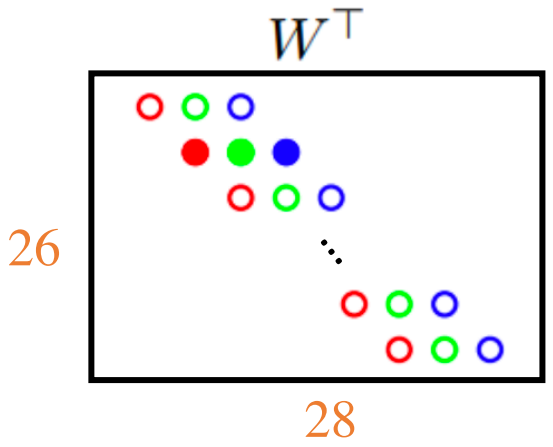
- Too many parameters: 100 × 784 + 100.
  - What if images are 640 × 480 × 3?
  - What if the first layer counts 1000 units?

Fully connected networks where neurons at one level are connected to the neurons in other layers are not feasible for signals of large resolutions and are computationally expensive in feedforward and backpropagation computations.

- Spatial organization of the input is destroyed.
  - The network is not invariant to transformations (e.g., translation).

Locally connected networks



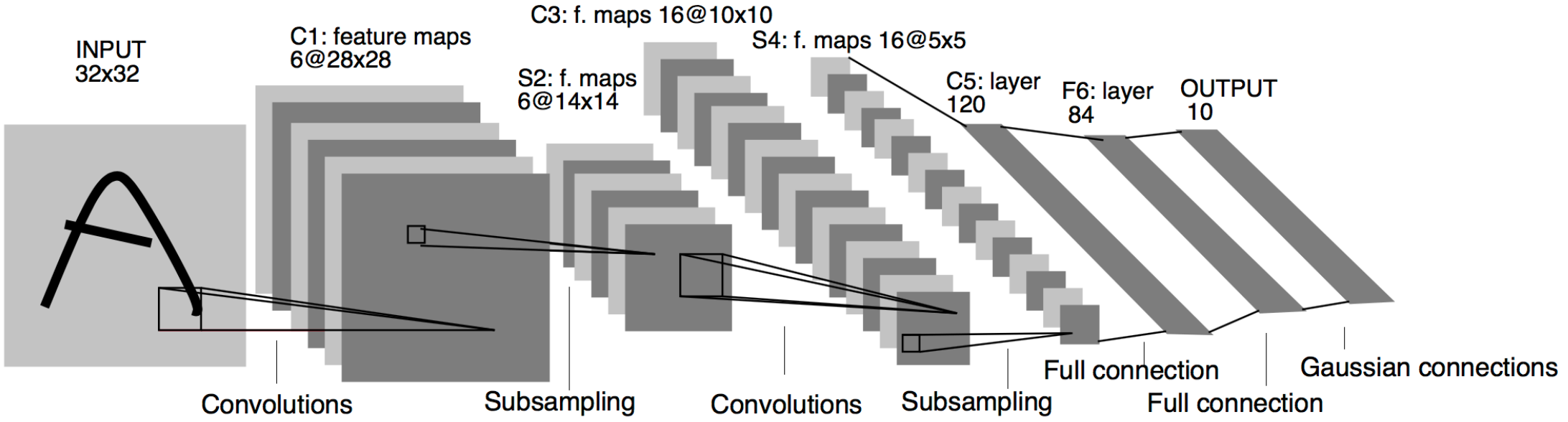


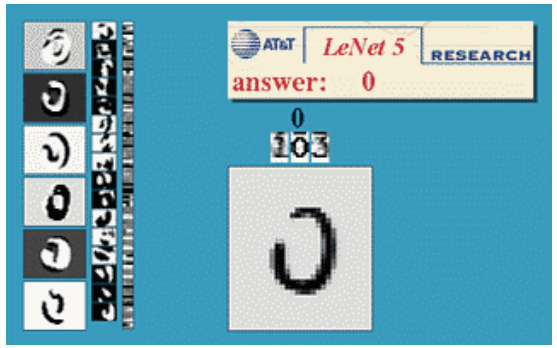
Instead, let us only keep a sparse set of connections, where all weights having the same color are shared.

- The resulting operation can be seen as shifting the same weight triplet (kernel).
- The set of inputs seen by each unit is its receptive field.

This is a 1D convolution, which can be generalized to more dimensions.

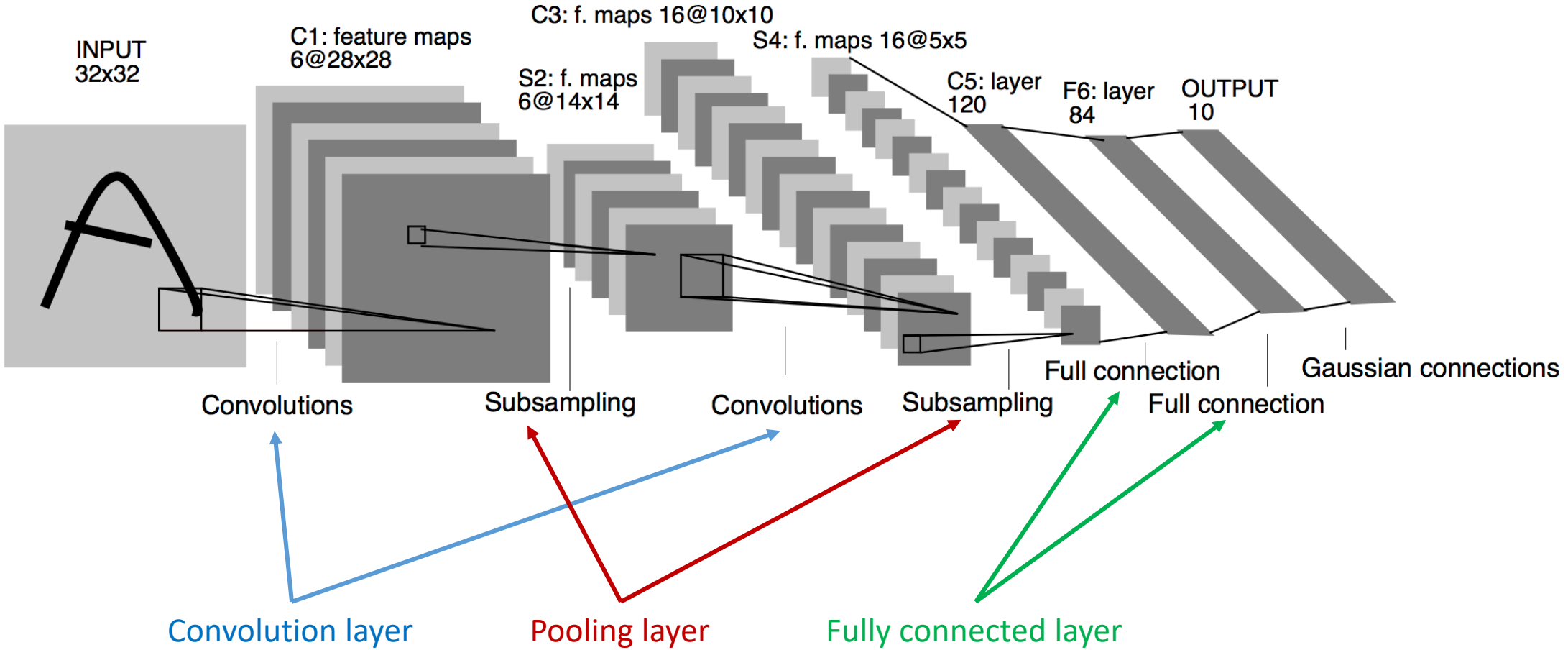
An example of convolutional network: LeNet 5





LeCun et al., 1998

An example of convolutional network: LeNet 5

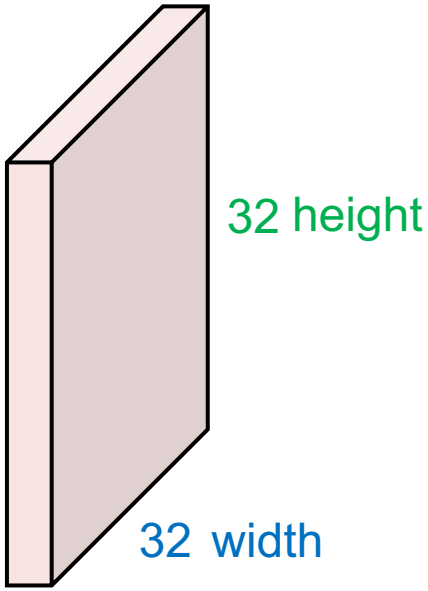


As we go deeper (left to right) the height and width tend to go down and the number of channels increased.

Common layer arrangement: Conv → pool → Conv → pool → fully connected → fully connected → output

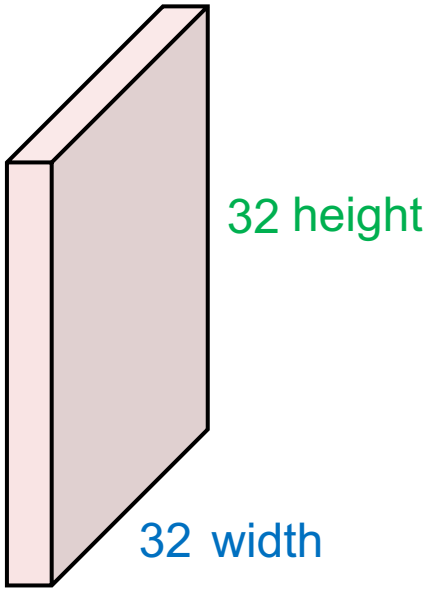
Convolution layer

3x32x32 image

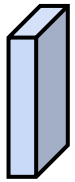


Convolution layer

3x32x32 image



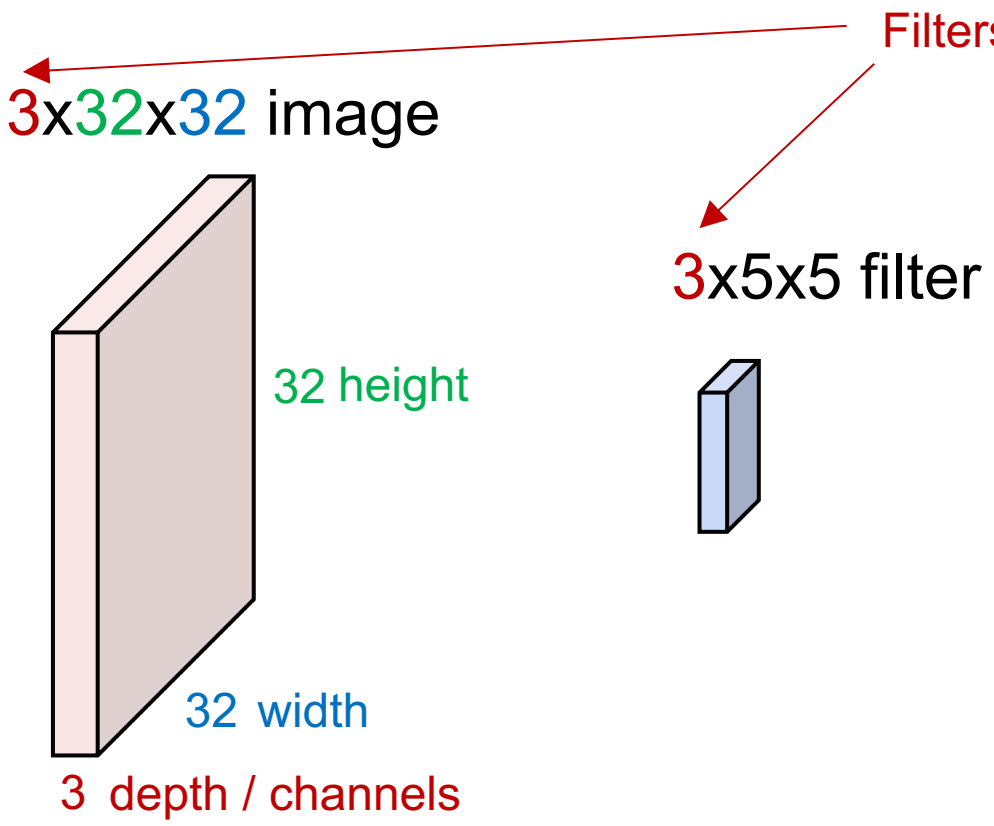
3x5x5 filter



Weights of filters are learned using backpropagation algorithm. The weights learned are known as *filters* or *kernels*

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

Convolution layer

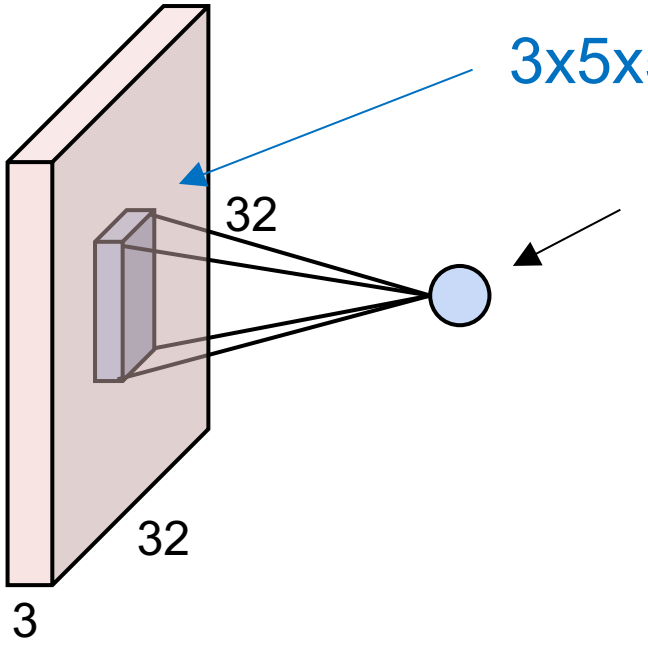


Weights of filters are learned using backpropagation algorithm. The weights learned are known as *filters* or *kernels*

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

Convolution layer

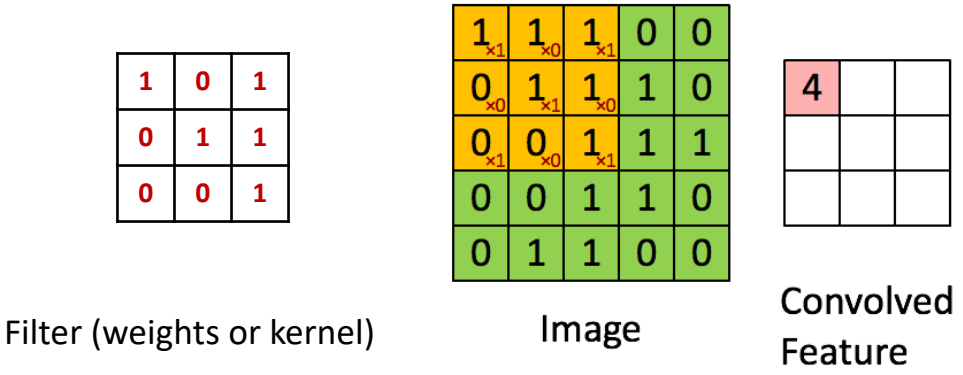
3x32x32 image



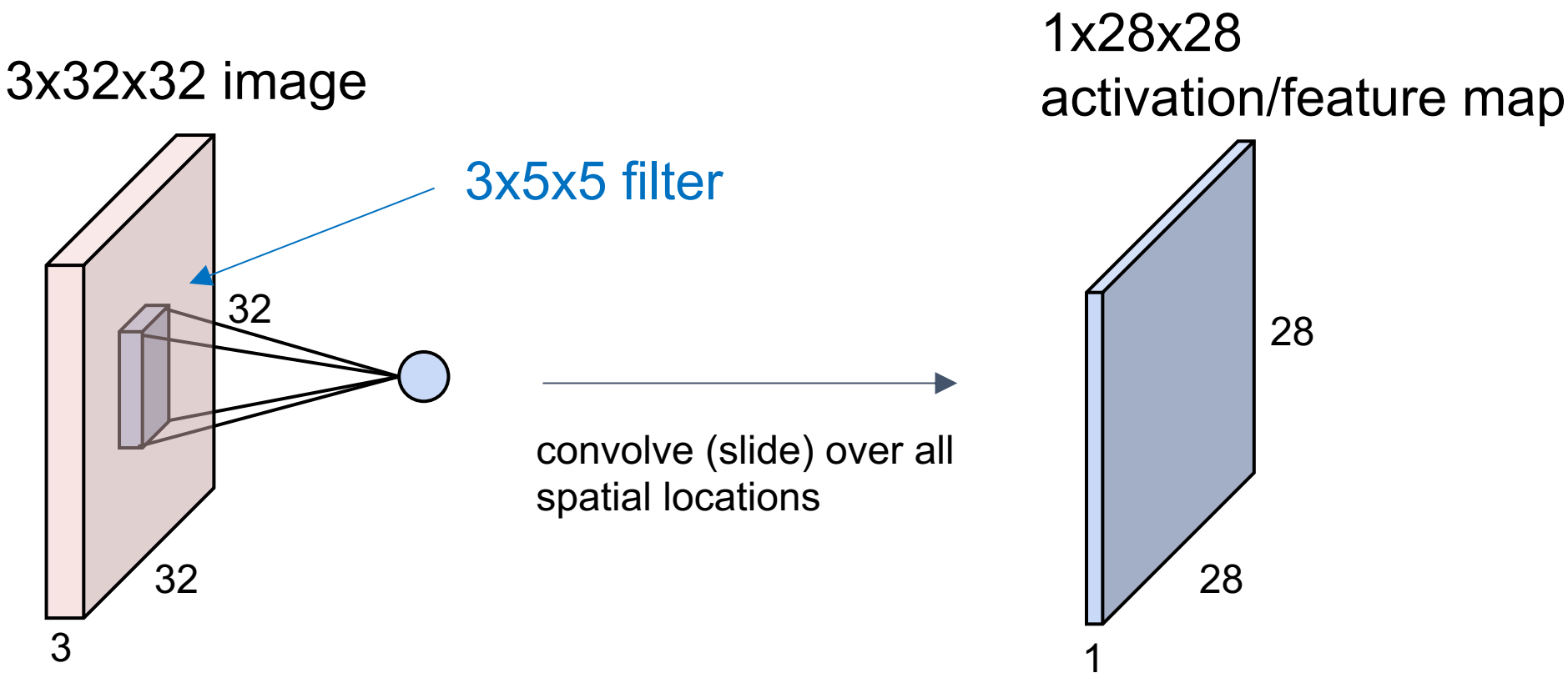
1 number:
the result of taking a dot product between the filter and a small 3x5x5 chunk of the image

(i.e. 3*5*5 = 75-dimensional dot product + bias)

An example of convolving a 1x5x5 image with a 1x3x3 filter



Convolution layer

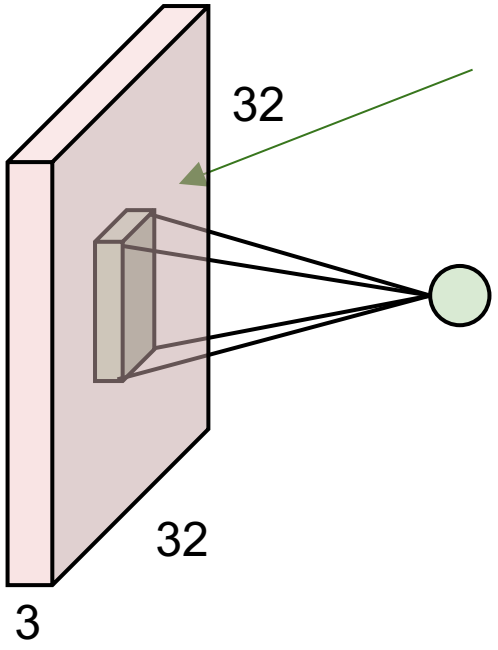


The activations are obtained by convolving the filters (weights) with the input activations. The output activation produced by a particular filter is known as a *activation map / feature map*

Convolution layer

Consider a second, green filter

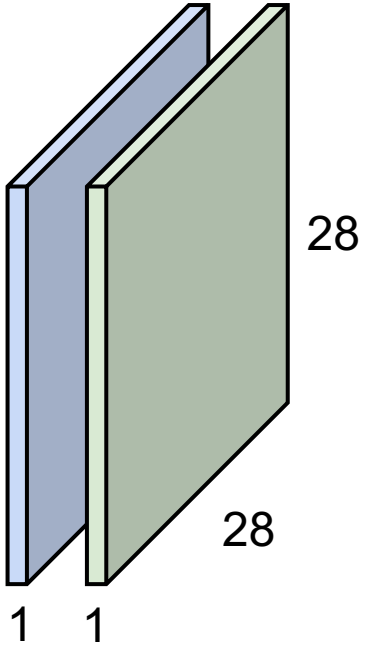
3x32x32 image



3x5x5 filter

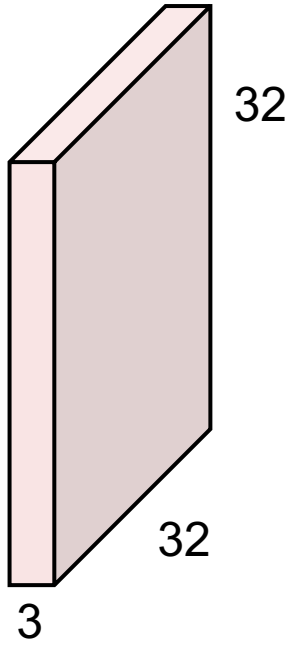
convolve (slide) over all spatial locations

Two 1x28x28 activation/feature maps

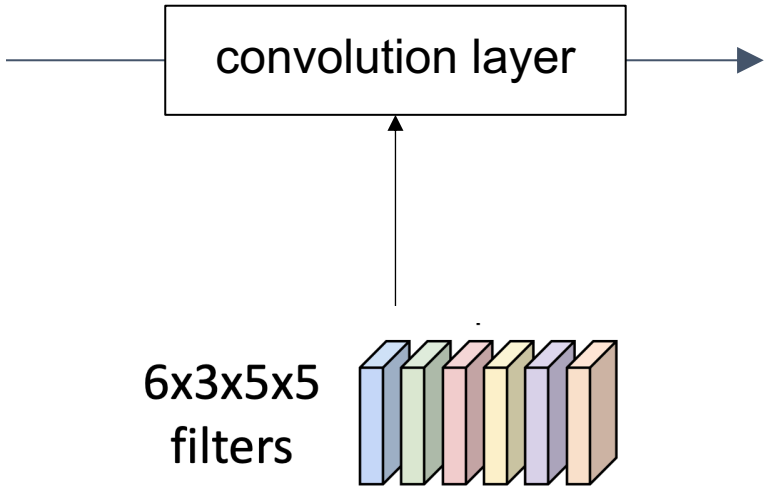


Convolution layer

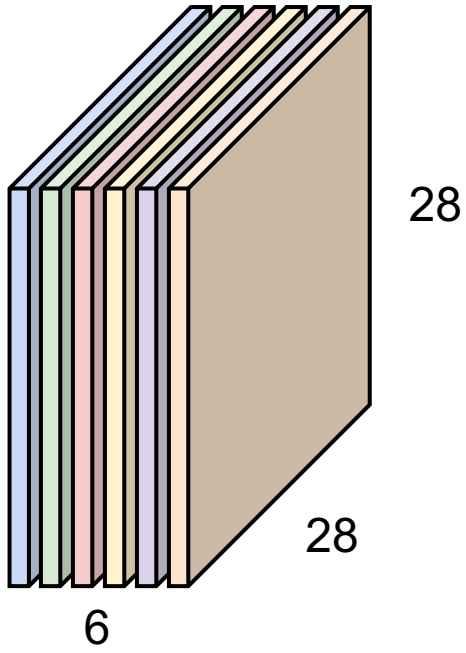
3x32x32 image



If we had six 3x5x5 filters, we'll get six separate activation maps:

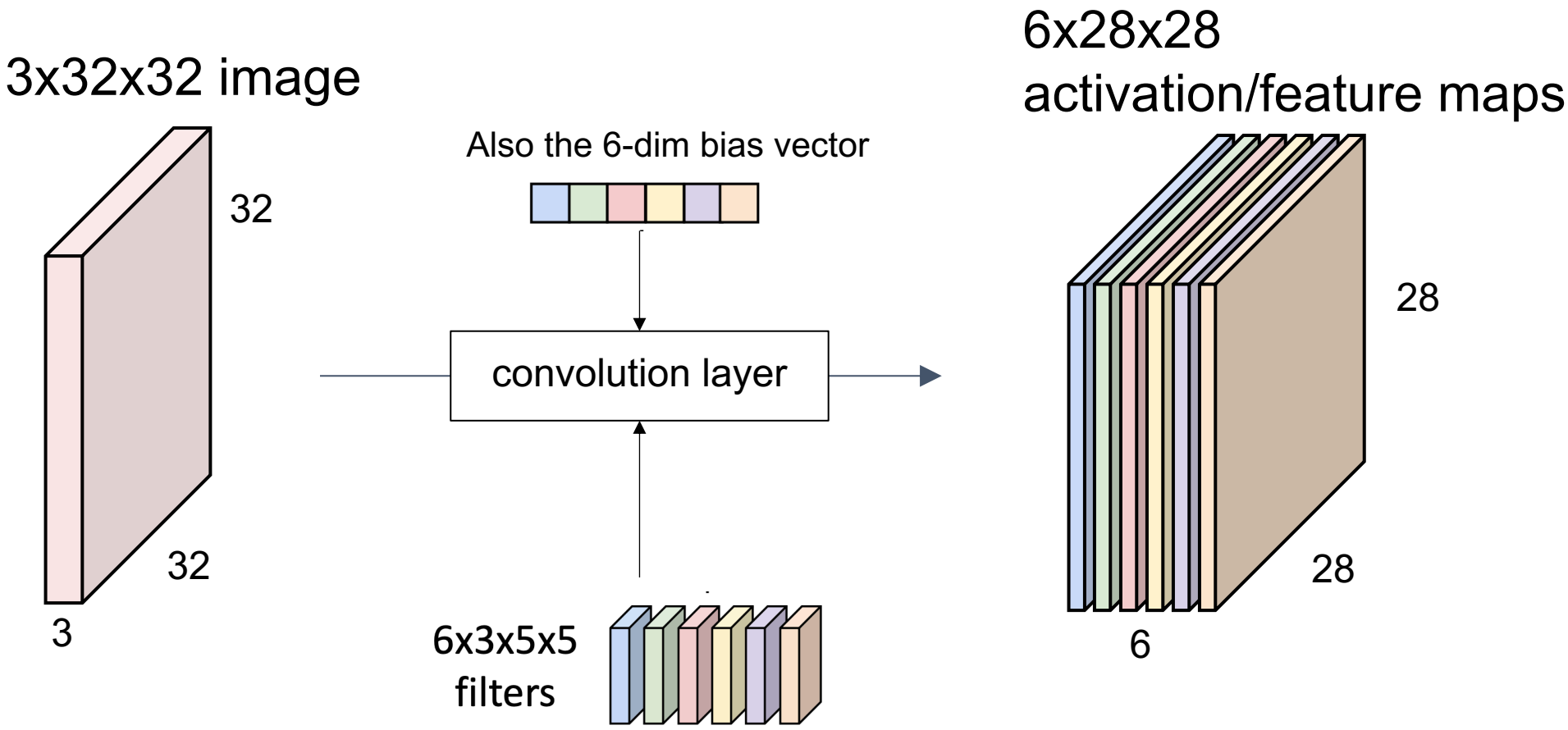


Six 1x28x28 activation/feature maps

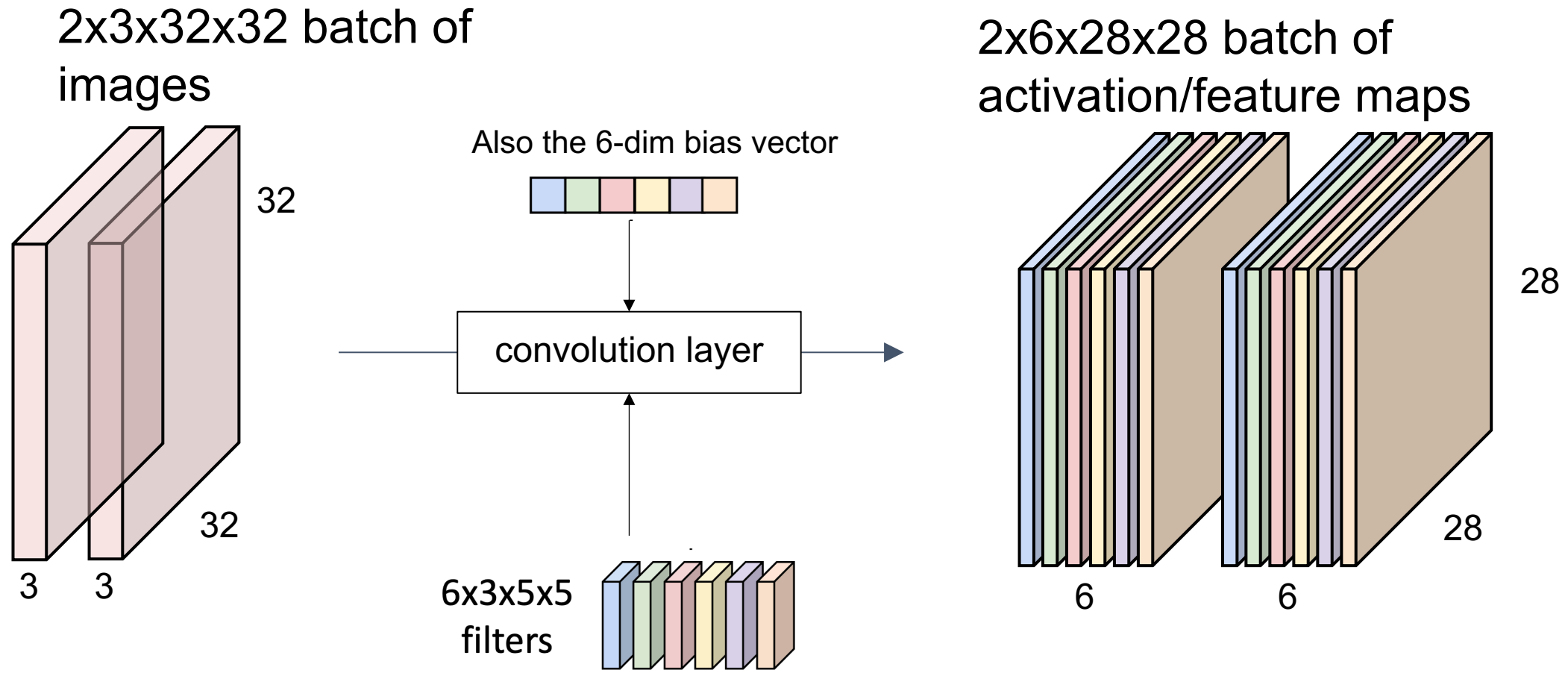


We stack these up to get a "new image" of size 6x28x28

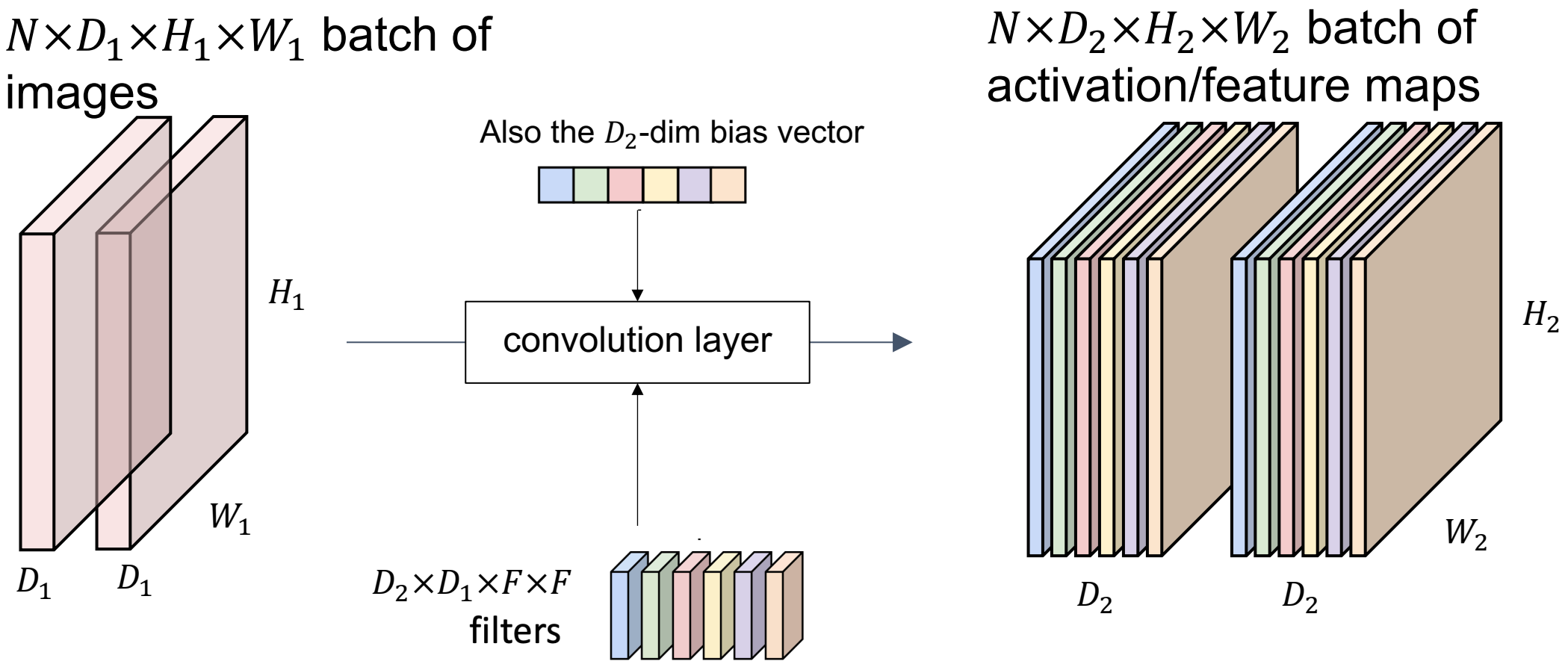
Convolution layer



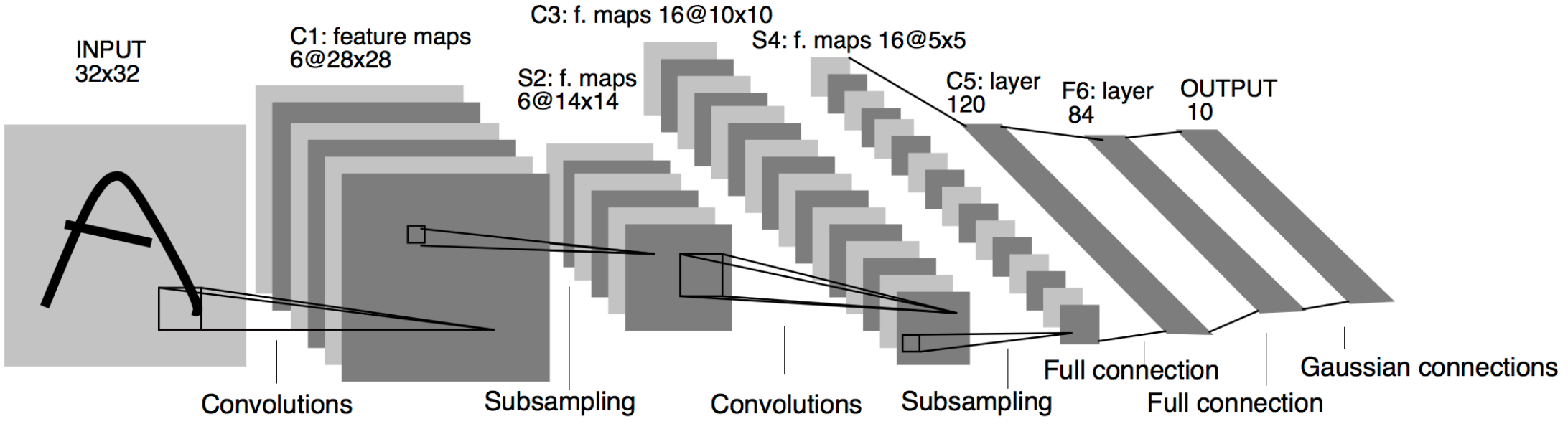
Convolution layer



Convolution layer



An example of convolutional network: LeNet 5

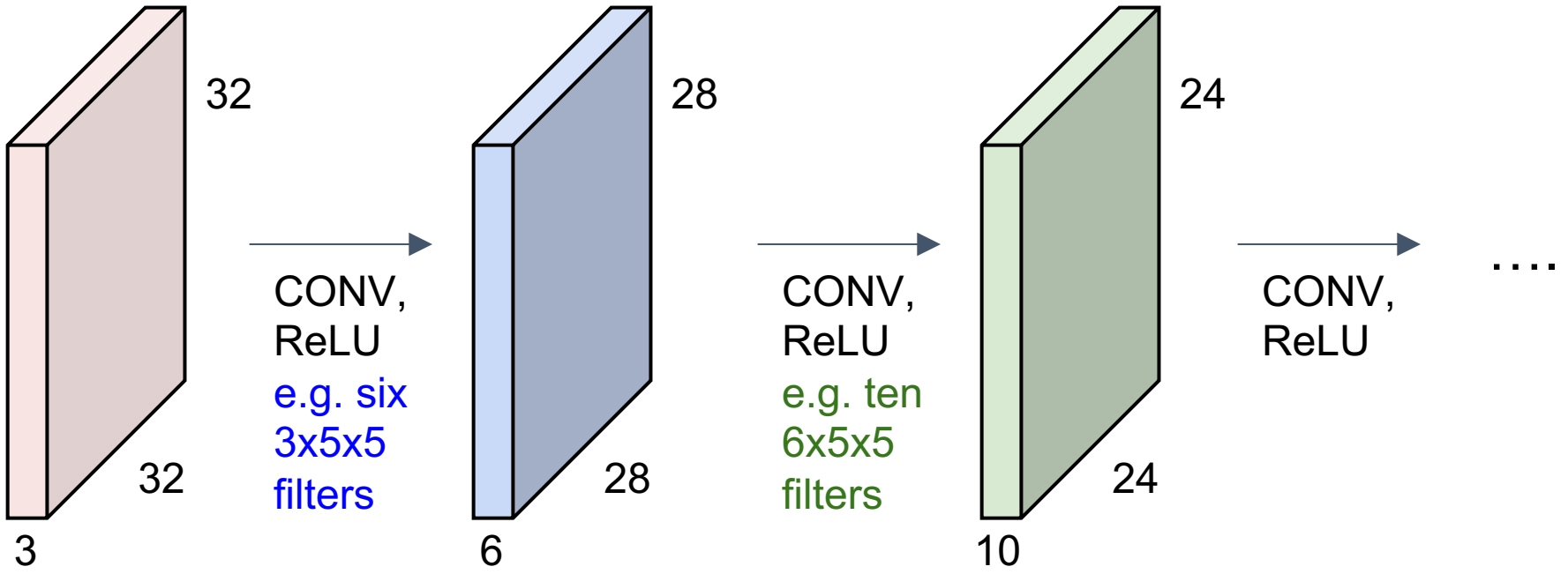


Convolution layer

Back to this example, 6@28x28 means that we have 6 feature maps of size 28*28
You can imagine that the first convolutional layer uses 6 filters and each filter is of size 5 x 5 (how do we know that? We will discuss that later)

Convolution layer

CNN is a sequence of convolutional layers, interspersed with activation functions



Convolution layer

Consider a kernel $\boldsymbol{w} = \{w(l,m)\}$, which has a size of $L \times M, L = 2a + 1, M = 2b + 1$

Synaptic input at location $p = (i,j)$ of the first hidden layer due to a kernel is given by

$$u(i,j) = \sum_{l=-a}^{a} \sum_{m=-b}^{b} x(i+l, j+m) w(l,m) + b$$

For instance, given $L = 3, M = 3$

$$u(i,j) = x(i-1, j-1)w(-1,-1) + x(i-1,j)w(-1,0) + \ \ldots$$
$$+x(i,j)w(0,0) + x(i+1, j+1)w(1,1) + b$$

Convolution layer

The output of the neuron at (i, j) of the convolution layer

$$y(i,j) = f(u(i,j))$$

where f is an activation function. For deep CNN, we typically use ReLU, $f(x) = \max(0, x)$.

Note that one weight tensor $\boldsymbol{w}_k = \{w_k(l,m)\}$ or kernel (filter) creates one feature map:

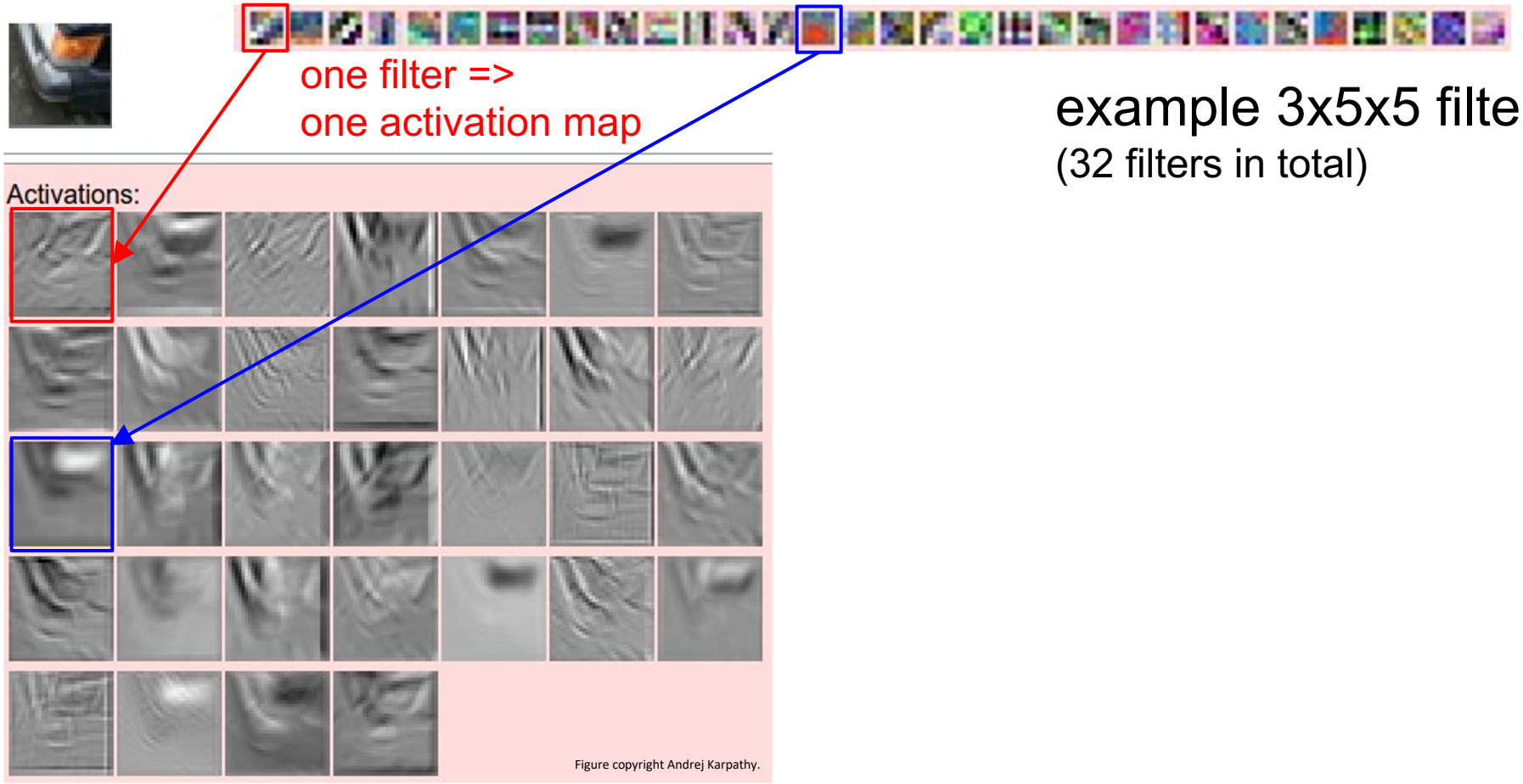
$$\boldsymbol{y}_k = \{y_k(i,j)\}$$

If there are K weight vectors $(\boldsymbol{w}_k)_{k=1}^K$, the convolutional layer is formed by K feature maps

$$\boldsymbol{y} = (\boldsymbol{y}_k)_{k=1}^K$$

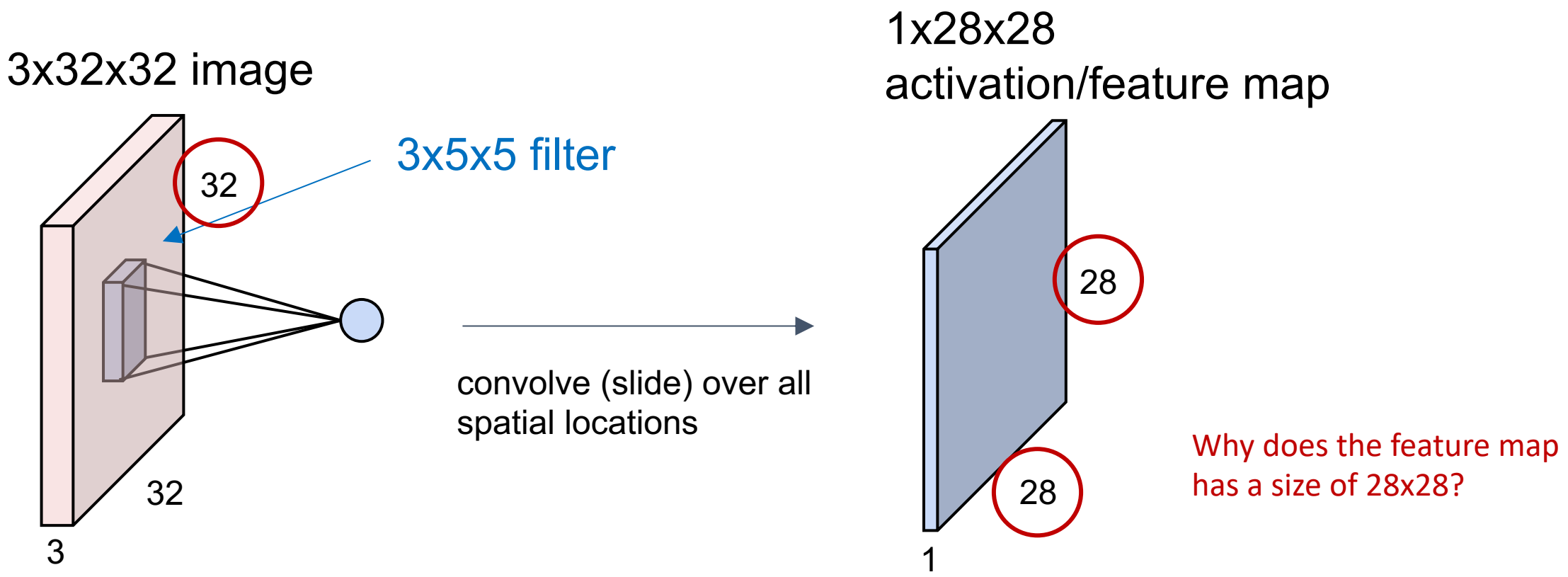
Convolution layer – visualization

3 channels



example 3x5x5 filters
(32 filters in total)

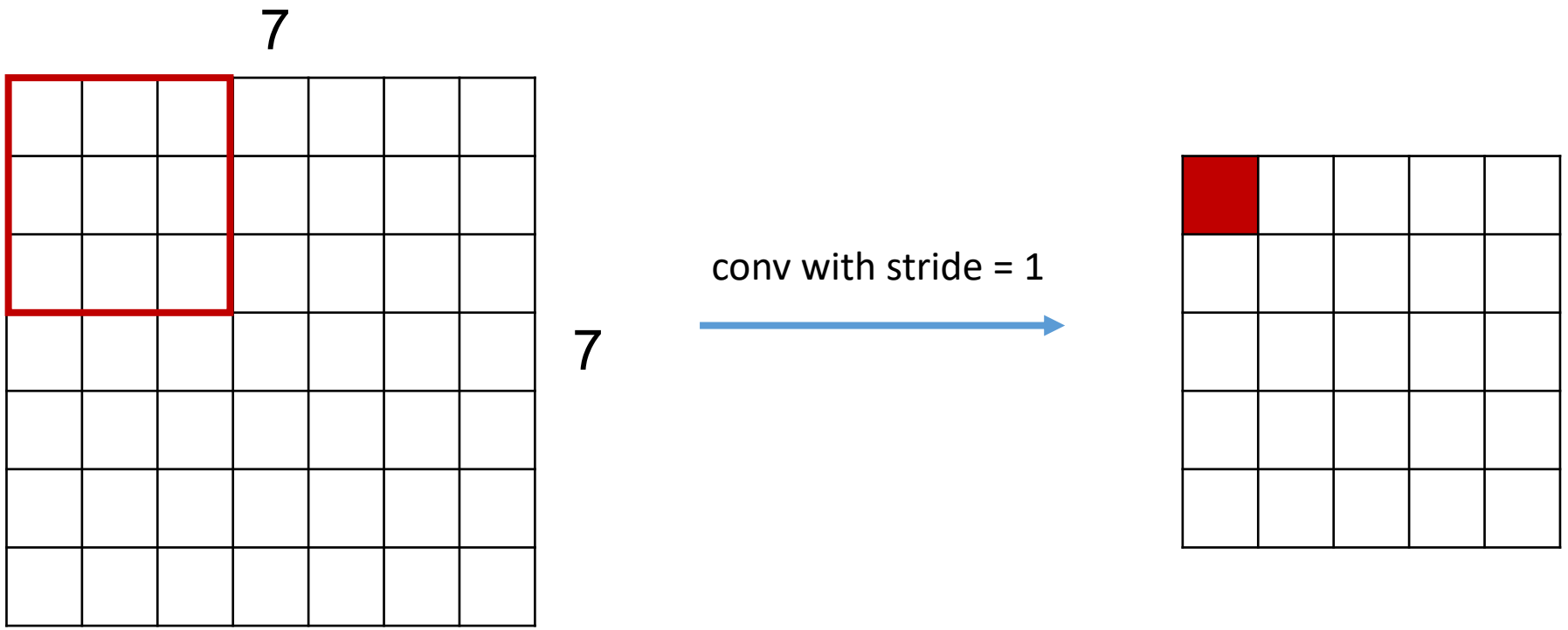
Convolution layer – spatial dimensions



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

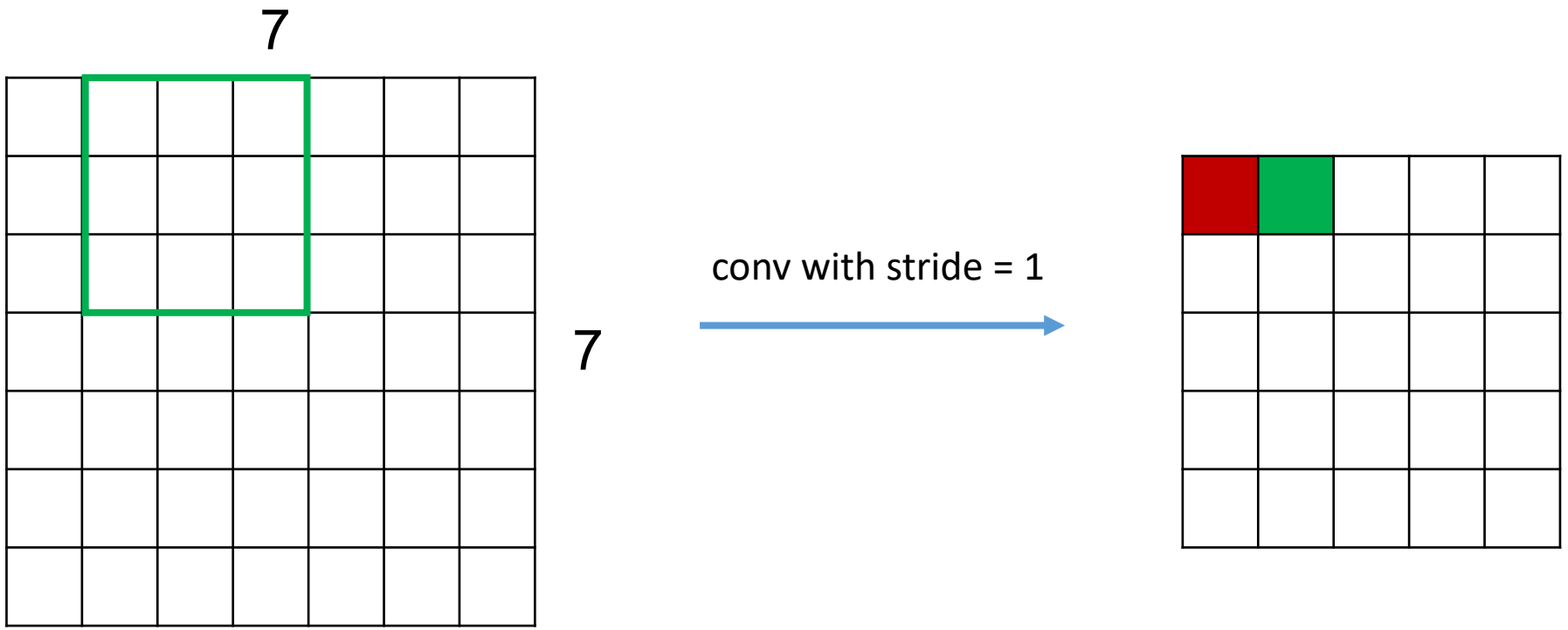
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

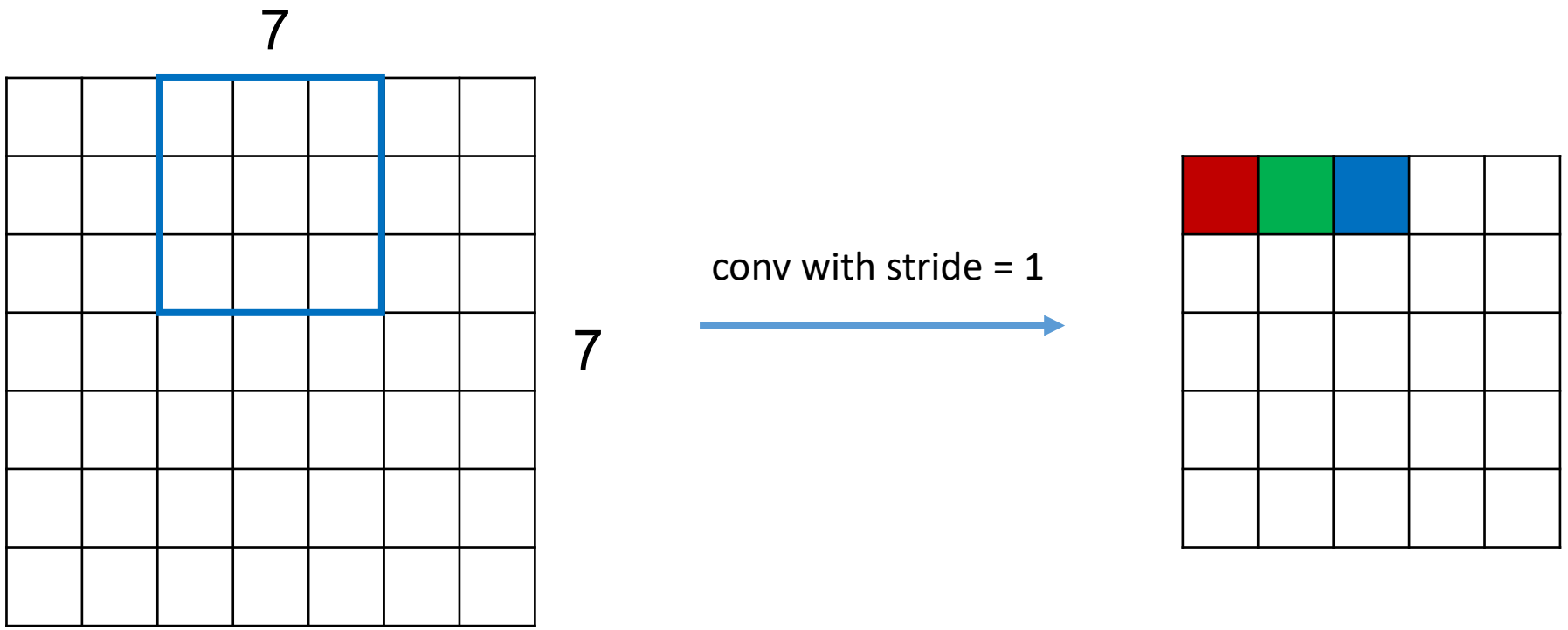
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

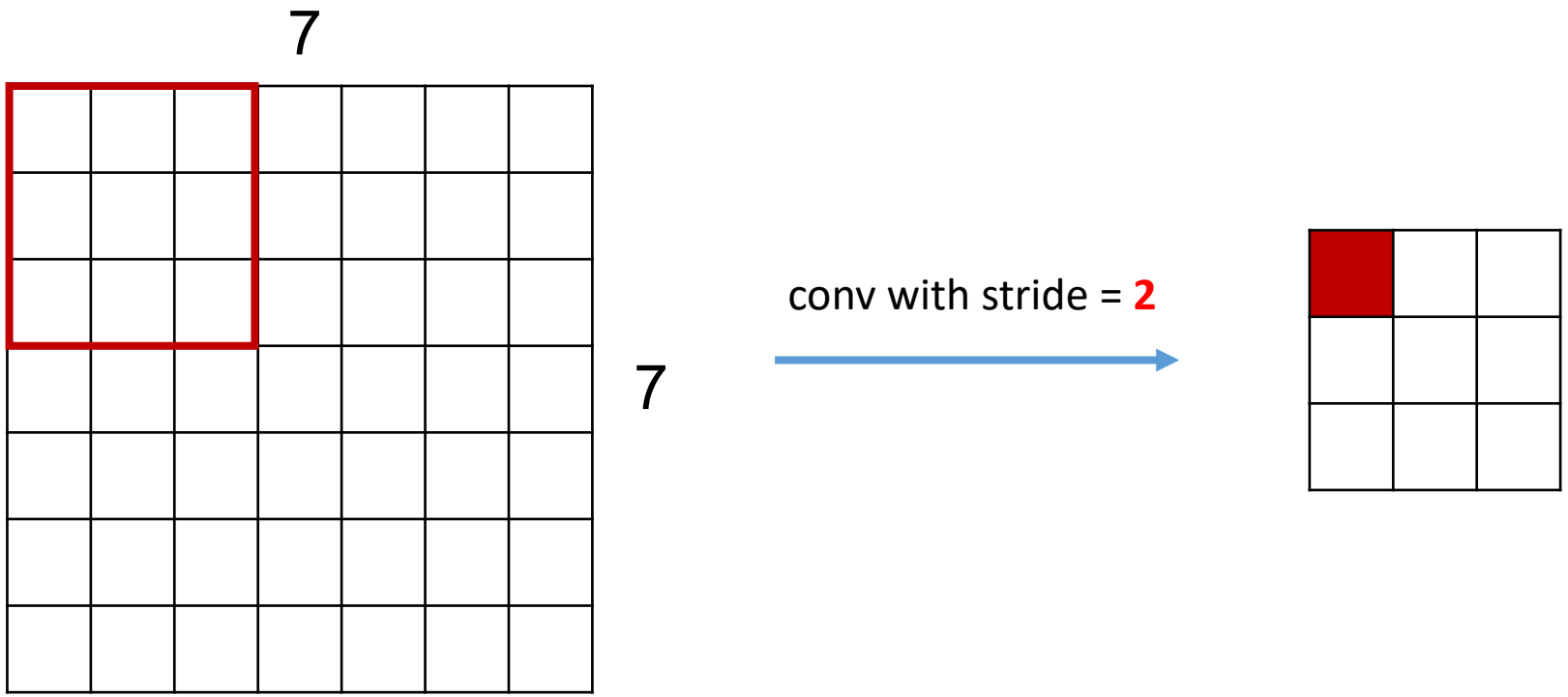
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

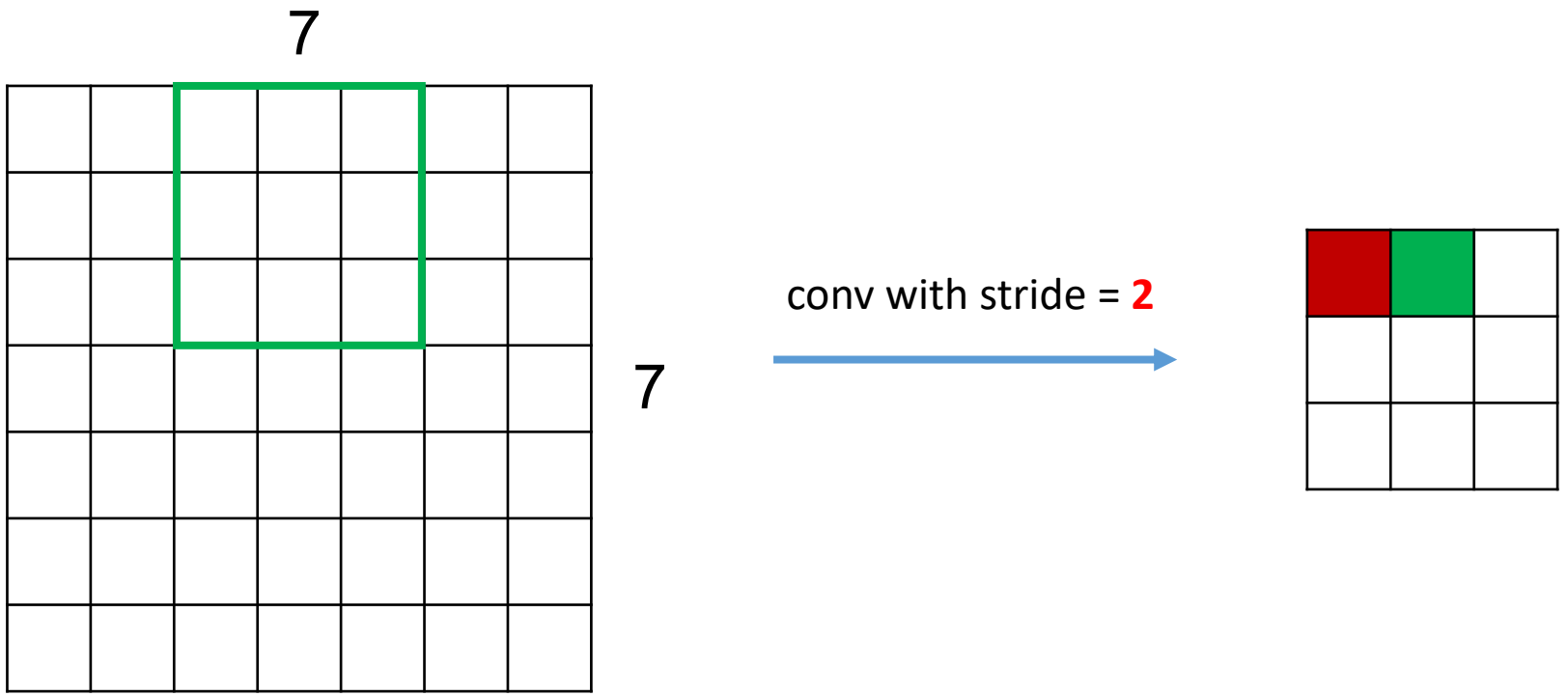
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

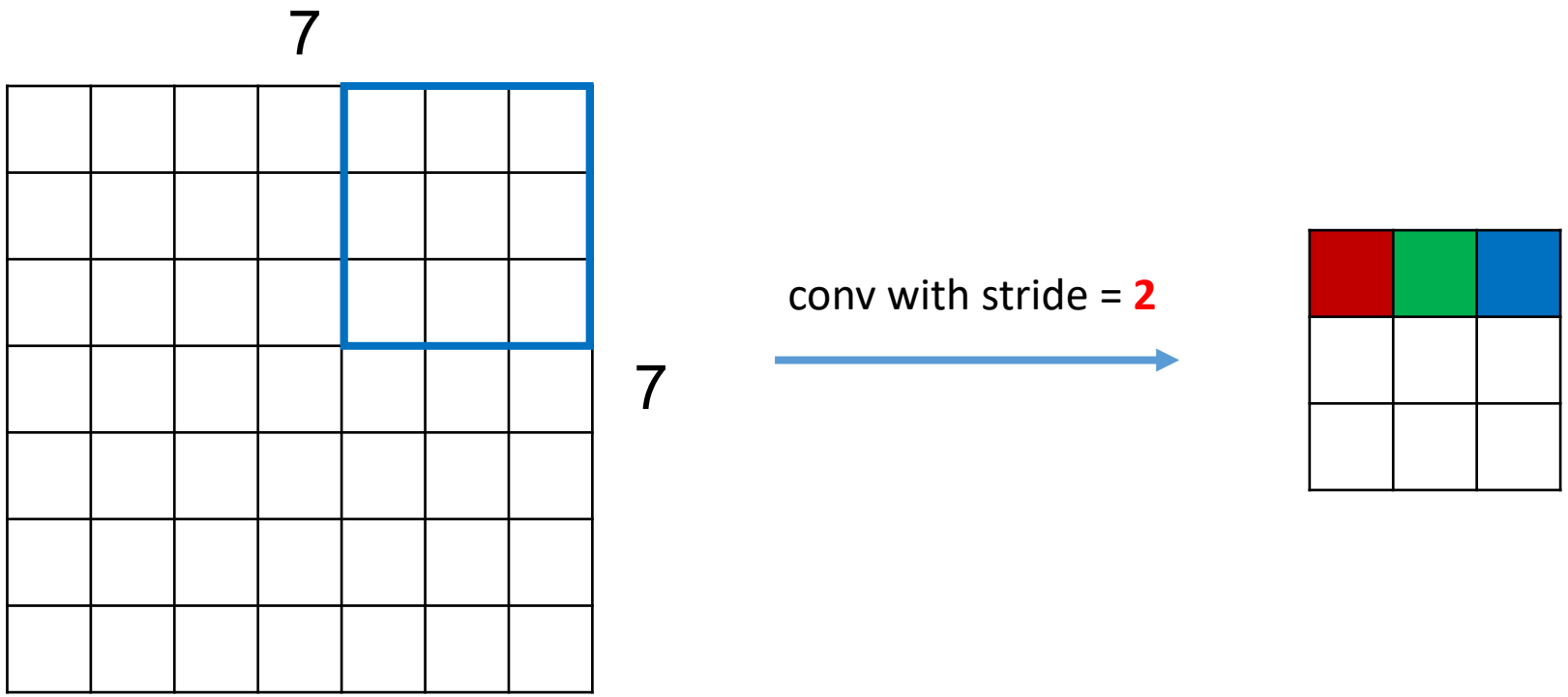
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



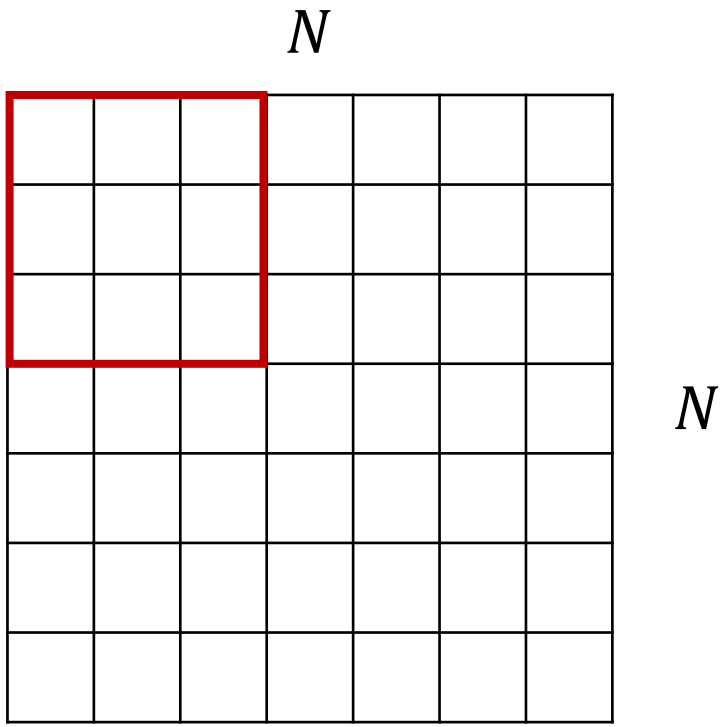
Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions



$N \times N$ input (spatially), assume $F \times F$ filter, and S stride

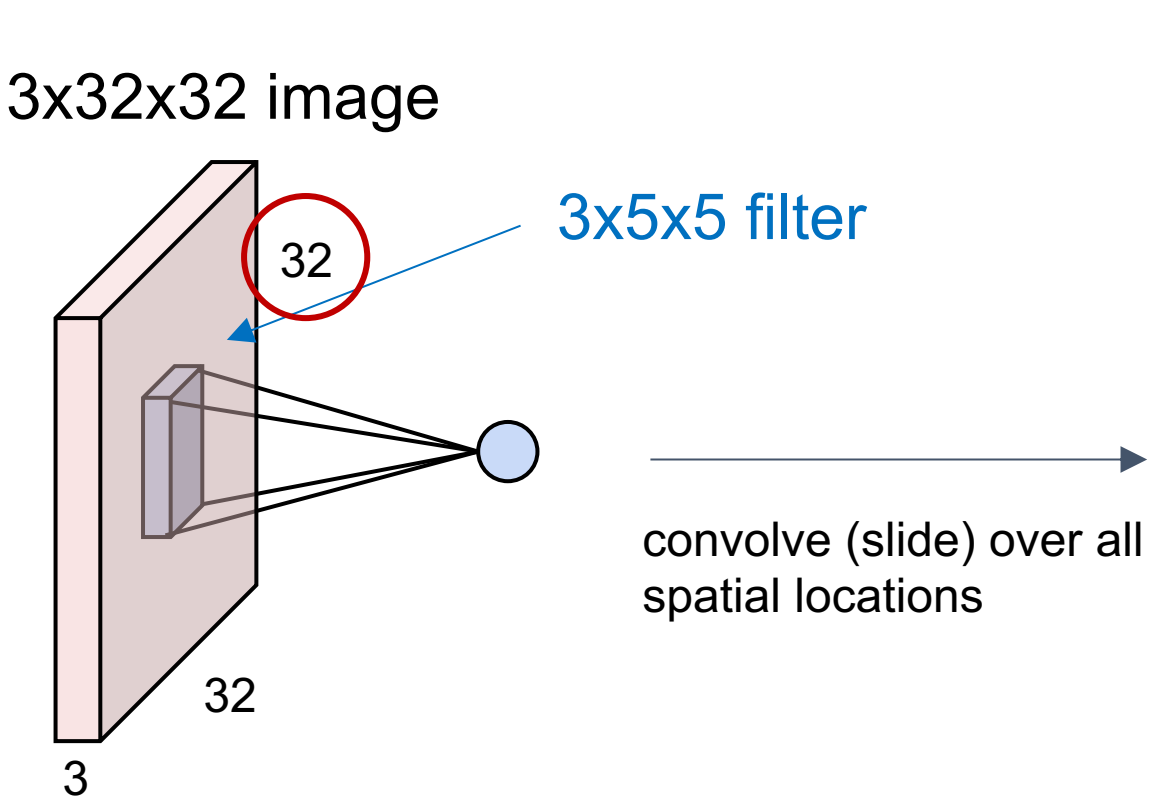
Output size = $\frac{N-F}{S} + 1$

e.g.

$$N = 7, F = 3$$

stride 1 => $(7 - 3)/1 + 1 = 5$
stride 2 => $(7 - 3)/2 + 1 = 3$
stride 3 => $(7 - 3)/3 + 1 = 2.33$:\

Convolution layer – spatial dimensions



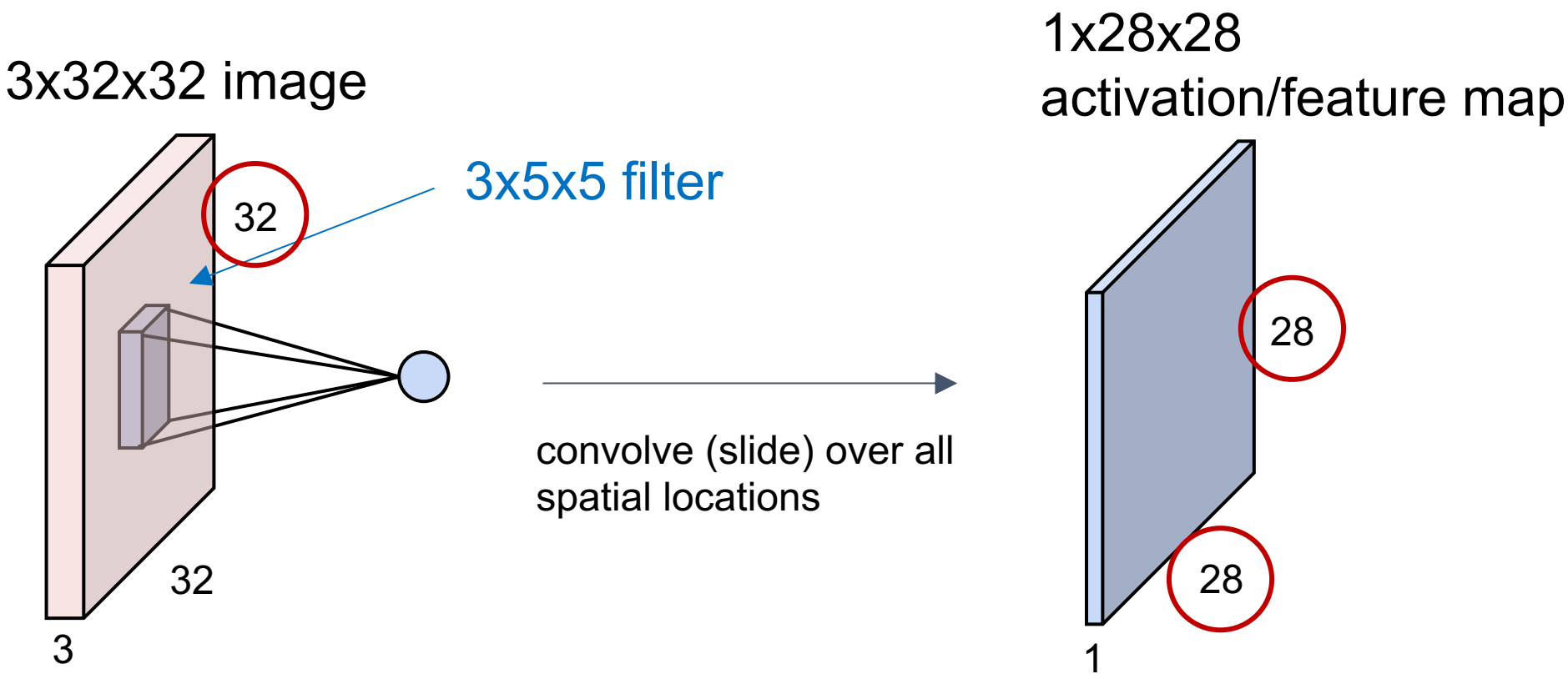
Why does the feature map has a size of 28x28?

Output size = $\frac{N-F}{S} + 1$

$N = 32, F = 5$

stride 1 => $(32 - 5)/1 + 1 = 28$

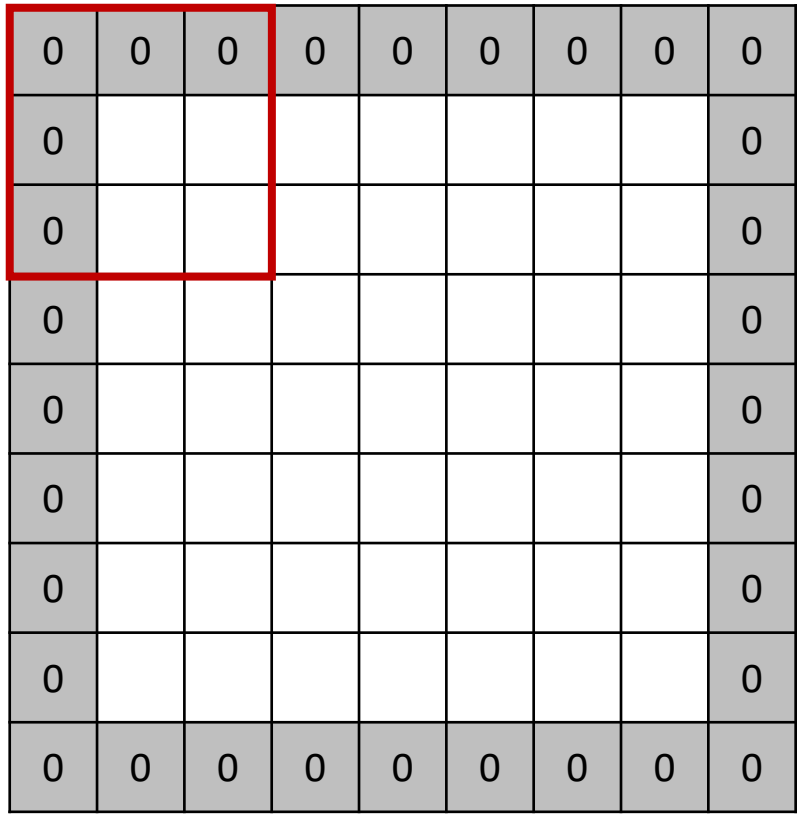
Convolution layer – spatial dimensions



The valid feature map is smaller than the input after convolution.
Without zero-padding, the width of the representation shrinks by the $F - 1$ at each layer
To avoid shrinking the spatial extent of the network rapidly, small filters have to be used

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized). Tensorflow pad zeros at the bottom and at the right.



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

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized). Tensorflow pad zeros at the bottom and at the right.

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

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

Recall that without padding, output size = $\frac{N-F}{S} + 1$

With padding, output size = $\frac{N-F+2P}{S} + 1$

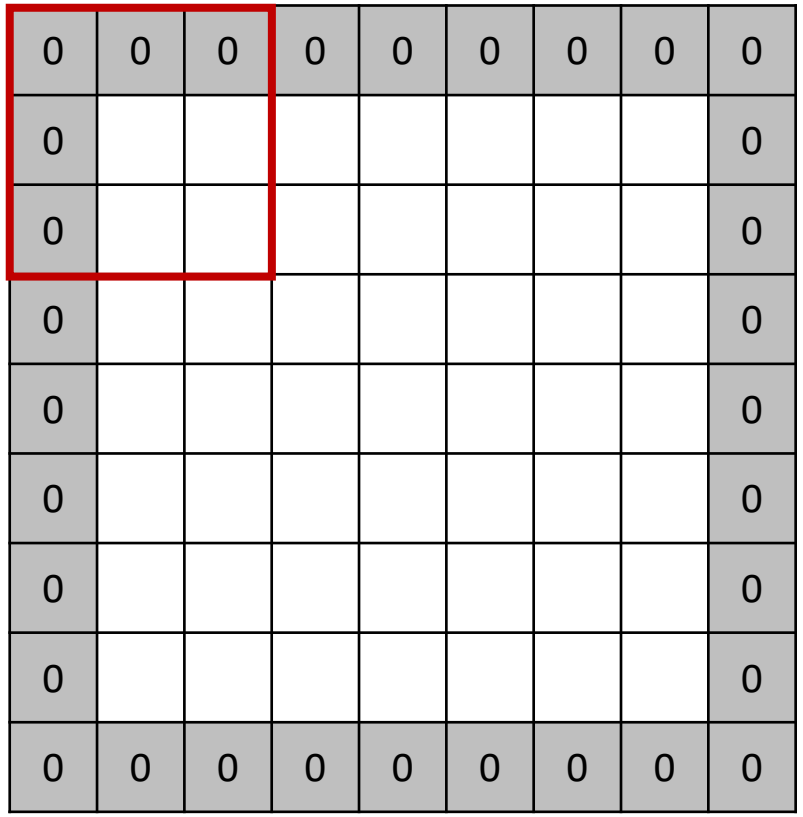
e.g.

$$N = 7, F = 3$$

stride 1 => $(7 - 3 + 2(1))/1 + 1 = 7$

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized). Tensorflow pad zeros at the bottom and at the right.



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

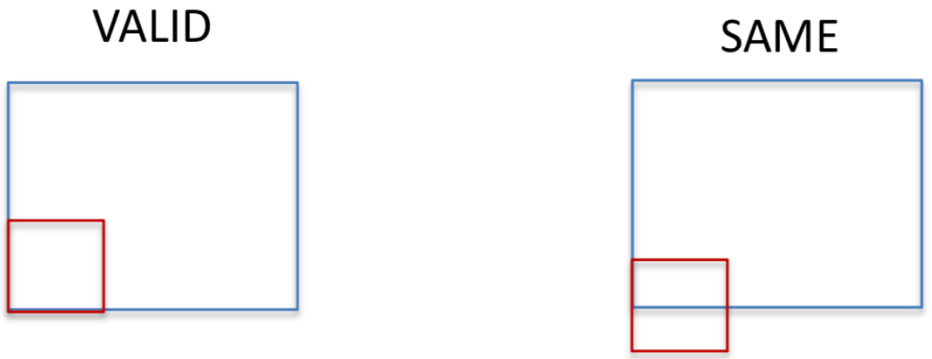
7×7 output!

In general, common to see CONV layers with stride 1, filters of size $F \times F$, 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

Zero-padding in Tensorflow

- There are two ways to handle the boundary:
  1. VALID: apply the filter wherever it completely overlaps with the input.
  2. SAME: apply the filter to make the output same size as input. The input is padded with zeros in obtaining the output.



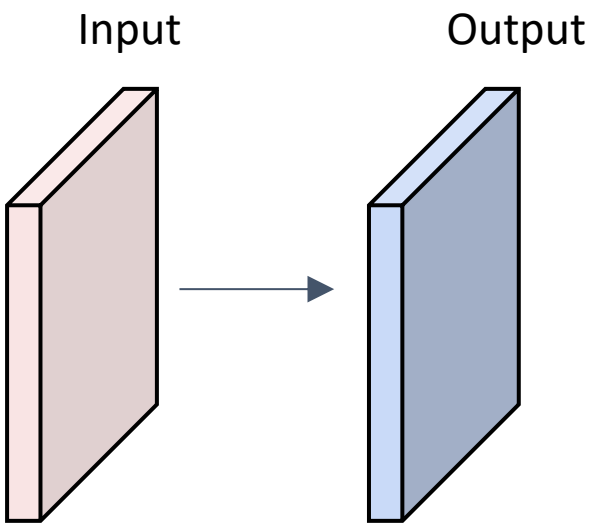
If input image is $I \times J$, and kernel size is $L \times M$, then the size of the output layer:
VALID: $(I - L + 1) \times (J - M + 1)$
SAME: $I \times J$

In Tensorflow, cases 'VALID' and 'SAME' are options for padding with convolution and pooling. Note that the input is scanned from top-left corner to right and down.

Example

Input volume: $3 \times 32 \times 32$
Ten $3 \times 5 \times 5$ filters with stride 1, pad 2

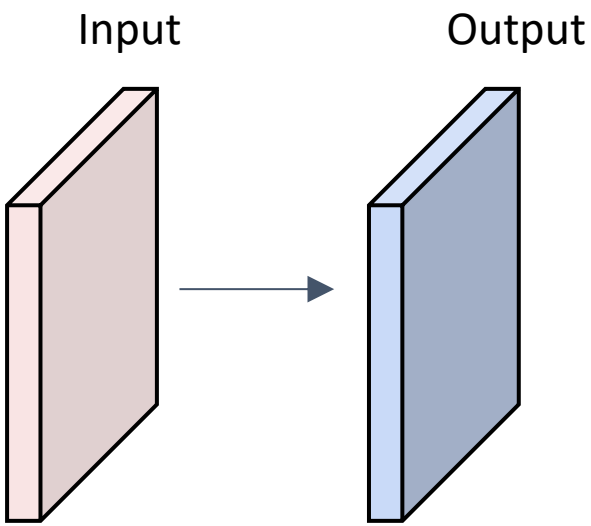
Output volume size: ?



Example

Input volume: $3\times32\times32$
Ten $3\times5\times5$ filters with stride 1, pad 2

Output volume size: ?



Output size =

$$\frac{N-F+2P}{S}+1$$

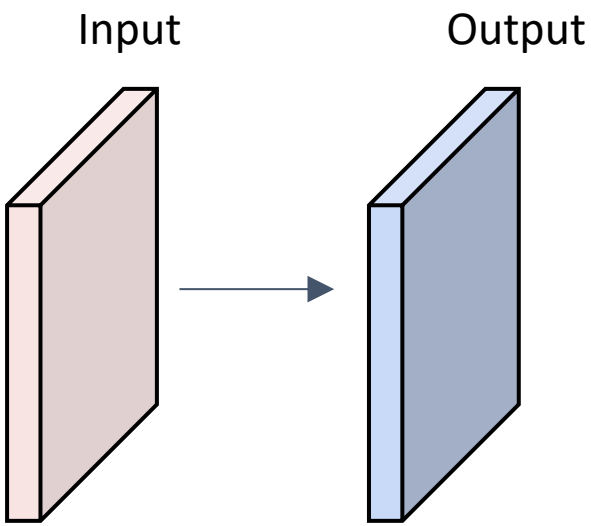
$$\frac{32-5+2(2)}{1}+1 = 32 \text{ spatially}$$

The output volume size is $10\times32\times32$

Example

Input volume: $3 \times 32 \times 32$
Ten $3 \times 5 \times 5$ filters with stride 1, pad 2

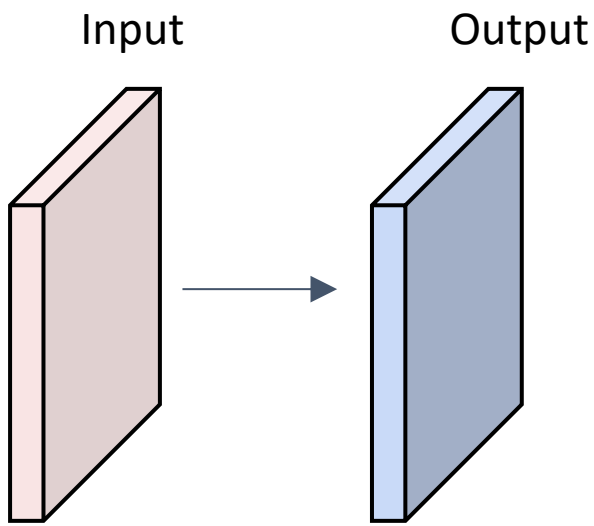
Number of parameters in this layer: ?



Example

Input volume: $3 \times 32 \times 32$
Ten $3 \times 5 \times 5$ filters with stride 1, pad 2

Number of parameters in this layer: ?



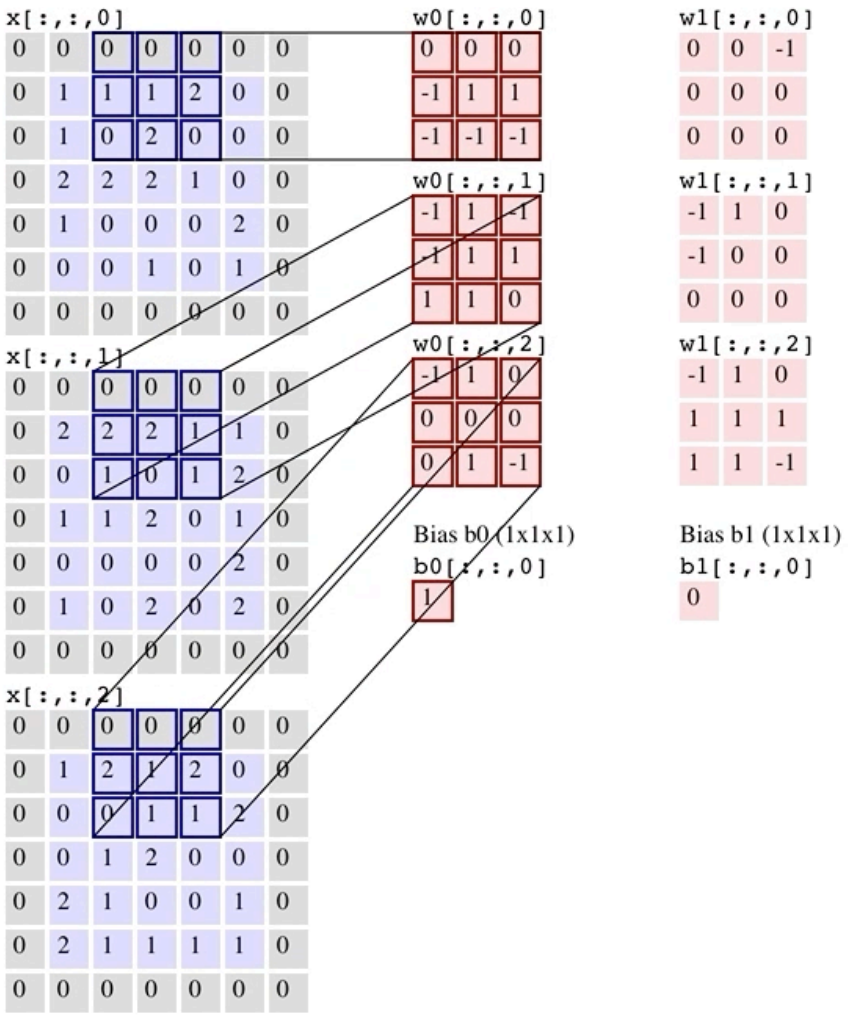
Each filter has $5 \times 5 \times 3 + 1 = 76$ params (+1 for bias)
=> $76 \times 10 = \mathbf{760}$

Convolution layer – a numerical example

Input volume (+pad 1)
$C \times I \times I = 3 \times 7 \times 7$

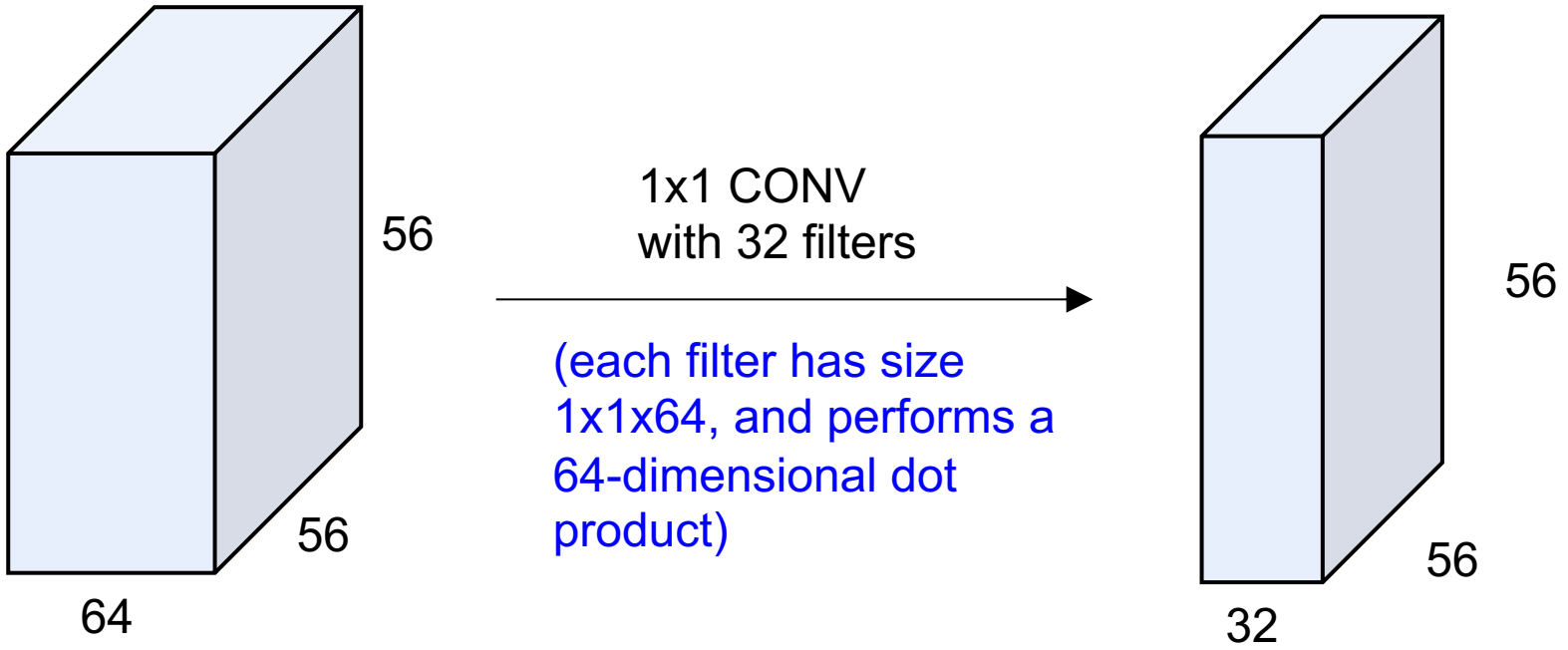
Filters W0, W1 volume
$C \times F \times F = 3 \times 3 \times 3$

Output volume
$C \times O \times O = 2 \times 3 \times 3$



- $C = 3$ channel inputs
- Two filters of size $F \times F = 3 \times 3$, and they are applied with a stride of 2.
- Each element of the output activations (green) is computed by elementwise multiplying the highlighted input (blue) with the filter (red), summing it up, and then offsetting the result by the bias.

Can we have 1x1 filter?



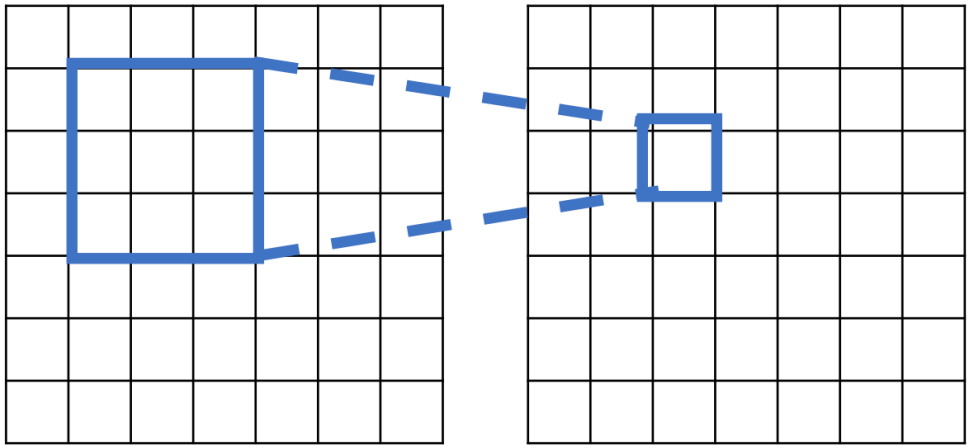
Dimension is reduced!

Receptive field

Sparse local connectivity - CNNs exploit spatially local correlations by enforcing local connectivity between neurons of adjacent layers.

Note that the receptive fields of the neurons are limited because of local connectivity.

For convolution with kernel size K, each element in the output depends on a $K \times K$ receptive field in the input



Receptive field

- Receptive field can be briefly defined as the region in the input space that a particular CNN's feature is looking at
- The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers
- This effect increases if the network include strided convolution or pooling
- Even though direct connections in a CNN are very sparse, units in the deeper layers can be indirectly connected to all or most of the input image

Receptive field

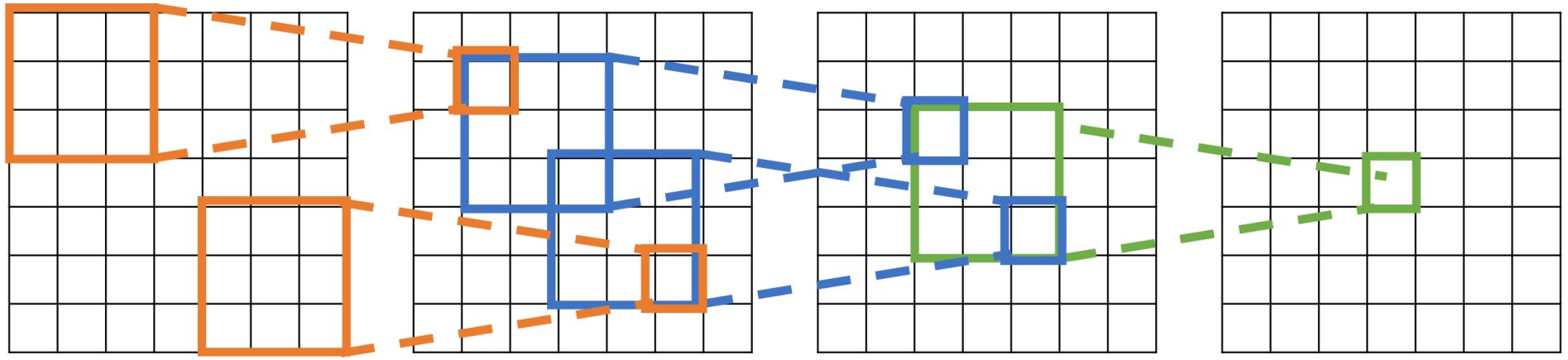
The receptive field at layer k is the area denoted $R_k \times R_k$ of the input that each pixel of the k-th activation map can 'see'.

By calling F_j the filter size of layer j and S_i the stride value of layer i and with the convention $S_0 = 1$, the receptive field at layer k can be computed with the formula:

$$R_k = 1 + \sum_{j=1}^{k} (F_j - 1) \prod_{i=0}^{j-1} S_i$$

Receptive field

Example



Input

Output

$$R_k = 1 + \sum_{j=1}^{k} (F_j - 1) \prod_{i=0}^{j-1} S_i$$

Filter size $F_1 = F_2 = F_3 = 3$

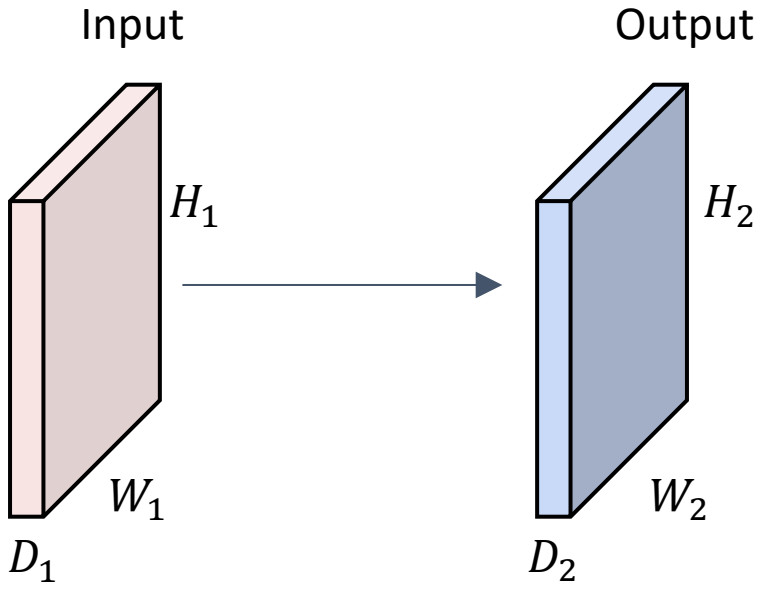
Stride $S_1 = S_2 = S_3 = 1$

$$R_3 = 1 + \sum_{j=1}^{3} (F_j - 1) \prod_{i=0}^{j-1} S_i = 1 + (2\times1) + (2\times1) + (2\times1) = 7$$

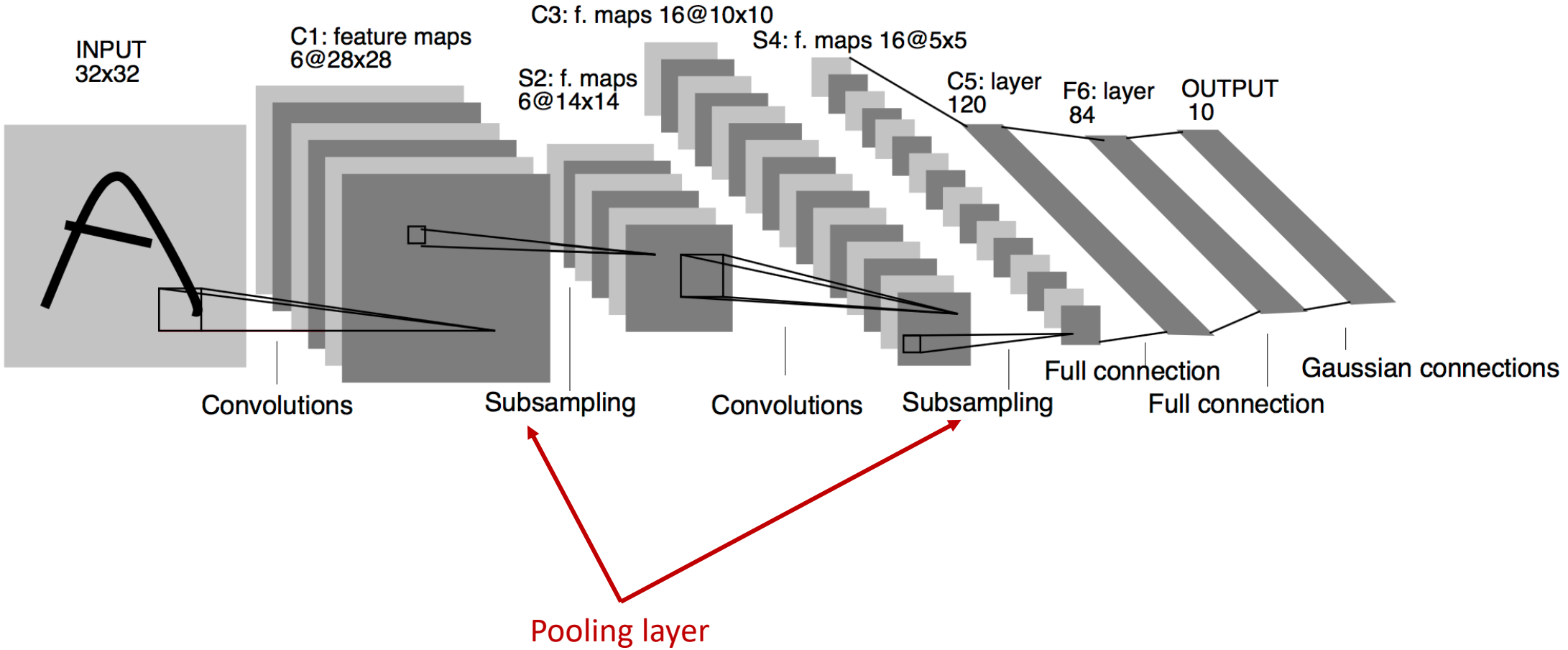
Convolution layer - summary

A convolution layer

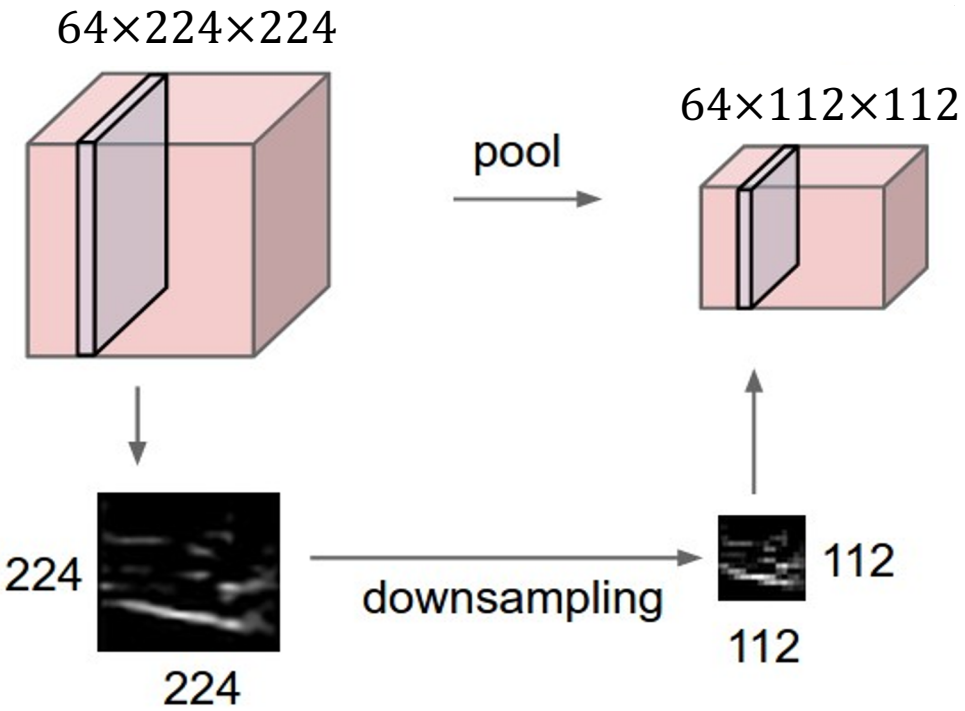
- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters
  - Number of filters K
  - Their spatial extent F
  - The stride S
  - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (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 $H_2 \times W_2$) is the result of a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias



An example of convolutional network: LeNet 5

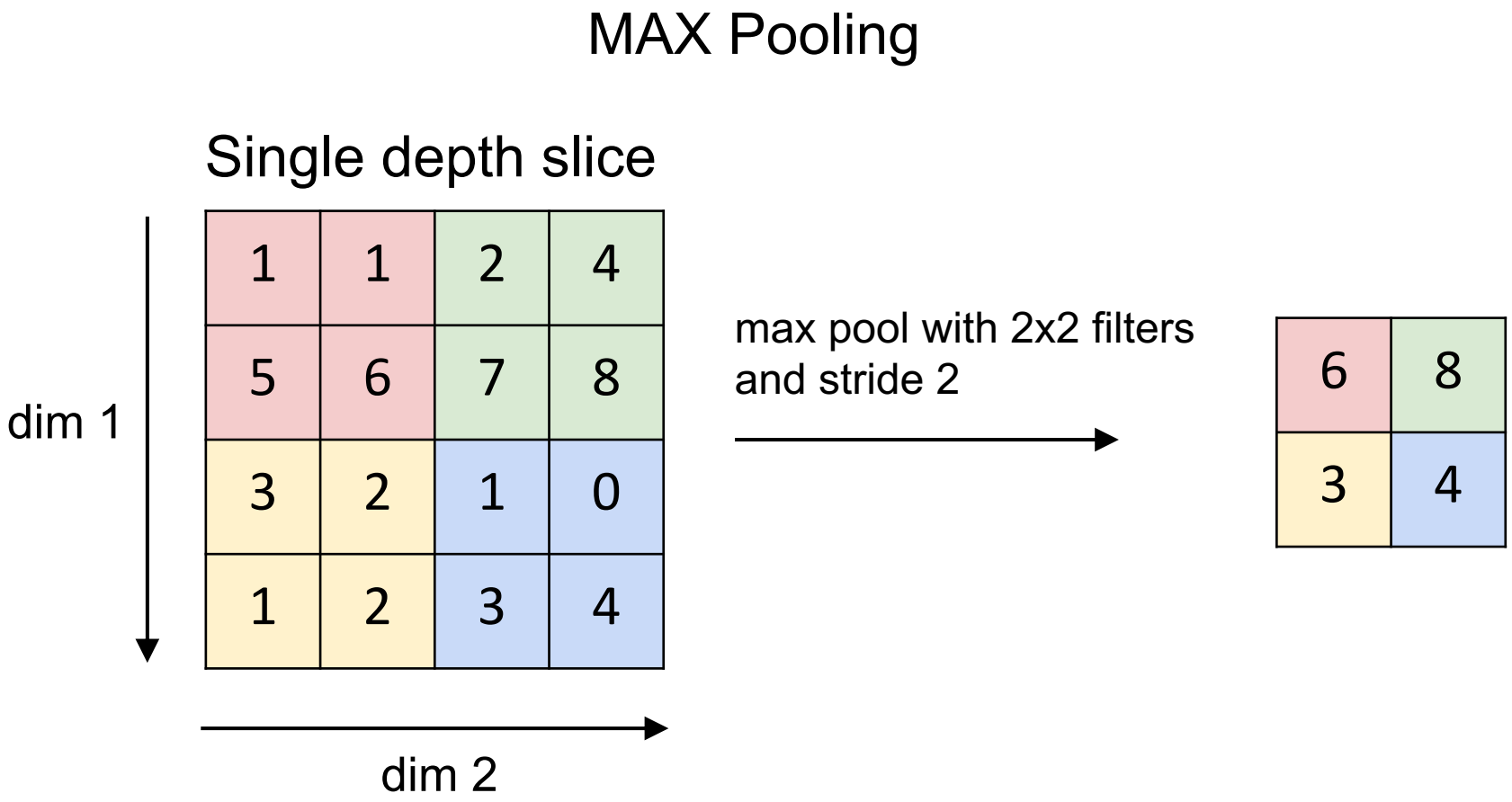


Pooling layer



- Operates over each activation map independently
- Either ‘max’ or ‘average’ pooling is used at the pooling layer. That is, the convolved features are divided into disjoint regions and pooled by taking either maximum or averaging.

Pooling layer



Pooling is intended to subsample the convolution layer.
The default stride for pooling is equal to the filter width.

Pooling layer

Consider pooling with non-overlapping windows $\{(l,m)\}_{l,m=-L/2,-M/2}^{L/2,M/2}$, of size $L \times M$

The max pooling output is the maximum of the activation inside the pooling window. Pooling of a feature map $\boldsymbol{y}$ at $p = (i,j)$ produce pooled feature

$$z(i,j) = \max_{l,m}\{y(i+l, j+m)\}$$

The mean pooling output is the mean of activations in the pooling window

$$z(i,j) = \frac{1}{L \times M} \sum_{l} \sum_{m} y(i+l, j+m)$$

Pooling layer

Why pooling?

A function f is invariant to g if $f(g(\boldsymbol{x})) = f(\boldsymbol{x})$.

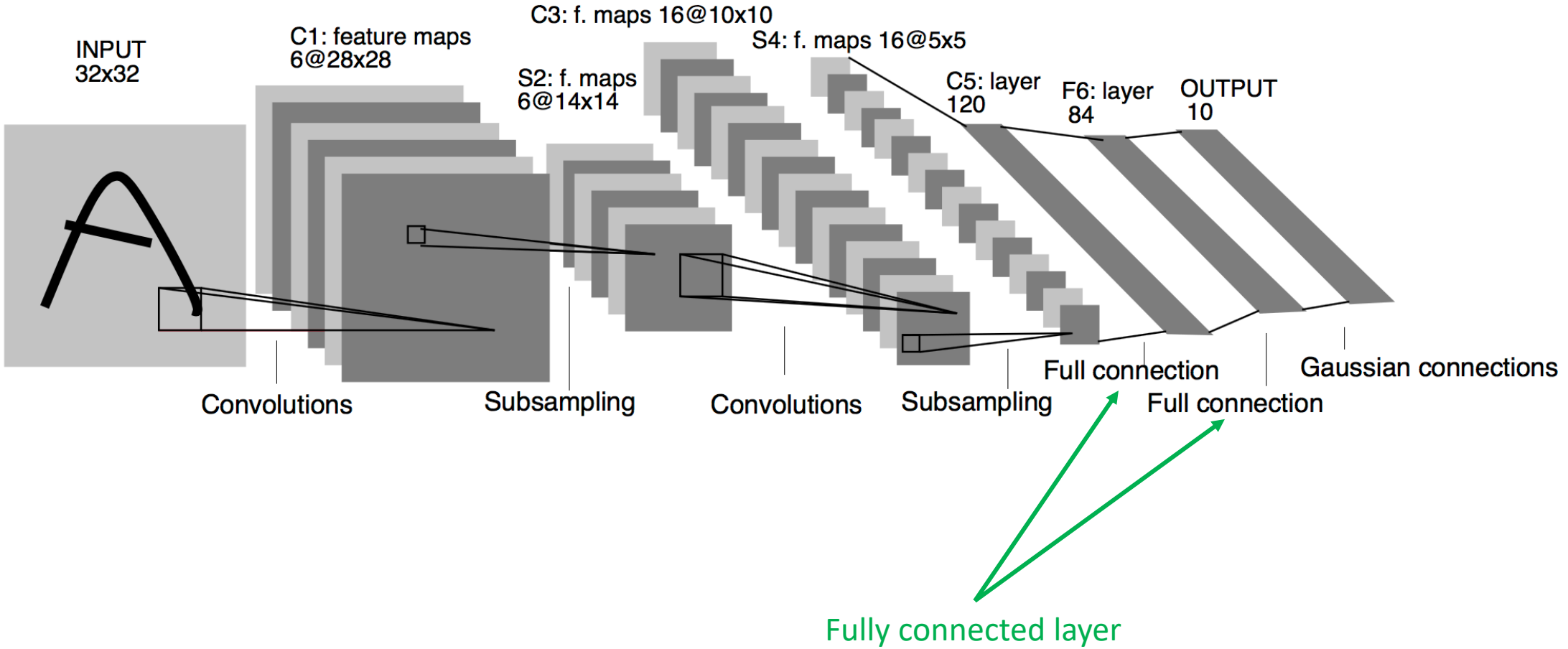
- Pooling layers can be used for building inner activations that are (slightly) invariant to small translations of the input.
- Invariance to local translation is helpful if we care more about the presence of a pattern rather than its exact position.

Pooling layer - summary

A pooling layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires two hyperparameters
  - Their spatial extent F
  - The stride S
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for pooling layers

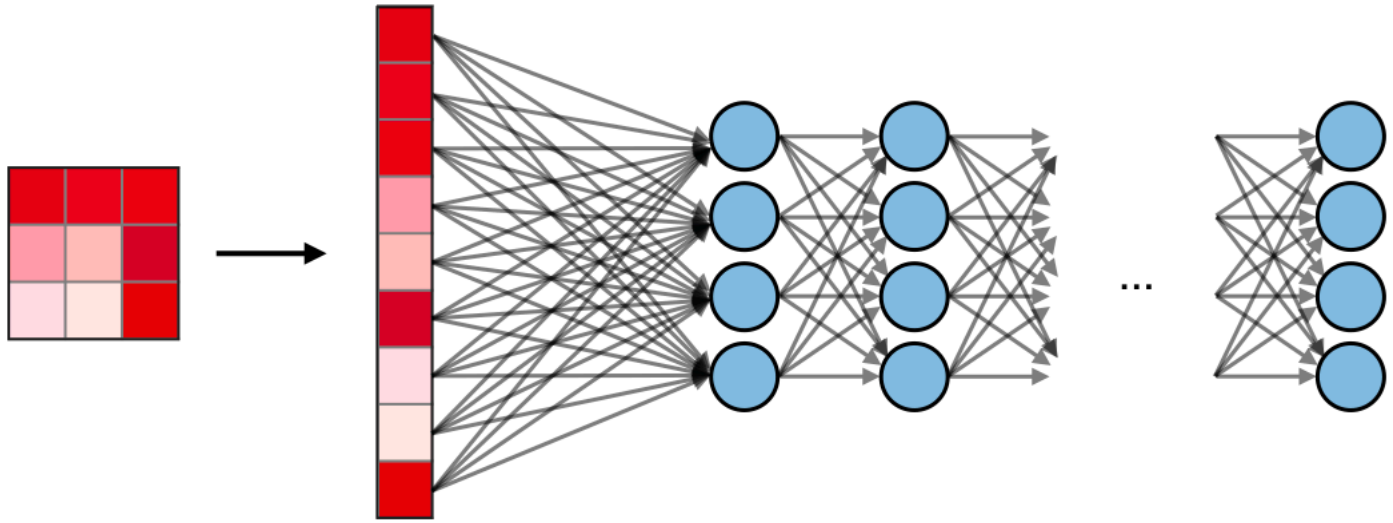
An example of convolutional network: LeNet 5



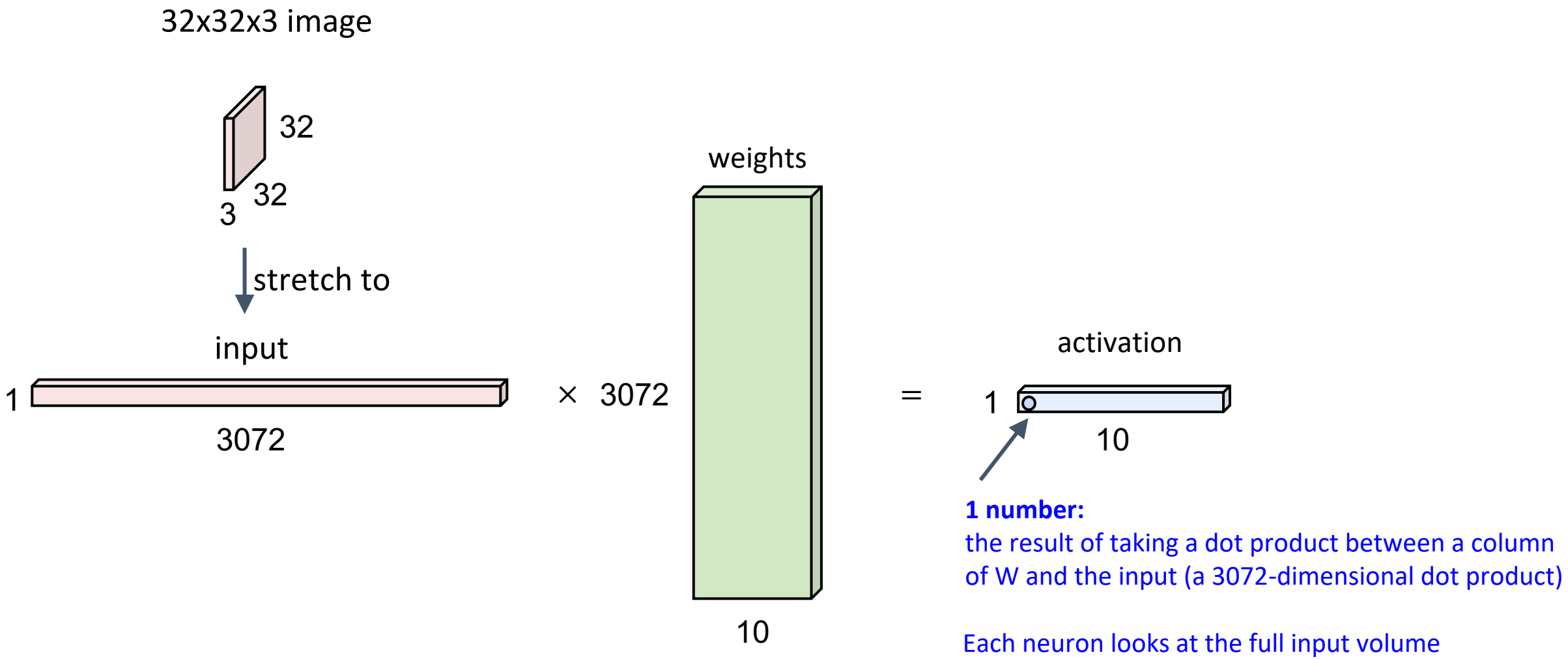
Fully connected layer

The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons.

If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.



Fully connected layer

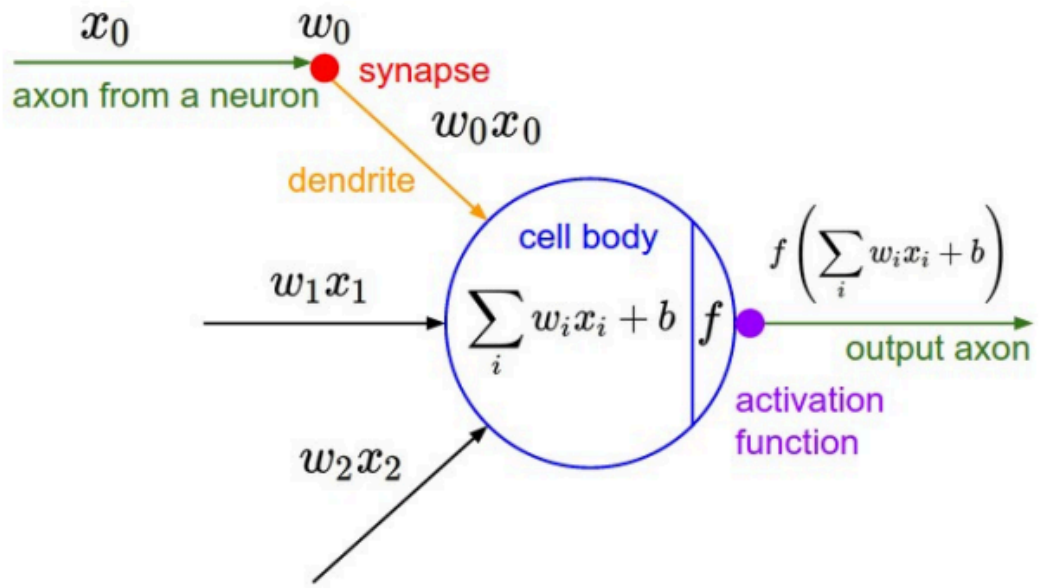


Activation function

Recall that the output of the neuron at (i, j) of the convolution layer

$$y(i,j) = f(u(i,j))$$

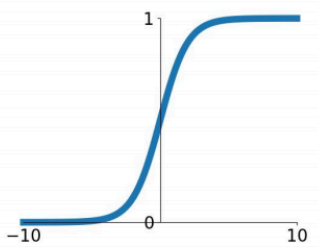
where u is the synaptic input and f is an activation function (It aims at introducing non-linearities to the network.).



Activation function

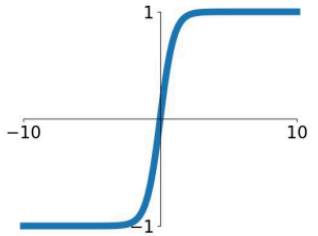
Sigmoid

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



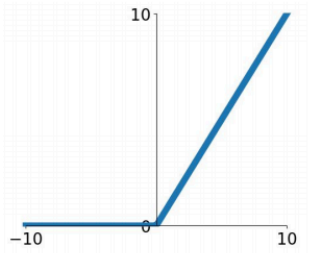
tanh

$$\tanh(x)$$



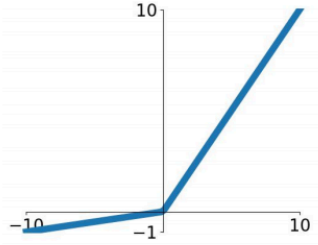
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

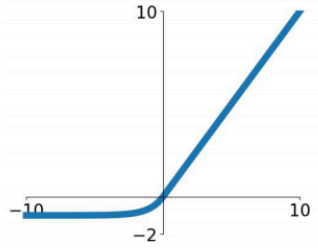


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Example 1

Given an input pattern $\boldsymbol{X}$:

$$\boldsymbol{X} = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.3 & -0.2 \\ 0.7 & 0.1 & 0.2 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}$$

The input pattern is received by a convolution layer consisting of one kernel (filter)

$\boldsymbol{w} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$ and bias = 0.05.

If convolution layer has a sigmoid activation function, find the outputs of the convolution layer if the padding is VALID at strides = 1.

If the pooling layer uses max pooling, has a pooling window size of 2x2, and strides = 2, find the activations at the pooling layer for VALID and SAME padding.

Example 1

$$\boldsymbol{I} = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.3 & -0.2 \\ 0.7 & 0.1 & 0.2 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}; \boldsymbol{w} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Synaptic input to the pooling-layer:

$$u(i,j) = \sum_l \sum_m x(i+l, j+m) w(l,m) + b$$

For VALID padding:

$u(1,1) = 0.5\times0 - 0.1\times1 + 0.2\times1 + 0.8\times1 + 0.1\times0 - 0.5\times1 - 1.0\times1 + 0.2\times1 + 0.0\times0 + 0.05 = -0.35$

$u(1,2) = -0.1\times0 + 0.2\times1 + 0.3\times1 + 0.1\times1 - 0.5\times0 + 0.5\times1 + 0.2\times1 + 0.0\times1 + 0.3\times0 + 0.05 = 1.35$

$u(1,3) = 0.2\times0 + 0.3\times1 + 0.5\times1 - 0.5\times1 + 0.5\times0 + 0.1\times1 + 0.0\times1 + 0.3\times1 - 0.2\times0 + 0.05 = 0.75$

$u(2,1) = 0.8\times0 + 0.1\times1 - 0.5\times1 - 0.1\times1 + 0.2\times0 + 0.0\times1 + 0.7\times1 + 0.1\times1 + 0.2\times0 + 0.05 = -0.55$

⋮

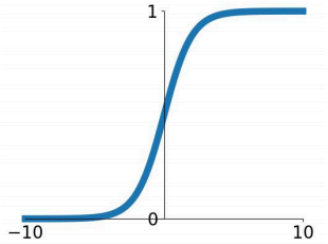
Example 1

Synaptic input to the convolution layer:

$$\boldsymbol{U} = \begin{pmatrix} -0.35 & 1.35 & 0.75 \\ -0.55 & 0.85 & 0.05 \\ 0.75 & 0.05 & 1.15 \end{pmatrix}$$

Sigmoid

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



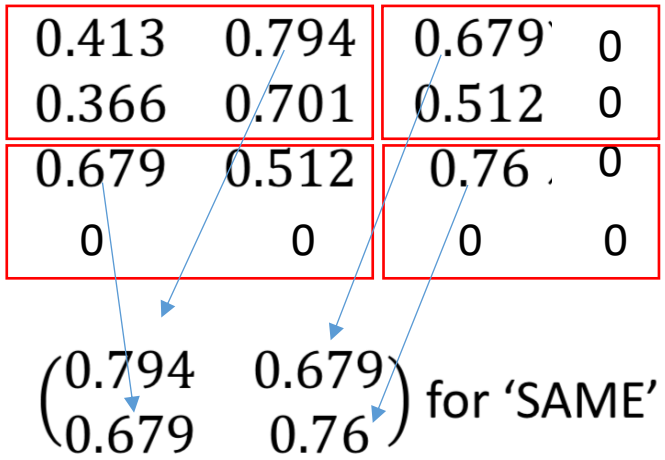
Output of the convolution layer:

$$f(\boldsymbol{U}) = \frac{1}{1+e^{-\boldsymbol{U}}} = \begin{pmatrix} 0.413 & 0.794 & 0.679 \\ 0.366 & 0.701 & 0.512 \\ 0.679 & 0.512 & 0.76 \end{pmatrix}$$

Output of the max pooling layer:

(0.794) for 'VALID' pooling

$\begin{pmatrix} 0.794 & 0.679 \\ 0.679 & 0.76 \end{pmatrix}$ for 'SAME' padding



See eg7.1.ipynb

Example 2

Inputs are digit images from MNIST database: http://yann.lecun.com/exdb/mnist/

Input image size = 28x28

First convolution layer consists of three filters $w_1 = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$, $w_2 = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$, $w_3 = \begin{pmatrix} 3 & 4 & 3 \\ 4 & 5 & 4 \\ 3 & 4 & 3 \end{pmatrix}$

Find the feature maps at the the convolution layer and pooling layer. Assume zero bias

For convolution layer, use a stride = 1 (default) and padding = 'VALID'.
For pooling layer, use a window of size 2x2 and a stride if 2 and 'VALID' padding.

Output size of convolution layer: 3x26x26
Output size of pooling layer: 3x13x13

See eg7.2.ipynb

Example 2

Original Image
28x28

Convolution layer
3 x 26 x26

Mean-Pooling layer
3 x 13 x 13

Example 2

Program segment:

```
# Load and prepare the MNIST dataset. Convert the samples from integers to floating-point numbers:
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

```
# Set filters
w = np.array([[[-1, 0, 1],[-2, 0, 2], [-1, 0, 1]],
     [[1, 2, 1],[0, 0, 0], [-1, -2, -1]],
     [[3, 4, 3], [4, 5, 4], [3, 4, 3]]]).astype(np.float32).reshape(3,3,1,3)
```

```
# Model definition
class MyModel(Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.seq = models.Sequential()
        self.seq.add(
            layers.Conv2D(filters=3, kernel_size=3, strides=1, padding='VALID', input_shape=[28, 28, 1], use_bias=False))

    def call(self, x):
        x1 = self.seq(x)
        x2 = tf.nn.sigmoid(x1)
        x3 = tf.nn.max_pool(x2, ksize=[1, 2, 2, 1], strides = [1, 2, 2, 1], padding = 'VALID')
        return x1, x3
```

See eg7.2.ipynb

Outline

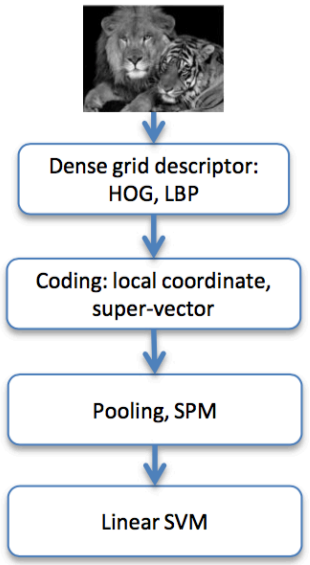
- Applications and success
- Basic components in CNN
- CNN architectures
- Training basics
- Optimizers

CNN Architectures

Deep networks for ImageNet

Year 2010

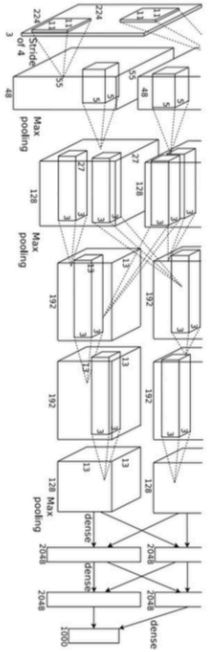
NEC-UIUC



[Lin CVPR 2011]

Year 2012

AlexNet



[Krizhevsky NIPS 2012]

Year 2014

GoogLeNet

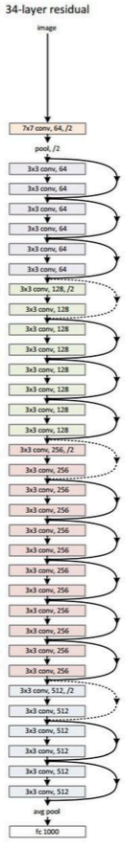
[Szegedy arxiv 2014]

VGG

[Simonyan arxiv 2014]

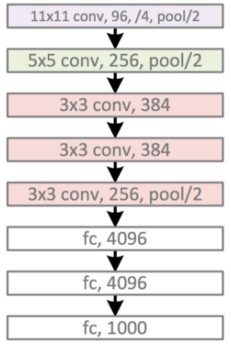
Year 2015

MSRA ResNet

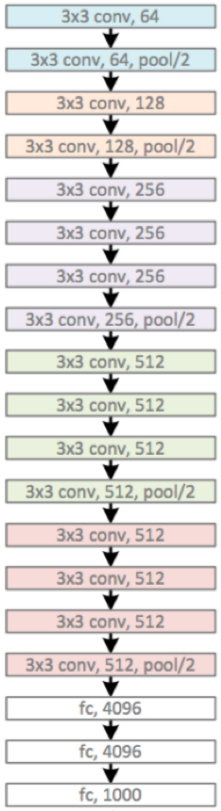


Deep architectures

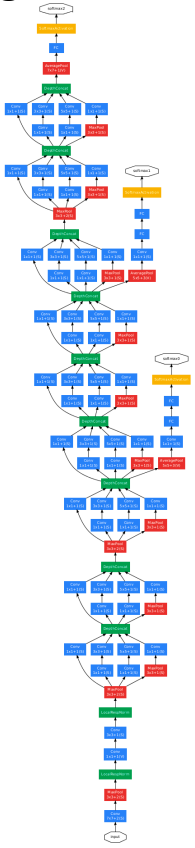
AlexNet 8 layers



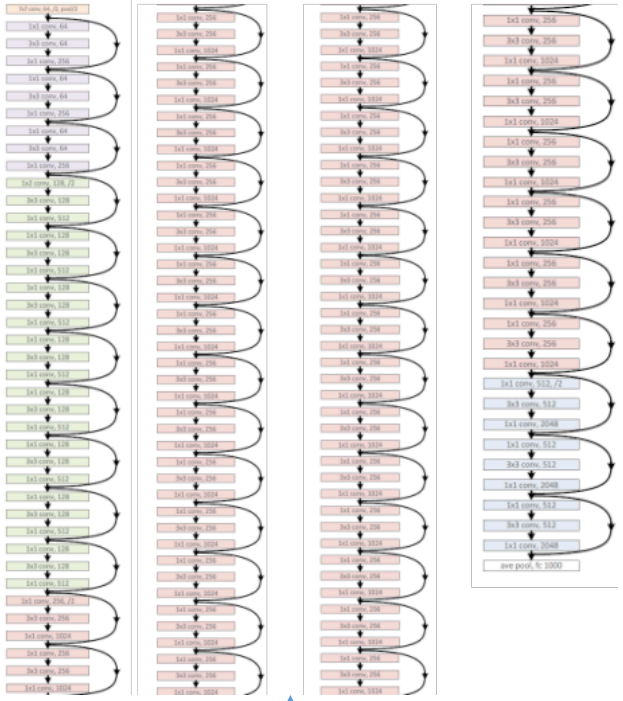
VGG 19 layers



GoogLeNet 22 layers



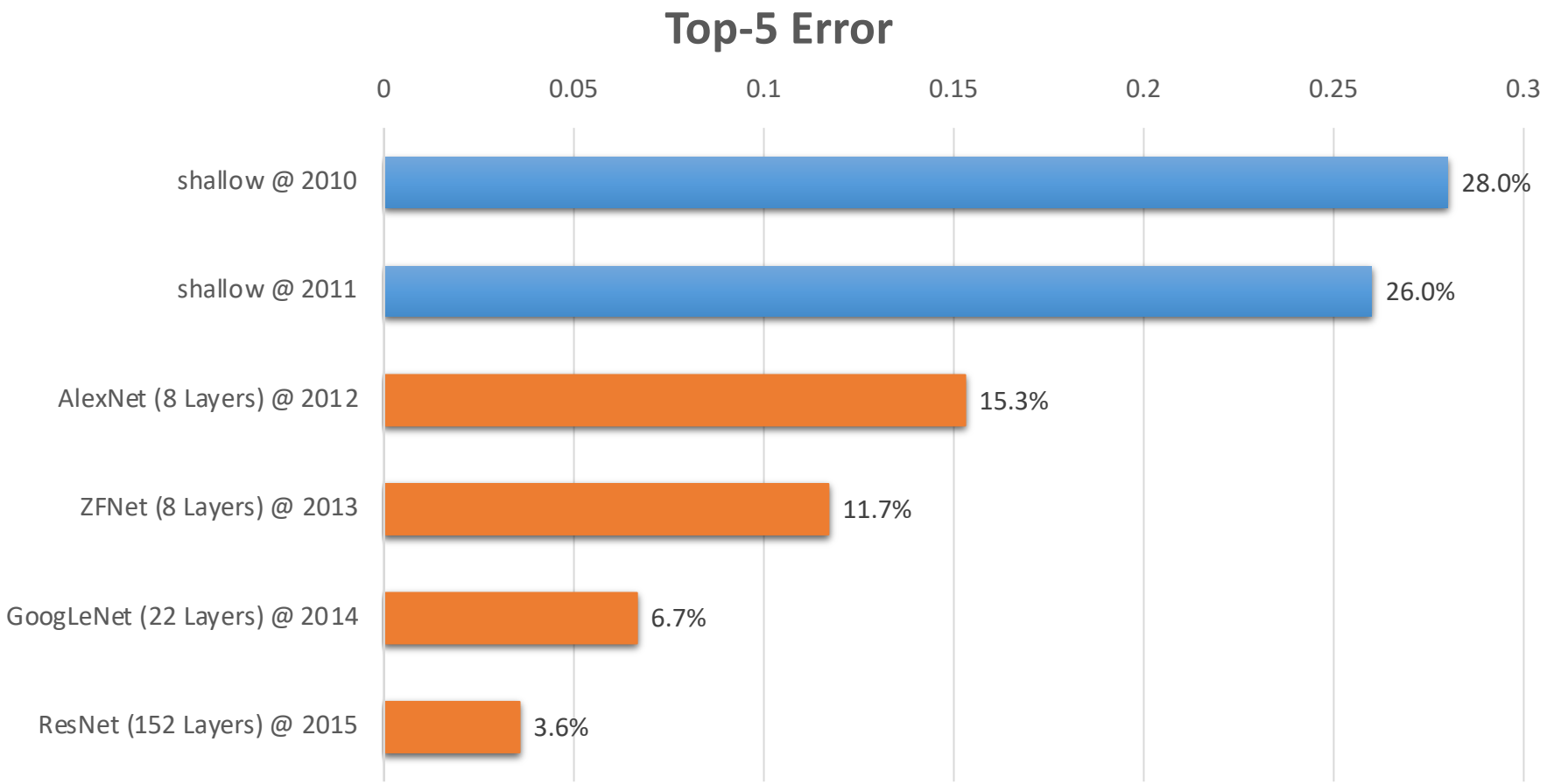
ResNet 152 layers



Deeper
More intricate and different connectivity structures

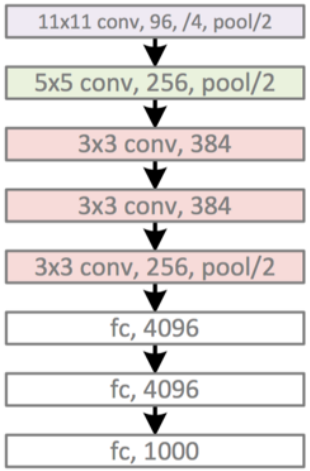
2012 2013 2014 2015 2016 2017 2018 ...

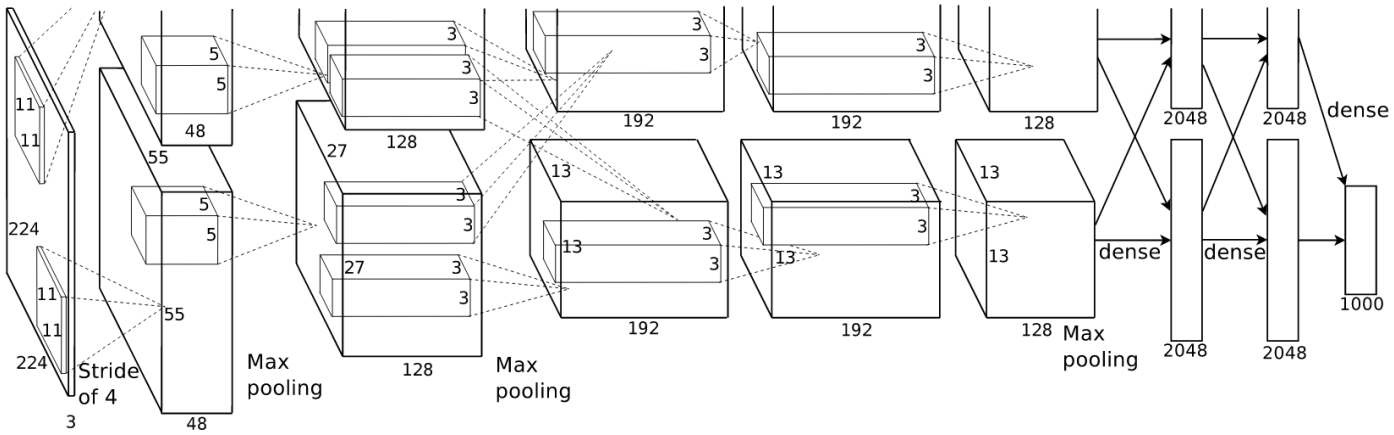
Performance of previous years on ImageNet



Depth is the key to high classification accuracy.

Deep architectures - AlexNet

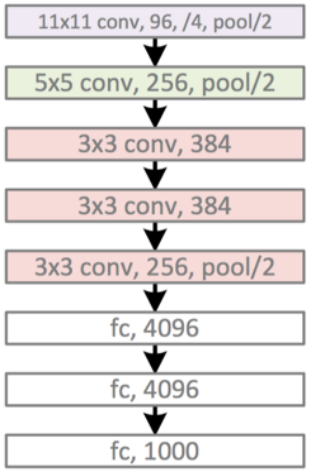


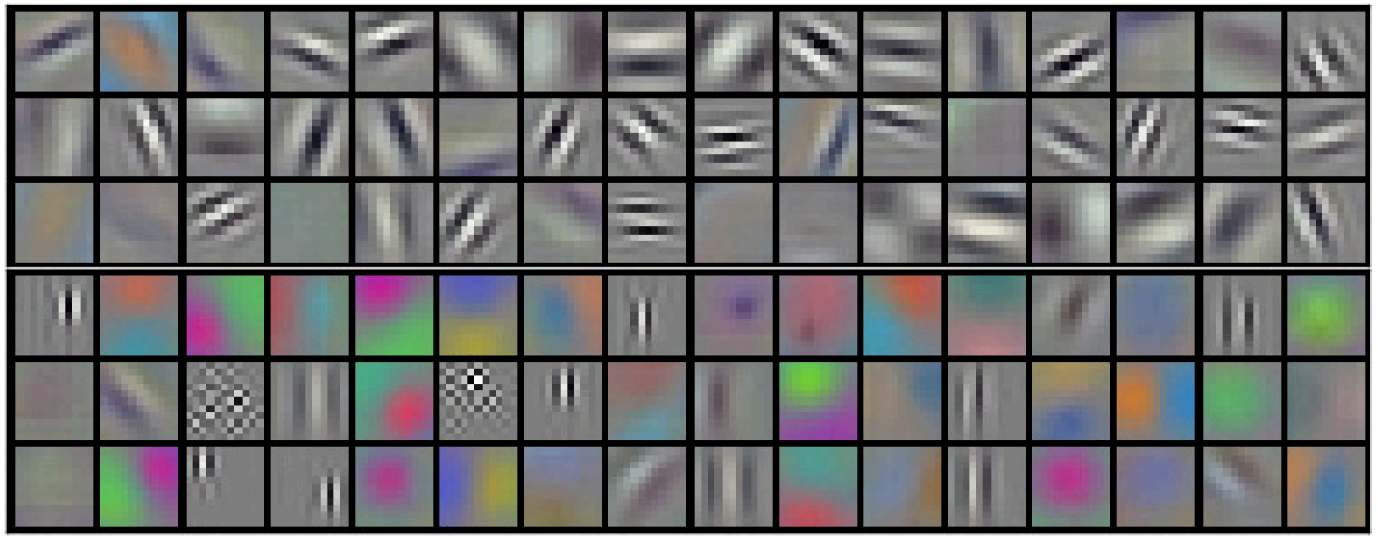


- The split (i.e. two pathways) in the image above are the split between two GPUs.
- Trained for about a week on two NVIDIA GTX 580 3GB GPU
- 60 million parameters
- Input layer: size 227x227x3
- 8 layers deep: 5 convolution and pooling layers and 3 fully connected layers

2012 **2013** **2014** **2015** **2016** **2017** **2018 …**

Deep architectures - AlexNet

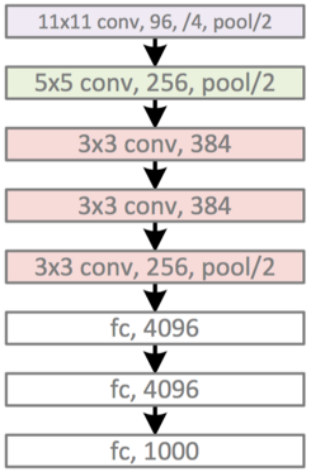


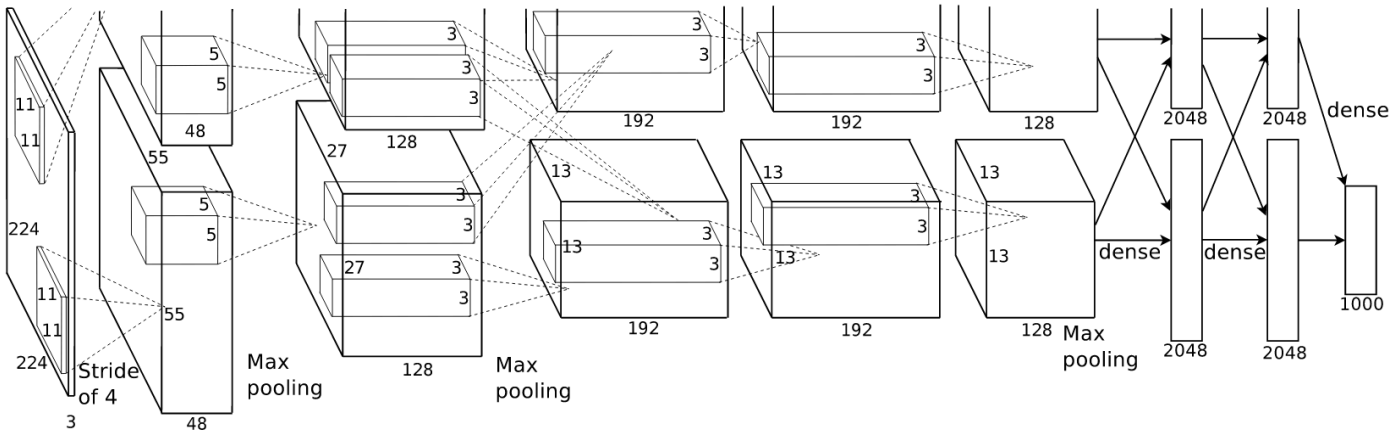


96 kernels learned by first convolution layer; 48 kernels were learned by each GPU

2012 2013 2014 2015 2016 2017 2018 ...

Deep architectures - AlexNet

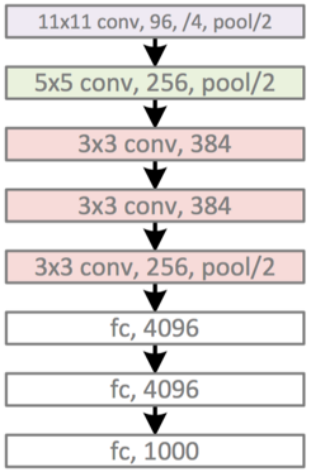




- Escape from a few layers
  - ReLU nonlinearity for solving gradient vanishing
  - Data augmentation
  - Dropout
  - Outperformed all previous models on ILSVRC by 10%

2012 2013 2014 2015 2016 2017 2018 …

Deep architectures - AlexNet



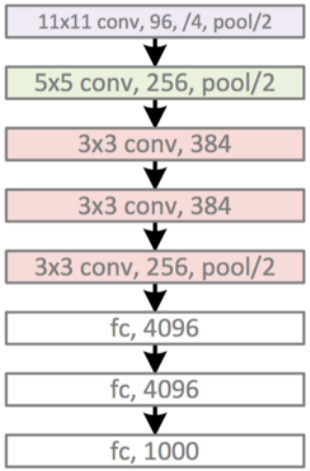
Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	**62,378,344**

First convolution layer: 96 kernels of size 11x11x3, with a stride of 4 pixels

Number of parameters = (11x11x3 + 1)*96 = 34,944

Note: There are no parameters associated with a pooling layer. The pool size, stride, and padding are hyperparameters.

Deep architectures - AlexNet



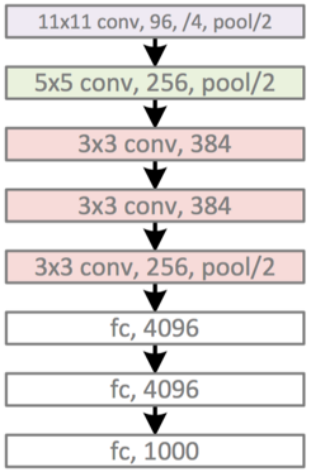
Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	**62,378,344**

Second convolution layer: 256 kernels of size 5x5x96

Number of parameters = (5x5x96 + 1)*256 = 614,656

2012 2013 2014 2015 2016 2017 2018 …

Deep architectures - AlexNet

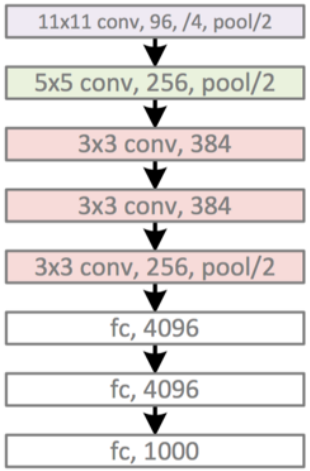


Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	**62,378,344**

First FC layer:
Number of neurons = 4096
Number of kernels in the previous Conv Layer = 256
Size (width) of the output image of the previous Conv Layer = 6

Number of parameters = (6x6x256*4096)+4096 = 37,752,832

Deep architectures - AlexNet



Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	**62,378,344**

The last FC layer:
Number of neurons = 1000
Number of neurons in the previous FC Layer = 4096

Number of parameters = (1000*4096)+1000 = 4,097,000

2012 **2013** **2014** **2015** **2016** **2017** **2018 …**

Outline

- Applications and success
- Basic components in CNN
- CNN architectures
- Training basics
- Optimizers

Training Basics

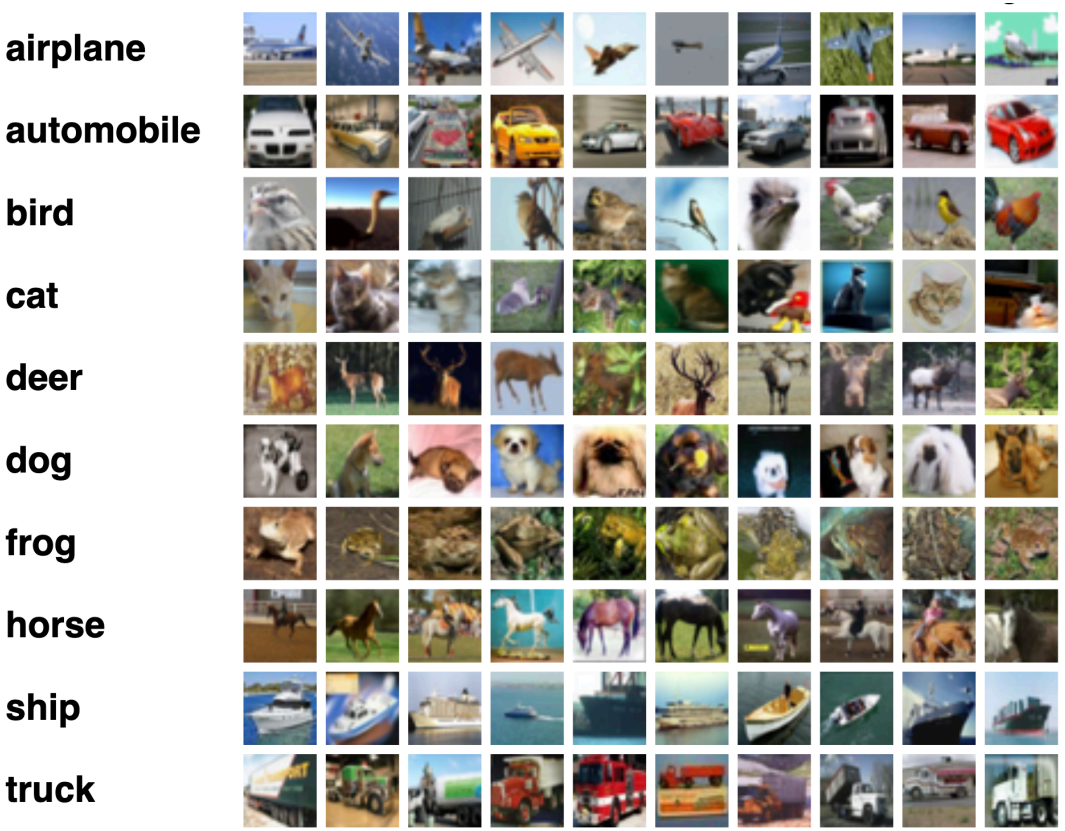
Multi-class classification

MNIST

- Size-normalized and centred 1x28x28 =784 inputs
- Training set = 60,000 images
- Testing set = 10,000 images

http://yann.lecun.com/exdb/mnist/

Multi-class classification



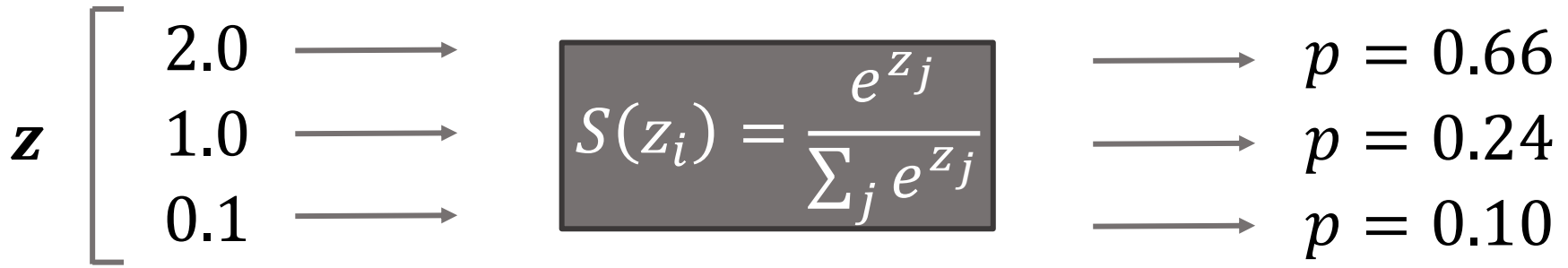
CIFAR-10

- 10 classes
- 6000 images per class
- 60000 images - 50000 training images and 10000 test images
- Each image has a size of 3x32x32, that is 3-channel color images of 32x32 pixels in size

https://www.cs.toronto.edu/~kriz/cifar.html

Softmax function

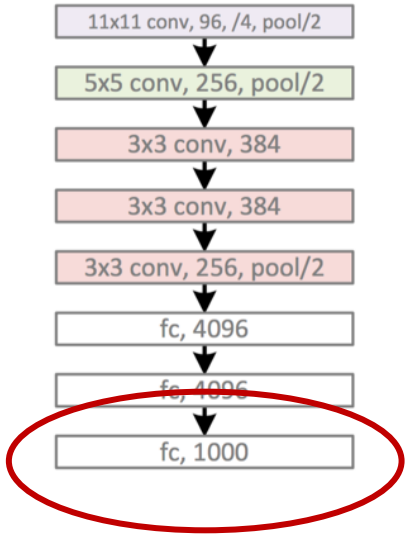
The softmax step can be seen as a generalized logistic function that takes as input a vector of scores $\boldsymbol{z} \in \mathbb{R}^n$ and outputs a vector of output probability $\boldsymbol{p} \in \mathbb{R}^n$ through a softmax function at the end of the architecture.



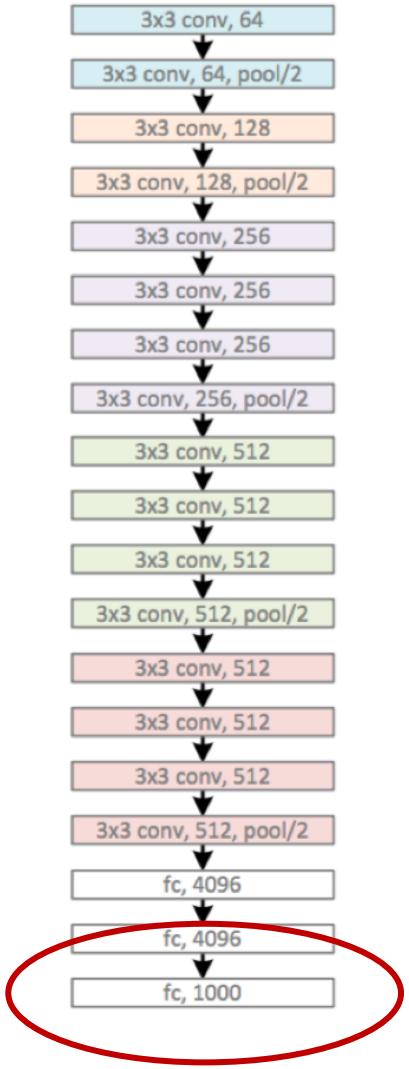
Logits: Numeric output of the last linear layer of a multi-class classification neural network

Softmax function

AlexNet



VGG



Where does the Softmax function fit in a CNN architecture?

Softmax's input is the output of the fully connected layer immediately preceding it, and it outputs the final output of the entire neural network. This output is a probability distribution of all the label class candidates.

Cross entropy loss

Loss function – In order to quantify how a given model performs, the loss function L is usually used to evaluate to what extent the actual outputs are correctly predicted by the model outputs.

Cross entropy loss (Multinomial Logistic Regression)

- The usual loss function for a multi-class classification problem
- Right after the Softmax function
- It takes in the input from the Softmax function output and the true label

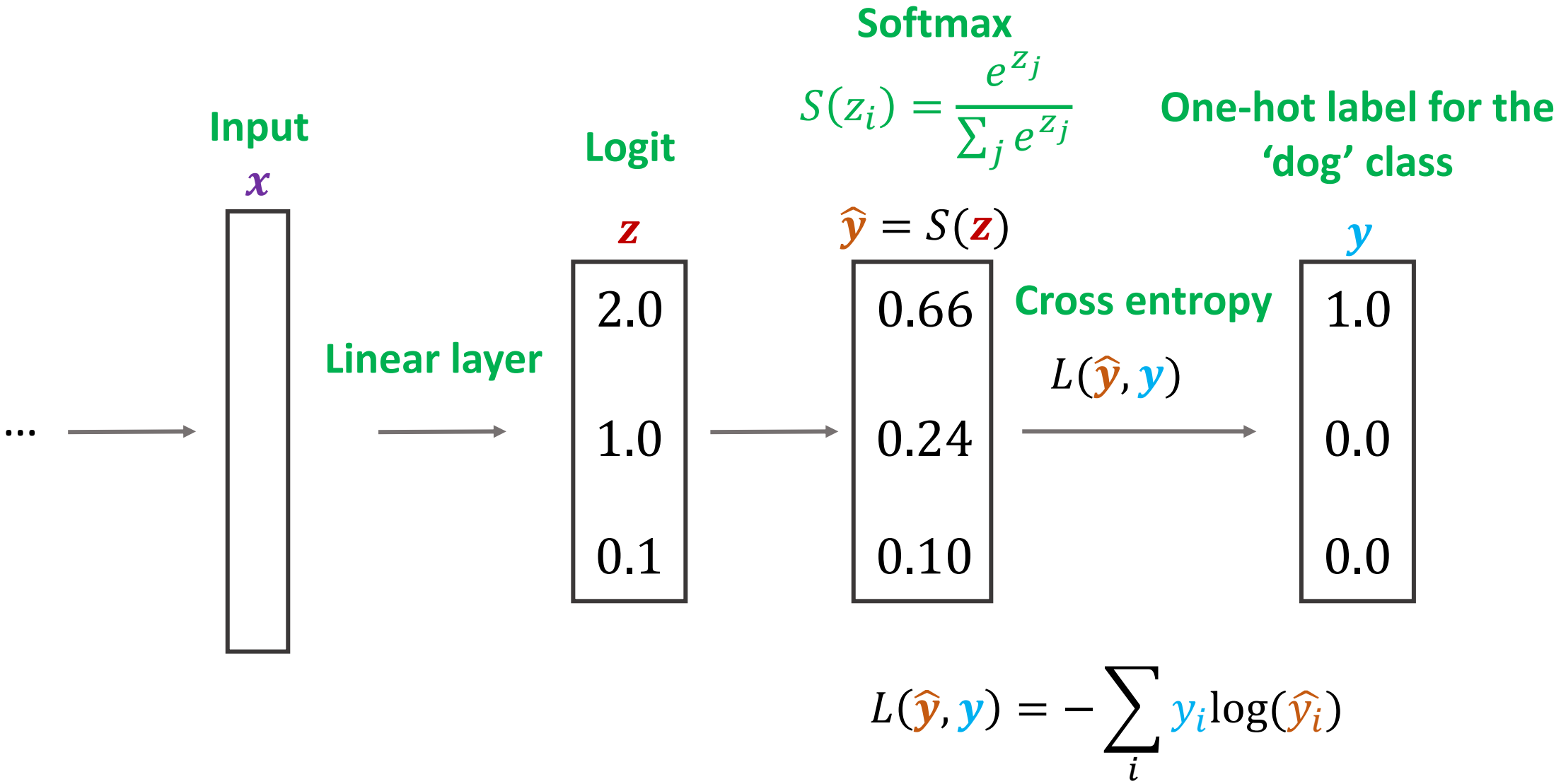
Cross entropy loss

One-hot encoded ground truth

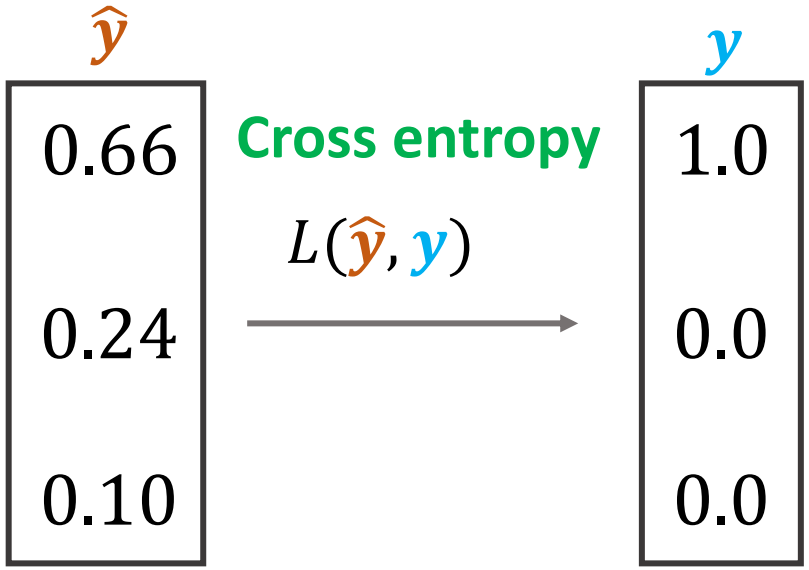
Example:

Class	One-hot vector
Dog	$[1 \quad 0 \quad 0]$
Cat	$[0 \quad 1 \quad 0]$
Bird	$[0 \quad 0 \quad 1]$

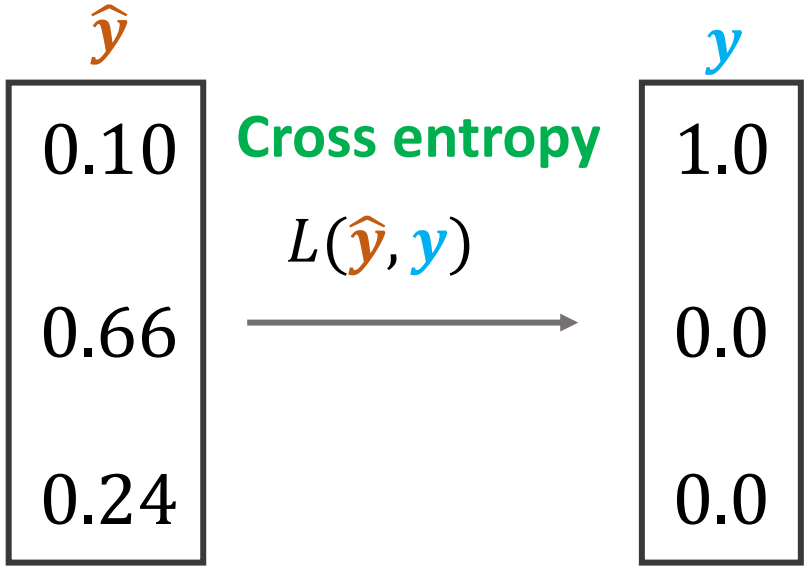
Cross entropy loss



Cross entropy loss



$$L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$
$$= -[(1 \times \log_2(0.66)) + (0 \times \log_2(0.24)) + (0 \times \log_2(0.10))]$$
$$= 0.6$$



$$L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$
$$= -[(1 \times \log_2(0.10)) + (0 \times \log_2(0.66)) + (0 \times \log_2(0.24))]$$
$$= 3.32$$

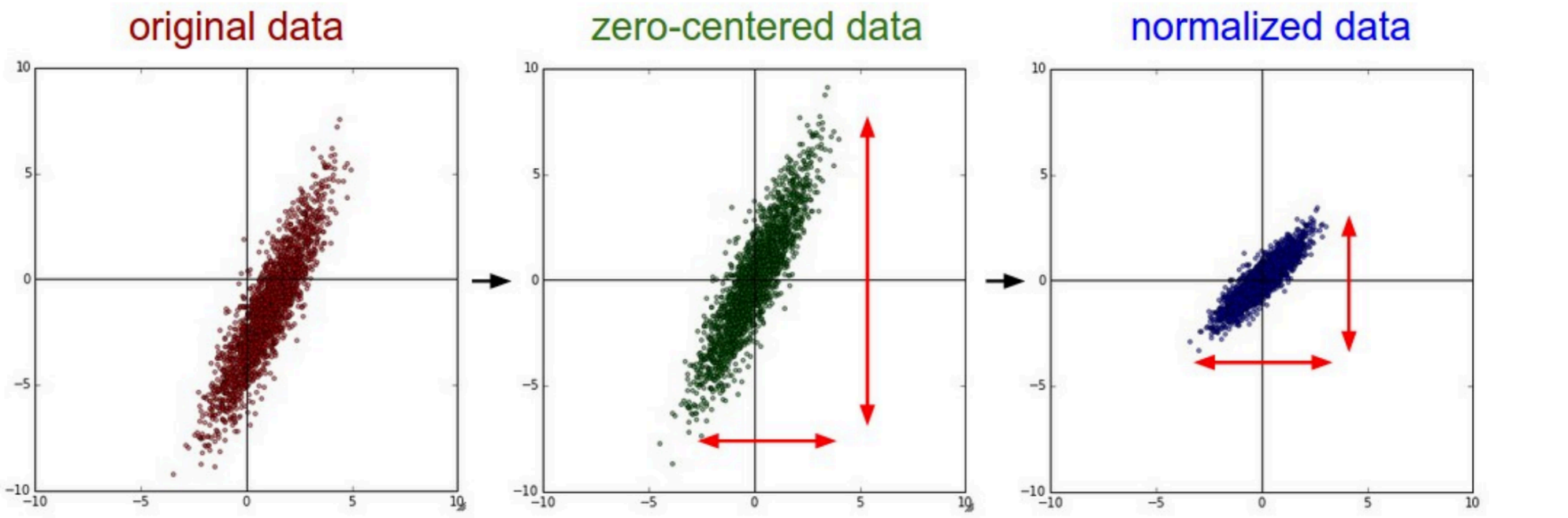
What is the min / max possible loss?

Epoch and mini-batch

- **Epoch**
  - In the context of training a model, epoch is a term used to refer to one iteration where the model sees the whole training set to update its weights.

- **Mini-batch gradient descent**
  - During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues.
  - Instead, the update step is done on mini-batches, where the number of data points in a batch is a hyperparameter that we can tune.

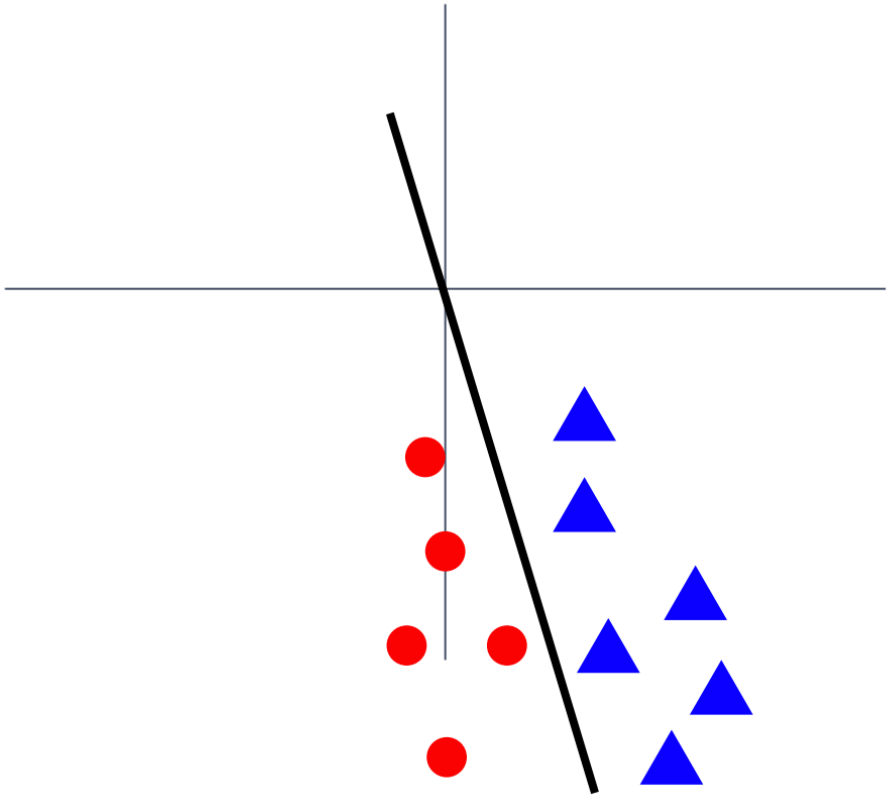
Data pre-processing



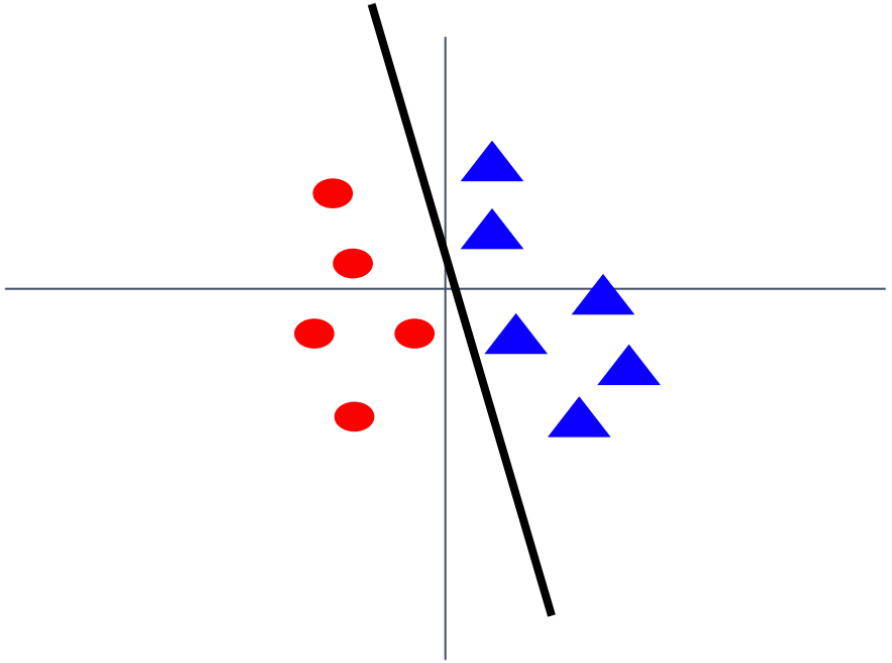
(Assume X [NxD] is data matrix,
each example in a row)

Data pre-processing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Data pre-processing for images

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Outline

- Applications and success
- Basic components in CNN
- CNN architectures
- Training basics
- Optimizers

Optimizers

Gradient descent (GD)

- Batch Gradient Descent
  - Full sum is expensive when N is large

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

- Stochastic Gradient Descent (SGD)
  - Approximate sum using a minibatch of examples
  - 32 / 64 / 128 common minibatch size
  - Additional hyperparameter on batch size

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

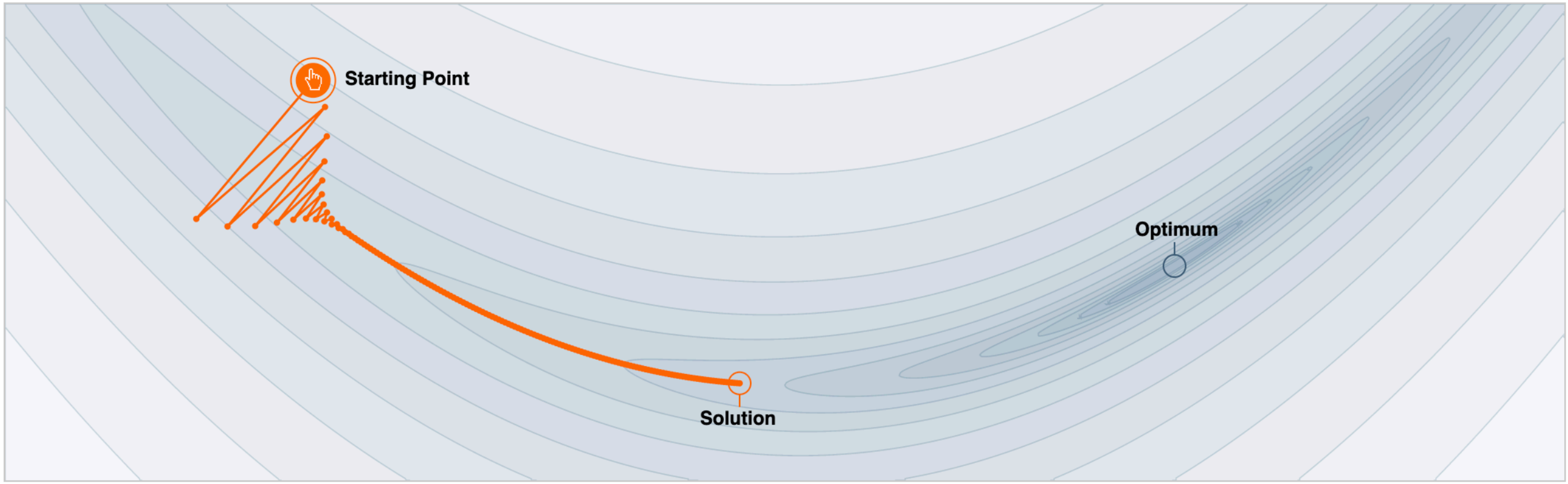
GD with Momentum

Deep neural networks have very complex error profiles. The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

When the error function has the form of a shallow ravine leading to the optimum and steep walls on the side, stochastic gradient descent algorithm tends to oscillate near the optimum. This leads to very slow converging rates. This problem is typical in deep learning architecture.

Momentum is one method of speeding the convergence along a narrow ravine.

GD with Momentum



Step-size α = 0.0030

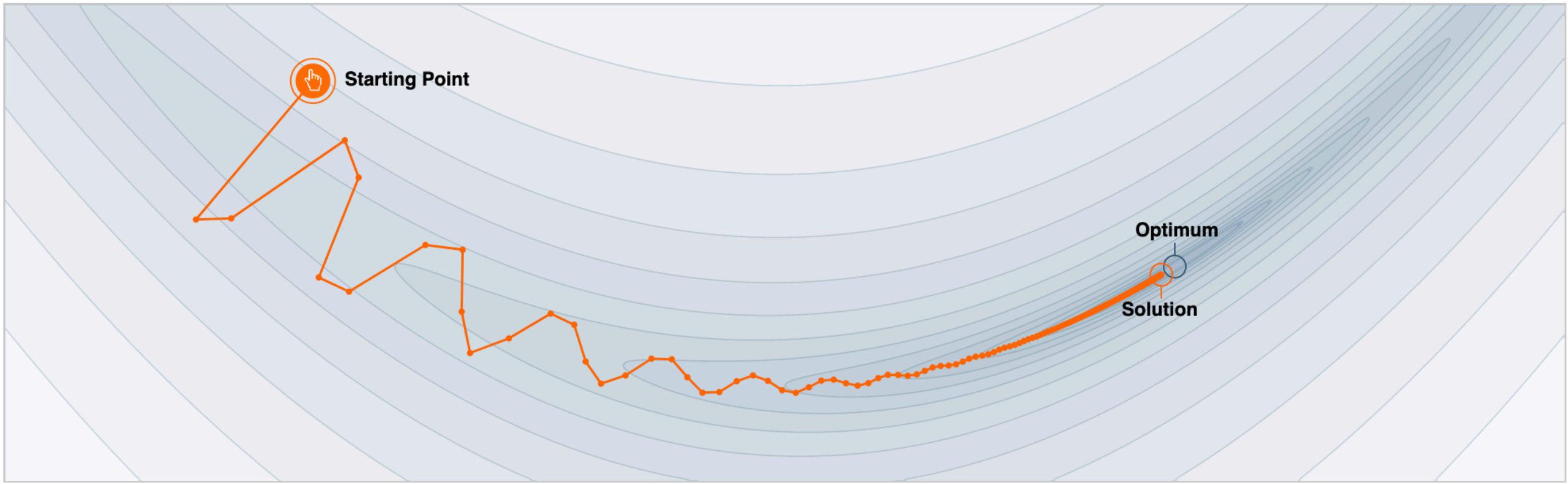
0 0.003 0.006

Momentum β = 0.0

0.00 0.500 0.990

Why Momentum Really Works: https://distill.pub/2017/momentum/

GD with Momentum



Step-size α = 0.0030

0 0.003 0.006

Momentum β = 0.80

0.00 0.500 0.990

Why Momentum Really Works: https://distill.pub/2017/momentum/

GD with Momentum

Momentum update is given by:

$$\boldsymbol{V} \leftarrow \gamma \boldsymbol{V} - \alpha \nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{V}$$

where $\boldsymbol{V}$ is known as the velocity term and has the same dimension as the weight vector $\boldsymbol{W}$.

The momentum parameter $\gamma \in [0,1]$ indicates how many iterations the previous gradients are incorporated into the current update.

The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Often, γ is initially set to 0.1 until the learning stabilizes and increased to 0.9 thereafter.

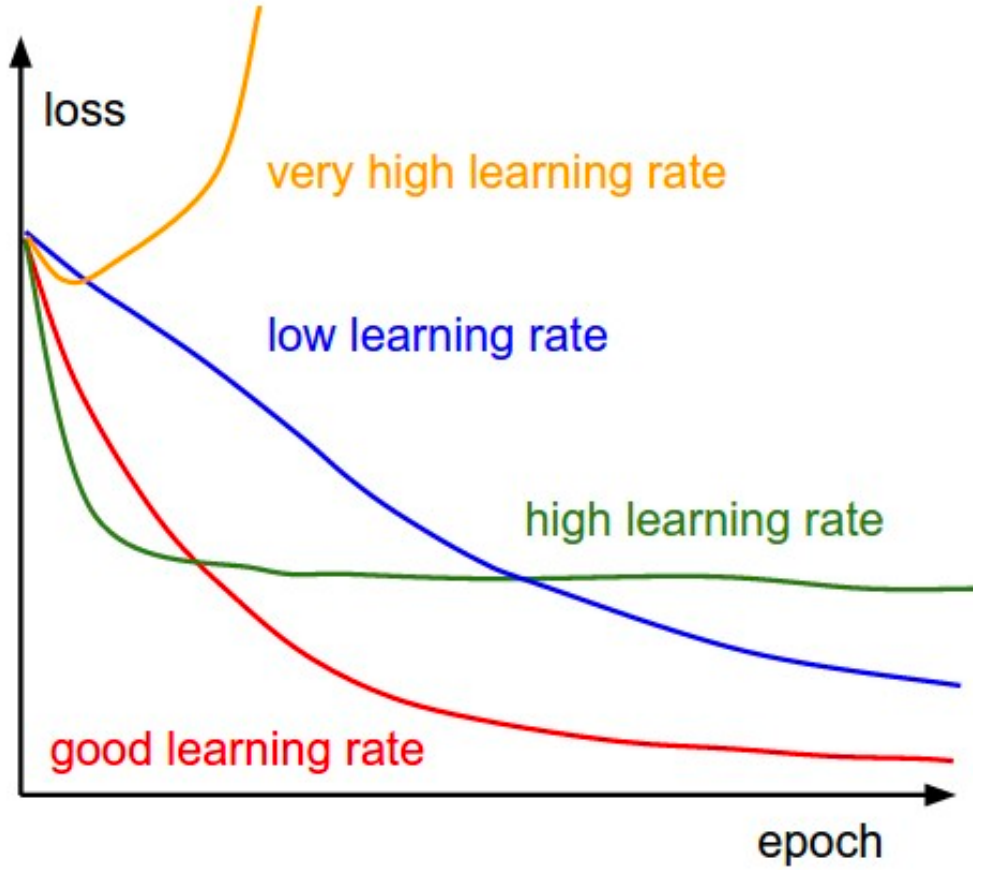
Learning rate

- **Learning rate**
  - The learning rate, often noted α or sometimes η, indicates at which pace the weights get updated. It can be fixed or adaptively changed.
  - The current most popular method is called Adam, which is a method that adapts the learning rate.
- **Adaptive learning rates**
  - Letting the learning rate vary when training a model can reduce the training time and improve the numerical optimal solution.
  - Algorithms with adaptive learning rates: AdaGrad, RMSprop, Adam
  - While Adam optimizer is the most commonly used technique, others can also be useful.

Adam: A Method for Stochastic Optimization, ICLR 2014

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

Learning rate



Annealing

One way to adapting the learning rate is to use an annealing schedule: that is, to start with a large learning factor and then gradually reducing it.

A possible annealing schedule (t – the iteration count):

$$\alpha(t) = \frac{\alpha}{\varepsilon + t}$$

α and ε are two positive constants. Initial learning rate $\alpha(0) = \alpha/\varepsilon$ and $\alpha(\infty) = 0$.

AdaGrad

Adaptive learning rates with annealing usually works with convex cost functions.

Learning trajectory of a neural network minimizing non-convex cost function passes through many different structures and eventually arrive at a region locally convex.

AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient. This improves the learning rates, especially in the convex regions of error function.

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + (\nabla_{\boldsymbol{W}} J)^2$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot (\nabla_{\boldsymbol{W}} J)$$

In other words, learning rate:

$$\tilde{\alpha} = \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}}$$

α and ε are two parameters.

RMSprop

RMSprop improves upon AdaGrad algorithms uses an exponentially decaying average to discard the history from extreme past so that it can converge rapidly after finding a convex region.

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)(\nabla_{\boldsymbol{W}} J)^2$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\sqrt{\varepsilon + \boldsymbol{r}}} \cdot (\nabla_{\boldsymbol{W}} J)$$

The decay constant ρ controls the length of the moving average of gradients.

Default value = 0.9.

RMSprop has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam Optimizer

Adams optimizer combines RMSprop and momentum methods. Adam is generally regarded as fairly robust to hyperparameters and works well on many applications.

Momentum term: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\nabla_{\boldsymbol{W}} J$

Learning rate term: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)(\nabla_{\boldsymbol{W}} J)^2$

$$\boldsymbol{s} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1}$$

$$\boldsymbol{r} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot \boldsymbol{s}$$

Note that $\boldsymbol{s}$ adds the momentum and $\boldsymbol{r}$ contributes to the adaptive learning rate.

Suggested defaults: $\alpha = 0.001, \rho_1 = 0.9, \rho_2 = 0.999$, and $\epsilon = 10^{-8}$

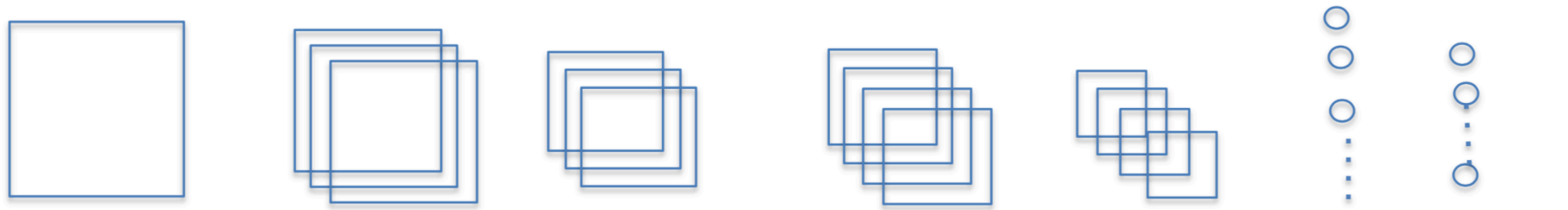
Example 3: MNIST digit recognition

MNIST database: 28x28 =784 inputs

Training set = 12000 images
Testing set = 2000 images

Input pixel values were normalized to [0, 1]

Example 3: Architecture of CNN



input image
28x28

Convolution 1
32 filters
5x5 size
SAME
32x28x28

Pooling 1
2x2 pooling
Stride = 2
32x14x14

Convolution 2
64 filters
5x5 size
SAME
64x14x14

Pooling 2
2x2 pooling
Stride = 2
64x7x7

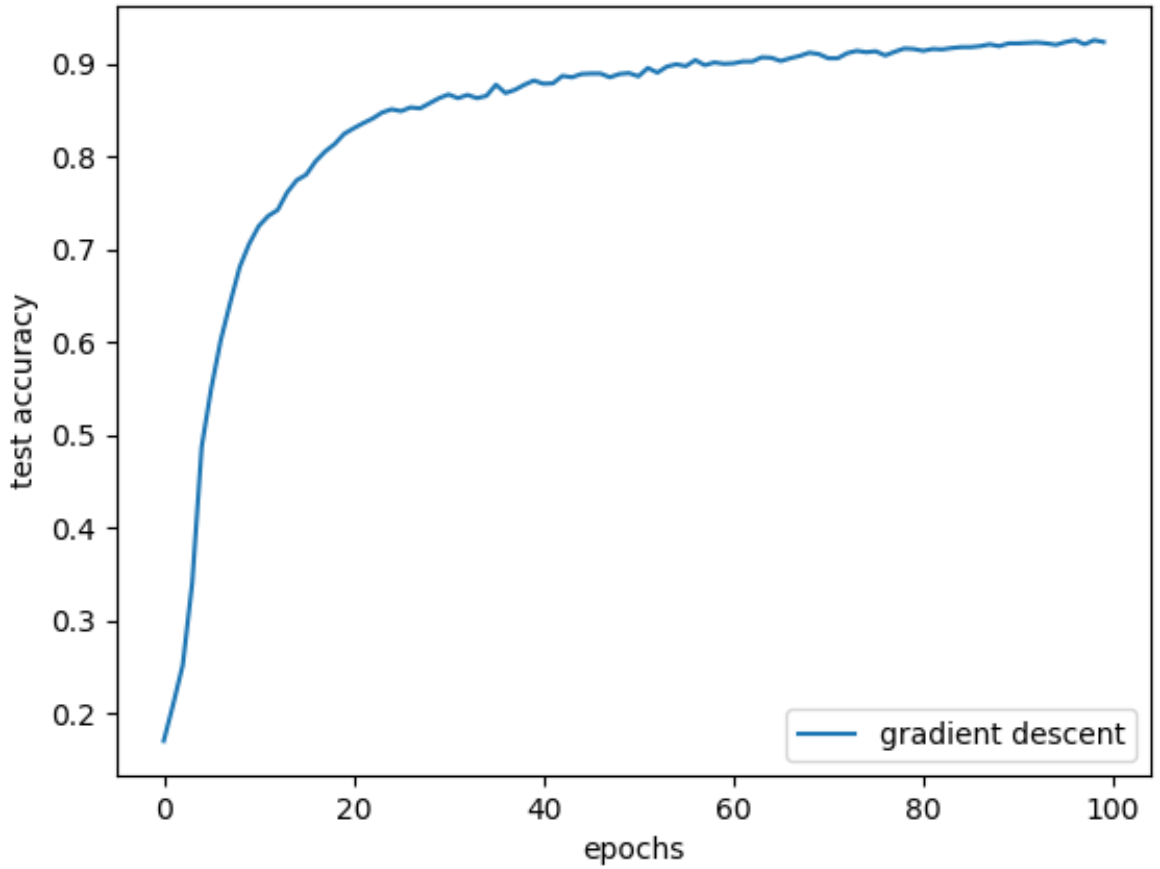
Fixed
1024

Output
Softmax
10

ReLU neurons
Gradient descent optimizer with batch-size = 128
Leaning parameter = 10^{-3}

See eg7.3.ipynb

Example 3: Training Curve



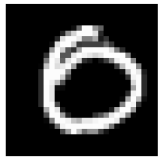
Example 3: Weights learned

Weights learned at convolution layer 1

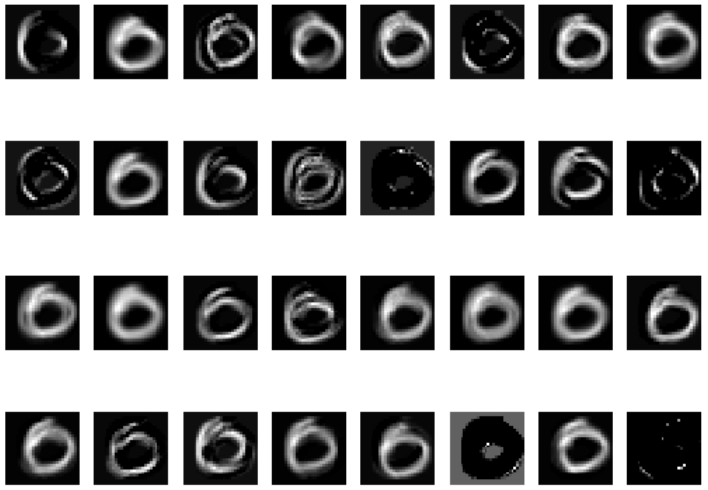
32x5x5

Example 3: Feature maps

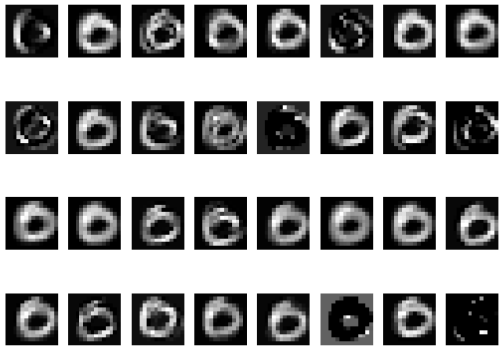
Input image
28x28



Feature maps at conv1
32x28x28

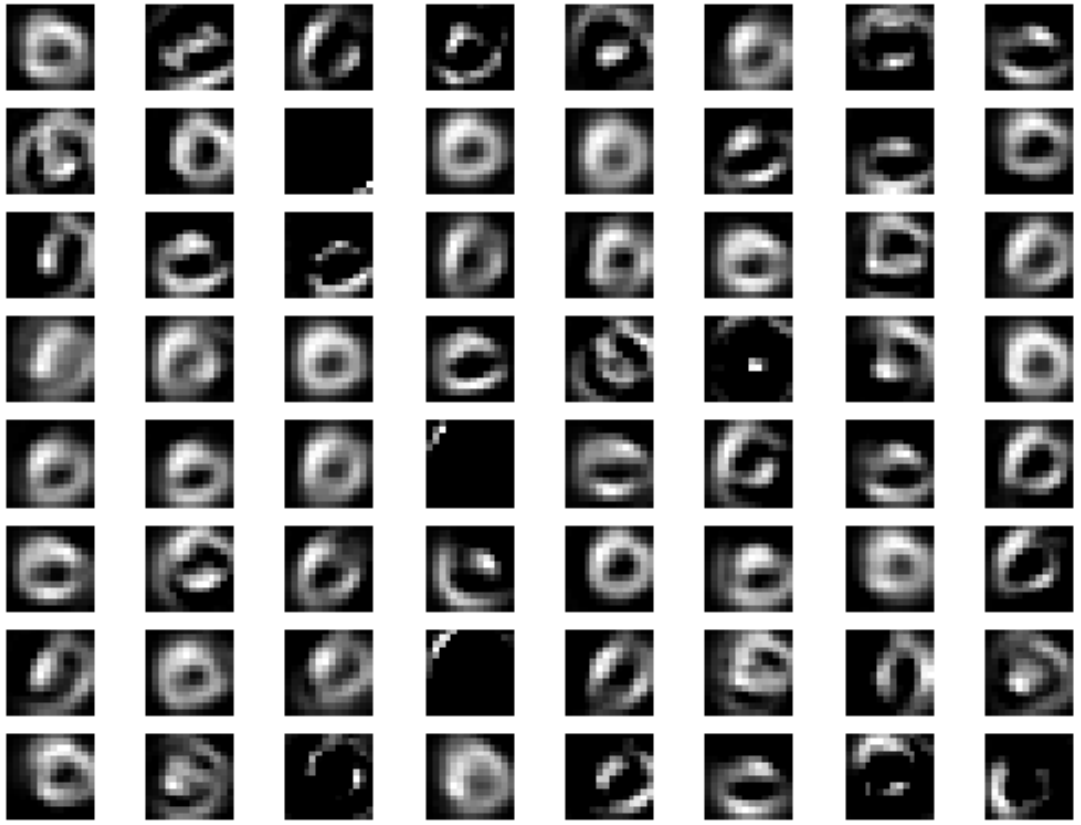


Feature maps at pool1
32x14x14



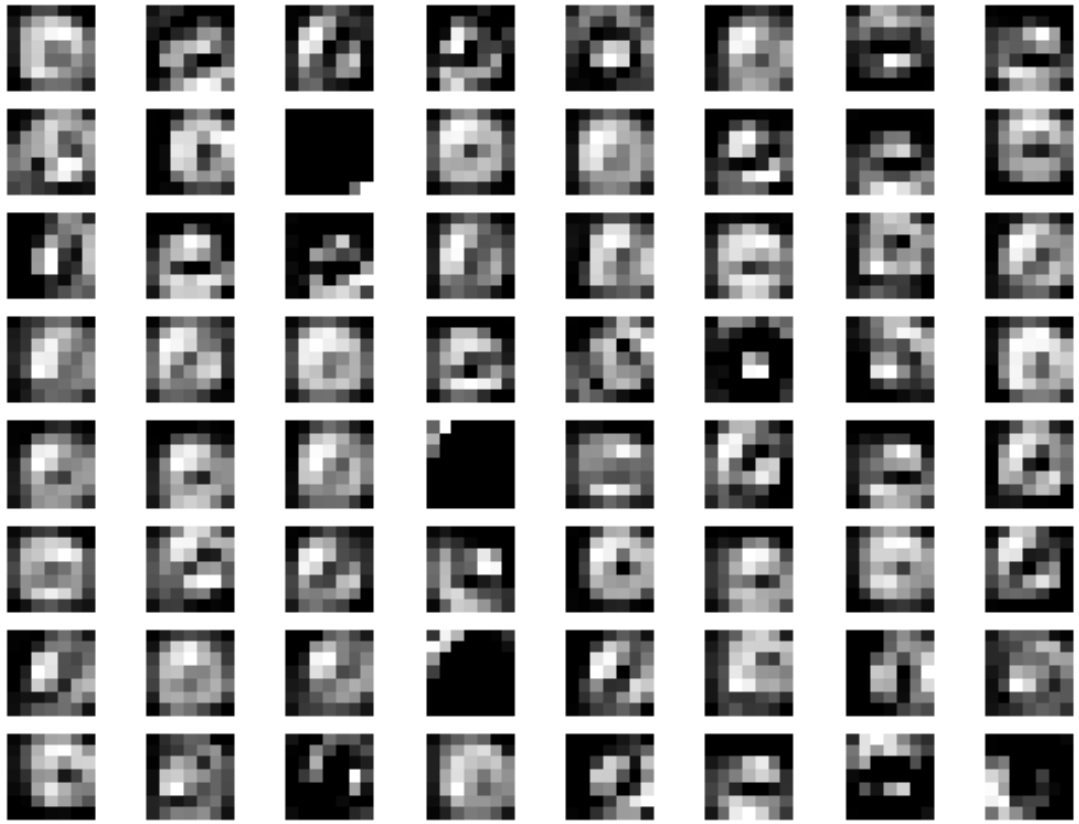
Example 3: Feature maps

Feature maps at conv2
64x14x14

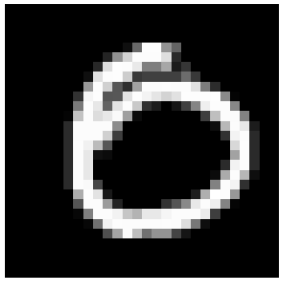


Example 3: Feature maps

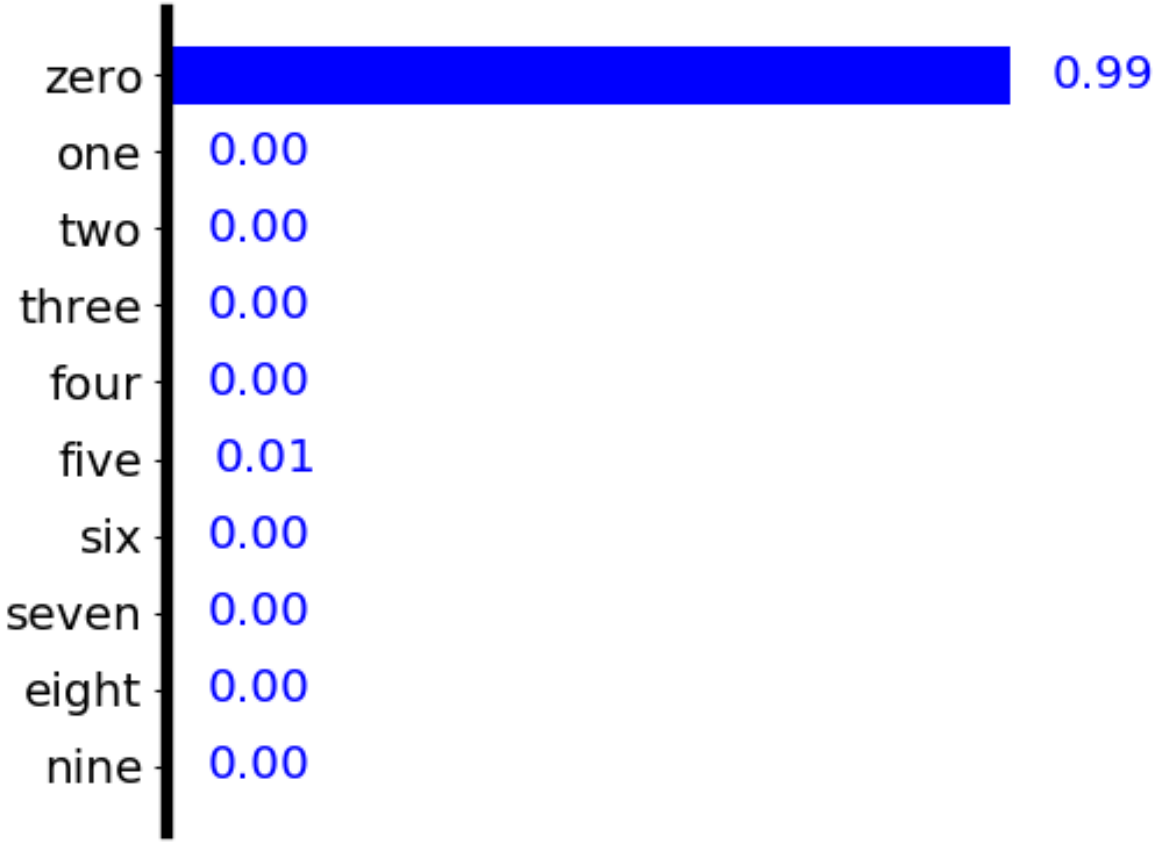
Feature maps at pool2
64x7x7



Example 3:Output

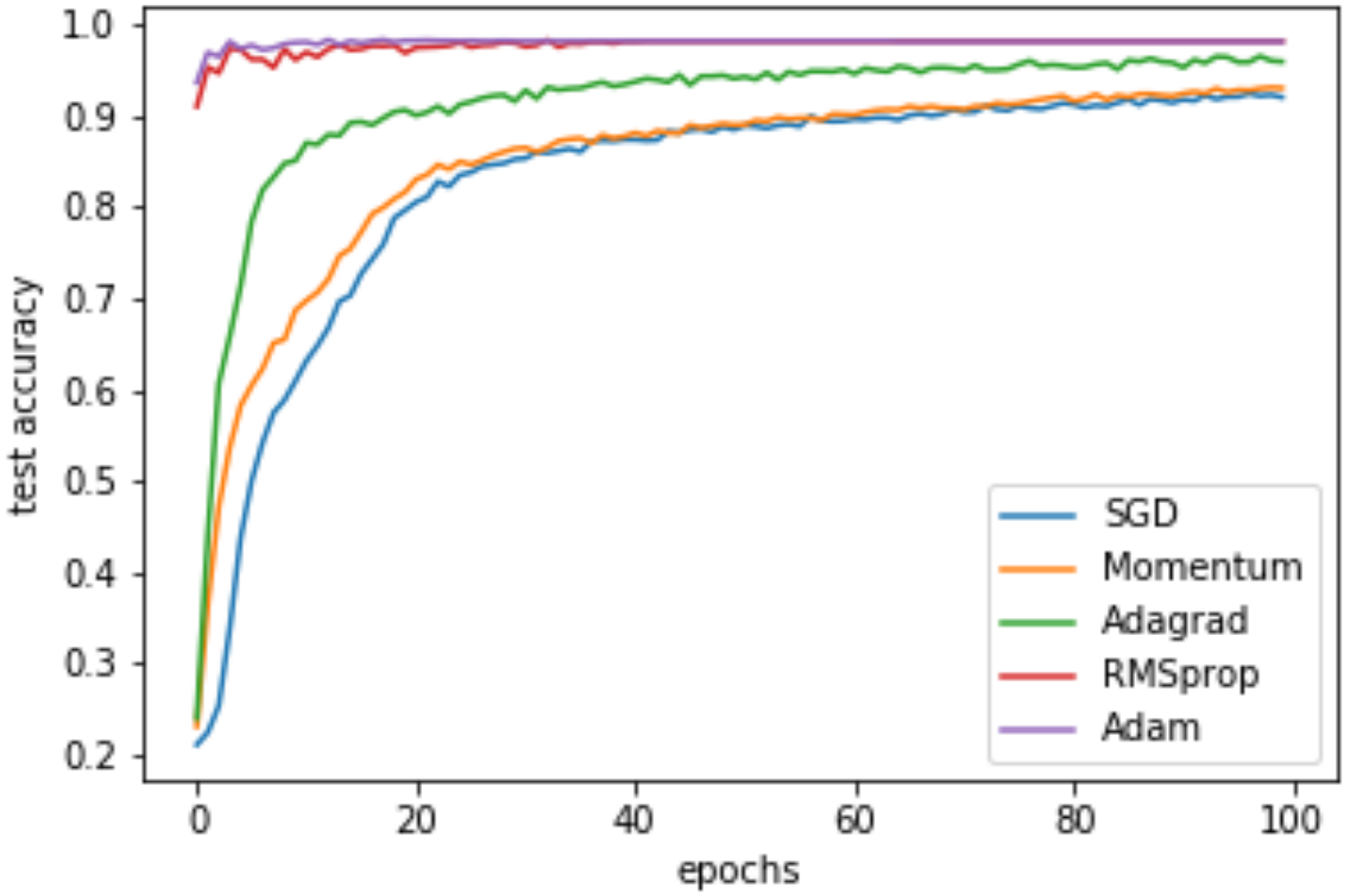


Input image



Output 10 scores

Example 4: MNIST recognition with CNN with different learning algorithms



See eg7.4.ipynb

Performance