SC 4001 CE/CZ 4042 Neural Network and Deep Learning

Last update: 25 Feb 2024

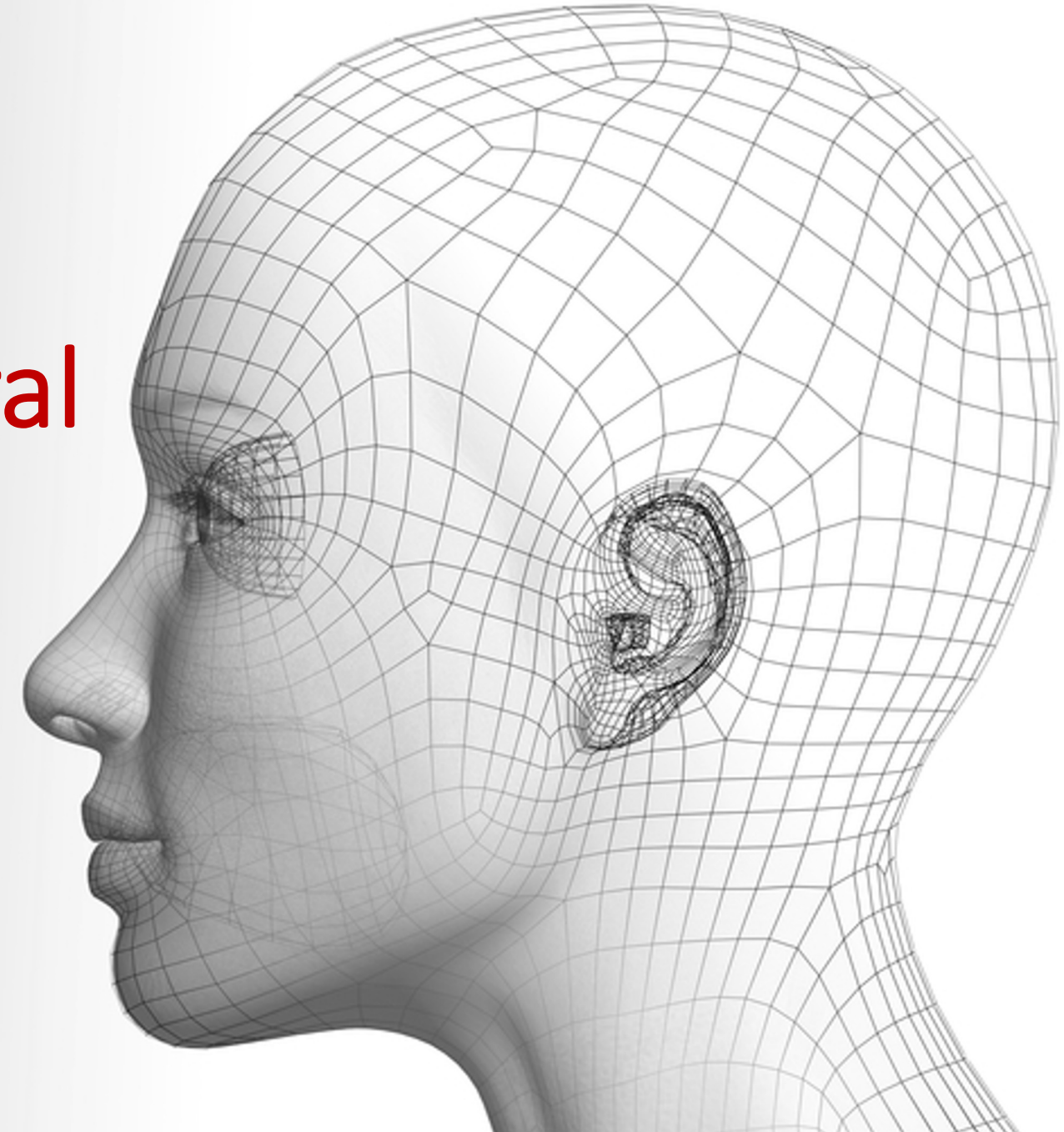
Convolutional Neural Networks I

Xingang Pan

潘新钢

https://xingangpan.github.io/

https://twitter.com/XingangP



Instructors

Xingang Pan

2012-2016

B.E.

Tsinghua University

2017-2021

PhD

The Chinese University of Hong Kong

Advisor : Xiaoou Tang

2021-2023

Post-doctoral Fellow

Department of Visual Computing and AI

Max Planck Institute for Informatics

2023-Now

Assistant Professor

School of Computer Science and Engineering

Nanyang Technological University

Research areas

- Generative models
- Image synthesis
- Computer vision
- Deep learning

Email: xingang.pan@ntu.edu.sg
Office: N4-02c-113
Office hour: By email appointment

Schedule

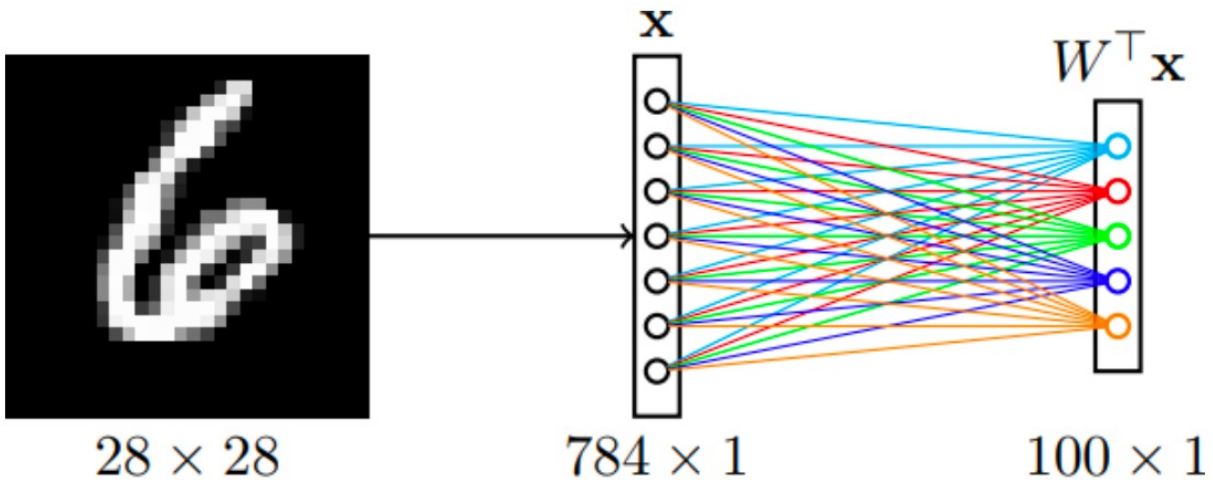
- **Week 7** – Convolutional Neural Network (CNN) I
- **Recess Week**
- **Week 8** – Convolutional Neural Network (CNN) II
- **Week 9** – Recurrent Neural Networks (RNN)
- **Week 10** – Attention
- **Week 11** – Autoencoders
- **Week 12** – Generative Adversarial Networks (GAN)
- **Week 13** – Revision and Selected Topics

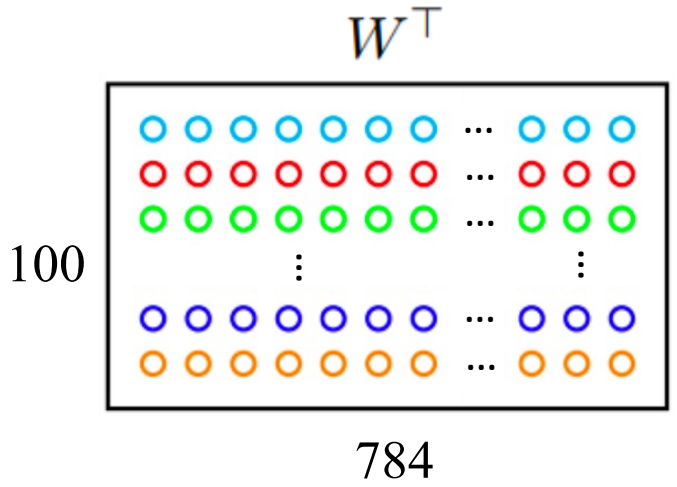
Every week: Tutorial of previous week + Lecture of current week

Outline

- Basic components in CNN
- Training a classifier
- Optimizers

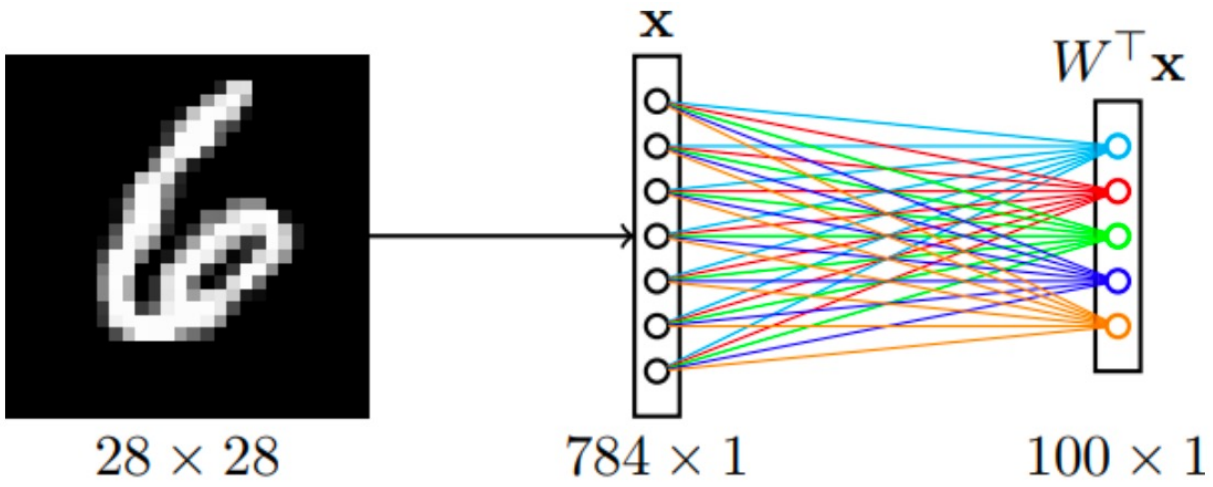
Motivation

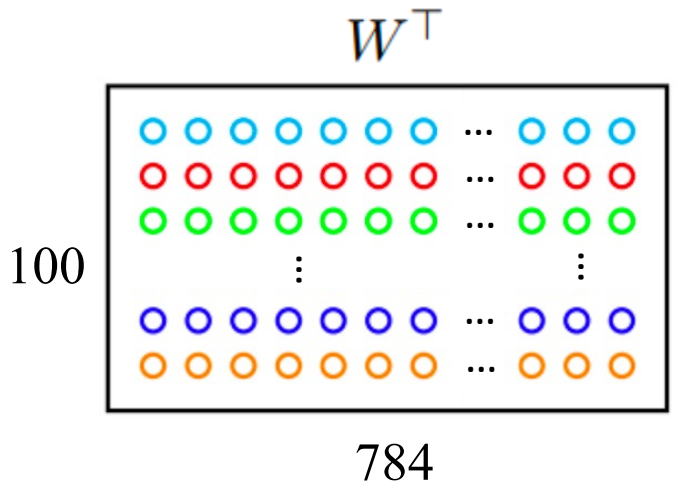




Let us consider the first layer of a MLP taking images as input. What are the problems with this architecture?

Motivation





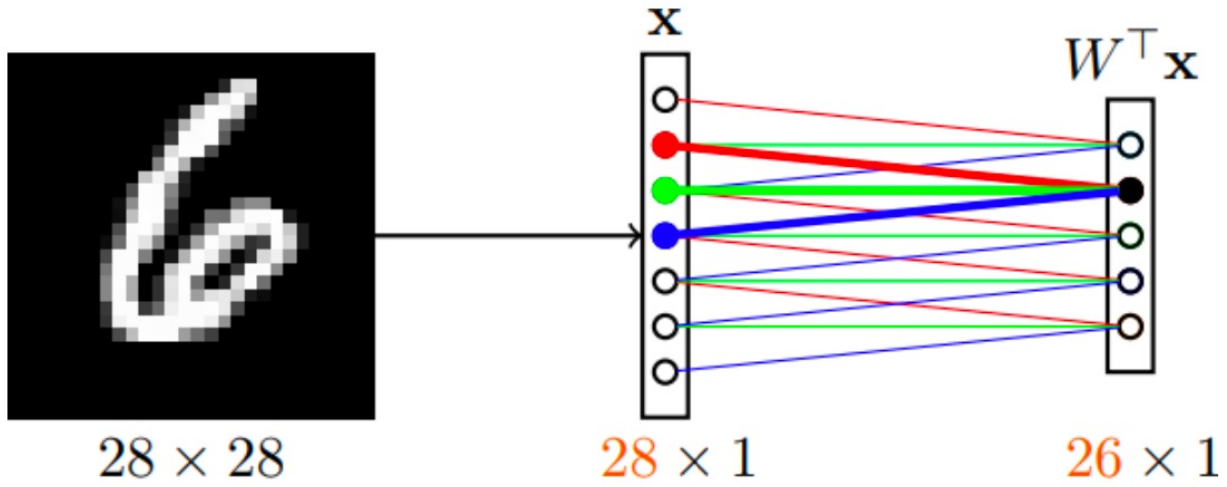
Issues

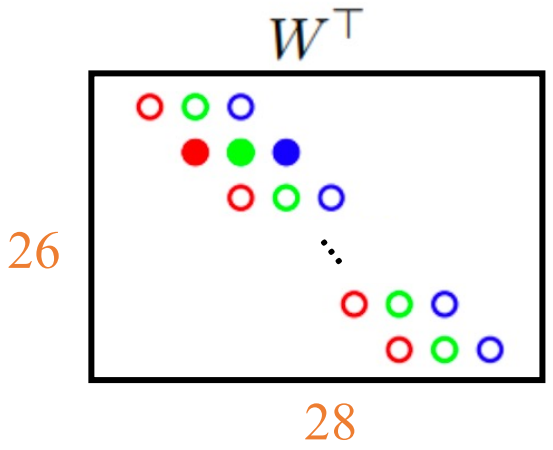
- Too many parameters: 100 × 784 + 100.
  - What if images are 640 × 480 × 3?
  - What if the first layer counts 1000 units?

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/equivariant 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.

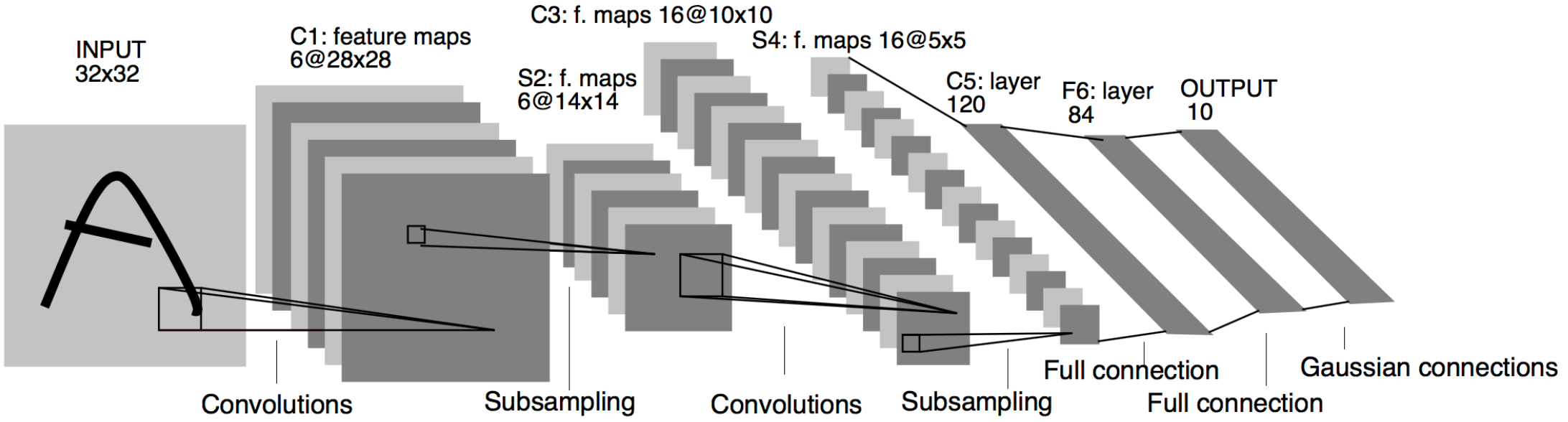
Basic Components in CNN

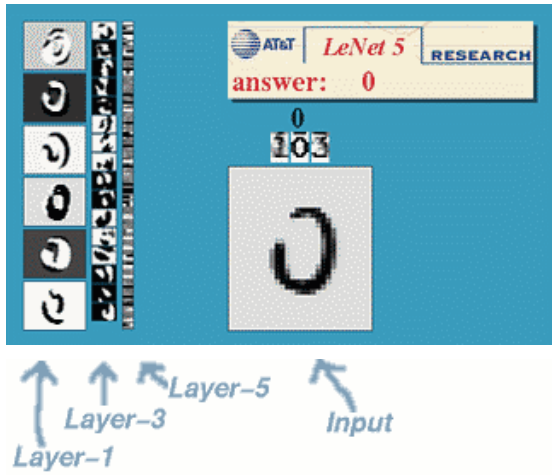
Credits

- 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

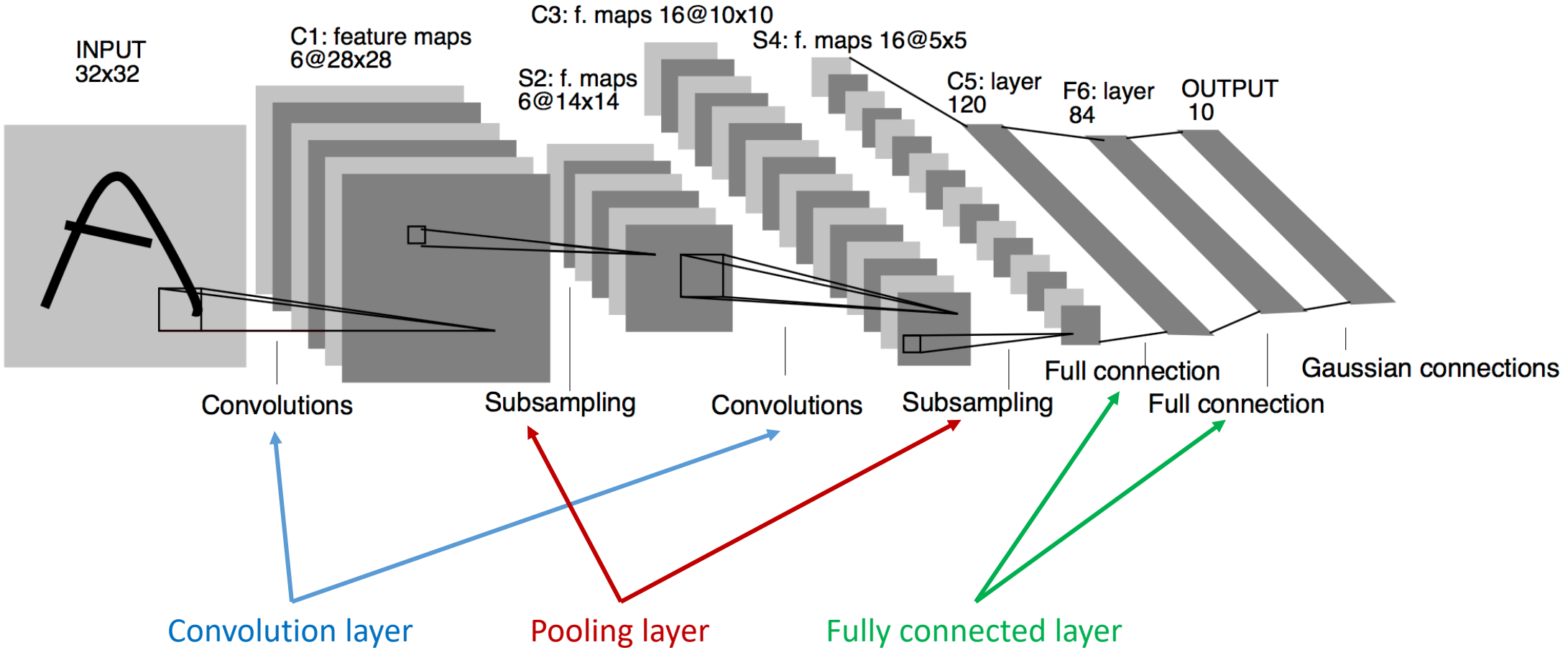
An example of convolutional network: LeNet 5





LeCun et al., 1998
Turing Award (2018)

An example of convolutional network: LeNet 5

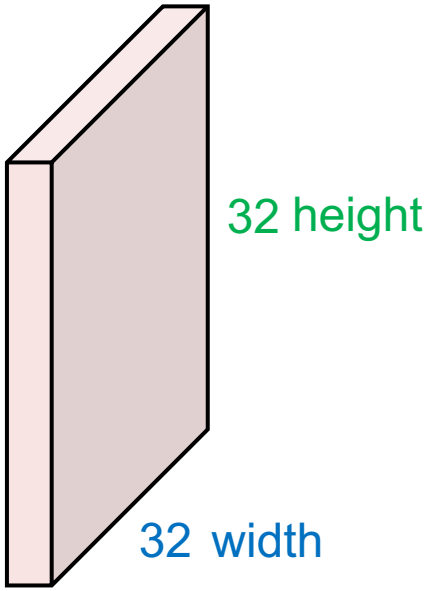


As we go deeper (left to right) the height and width tend to go down and the number of channels increased.

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

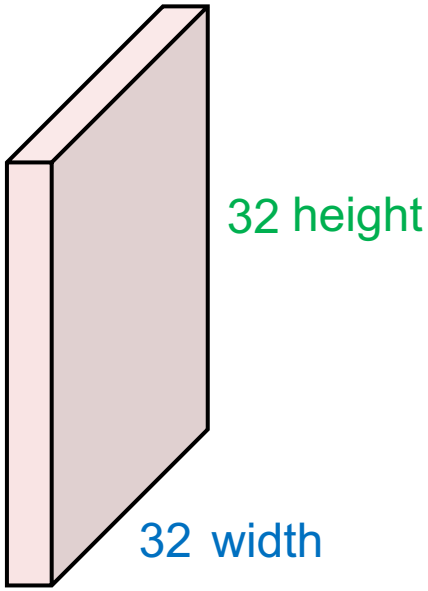
Convolution layer

3x32x32 image

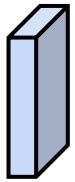


Convolution layer

3x32x32 image



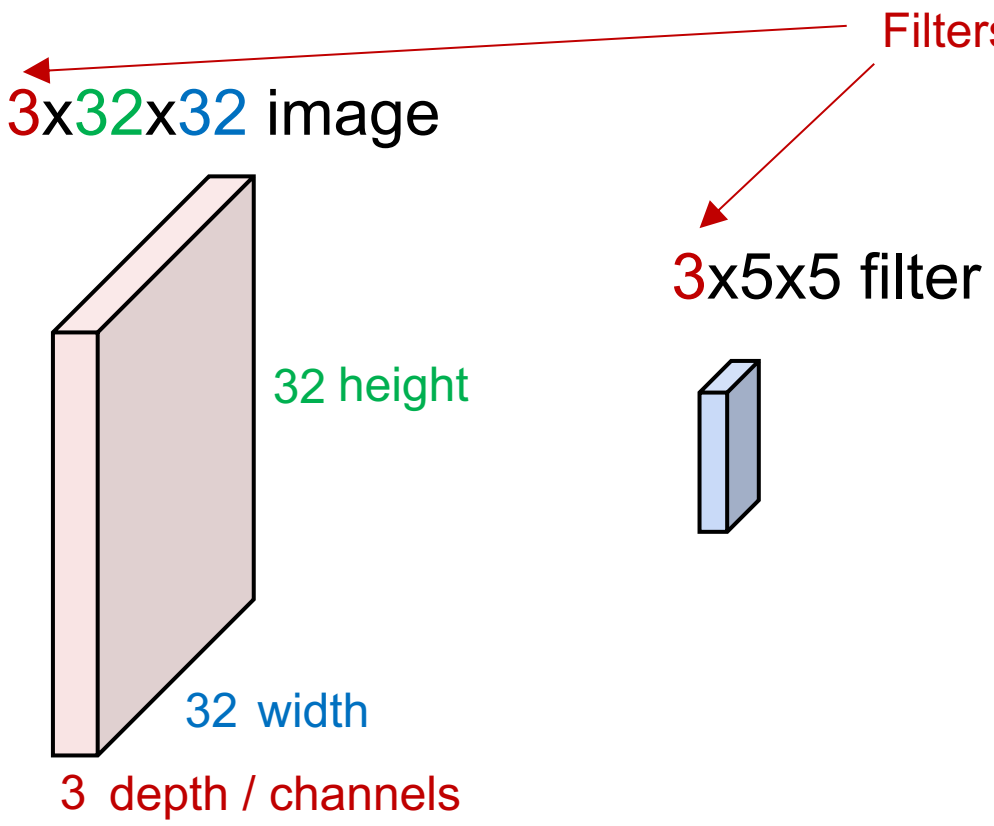
3x5x5 filter



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

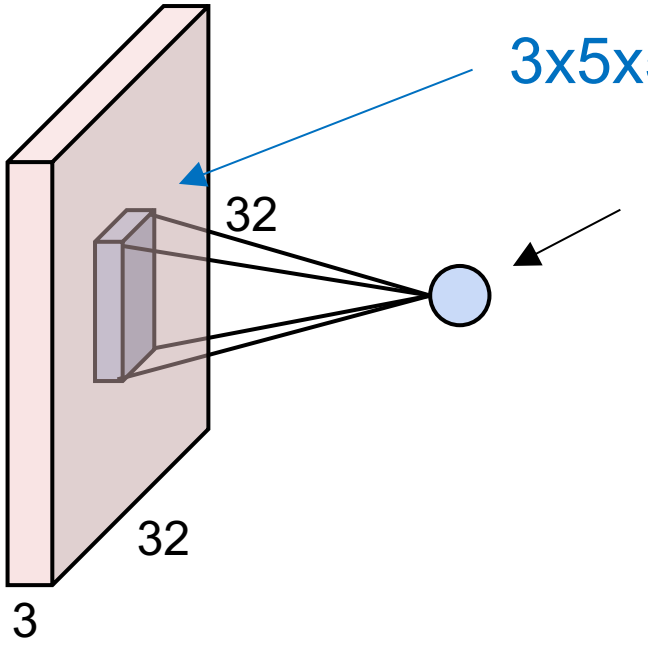


Weights of filters are learned using backpropagation algorithm. The weights learned are known as *filters* or *kernels*

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

Convolution layer

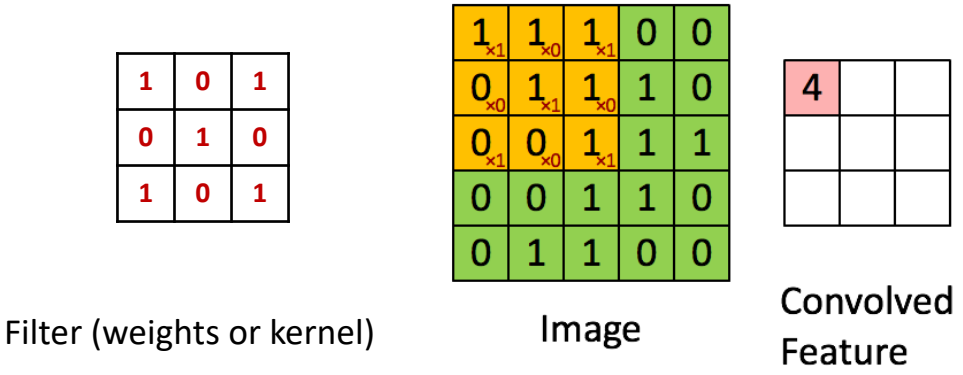
3x32x32 image



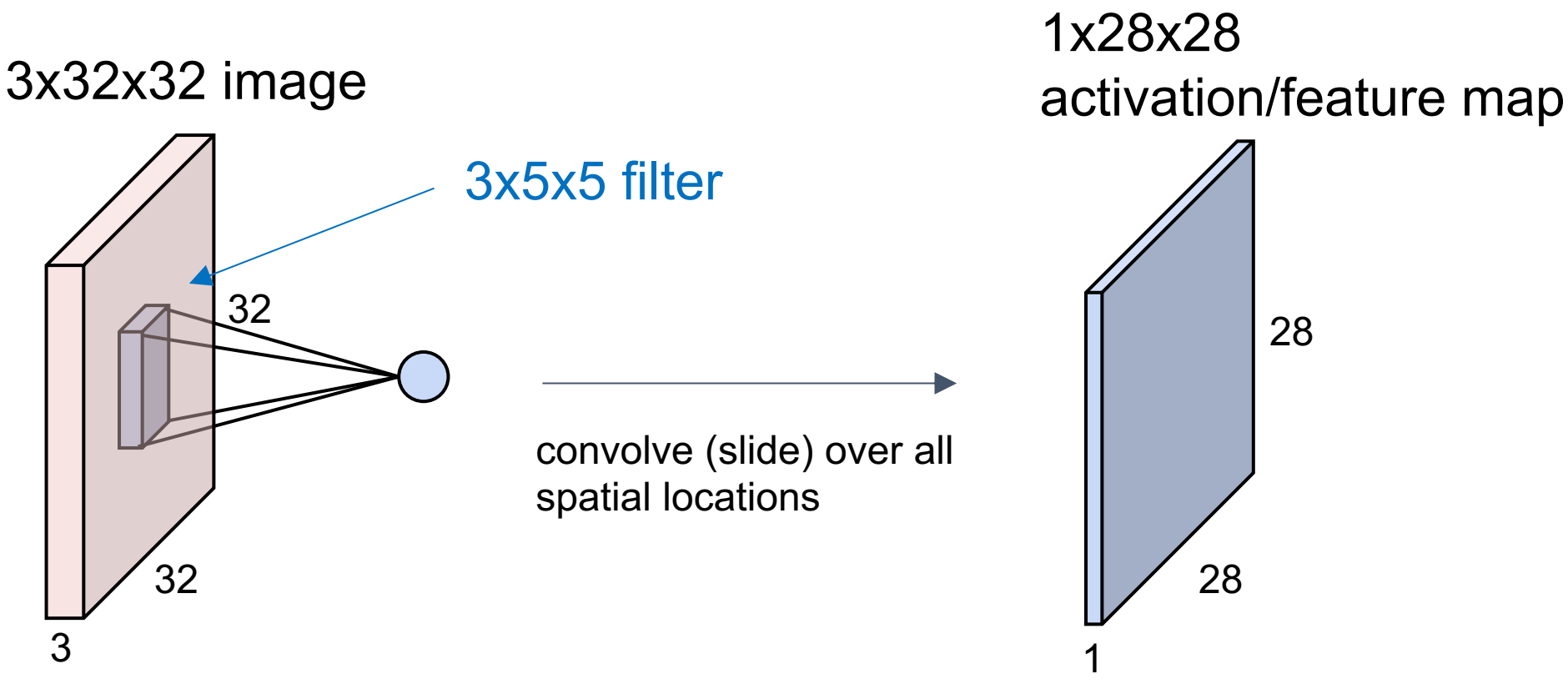
1 number:
the result of taking a dot product between the filter and a small 3x5x5 chunk of the image

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

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



Convolution layer

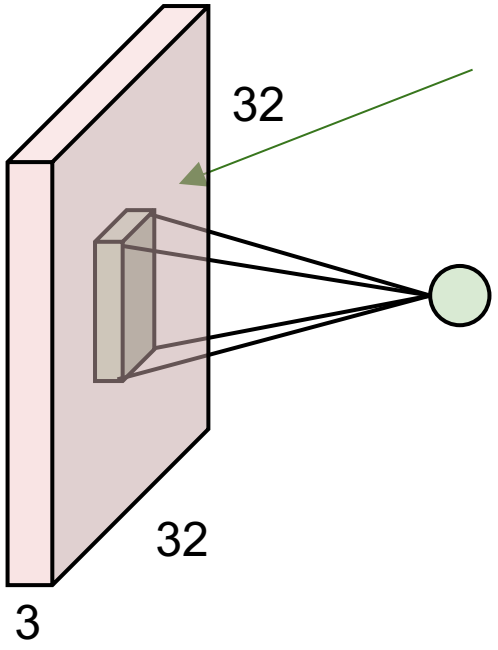


The activations are obtained by convolving the filters (weights) with the input activations. The output activation produced by a particular filter is known as a *activation map / feature map*

Convolution layer

Consider a second, green filter

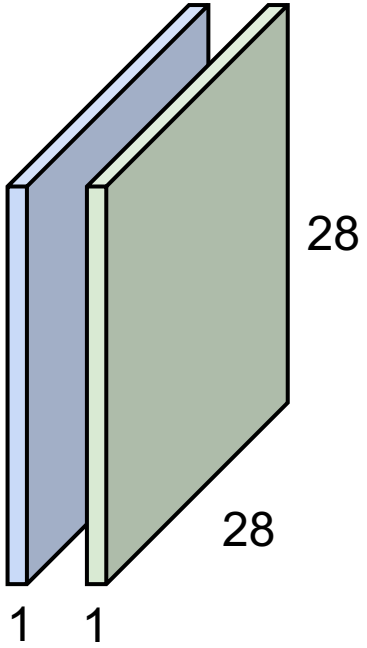
3x32x32 image



3x5x5 filter

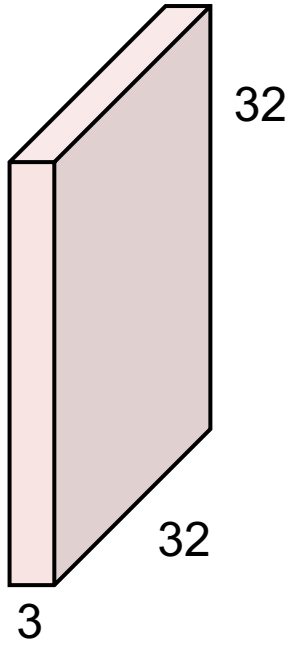
convolve (slide) over all spatial locations

Two 1x28x28 activation/feature maps

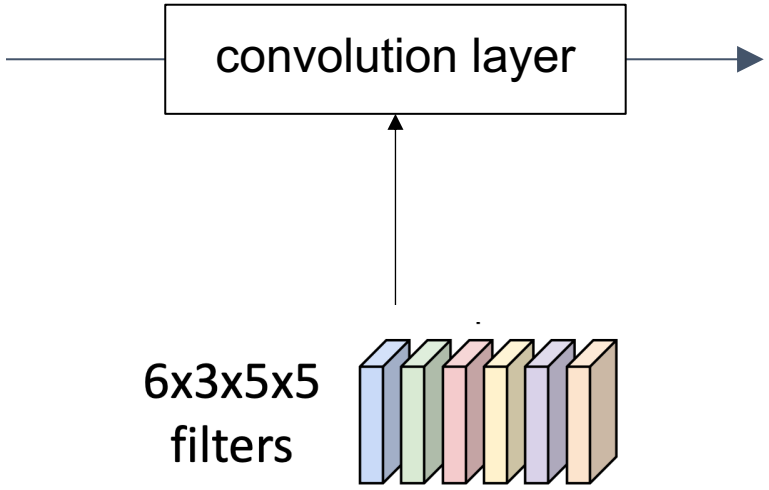


Convolution layer

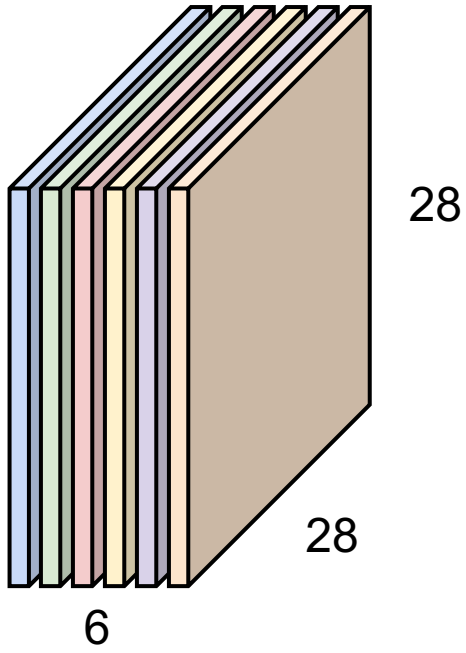
3x32x32 image



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

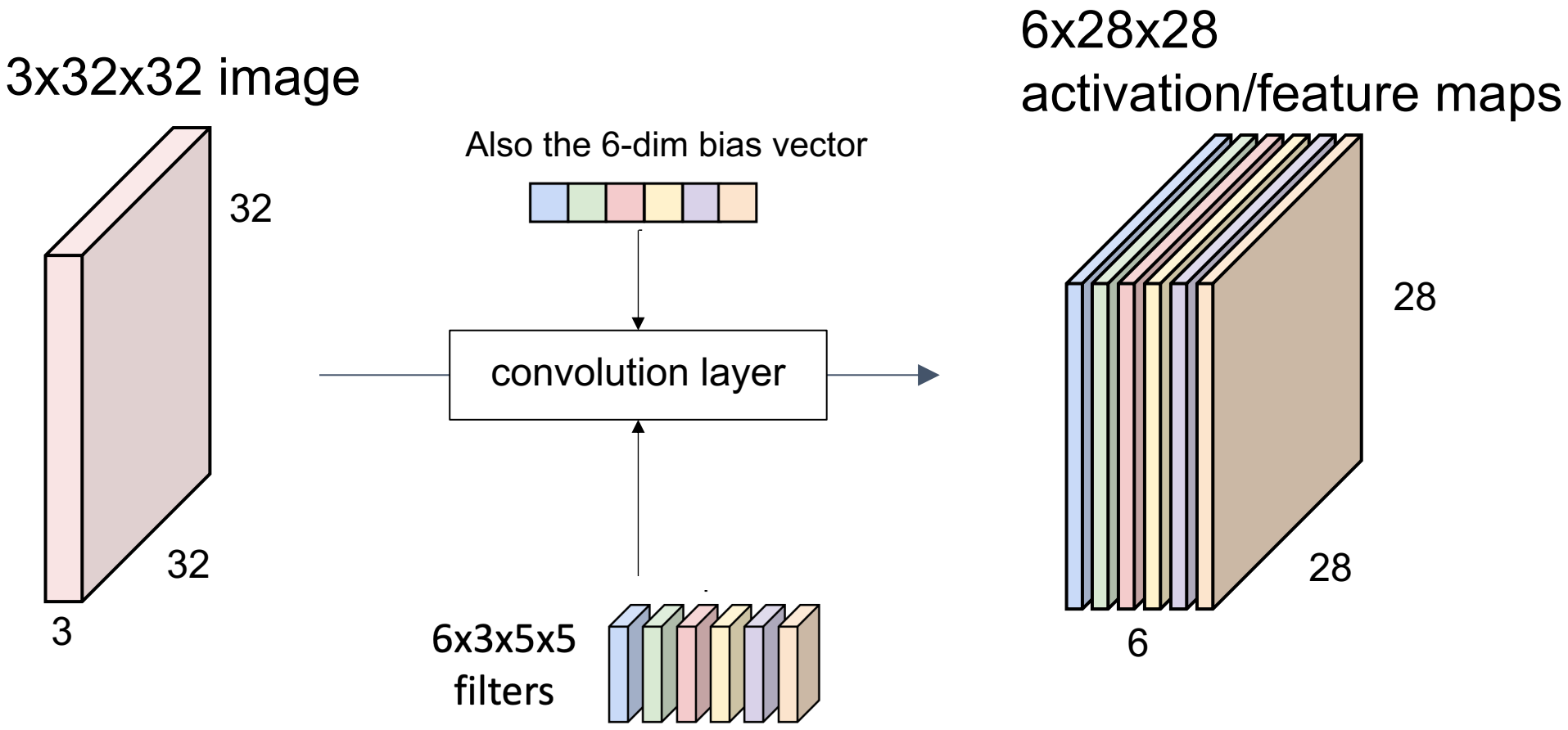


Six 1x28x28 activation/feature maps

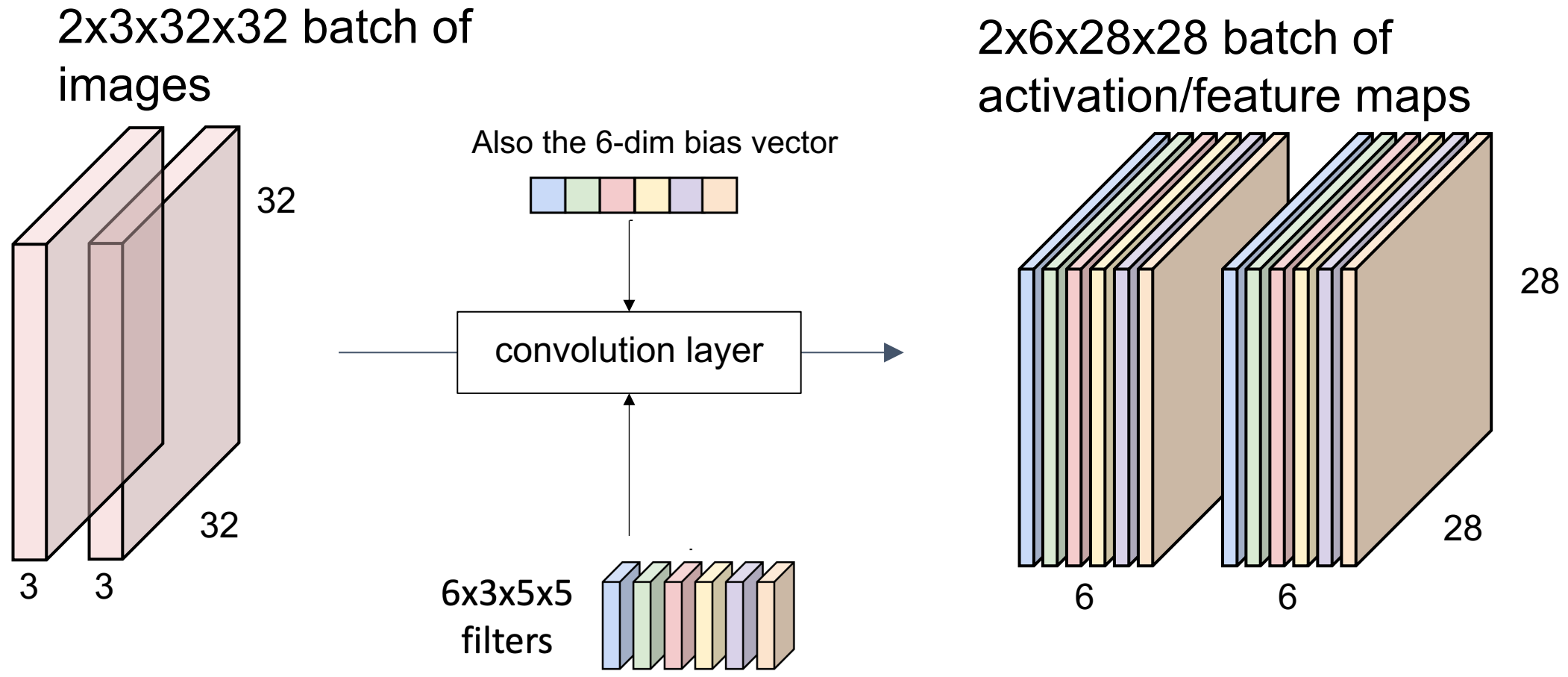


We stack these up to get a "new image" of size 6x28x28

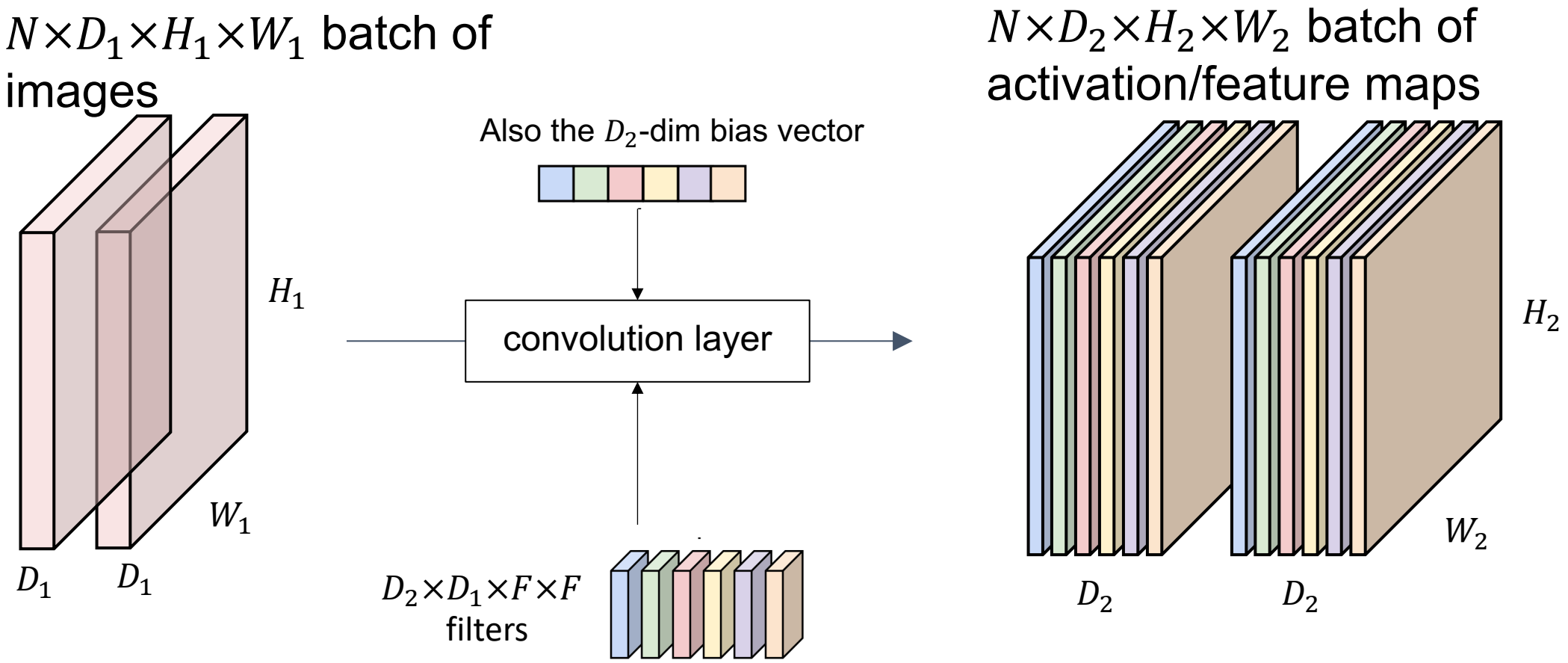
Convolution layer



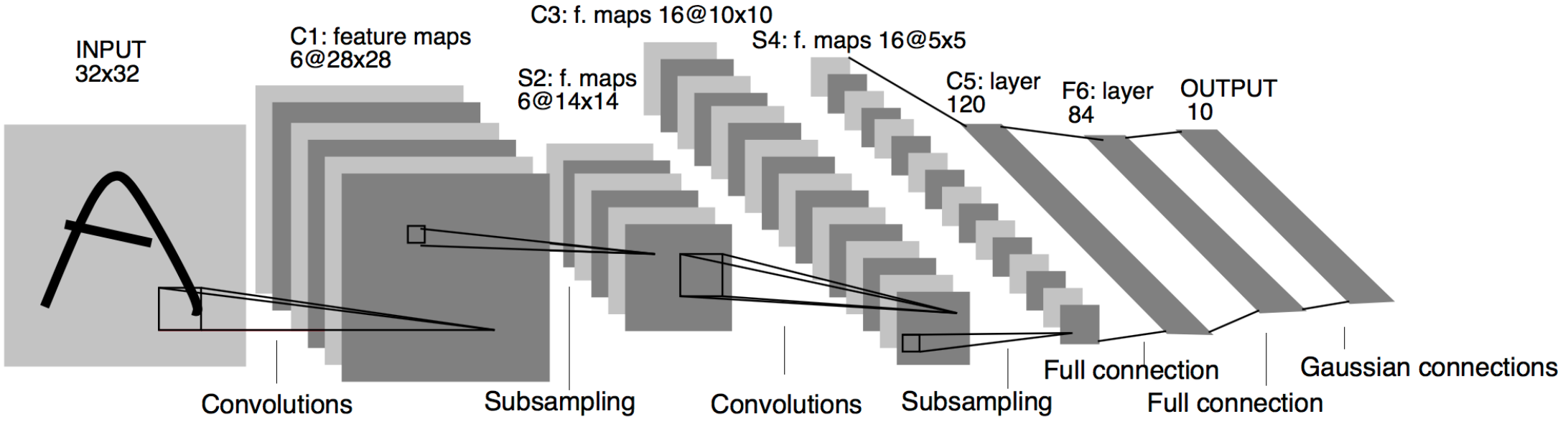
Convolution layer



Convolution layer



An example of convolutional network: LeNet 5

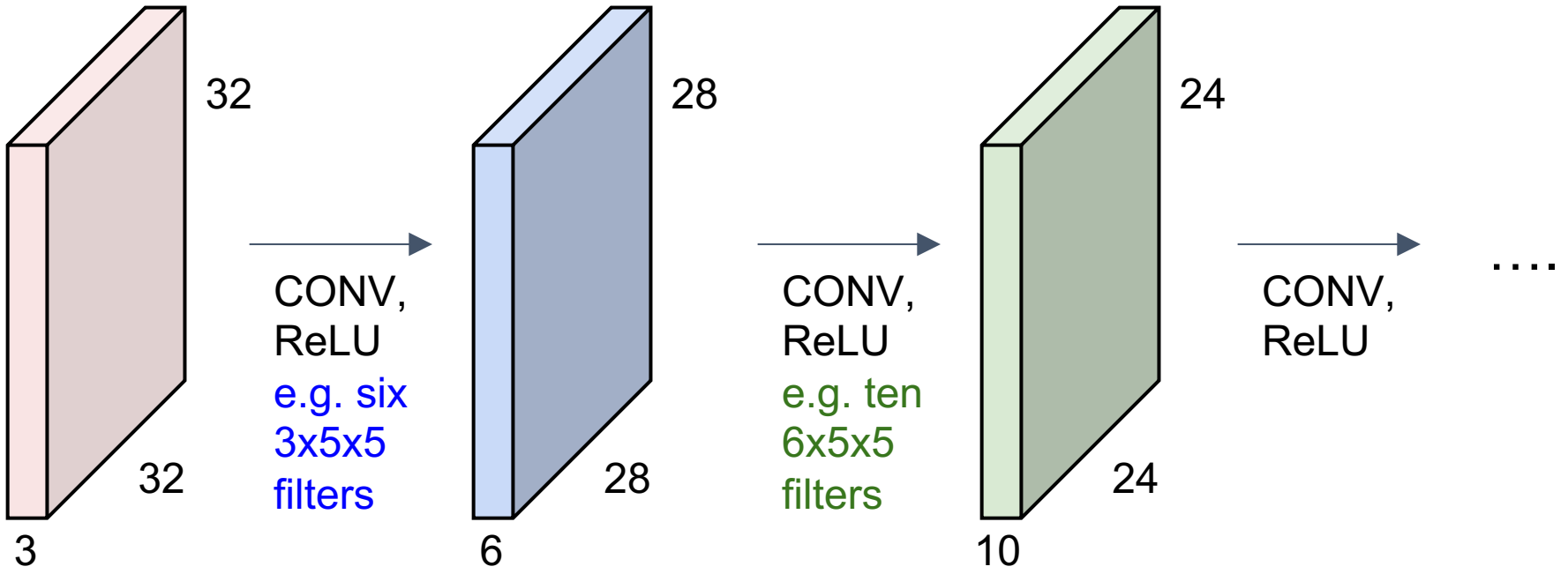


Convolution layer

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

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

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

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

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:

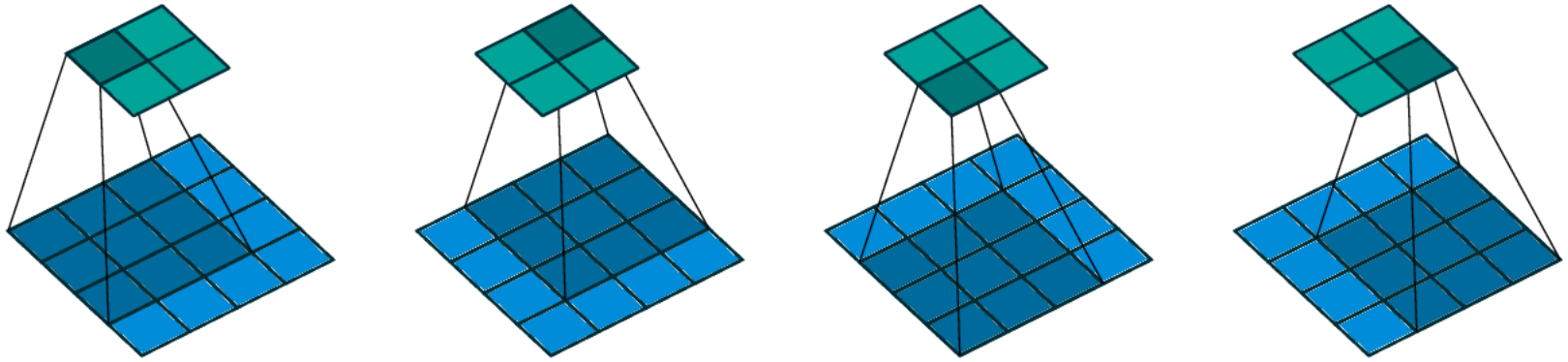
$$\boldsymbol{y}_k = \{y_k(i,j)\}$$

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

Convolution by doing a sliding window



As a guiding example, let us consider the convolution of single-channel tensors $\mathbf{x} \in \mathbb{R}^{4\times4}$ and $\mathbf{w} \in \mathbb{R}^{3\times3}$:

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

Convolution layer

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

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

$$u(1,1) = 4\times1 + 5\times4 + 8\times1 + 1\times1 + 8\times4 + 8\times3 + 3\times3 + 6\times3 + 6\times1 = 122$$

Convolution layer

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

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

$$u(1,1) = 4\times1 + 5\times4 + 8\times1 + 1\times1 + 8\times4 + 8\times3 + 3\times3 + 6\times3 + 6\times1 = 122$$
$$u(1,2) = 5\times1 + 8\times4 + 7\times1 + 8\times1 + 8\times4 + 8\times3 + 6\times3 + 6\times3 + 4\times1 = 148$$

Convolution layer

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

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

$$u(1,1) = 4\times1 + 5\times4 + 8\times1 + 1\times1 + 8\times4 + 8\times3 + 3\times3 + 6\times3 + 6\times1 = 122$$
$$u(1,2) = 5\times1 + 8\times4 + 7\times1 + 8\times1 + 8\times4 + 8\times3 + 6\times3 + 6\times3 + 4\times1 = 148$$
$$u(2,1) = 1\times1 + 8\times4 + 8\times1 + 3\times1 + 6\times4 + 6\times3 + 6\times3 + 5\times3 + 7\times1 = 126$$

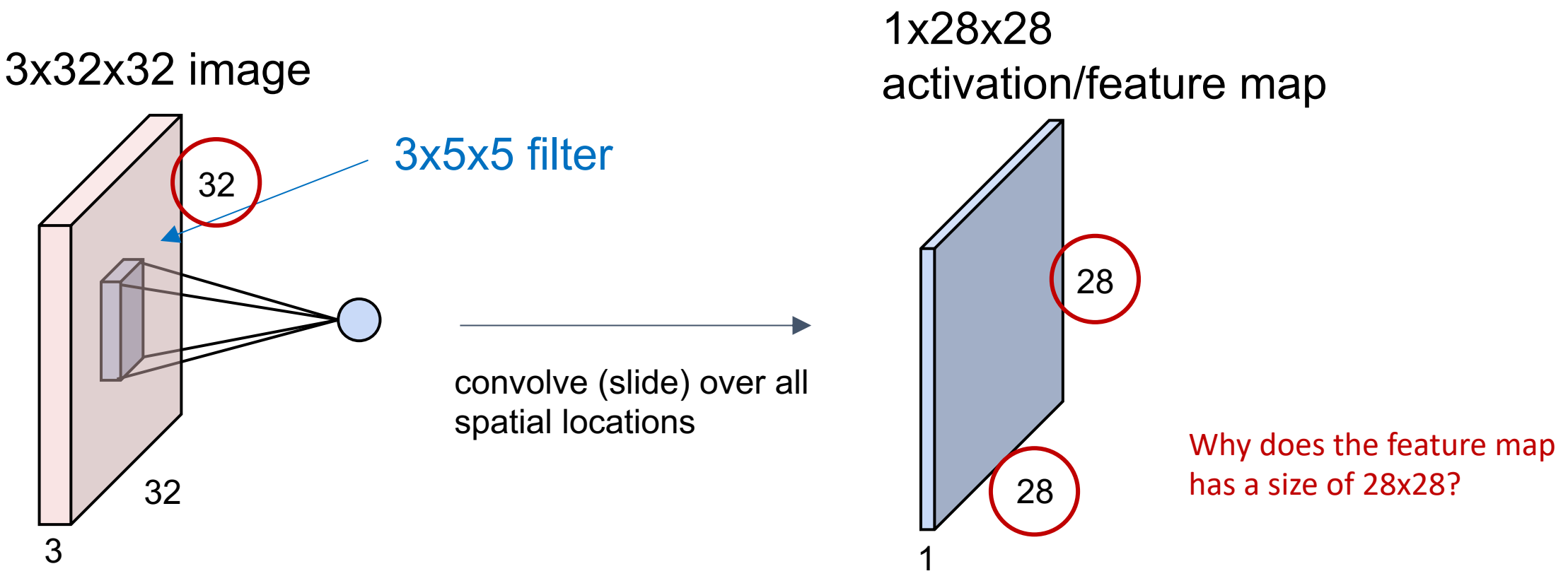
Convolution layer

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

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

$$u(1,1) = 4\times1 + 5\times4 + 8\times1 + 1\times1 + 8\times4 + 8\times3 + 3\times3 + 6\times3 + 6\times1 = 122$$
$$u(1,2) = 5\times1 + 8\times4 + 7\times1 + 8\times1 + 8\times4 + 8\times3 + 6\times3 + 6\times3 + 4\times1 = 148$$
$$u(2,1) = 1\times1 + 8\times4 + 8\times1 + 3\times1 + 6\times4 + 6\times3 + 6\times3 + 5\times3 + 7\times1 = 126$$
$$u(2,2) = 8\times1 + 8\times4 + 8\times1 + 6\times1 + 6\times4 + 4\times3 + 5\times3 + 7\times3 + 8\times1 = 134$$

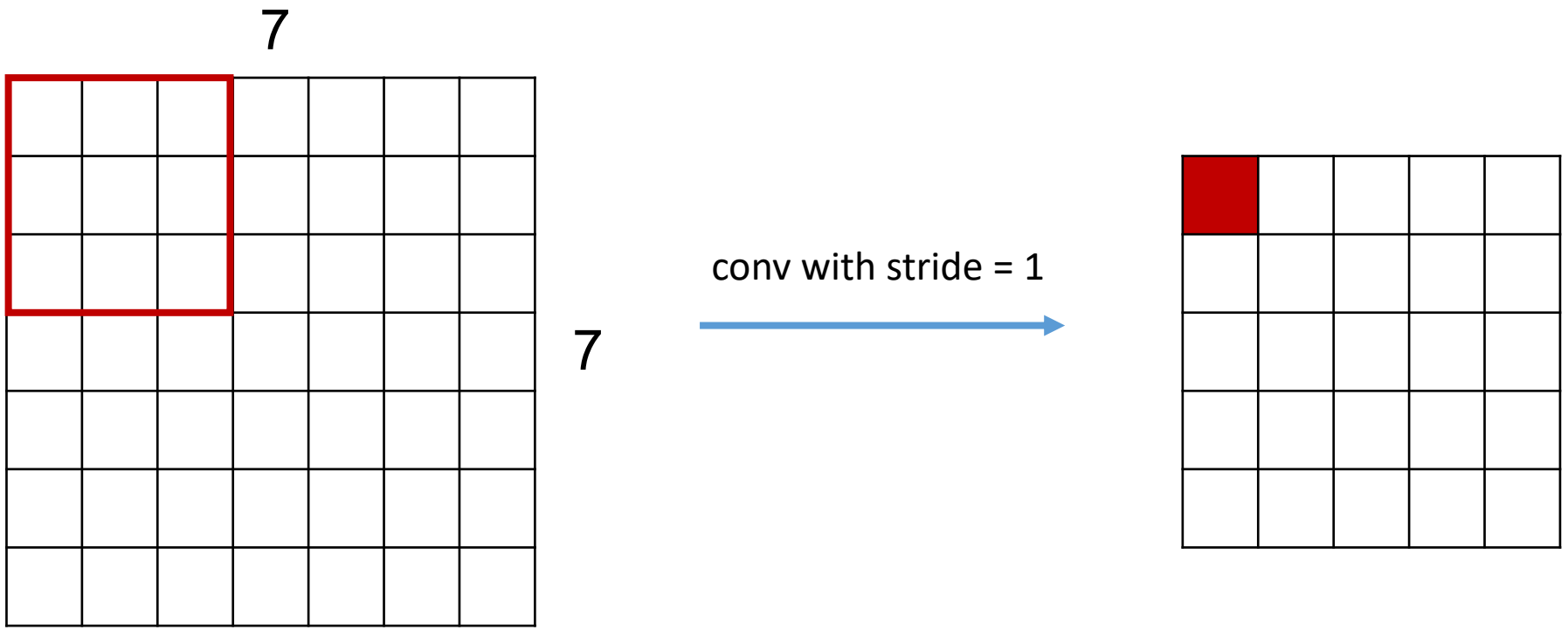
Convolution layer – spatial dimensions



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

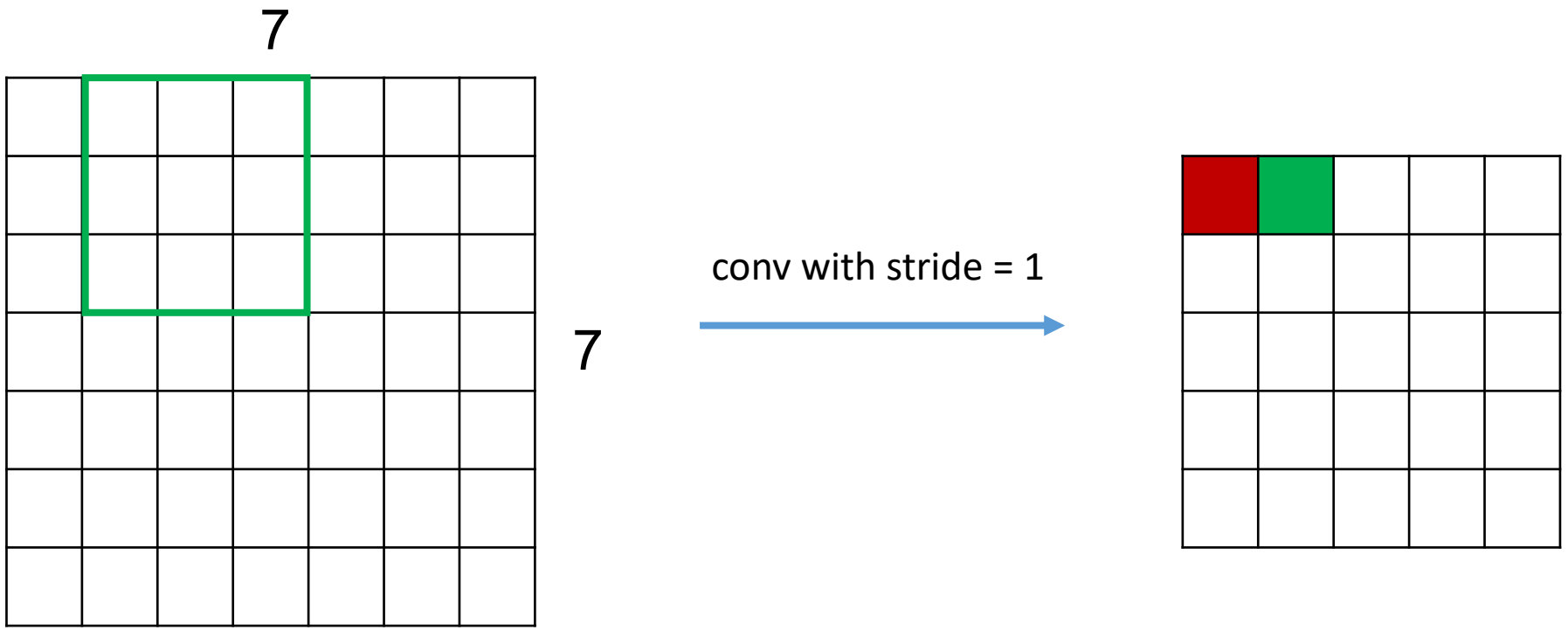
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

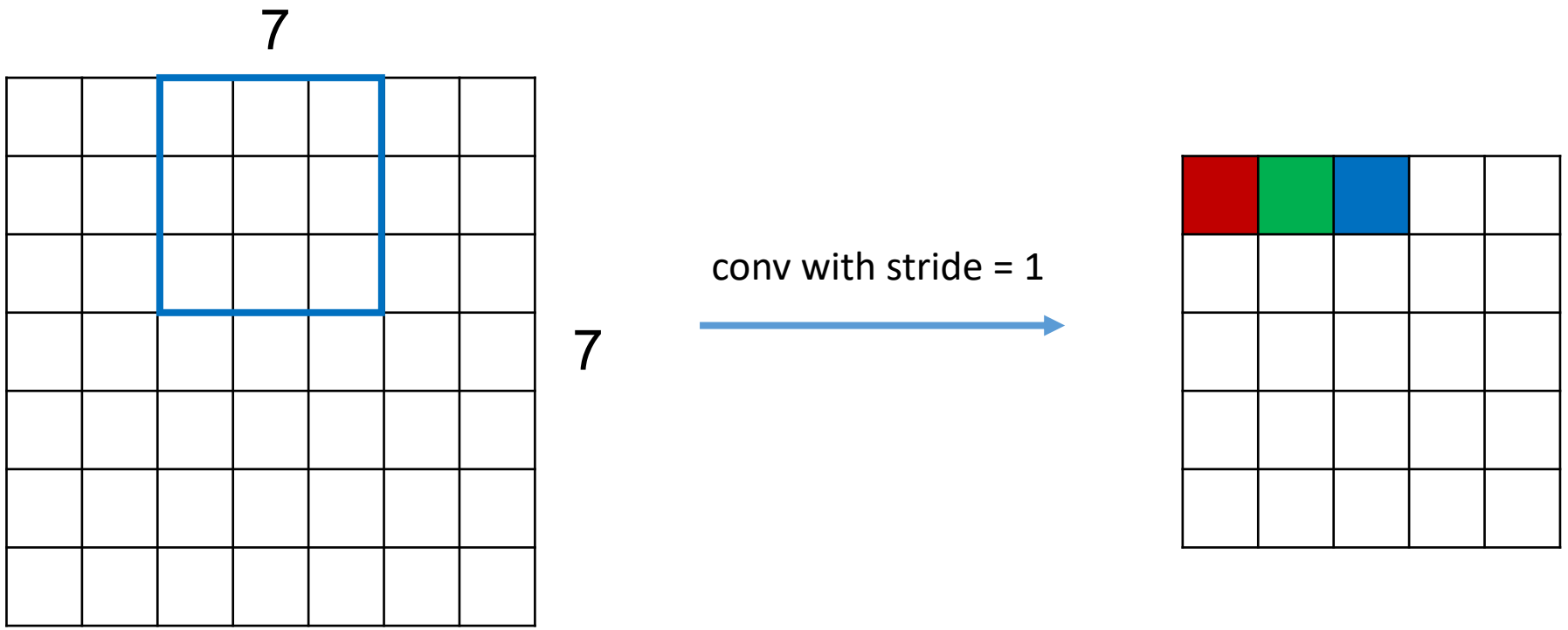
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

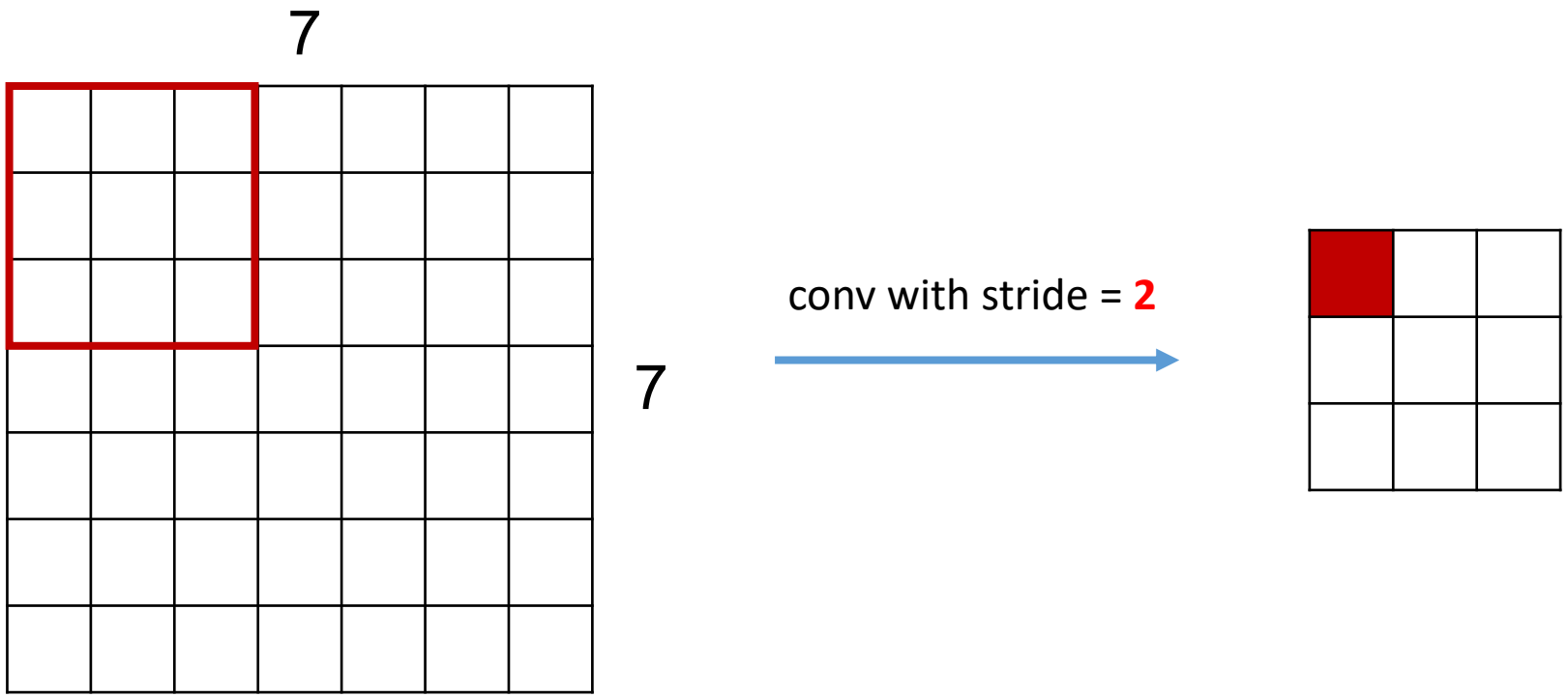
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

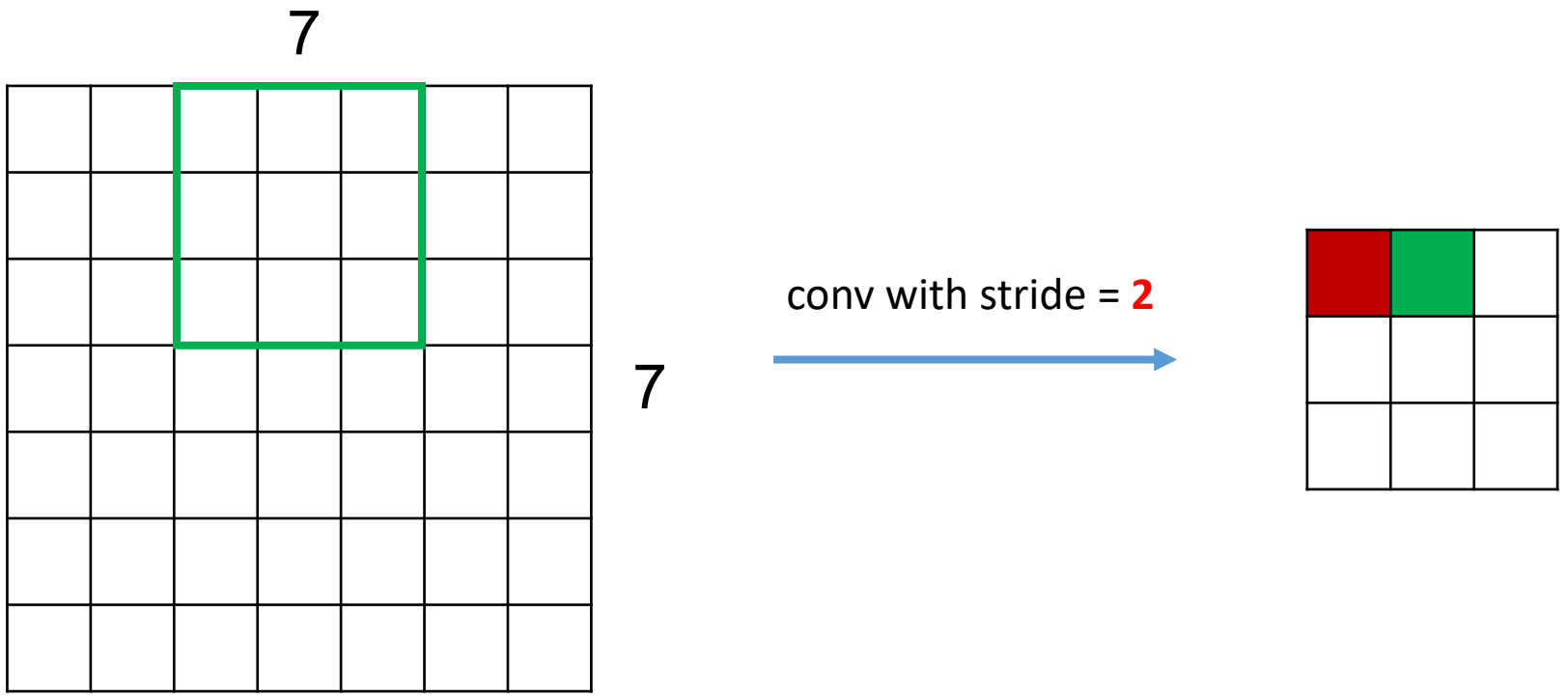
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

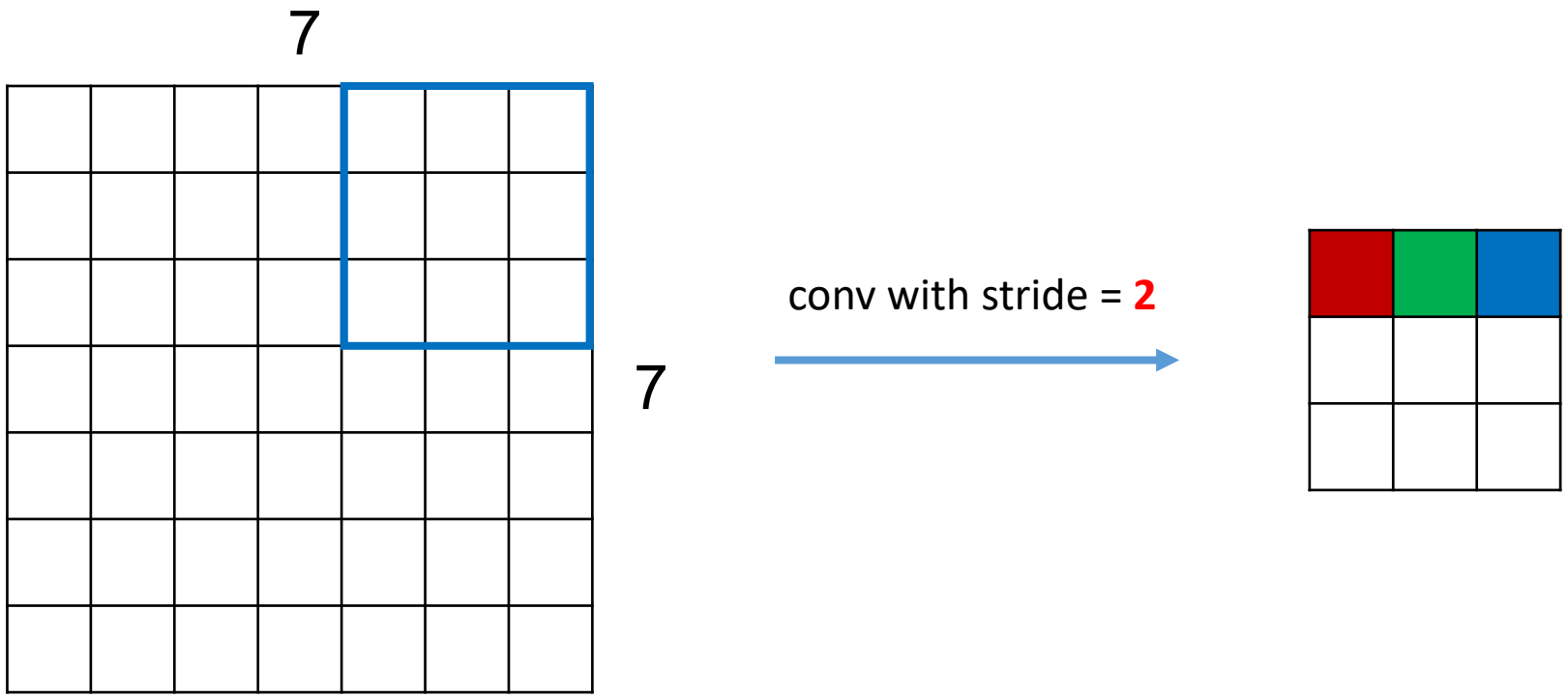
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



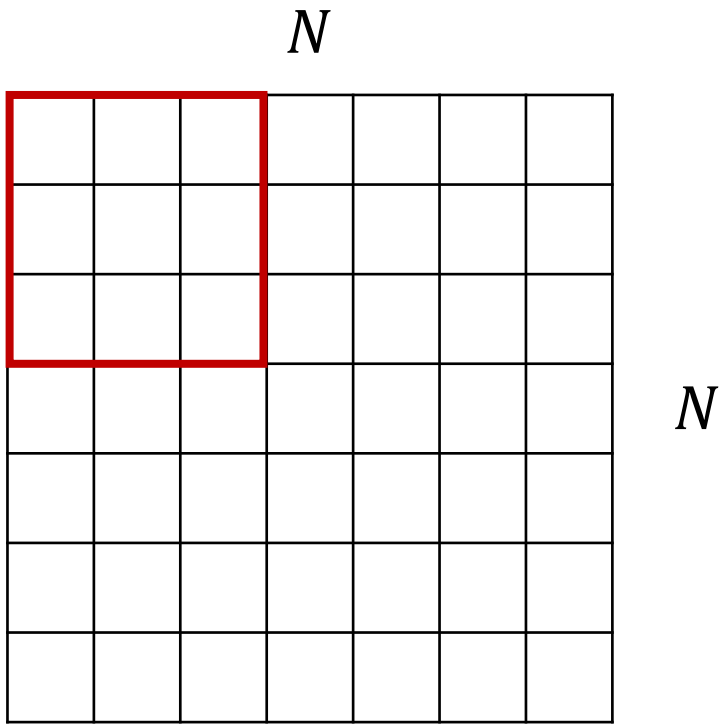
Convolution layer – spatial dimensions

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

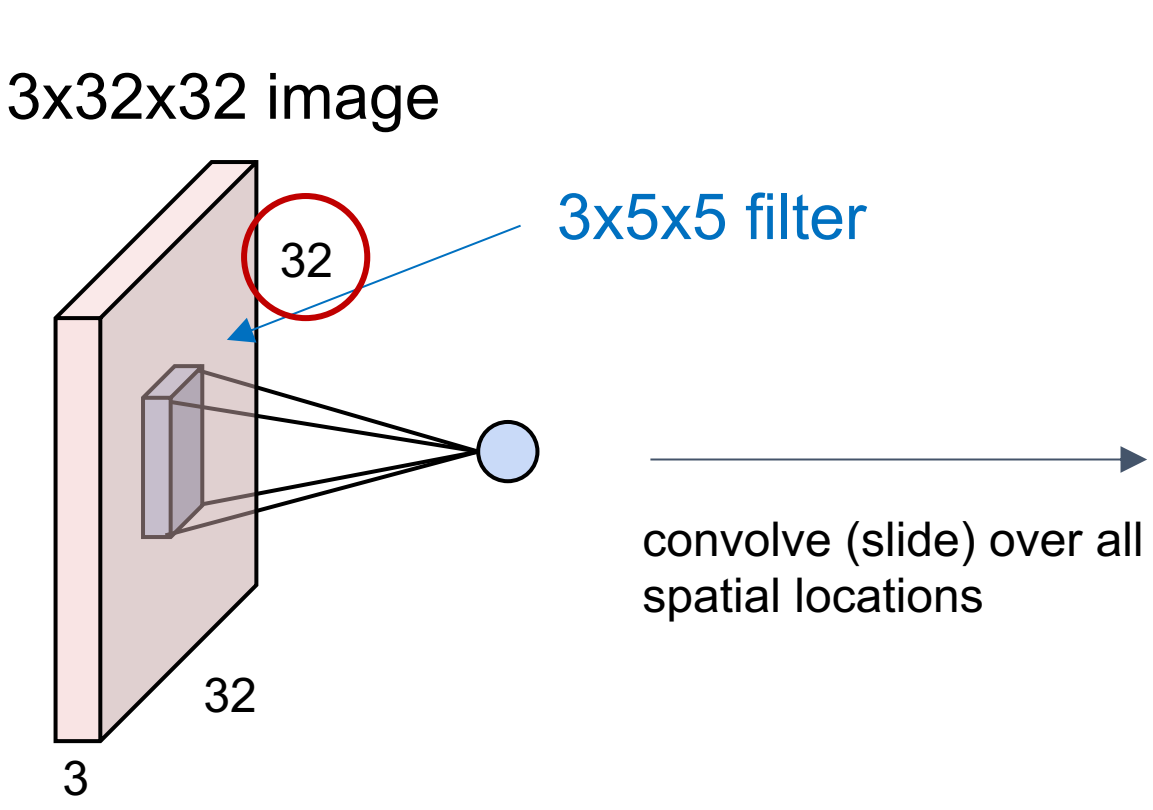
Output size = $\frac{N-F}{S} + 1$

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



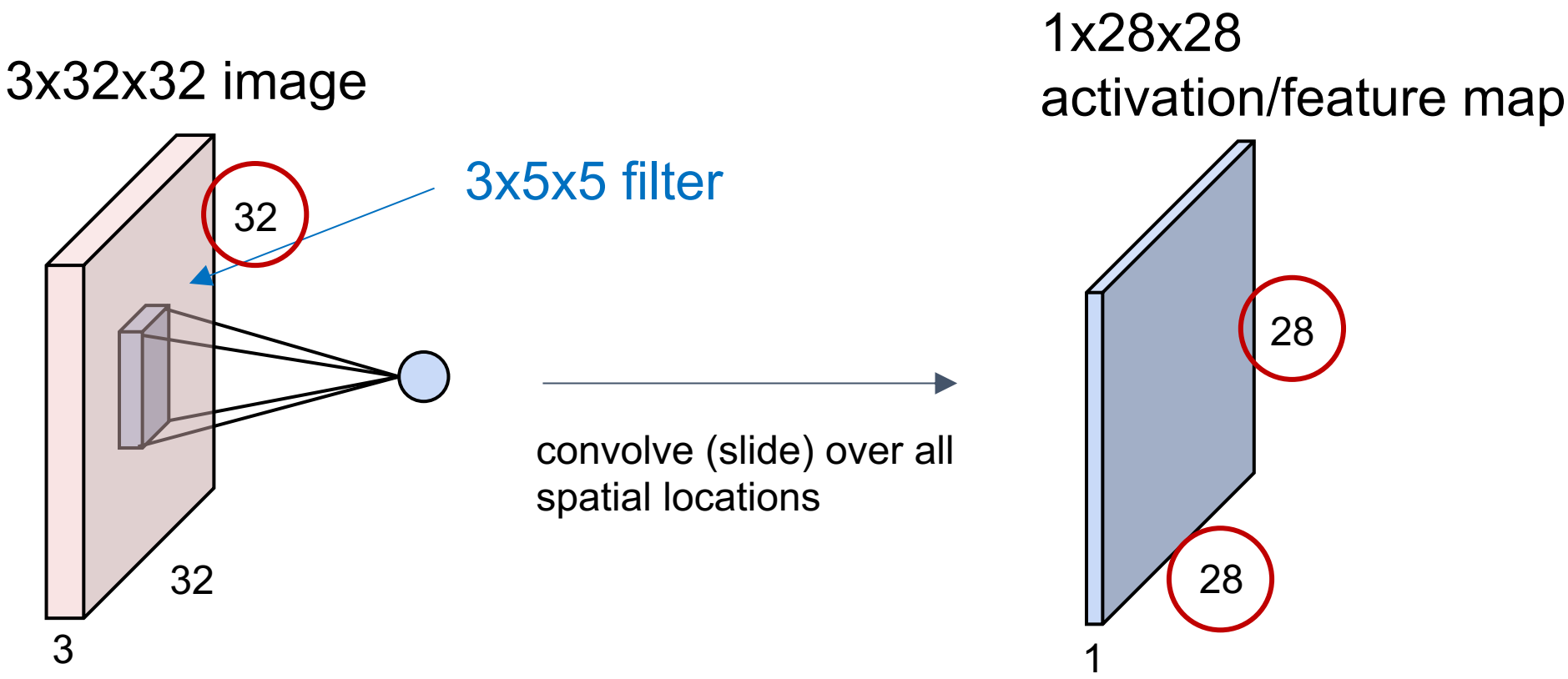
Why does the feature map has a size of 28x28?

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).

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?

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, e.g., 'constant', 'reflect', and 'replicate').

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

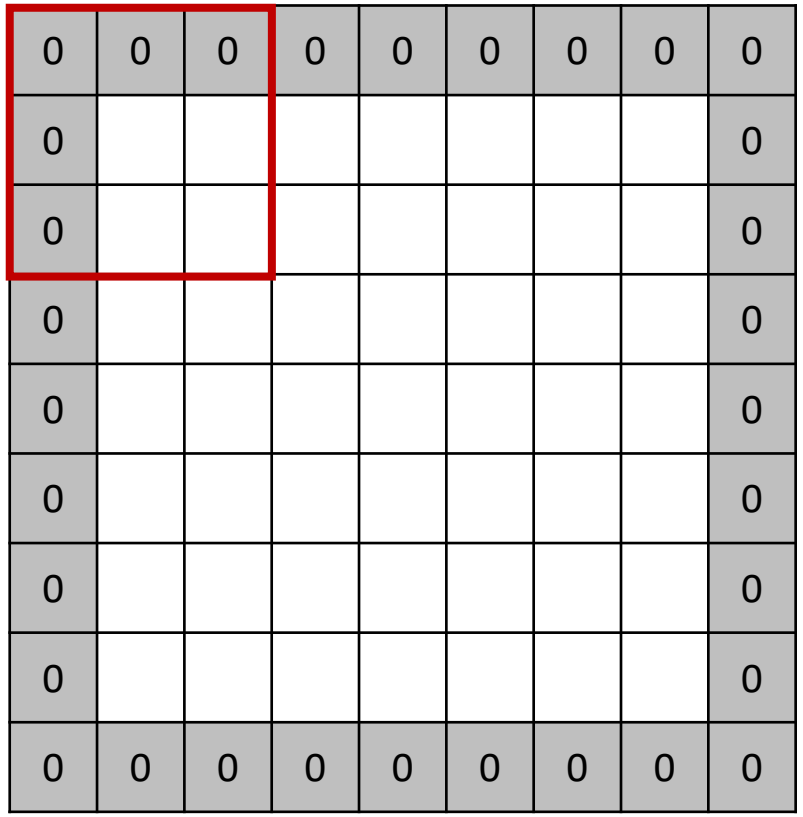
e.g.

$$N = 7, F = 3$$

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).



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

7×7 output!

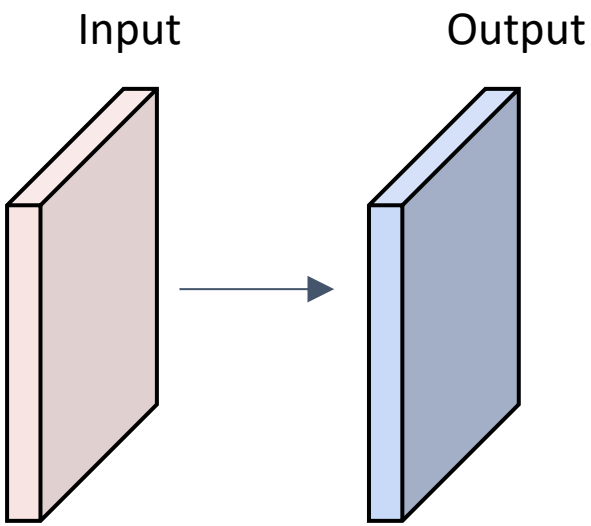
In general, common to see CONV layers with stride 1, filters of size $F \times F$, and zero-padding with $(F-1)/2$. (will preserve size spatially)

e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Example

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

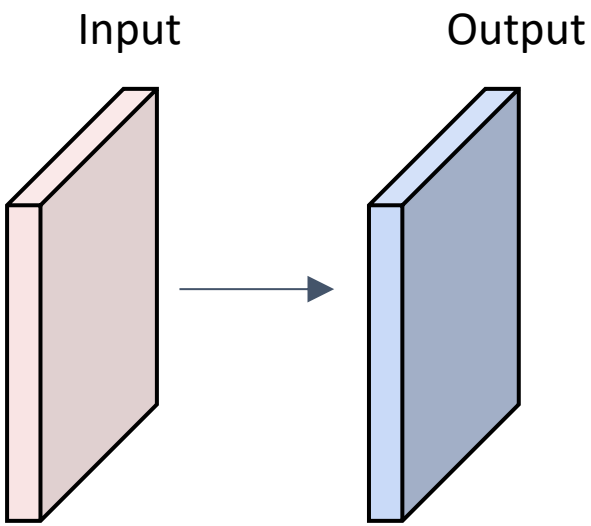
Output volume size: ?



Example

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

Output volume size: ?



Output size =

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

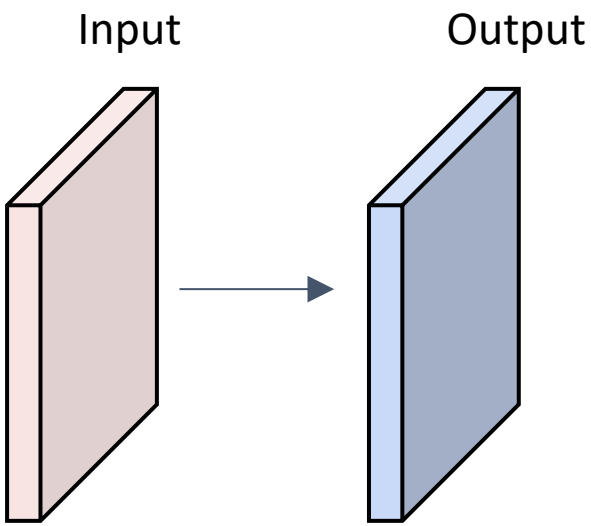
$$\frac{32-5+2(2)}{1}+1 = 32 \text{ spatially}$$

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

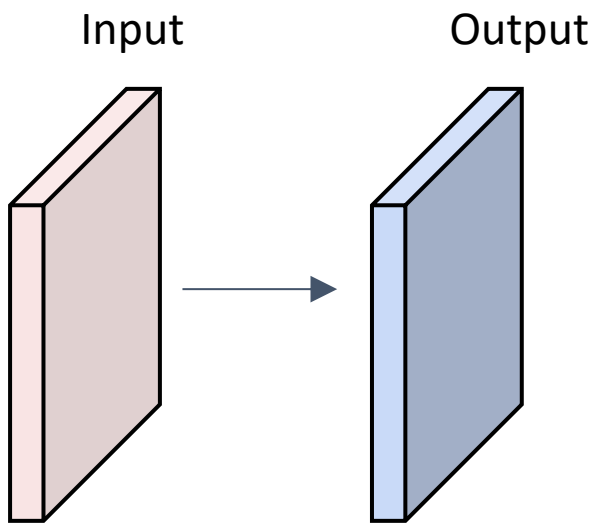
Number of parameters in this layer: ?



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: ?

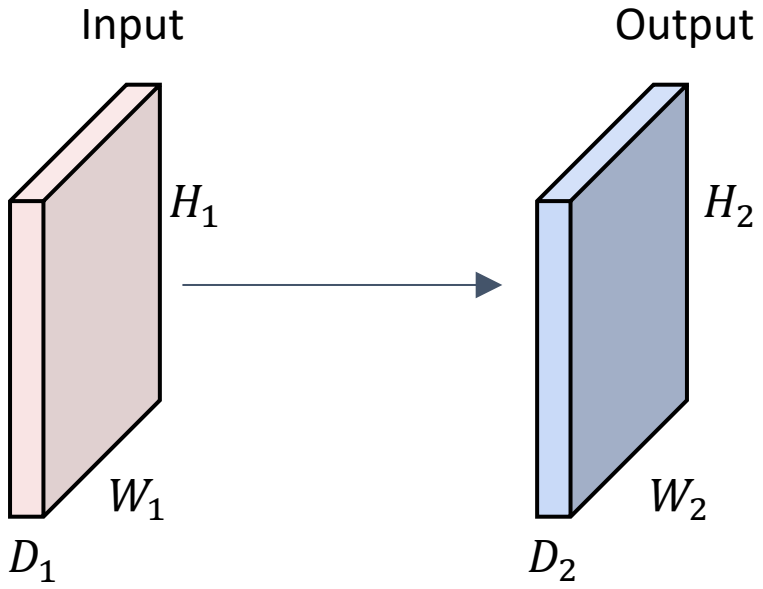


Each filter has $5 \times 5 \times 3 + 1 = 76$ params (+1 for bias)
=> $76 \times 10 = \mathbf{760}$

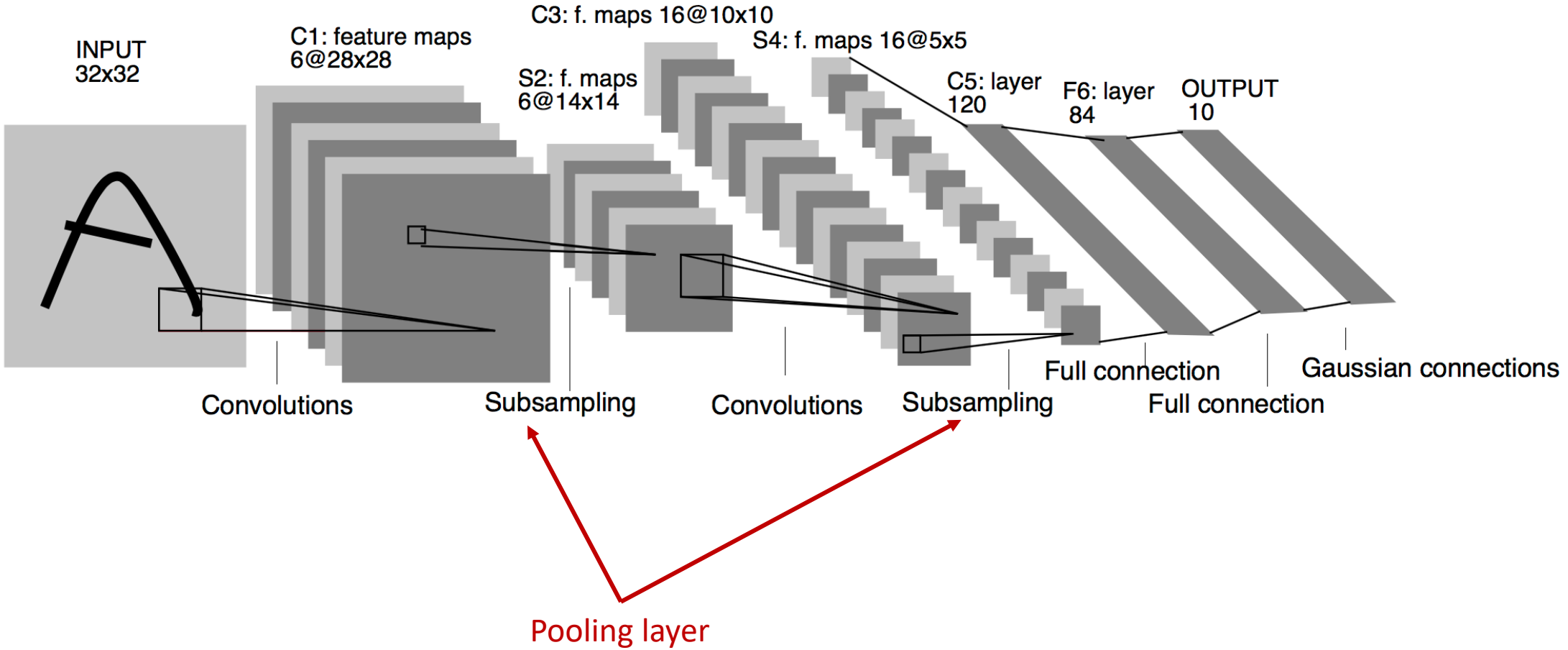
Convolution layer - summary

A convolution layer

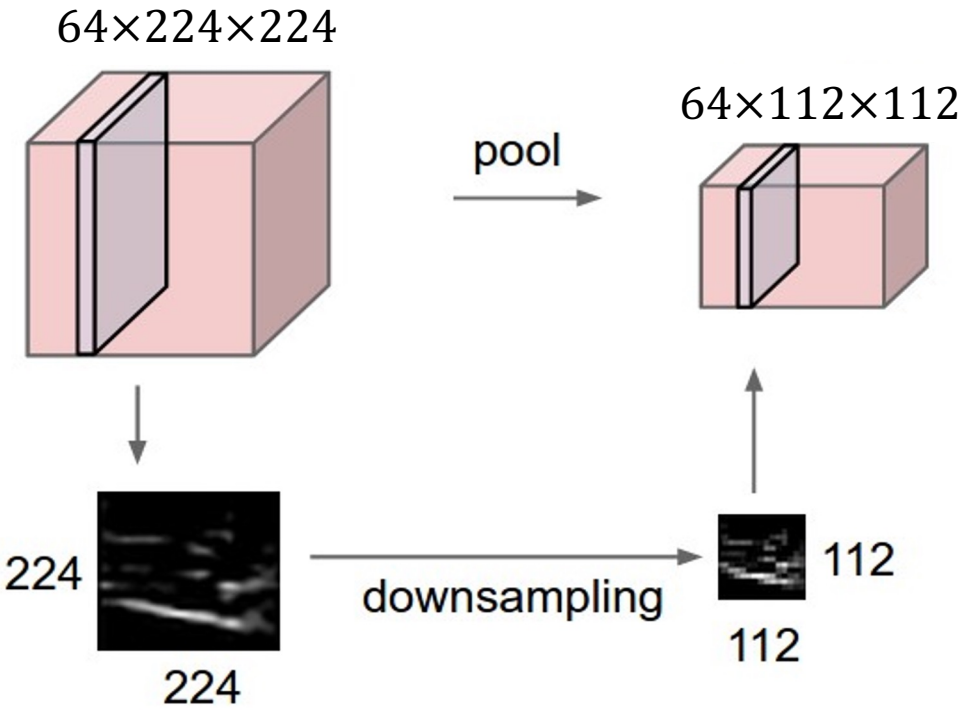
- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters
  - Number of filters K
  - Their spatial extent F
  - The stride S
  - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (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

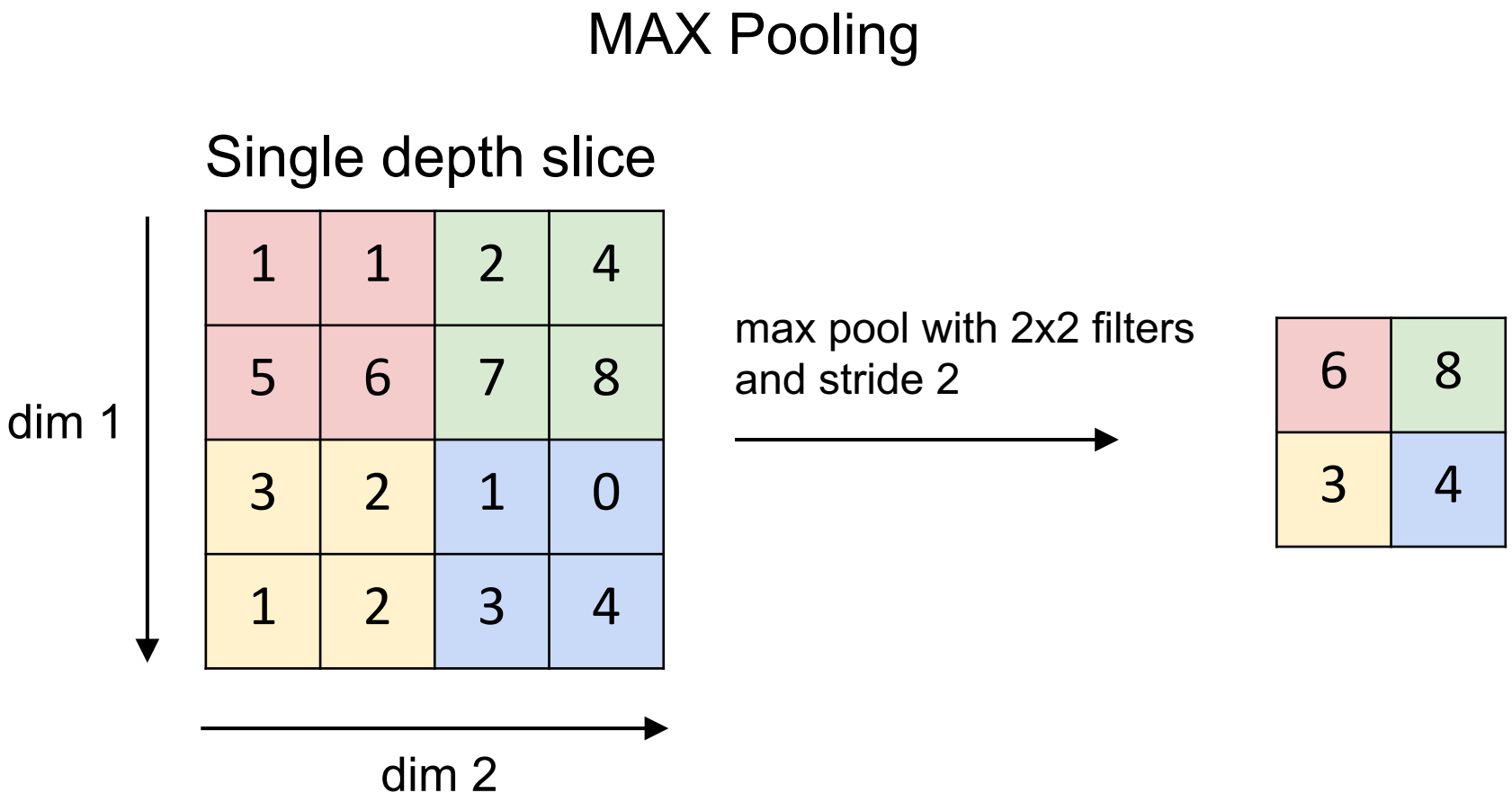


Pooling layer



- 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 $f(g(\boldsymbol{x})) = f(\boldsymbol{x})$.

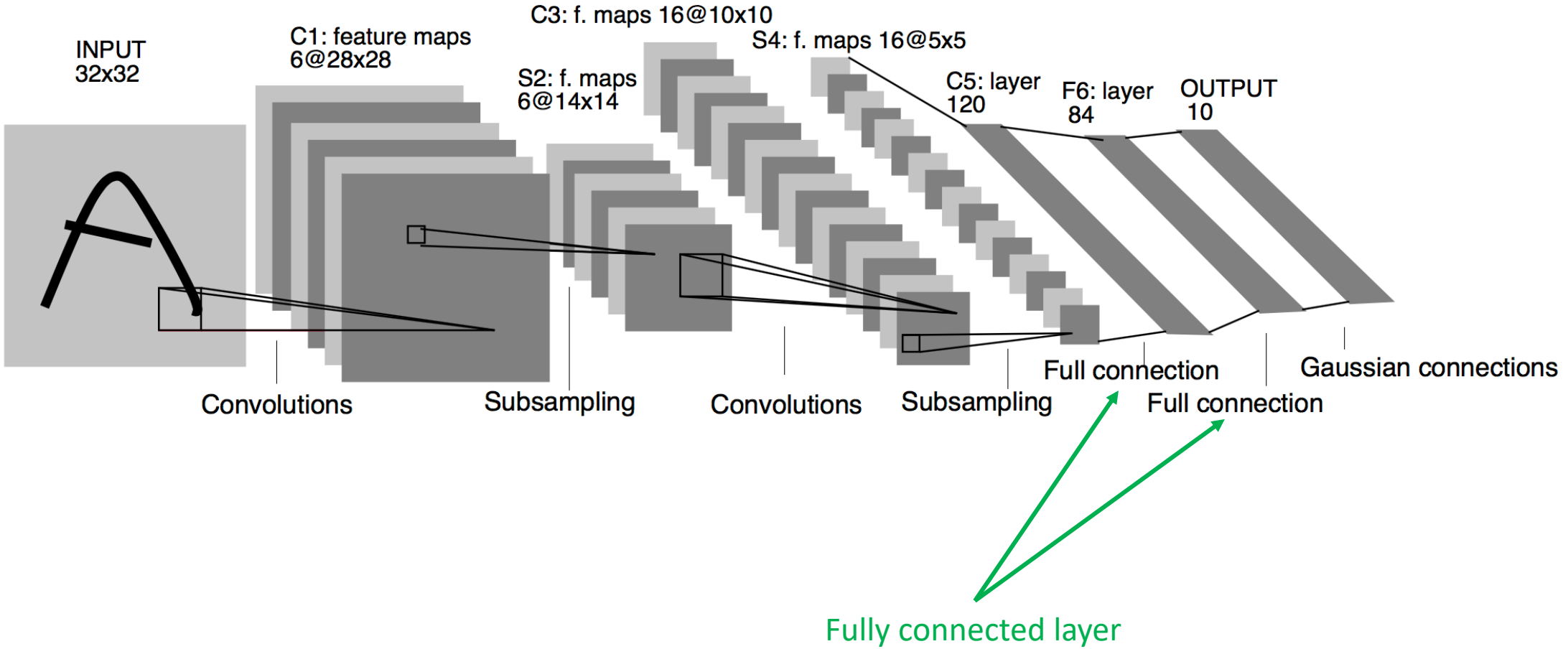
- Pooling layers can be used for building inner activations that are (slightly) invariant to small translations of the input.
- Invariance to local translation is helpful if we care more about the presence of a pattern rather than its exact position.

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

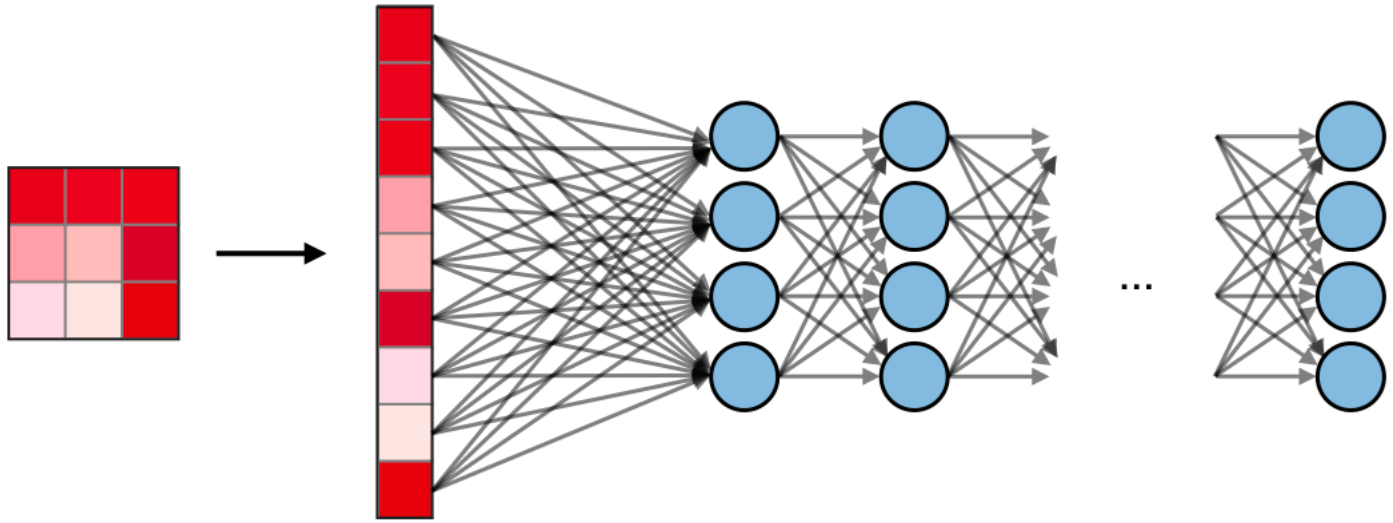
An example of convolutional network: LeNet 5



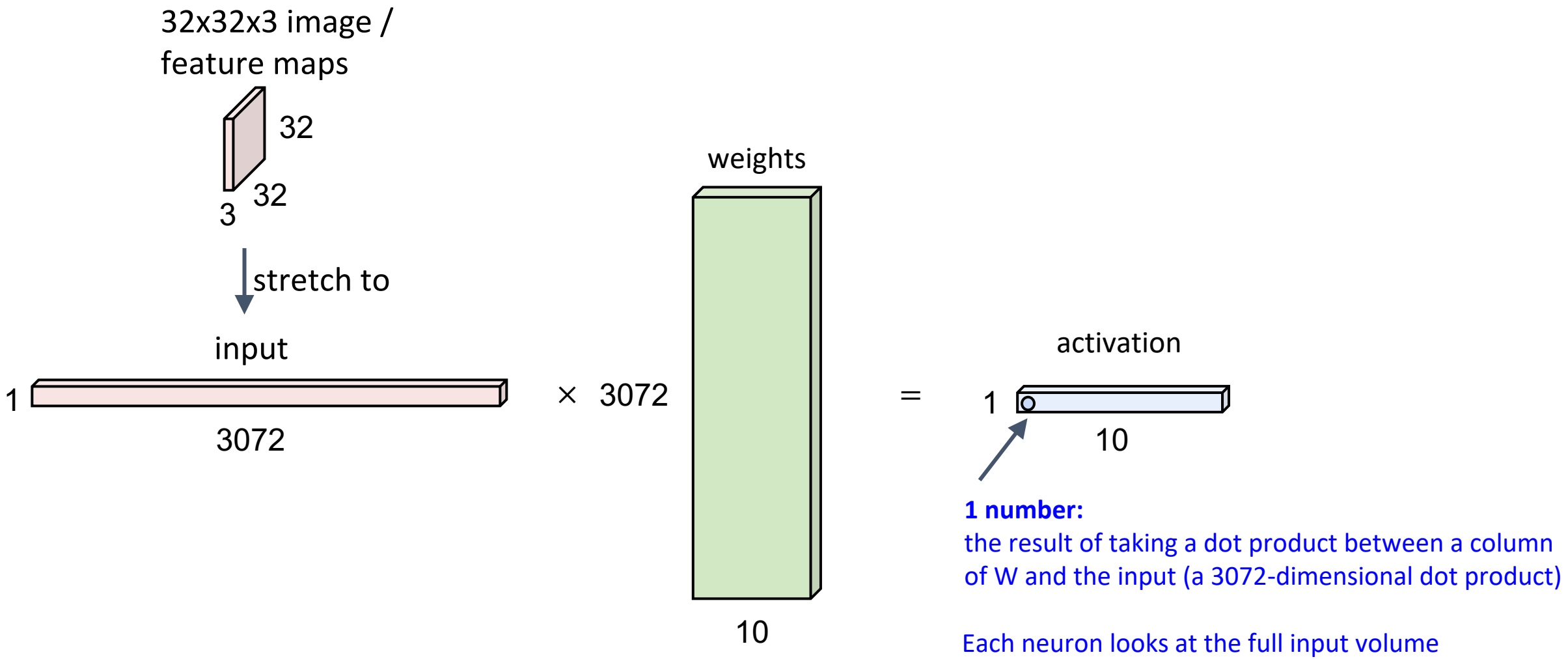
Fully connected layer

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

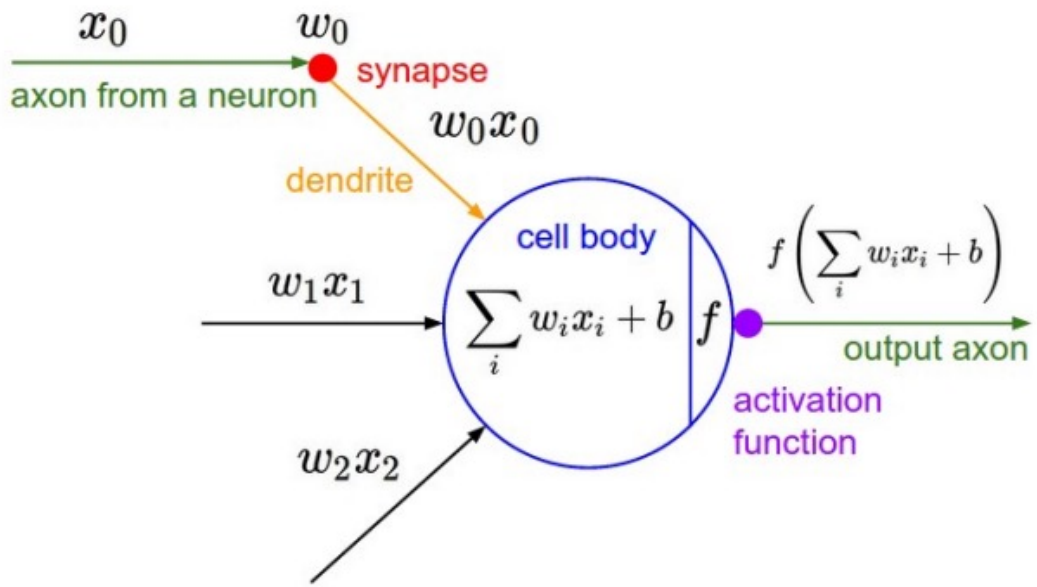


Activation function

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

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

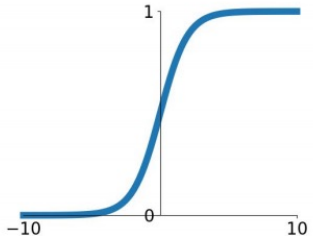
where u is the synaptic input and f is an activation function (It aims at introducing non-linearities to the network.).



Activation function

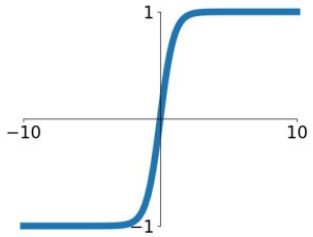
Sigmoid

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



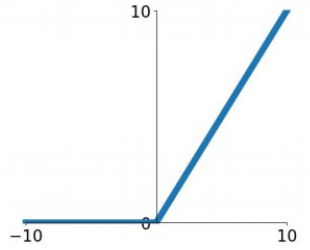
tanh

$$\tanh(x)$$



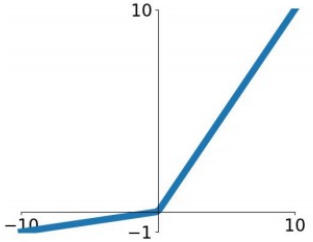
ReLU

$$\max(0, x)$$



Leaky ReLU

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

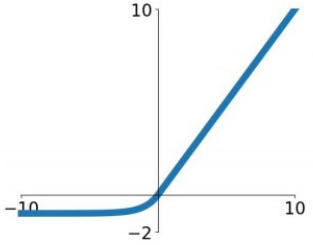


Maxout

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

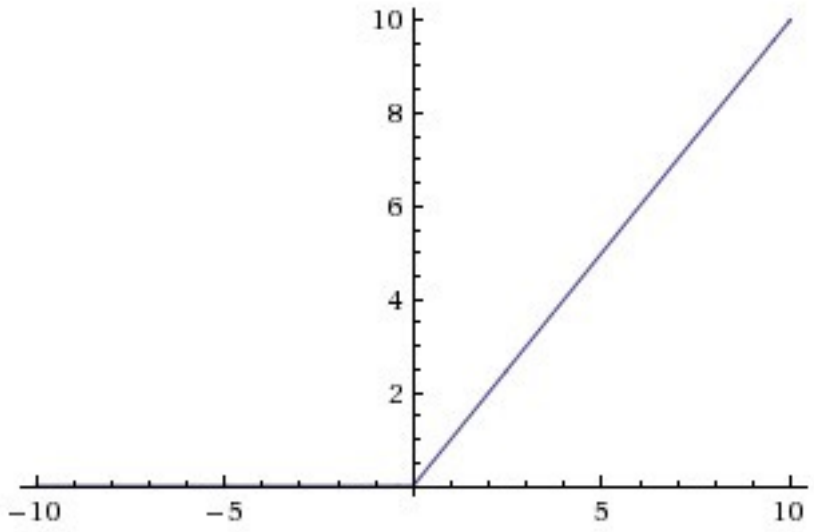
ELU

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



The activation function is usually an abstraction representing the rate of firing in the cell

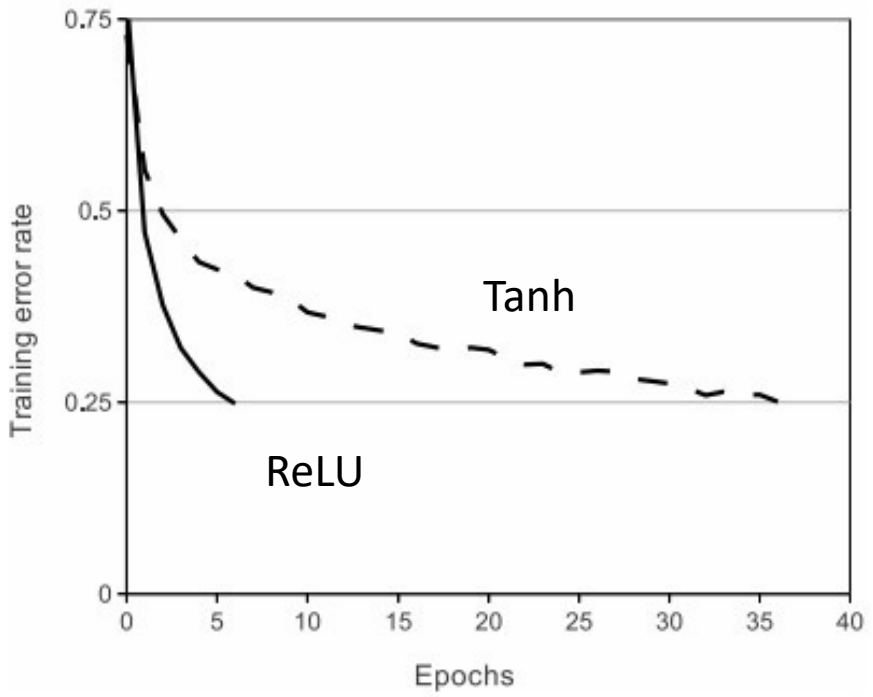
Activation function – ReLU



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

- Rectified Linear Unit (ReLU) activation function, which is zero when x < 0 and then linear with slope 1 when x > 0

Activation function – ReLU



- (+) It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.

- (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.

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,

a) Find the outputs of the convolution layer if the padding is VALID at strides = 1
b) Assume the pooling layer uses max pooling, has a pooling window size of 2x2, and strides = 2, find the activations at the pooling layer.
c) Repeat (a) and (b) using convolution layer with SAME padding

Example 1

Find the outputs of the convolution layer if the padding is VALID at strides = 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:

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

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

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

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

⋮

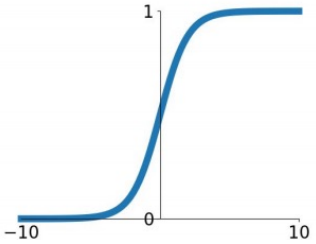
Example 1

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:

$$(0.794)$$

Now find the outputs of the convolution layer if the padding is SAME at strides = 1

See eg6.1.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:

$$w_1 = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \quad w_2 = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}, \quad w_3 = \begin{pmatrix} 3 & 4 & 3 \\ 4 & 5 & 4 \\ 3 & 4 & 3 \end{pmatrix}$$

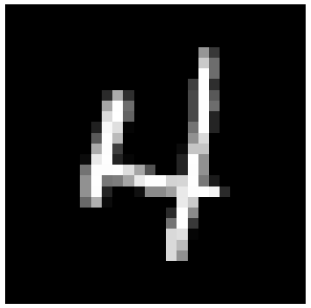
Find the feature maps at the the convolution layer and pooling layer. Assume zero bias

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.

See eg6.2.ipynb

Example 2



Original Image 28 x 28

Output of convolution layer
3 x 26 x 26

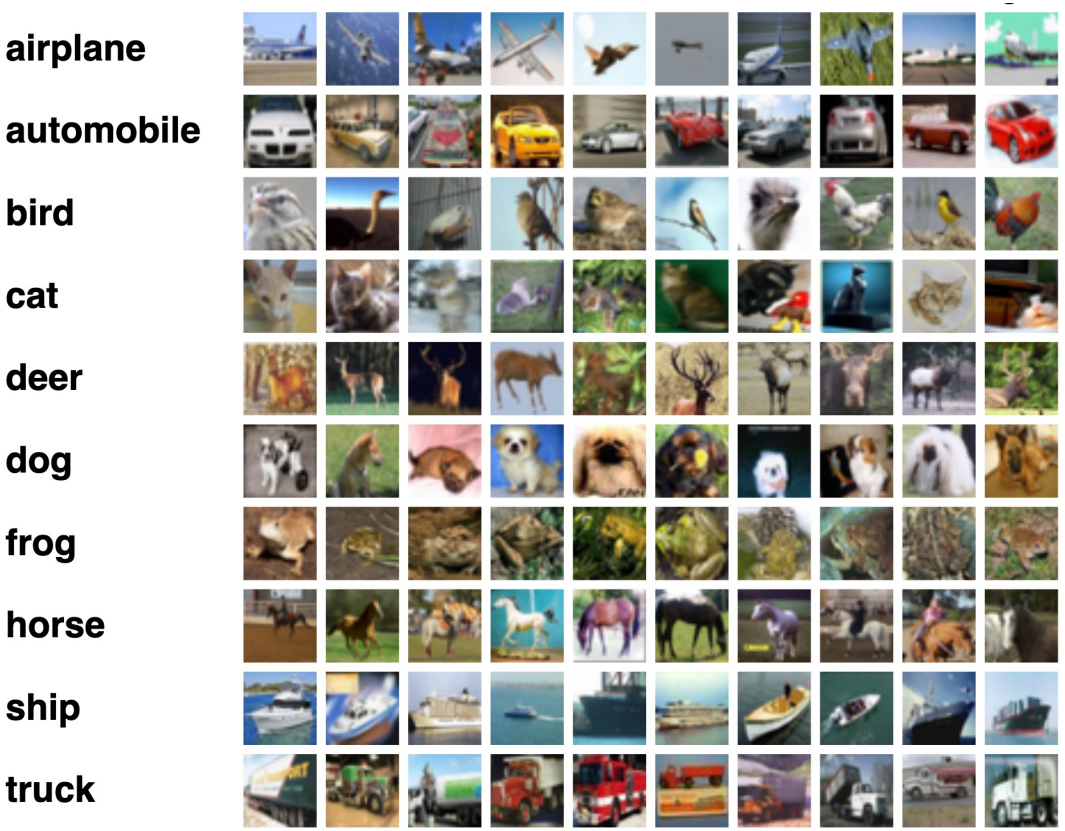
Output of pooling layer
3 x 13 x 13

Outline

- Basic components in CNN
- Training a classifier
- Optimizers

Training a Classifier

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

Multi-class classification

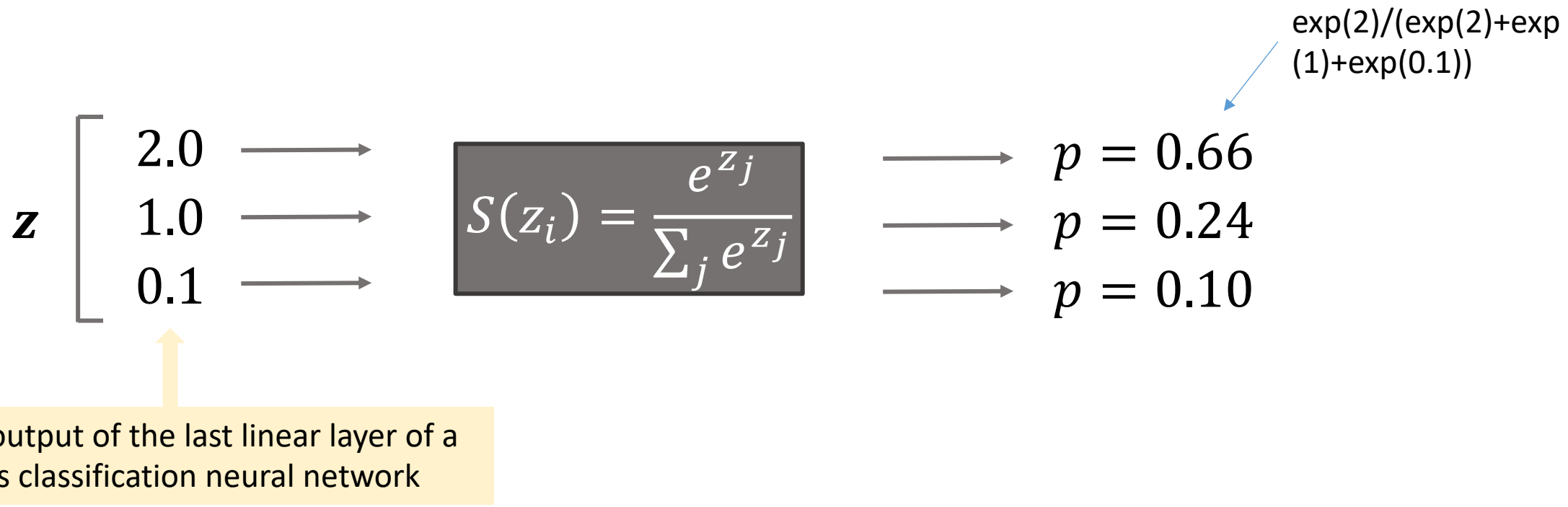
MNIST

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

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

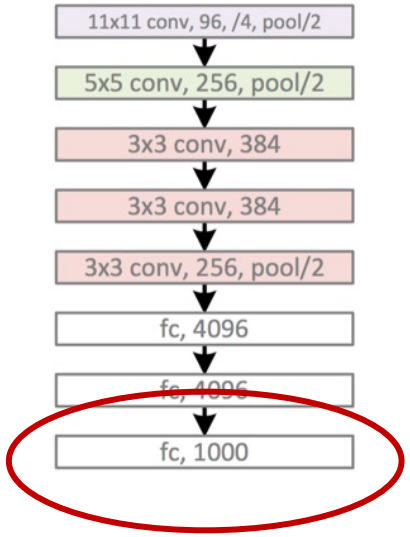
Softmax function

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.

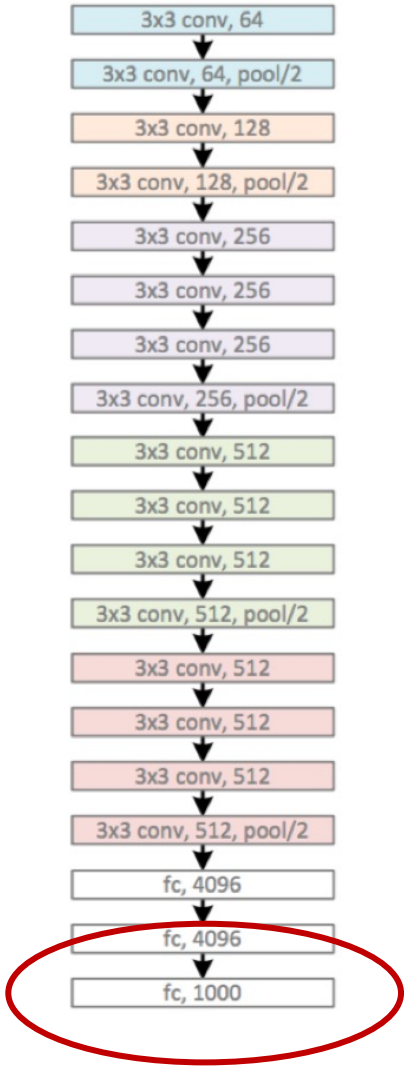


Softmax function

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

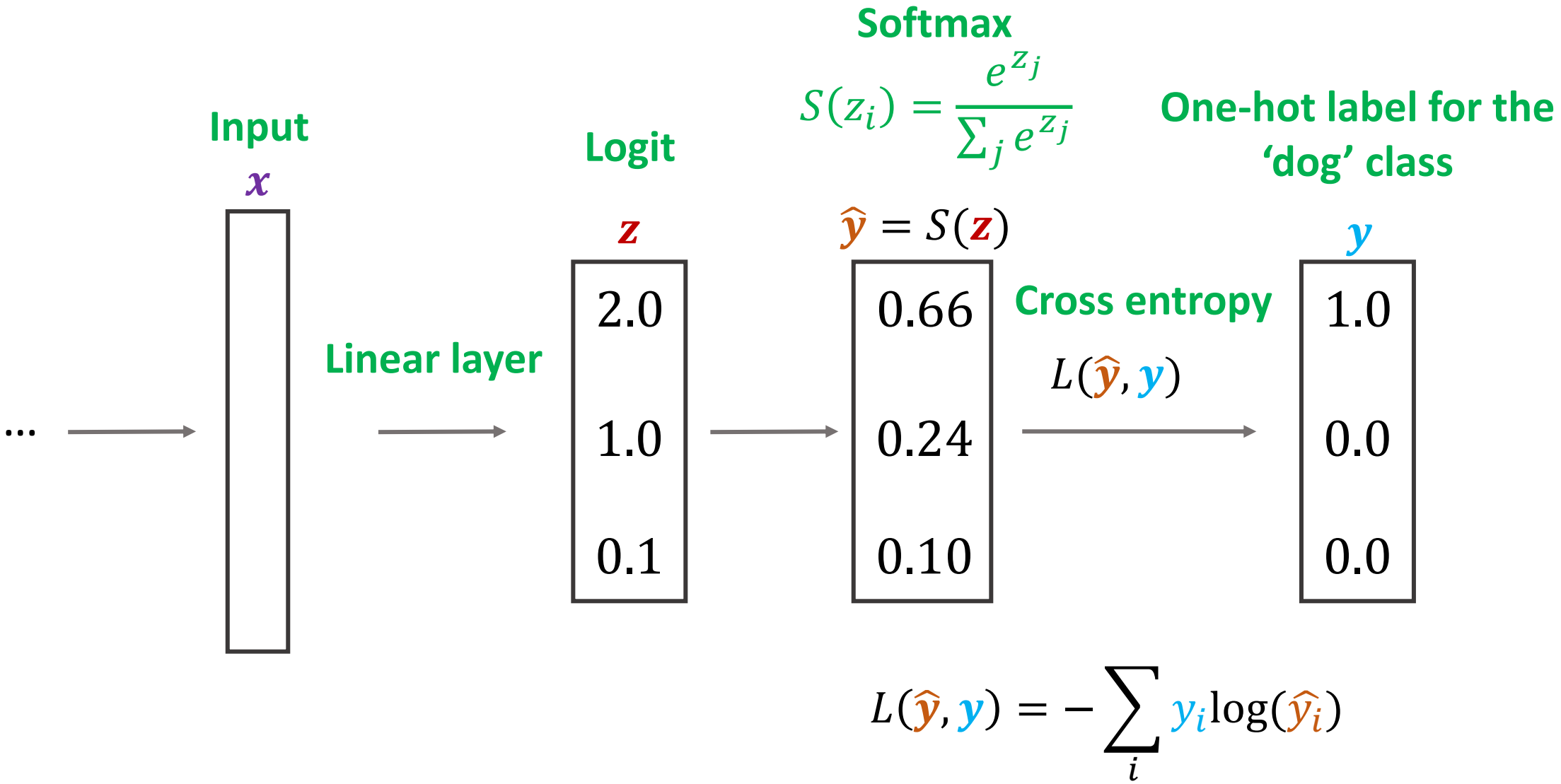
Cross entropy loss

One-hot encoded ground truth

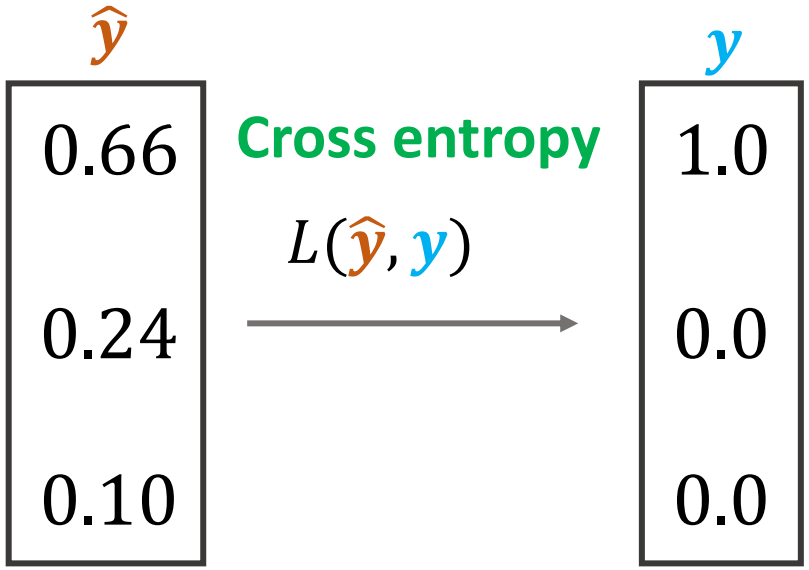
Example:

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

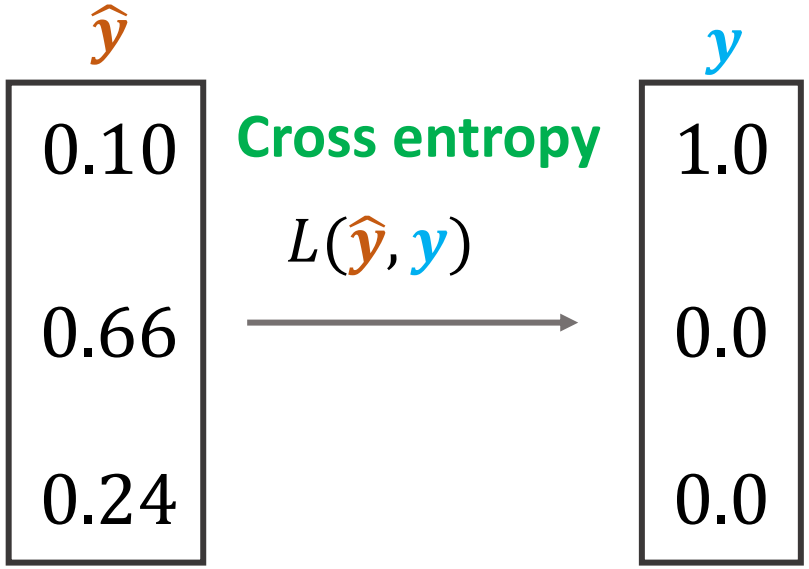
Cross entropy loss



Cross entropy loss



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



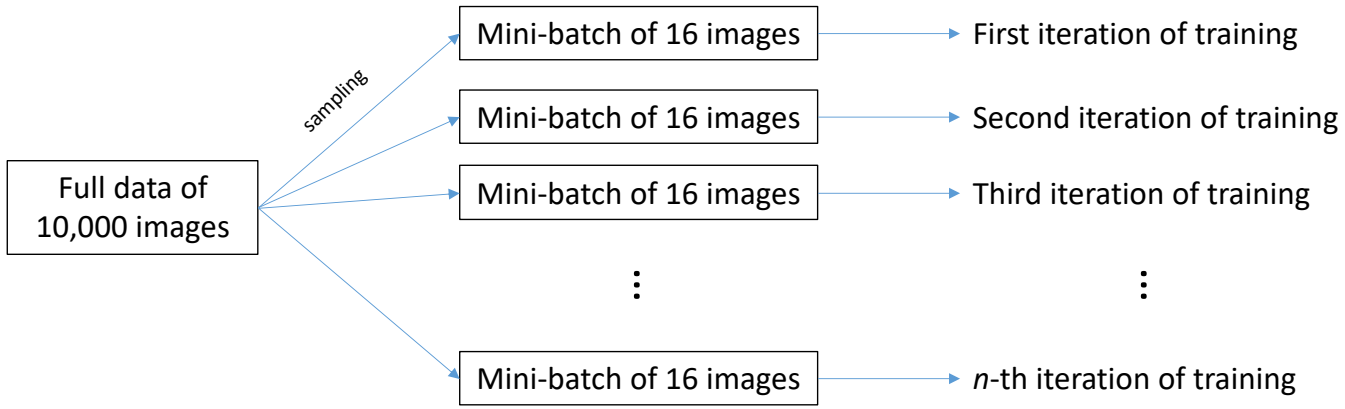
$$L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$
$$= -[(1 \times \log_2(0.10)) + (0 \times \log_2(0.66)) + (0 \times \log_2(0.24))]$$
$$= 3.32$$

What is the min / max possible loss?

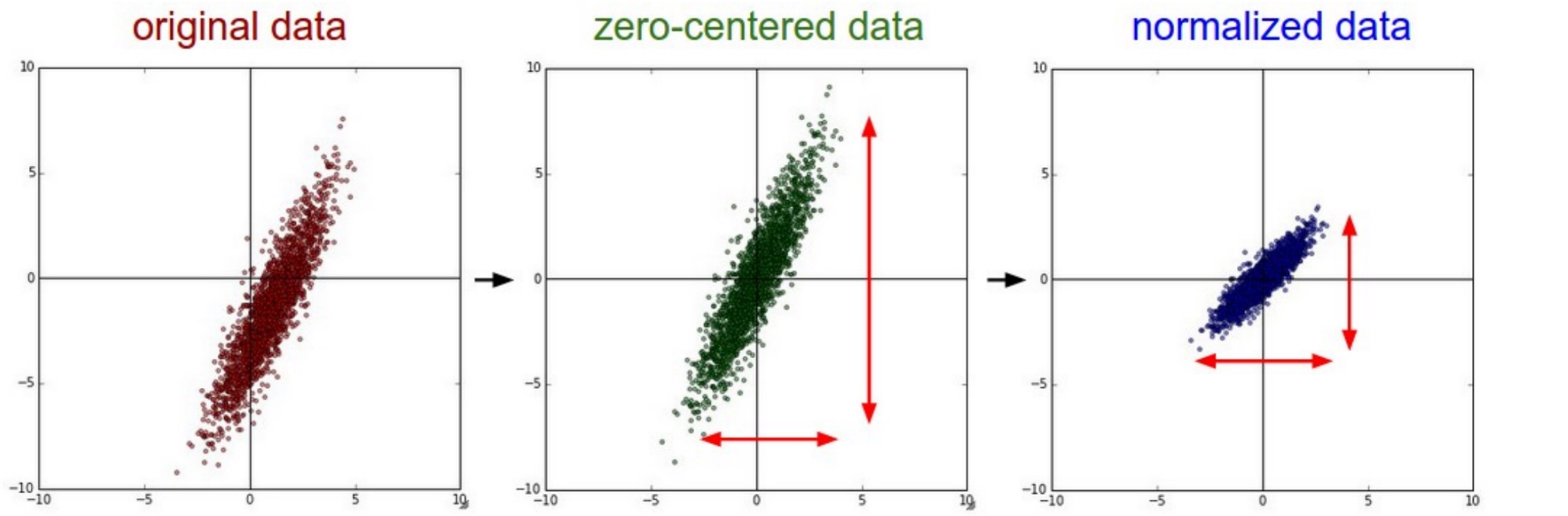
Epoch and mini-batch

- **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** – 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.



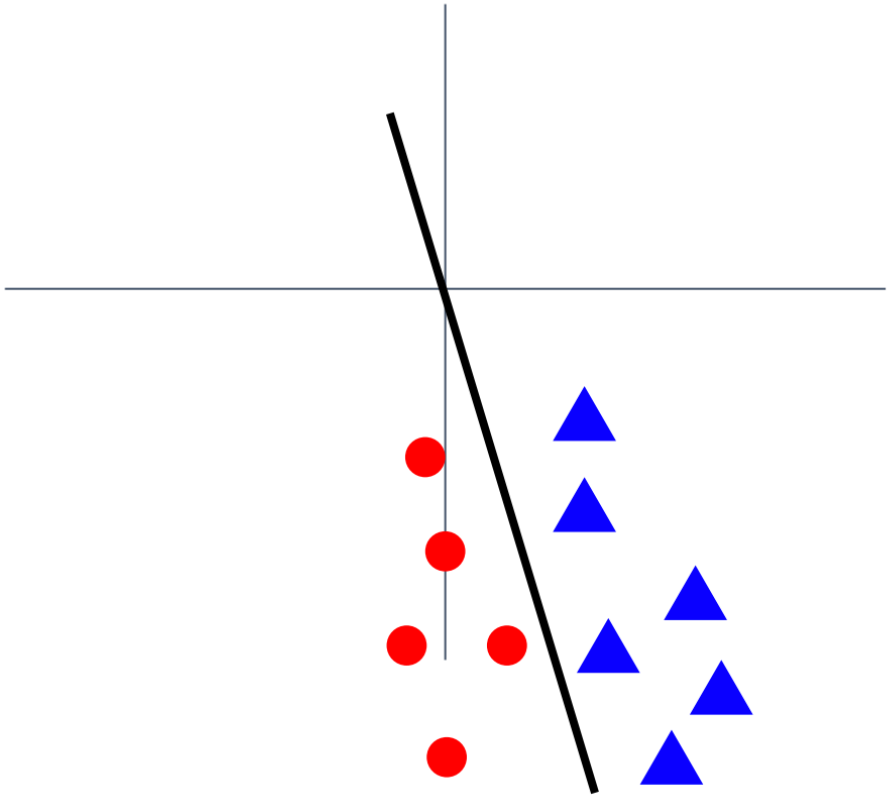
Data pre-processing



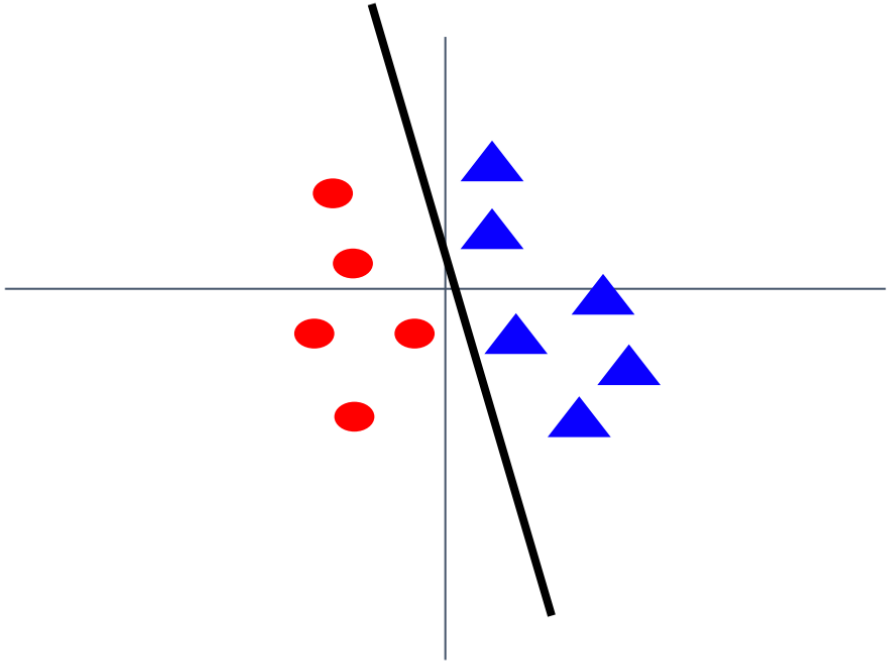
(Assume X [NxD] is data matrix,
each example in a row)

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

- Basic components in CNN
- Training a classifier
- Optimizers

Optimizers

Optimization

- A CNN as composition of functions

$$f_{\boldsymbol{w}}(\boldsymbol{x}) = f_L(\dots\ (f_2(f_1(\boldsymbol{x}; \boldsymbol{w}_1); \boldsymbol{w}_2)\ \dots; \boldsymbol{w}_L)$$

- Parameters

$$\boldsymbol{w} = (\boldsymbol{w_1}, \boldsymbol{w_2}, \dots \boldsymbol{w_L})$$

- Empirical loss function

$$L(\boldsymbol{w}) = \frac{1}{n}\sum_i l(y_i, f_{\boldsymbol{w}}(\boldsymbol{x_i}))$$

$$\boldsymbol{w}^* = \text{argmin}_{\boldsymbol{w}}\, L(\boldsymbol{w})$$

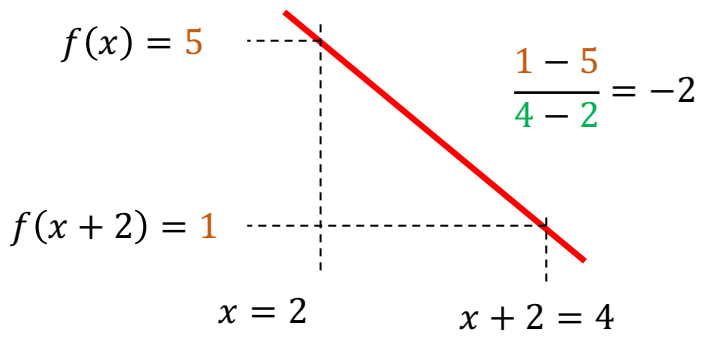
Random search is a bad idea

Follow the slope

Optimization

- In 1-dimension, the derivative of a function gives the slope:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



- In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension
- The direction of steepest descent is the **negative gradient**

Gradient descent (GD)

- Iteratively step in the direction of the negative gradient
- Gradient descent

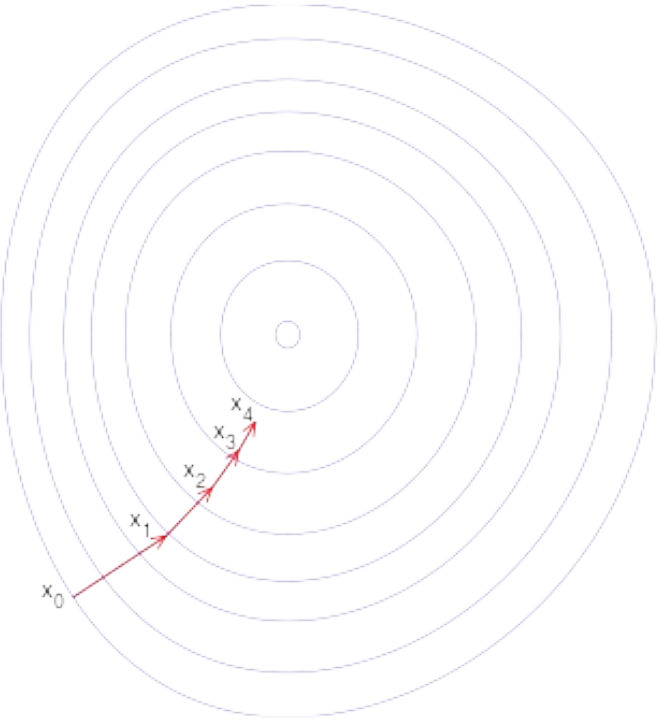
New weight

$$\boldsymbol{w^{t+1}} = \boldsymbol{w^{t}} - \eta_t \frac{\partial \boldsymbol{f}(\boldsymbol{w^{t}})}{\partial \boldsymbol{w}}$$

Old weight

Learning rate

Gradient



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
```

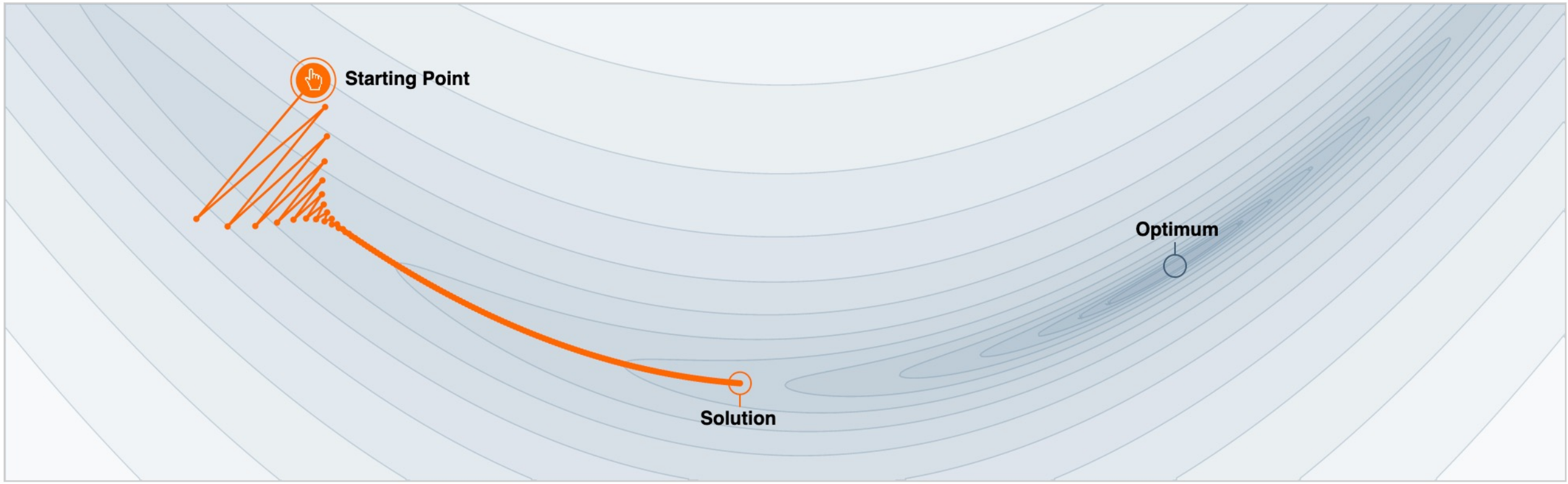
GD with Momentum

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

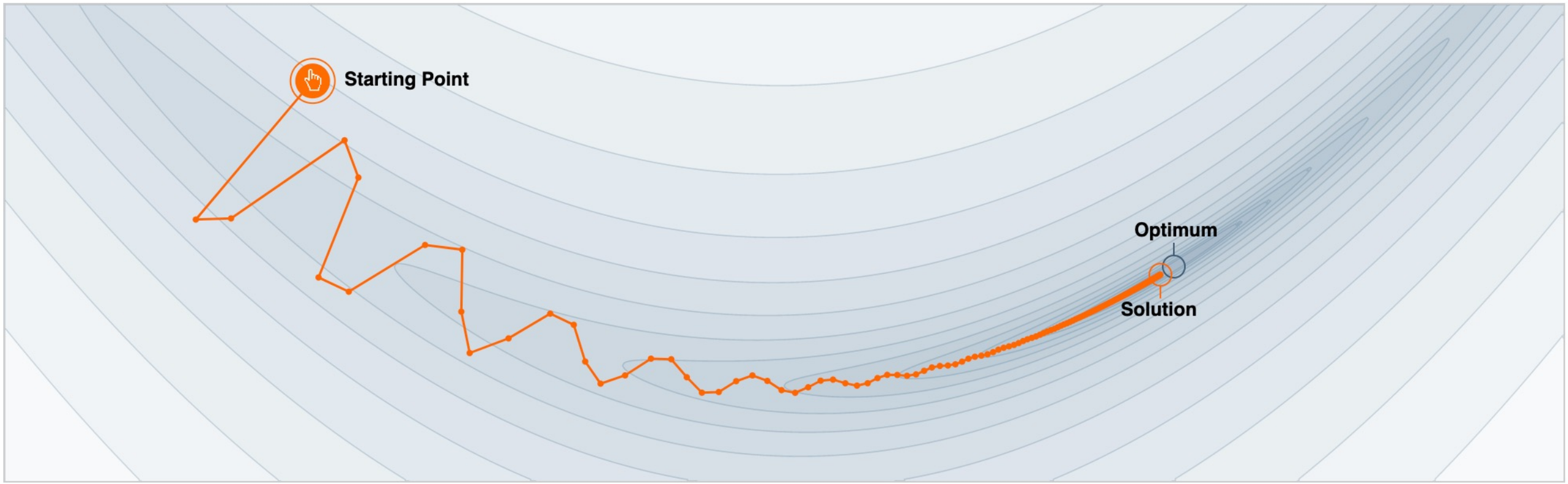
0 0.003 0.006

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:

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

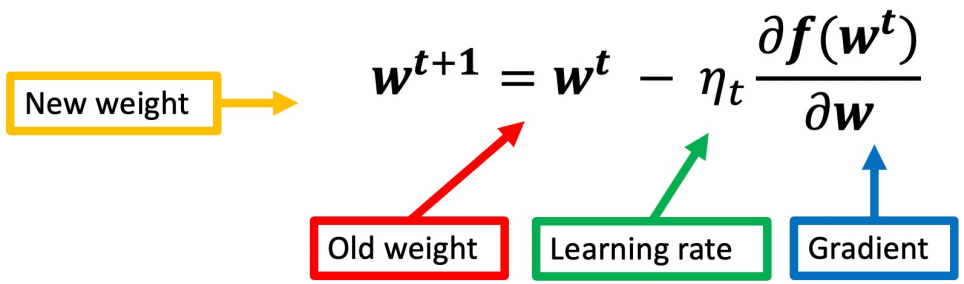
where $\boldsymbol{V}$ is known as the velocity term and has the same dimension as the weight vector $\boldsymbol{W}$.

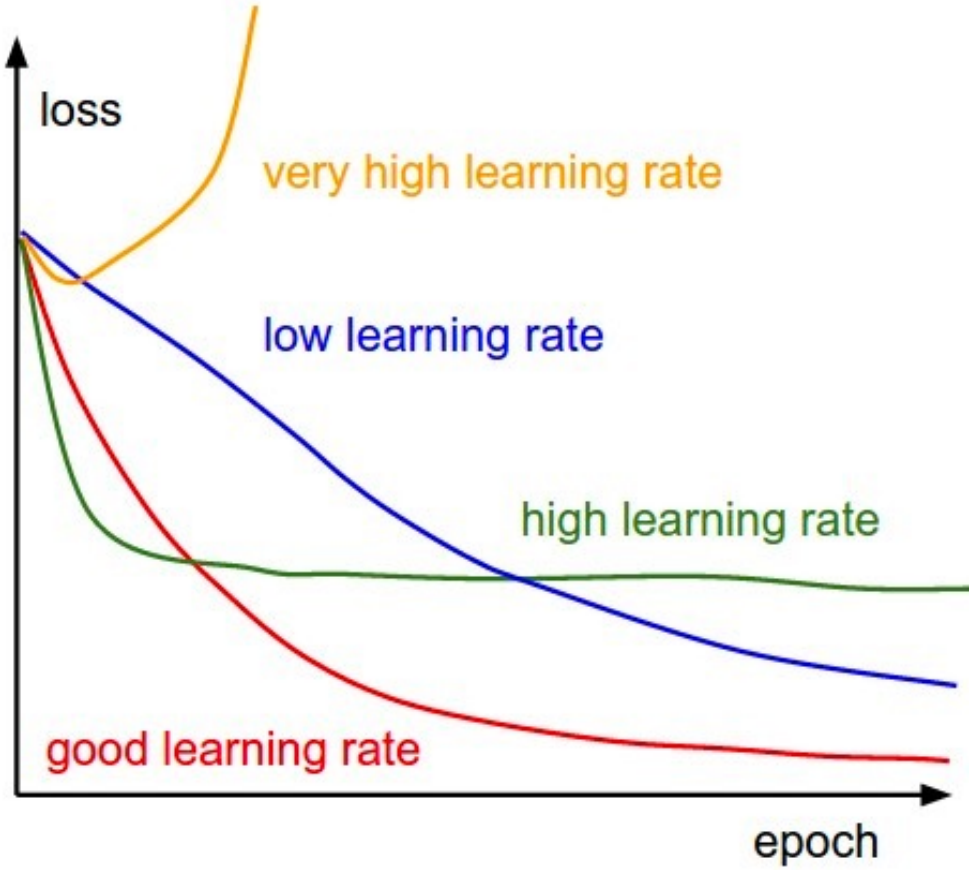
The momentum parameter $\gamma \in [0,1]$ indicates how many iterations the previous gradients are incorporated into the current update.

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.

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.
  - While Adam optimizer is the most commonly used technique, others can also be useful.

- **Algorithms with adaptive learning rates:**
  - AdaGrad torch.optim.Adagrad()
  - RMSprop torch.optim.RMSprop()
  - Adam torch.optim.Adam()

Adam: A Method for Stochastic Optimization, ICLR 2014

https://pytorch.org/docs/stable/optim.html

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 $\epsilon = 10^{-8}$

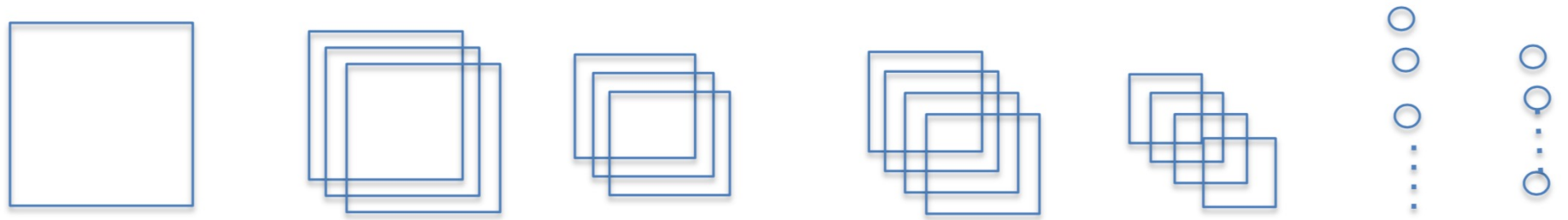
Example 3: MNIST digit recognition

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

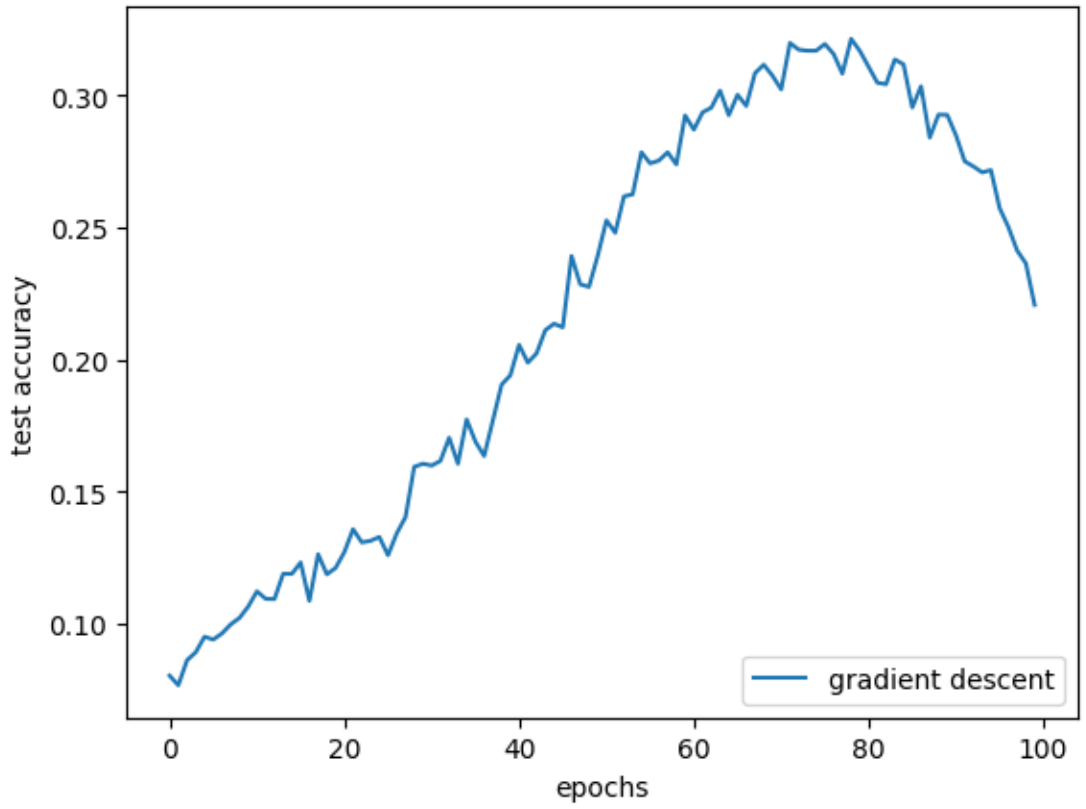
Fixed
1024

Output
Softmax
10

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

See eg6.3.ipynb

Example 3: Training Curve



Example 3: Weights learned

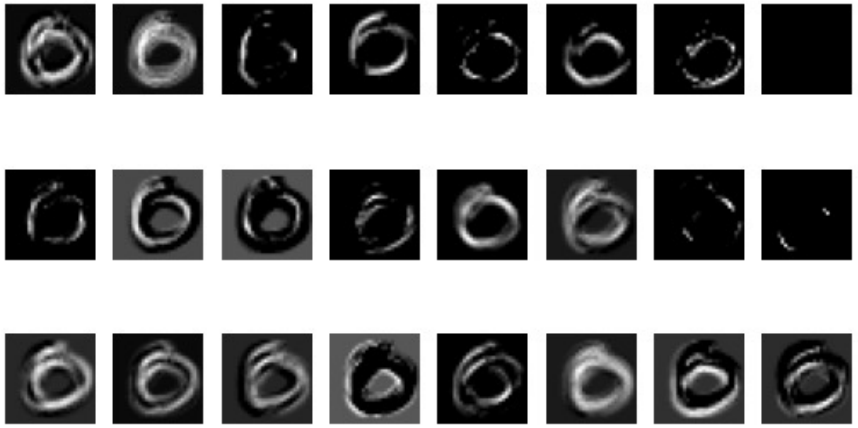
Weights learned at convolution layer 1

32x5x5

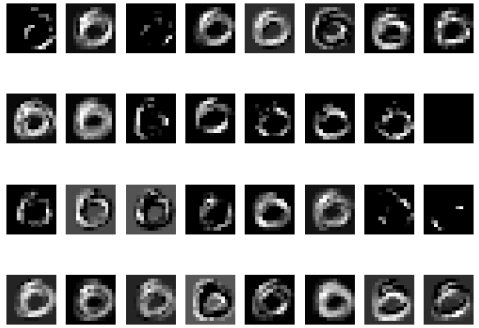
Example 3: Feature maps

Input image
28x28

Feature maps at conv1
32x28x28

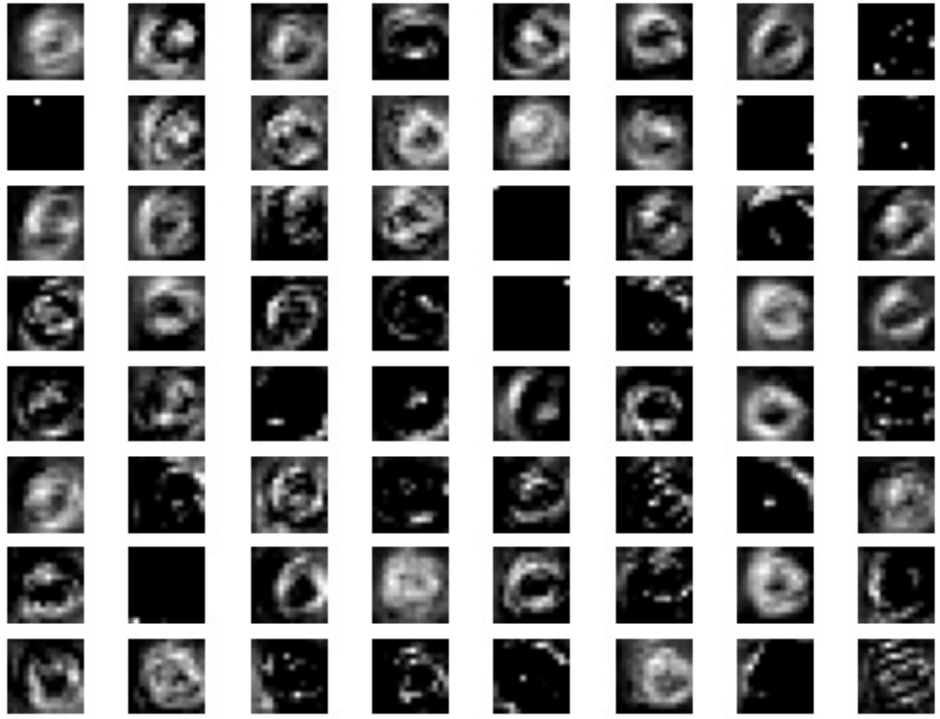


Feature maps at pool1
32x14x14



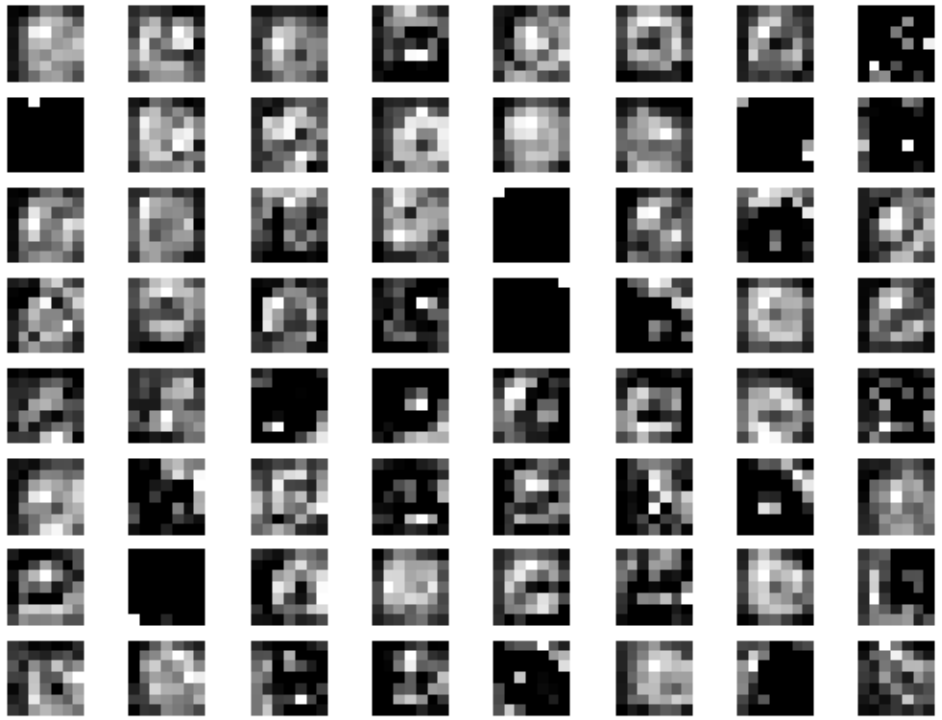
Example 3: Feature maps

Feature maps at conv2
64x14x14

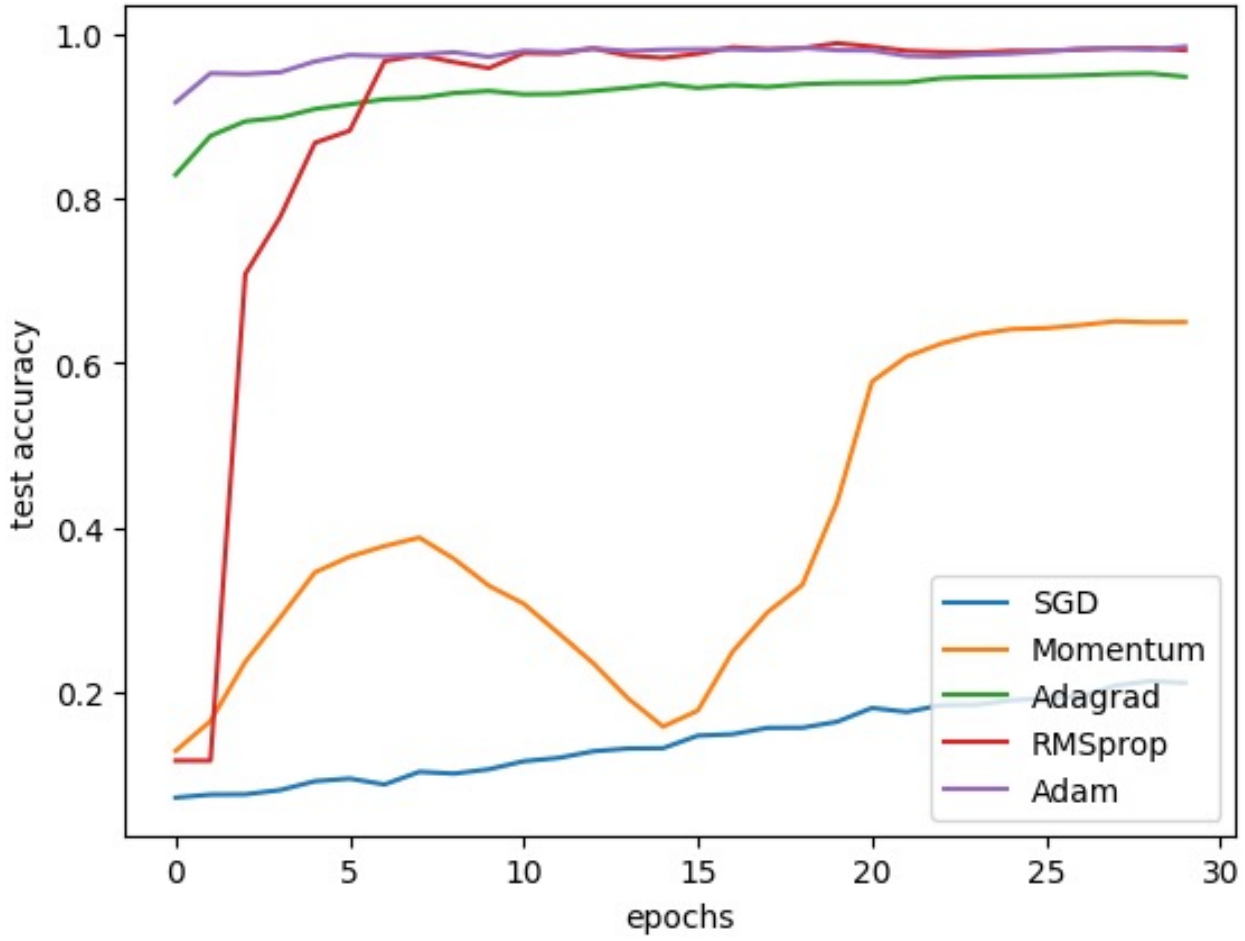


Example 3: Feature maps

Feature maps at pool2
64x7x7



Example 4: MNIST recognition with CNN with different learning algorithms



See eg6.4.ipynb

Next lecture

- CNN Architectures

  You learn some classic architectures

- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution

  You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network

- Batch normalization

  You learn an important technique to improve the training of modern neural networks

- Prevent overfitting
  - Transfer learning
  - Data augmentation

  You learn two important techniques to prevent overfitting in neural networks