19. Convolutional Networks

STA3142 Statistical Machine Learning

Kibok Lee

Assistant Professor of

Applied Statistics / Statistics and Data Science

May {21, 23}, 2024

** Slides adapted from EECS498/598 @ Univ. of Michigan by Justin Johnson*

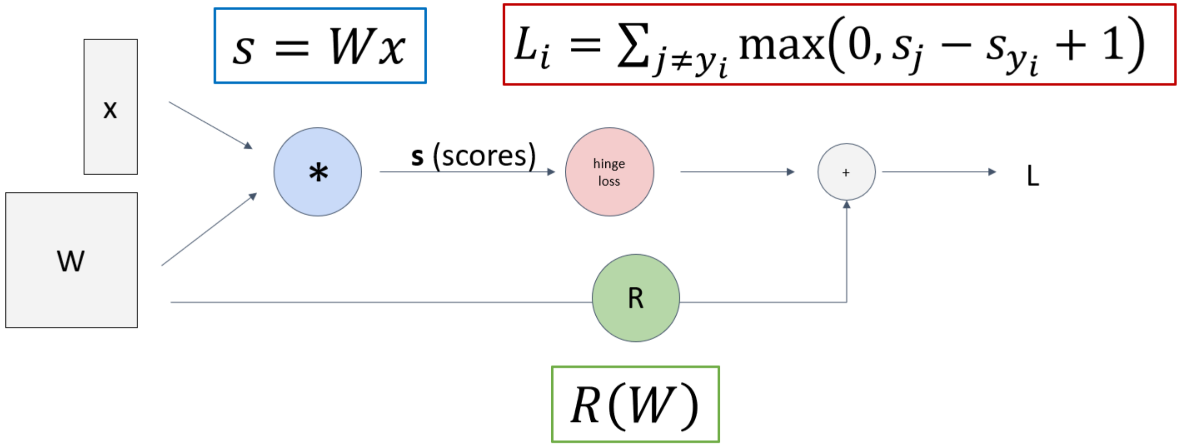
연세대학교
YONSEI UNIVERSITY

Assignment 4

- Due ~~Friday 5/17~~ **Wednesday 5/22, 11:59pm**
- Topics
  - K-Means and Gaussian Mixture Models -> bug in description fixed
  - Principal Component Analysis
- Please read the instruction carefully!
  - Submit one pdf and one zip file separately
  - Write your code only in the designated spaces
  - Do not import additional libraries
  - …
- If you feel difficult, consider to take option 2.

Recap: Backpropagation

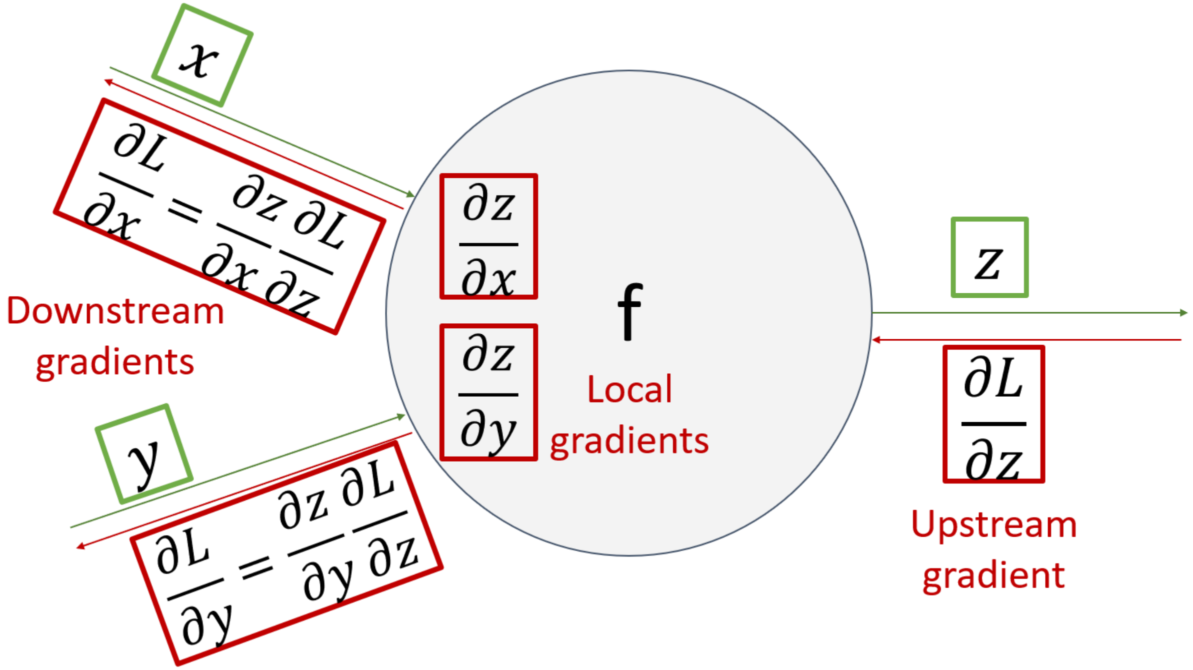
Represent complex expressions as **computational graphs**



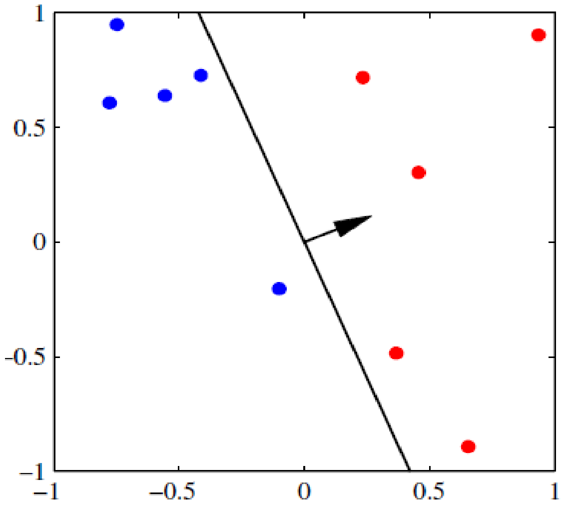
Forward pass computes outputs

Backward pass computes gradients

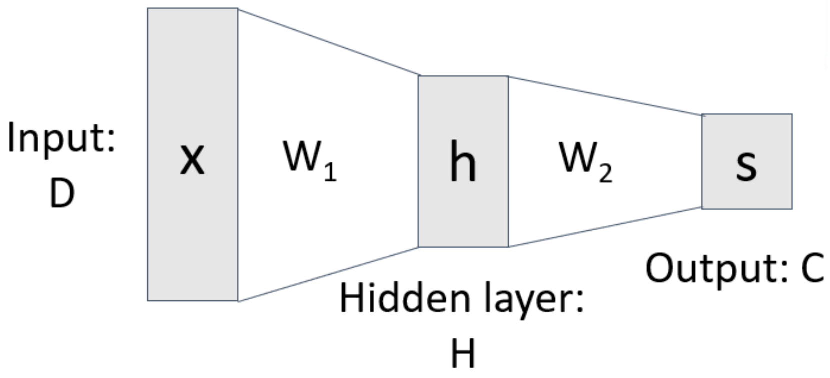
During the backward pass, each node in the graph receives **upstream gradients** and multiplies them by **local gradients** to compute **downstream gradients**



$$f(x) = Wx + b$$



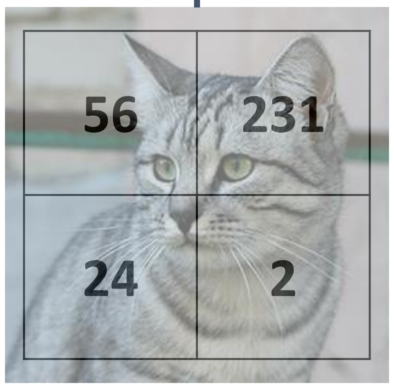
$$f(x) = W_2\, g(W_1 x + b_1) + b_2$$



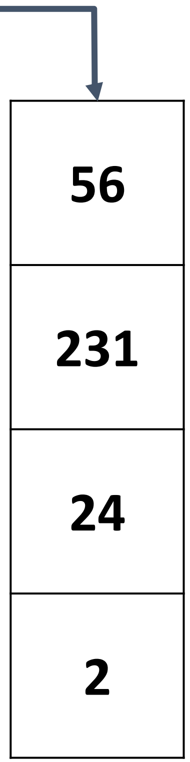
Problem: So far our classifiers don't respect the spatial structure of images!

Solution: Define new computational nodes that operate on images!

Stretch pixels into column



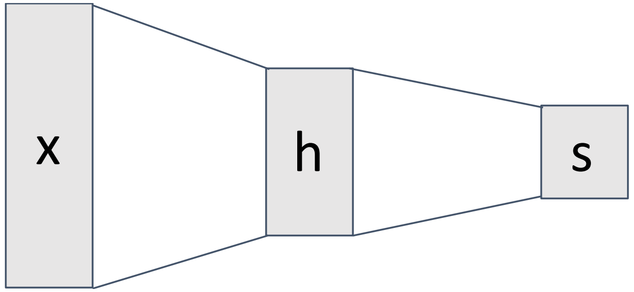
Input image
(2, 2)



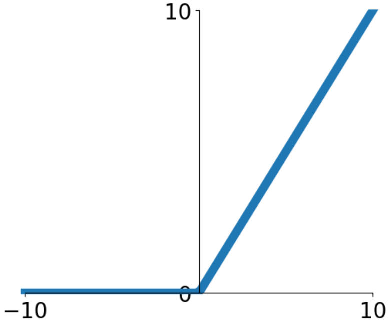
(4,)

Components of a Fully-Connected Network

Fully-Connected Layers

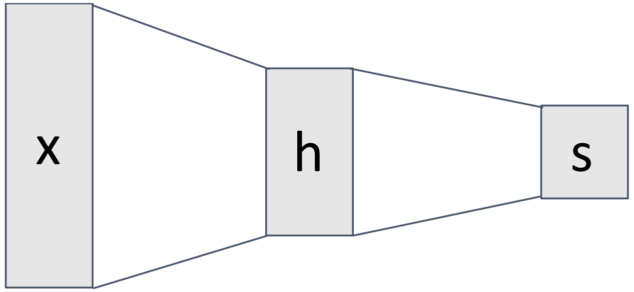


Activation Function

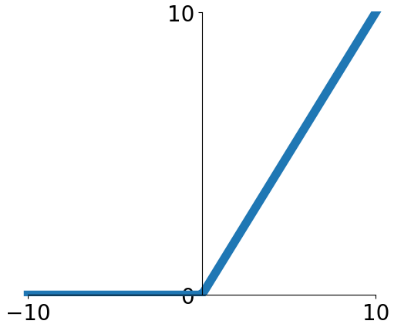


Components of a Convolutional Network

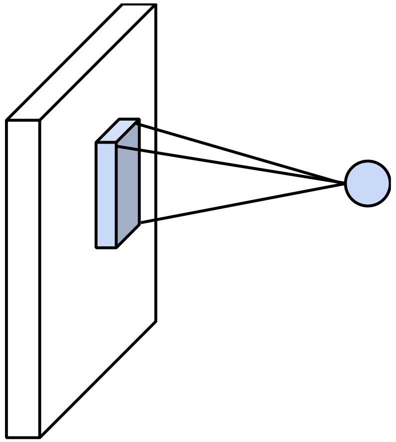
Fully-Connected Layers



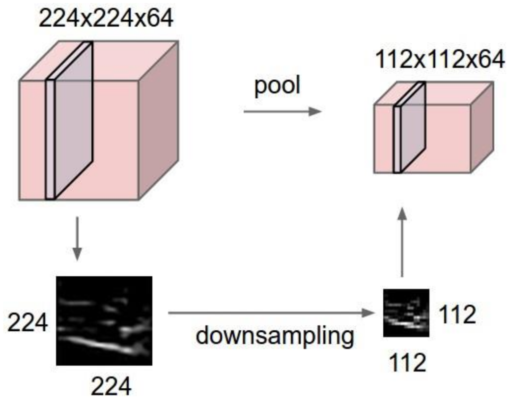
Activation Function



Convolution Layers



Pooling Layers

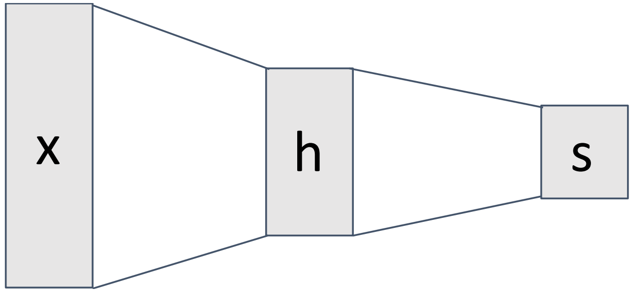


Normalization

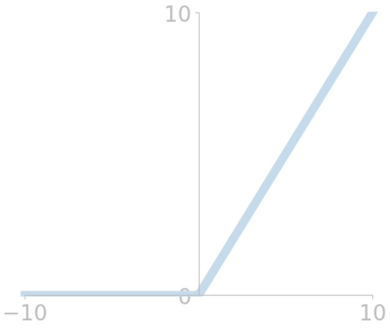
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Components of a Convolutional Network

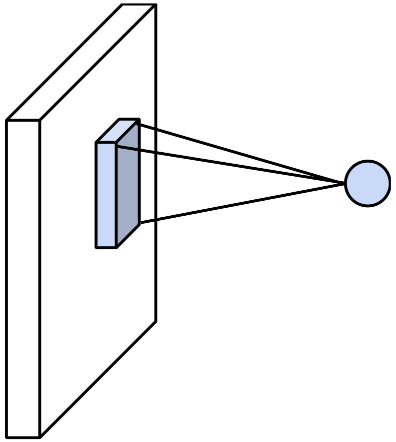
Fully-Connected Layers



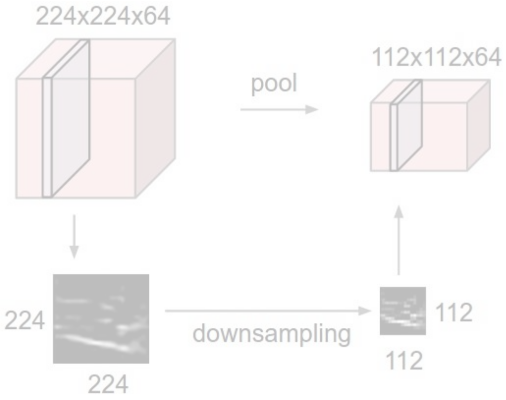
Activation Function



Convolution Layers



Pooling Layers

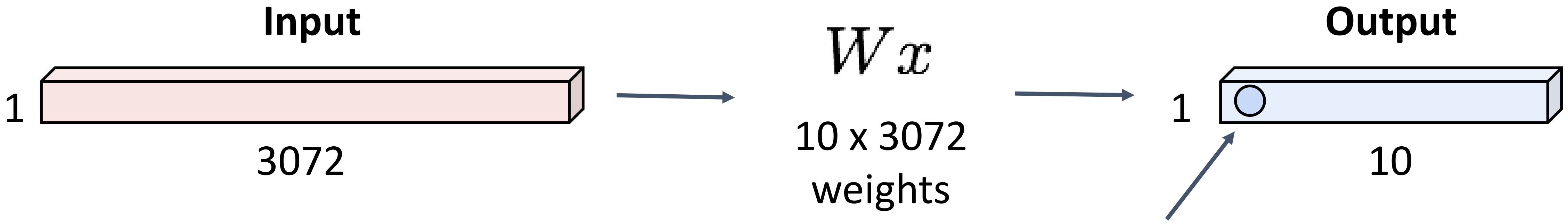


Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Fully-Connected Layer

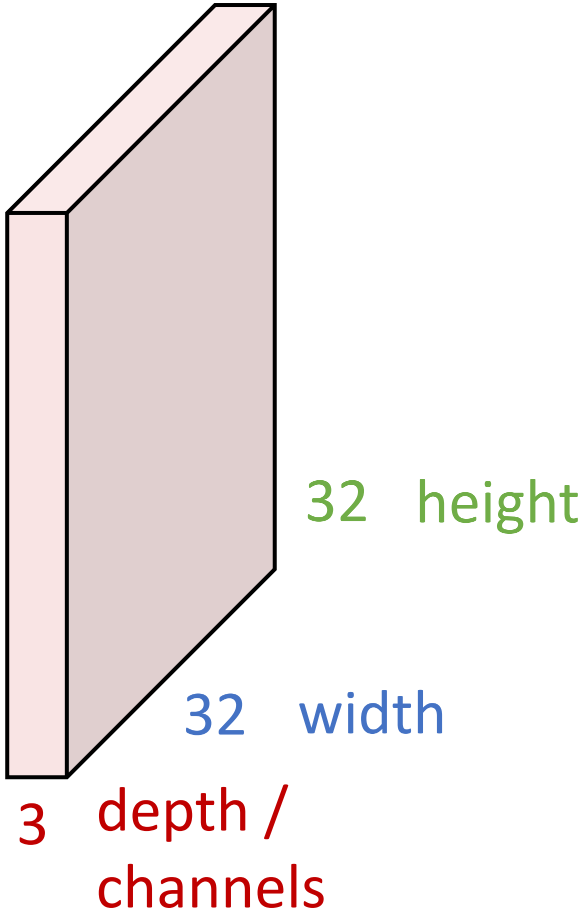
32x32x3 image -> stretch to 3072 x 1



1 number:
the result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)

Convolution Layer

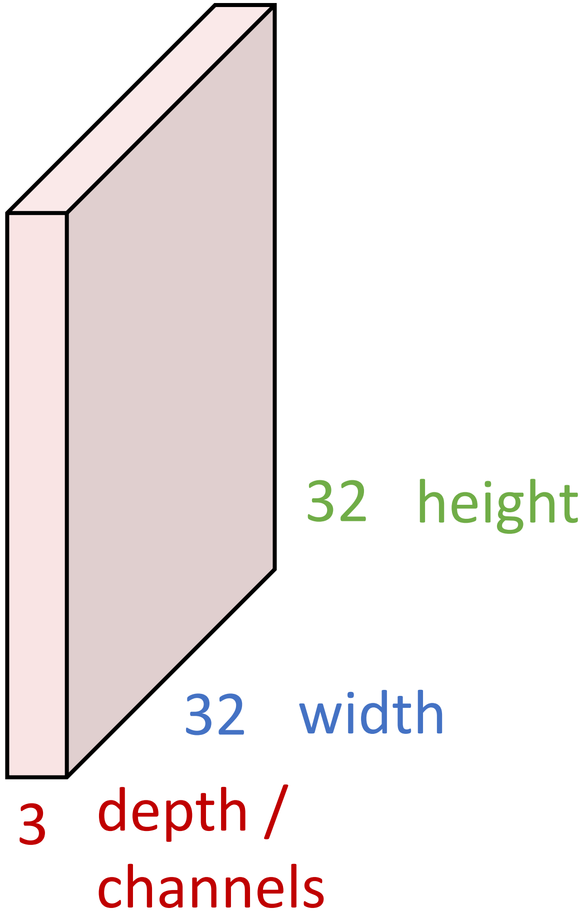
3x32x32 image: preserve spatial structure



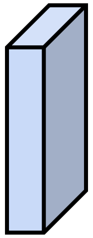
Convolution Layer

Filters always extend the full depth of the input volume

3x32x32 image

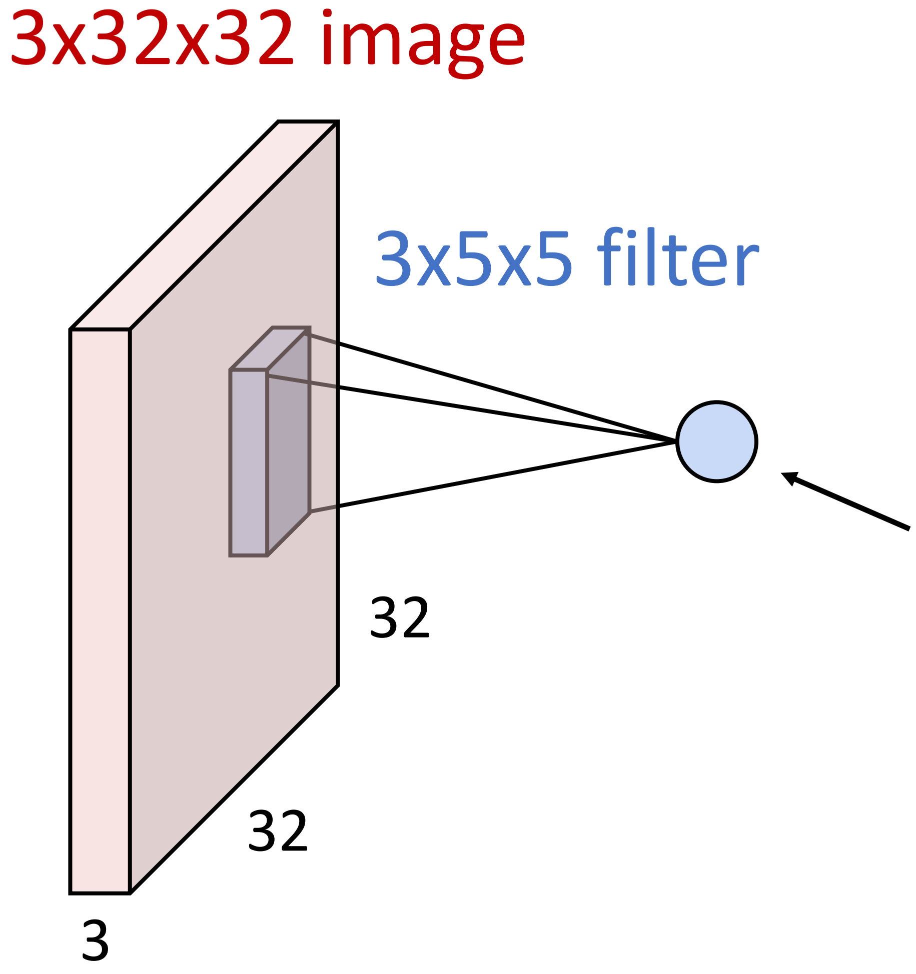


3x5x5 filter



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

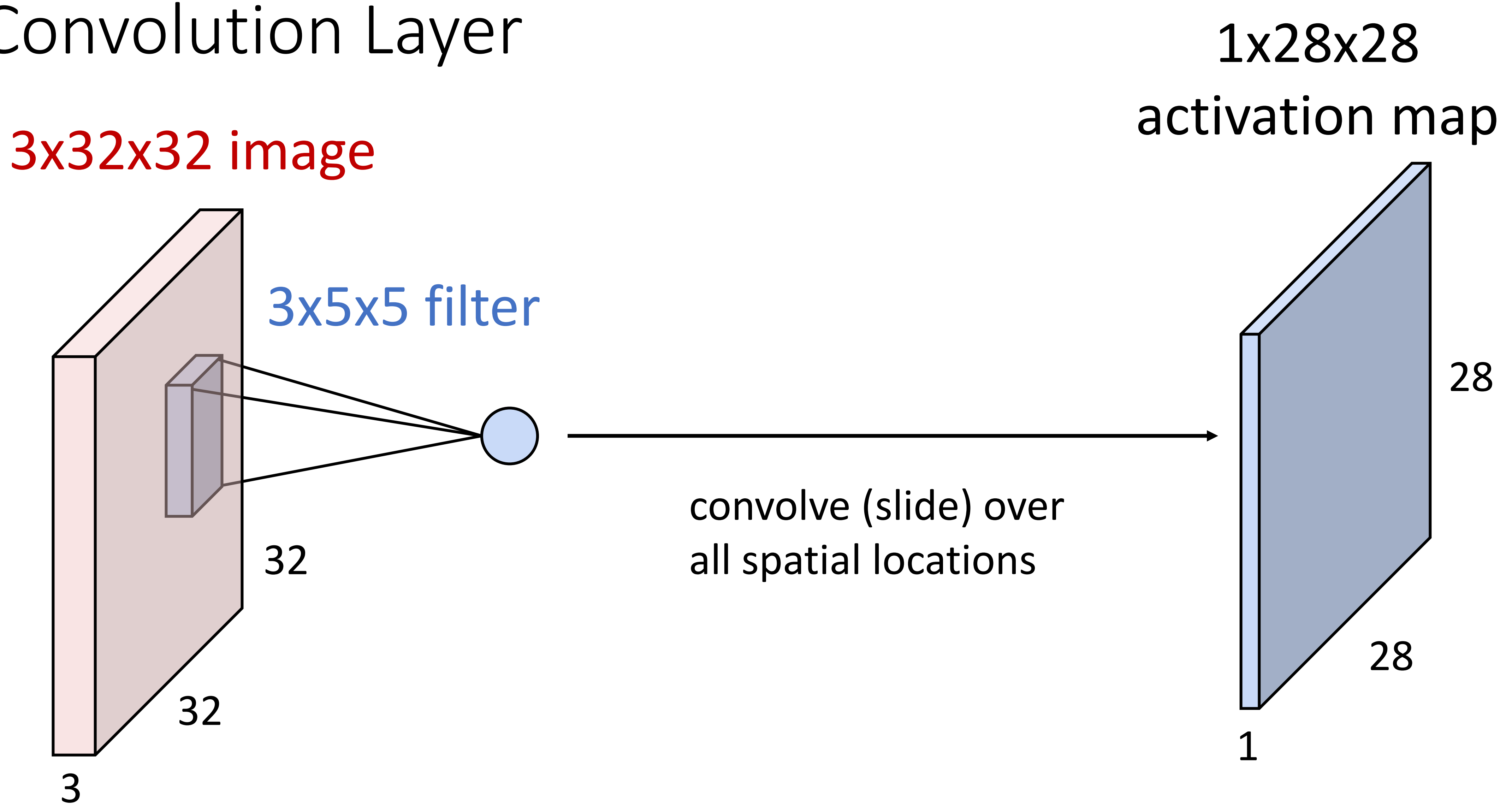
Convolution Layer



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)

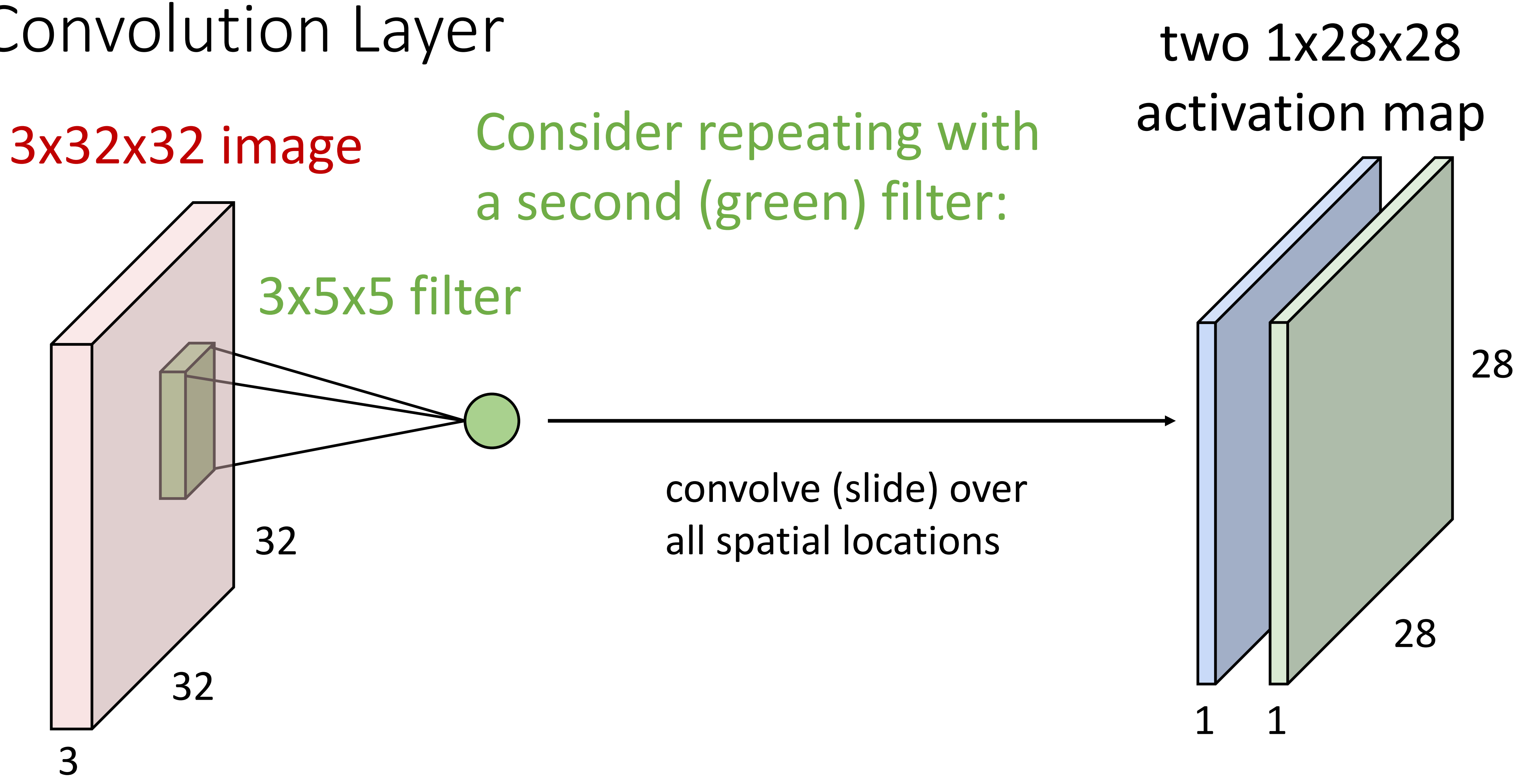
$$w^T x + b$$

Convolution Layer

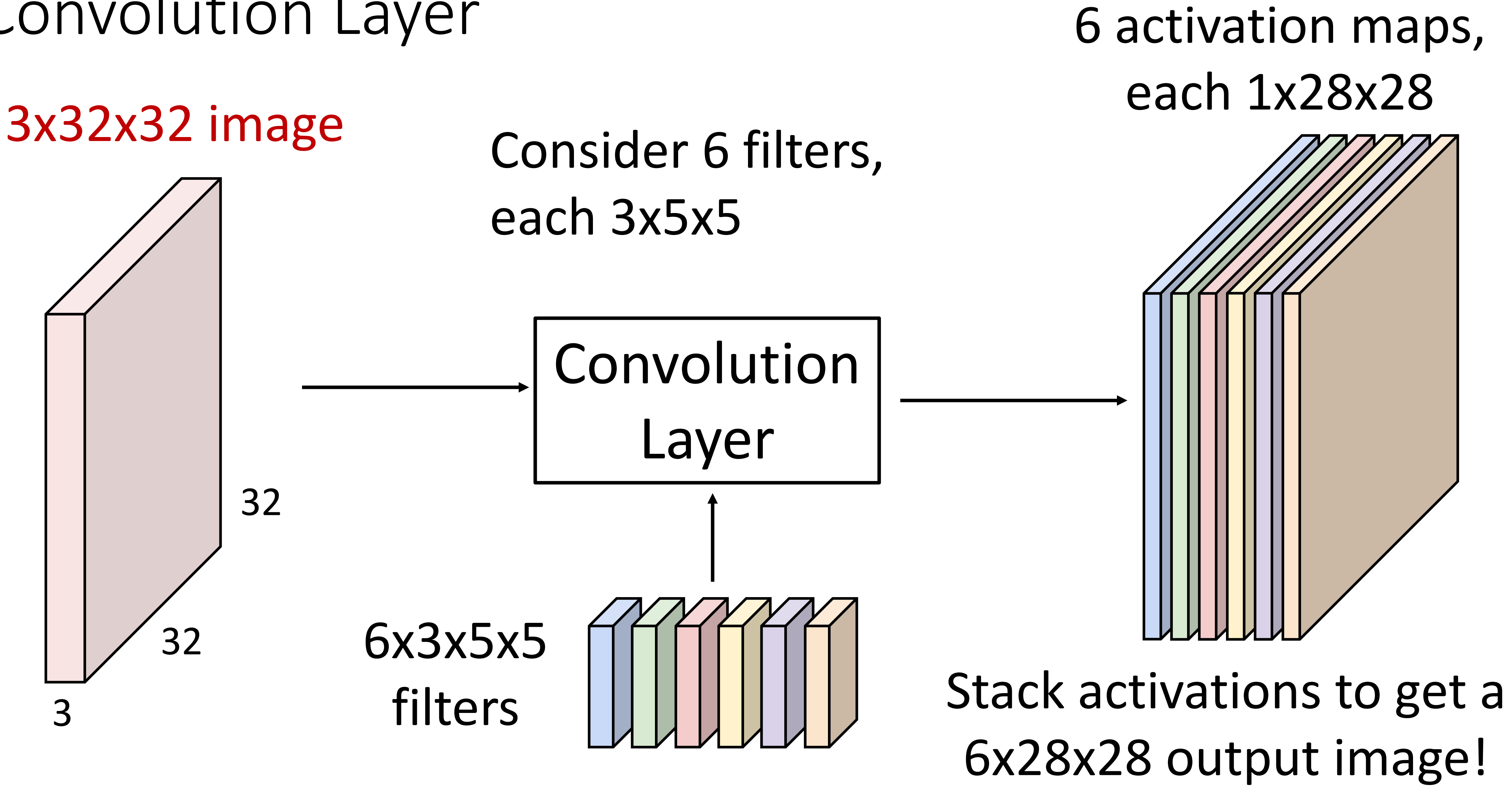


Convolution Layer

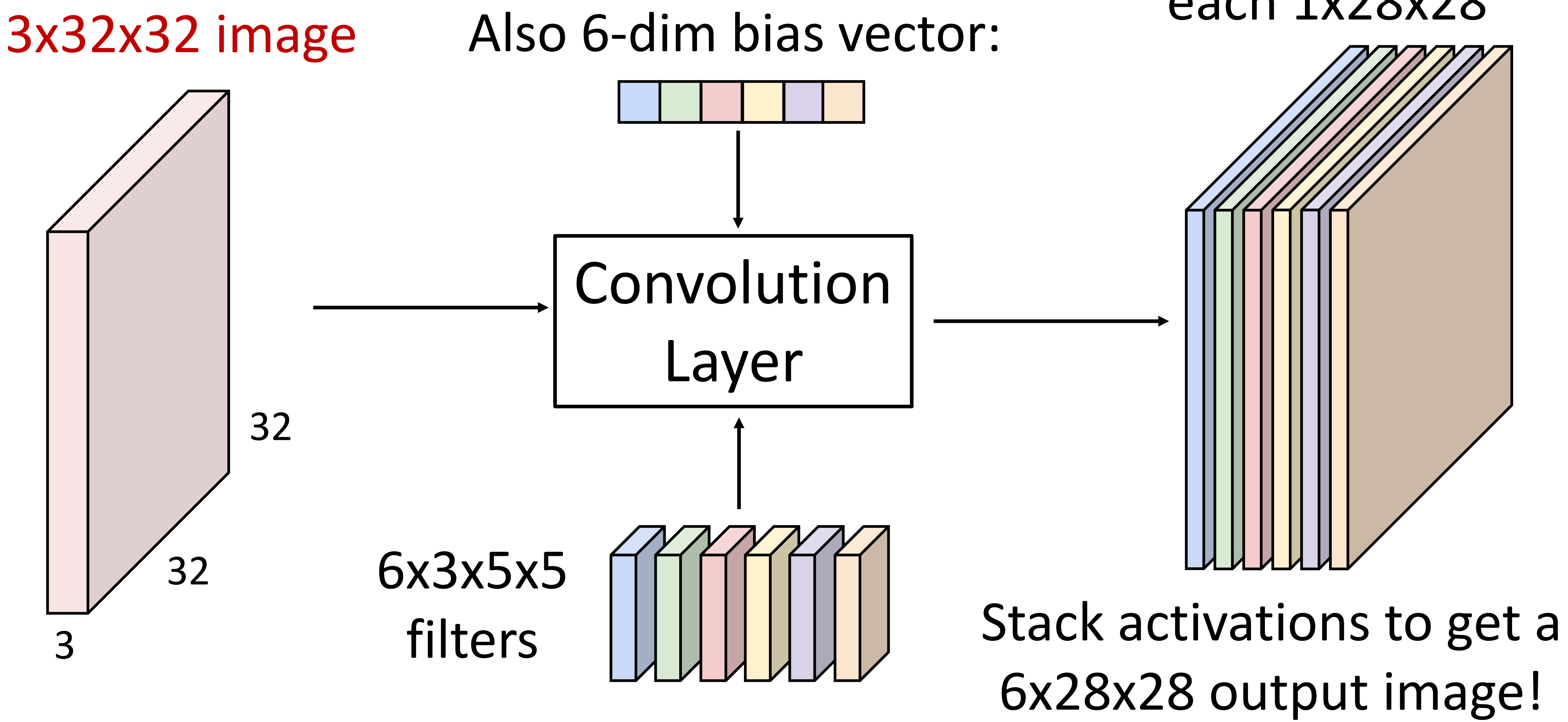
Consider repeating with a second (green) filter:



Convolution Layer



Convolution Layer

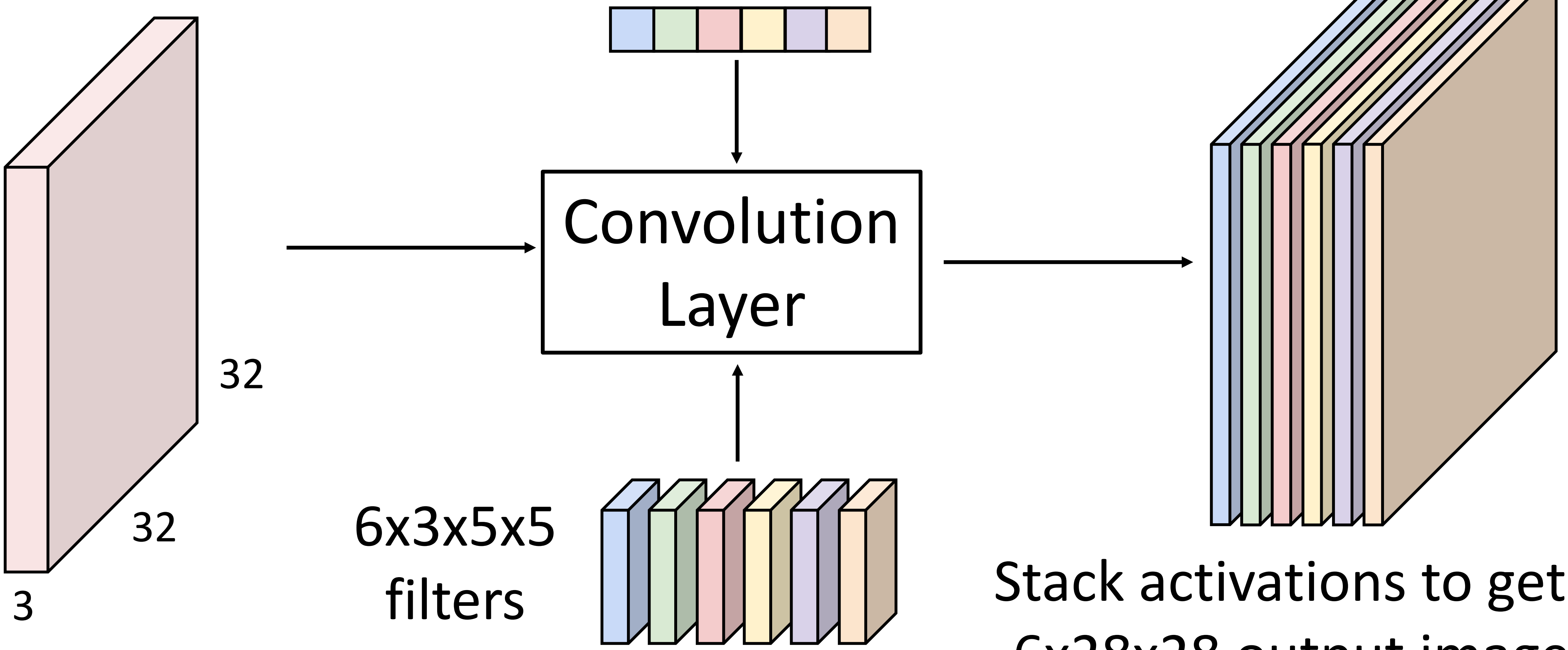


Convolution Layer

28x28 grid, at each point a 6-dim vector

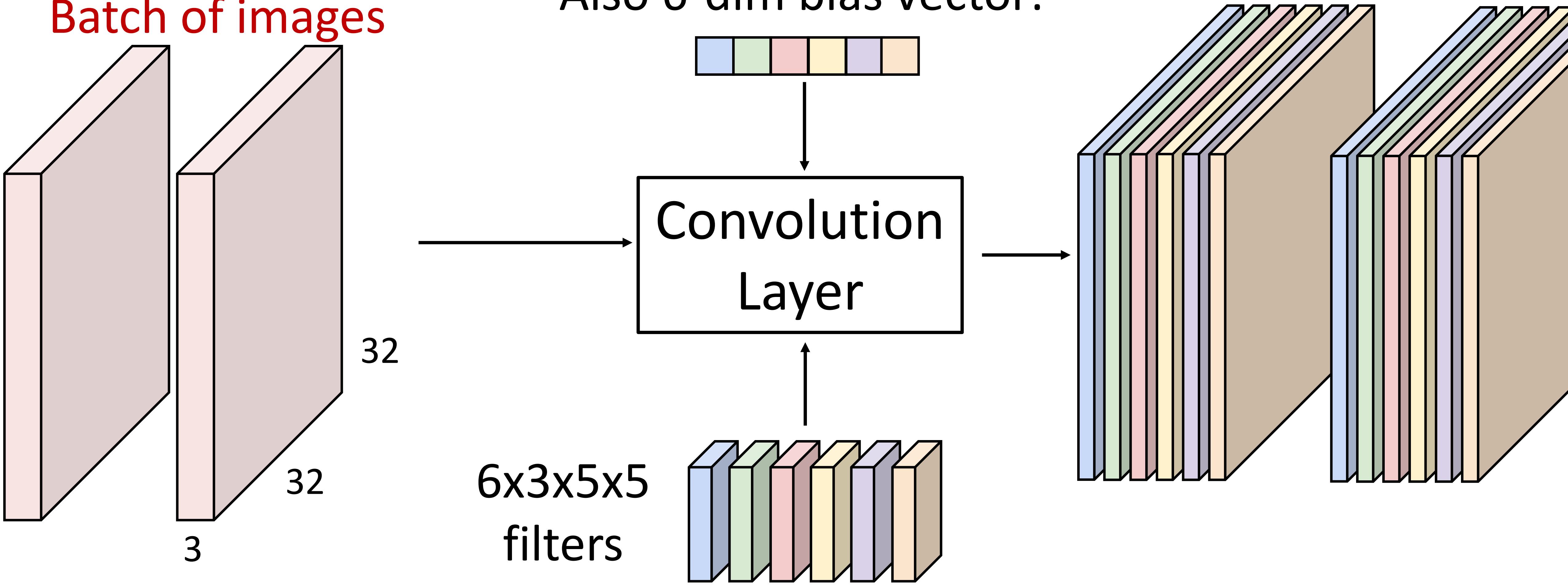
3x32x32 image

Also 6-dim bias vector:

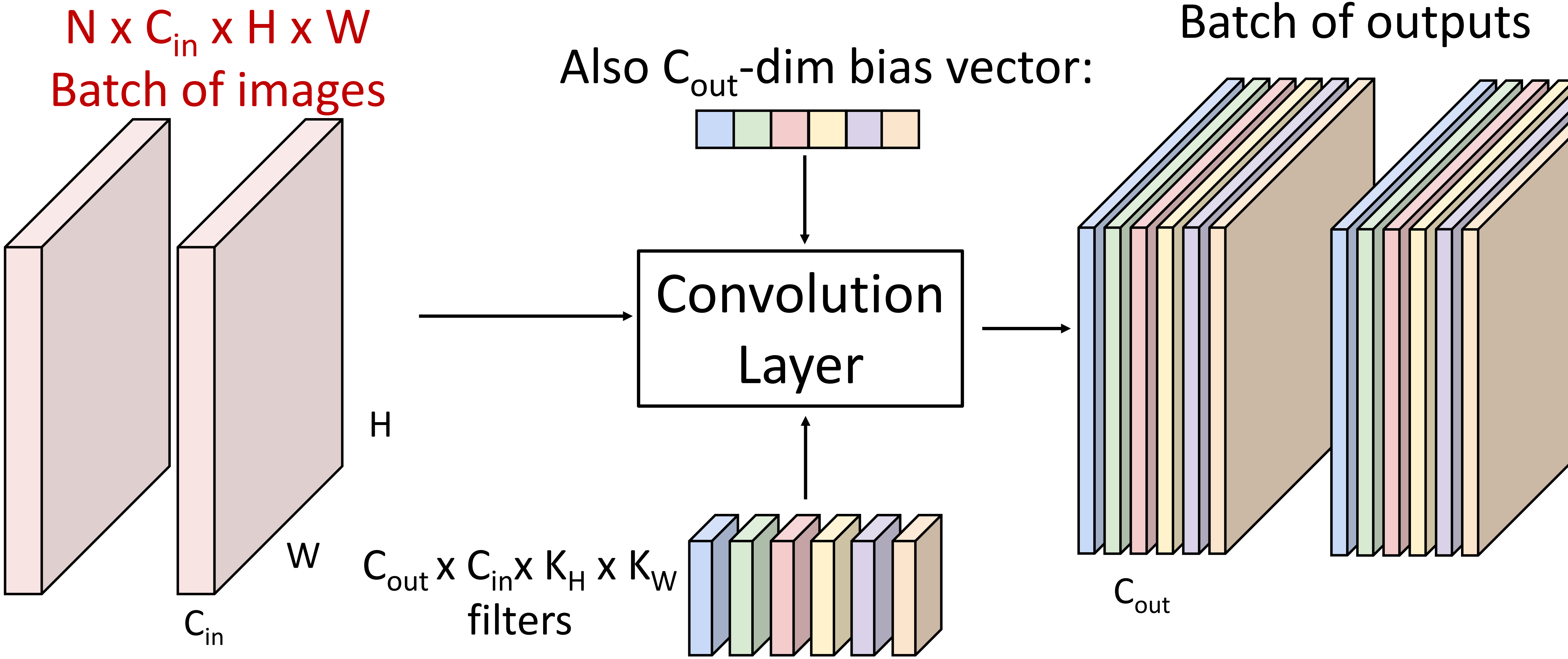


Stack activations to get a 6x28x28 output image!

Convolution Layer



Convolution Layer

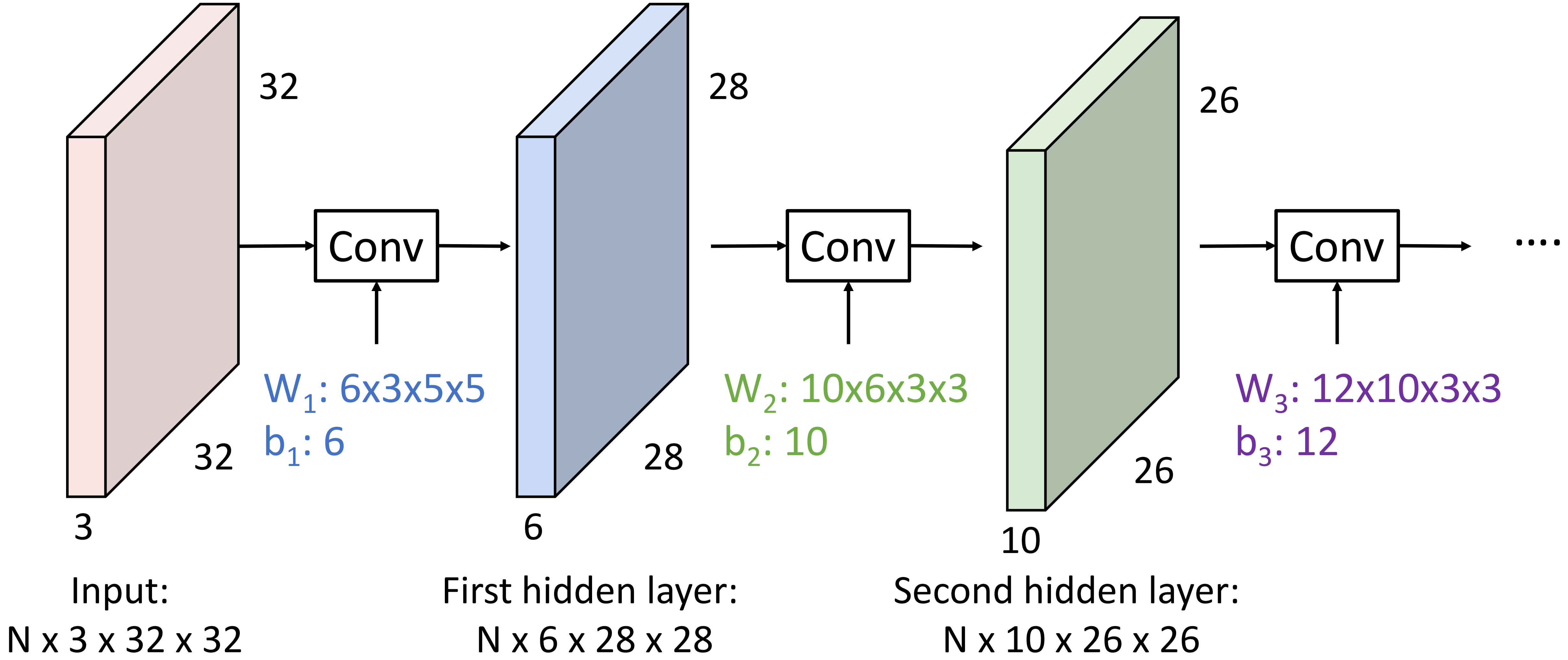


Stacking Convolutions

Q: What happens if we stack two convolution layers?

(Recall $y=W_2W_1x$ is a linear classifier)

A: We get another convolution!



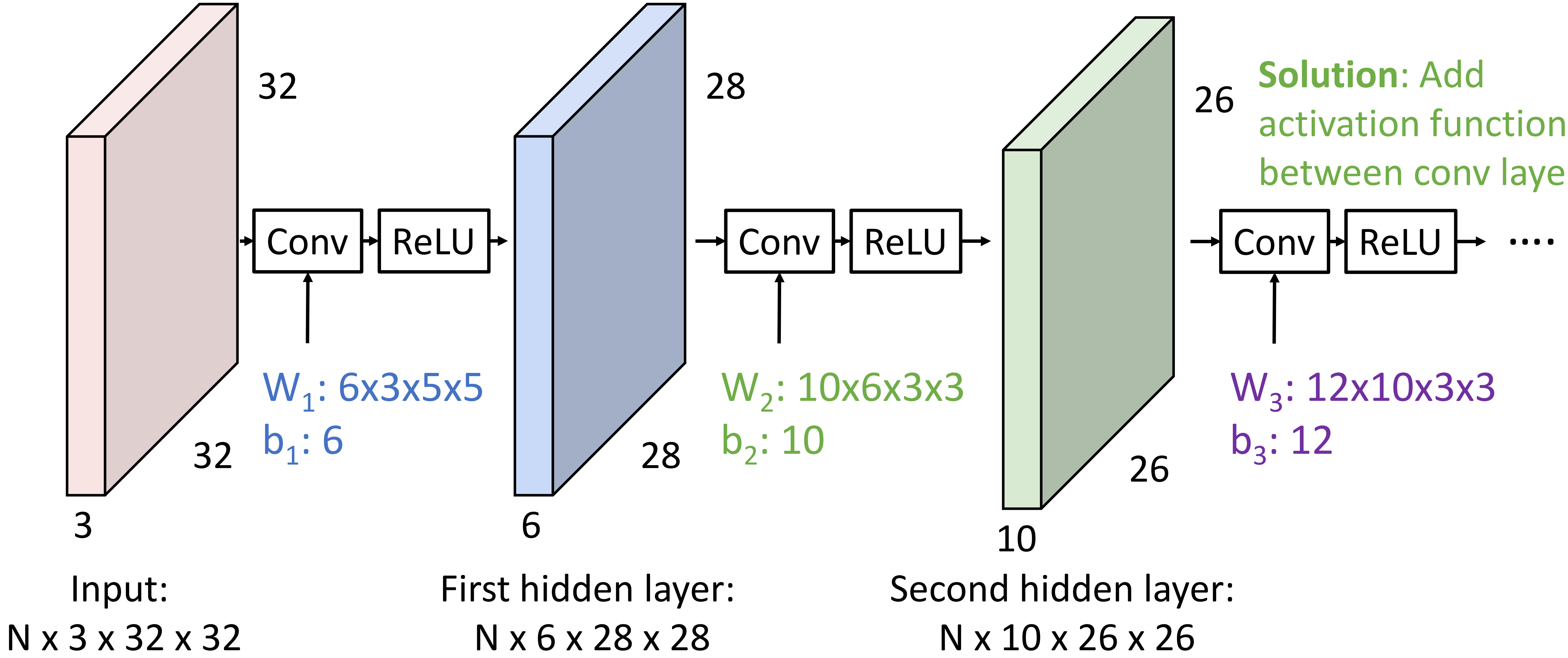
Stacking Convolutions

Q: What happens if we stack two convolution layers?

(Recall $y=W_2W_1x$ is a linear classifier)

A: We get another convolution!

Solution: Add activation function between conv layers



Discrete Convolution

- The convolution of an image x with kernel k is computed as follows:

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r-p,r-q}$$

- Example:

$\tilde{k} = \{k \text{ with rows and columns flipped}\}$

$$\underbrace{\begin{bmatrix} 1 & .5 \\ .25 & 0 \end{bmatrix}}_{\tilde{k}}$$

$$\underbrace{\begin{bmatrix} 0 & 80 & 40 \\ 20 & 40 & 0 \\ 0 & 0 & 40 \end{bmatrix}}_{x} * \underbrace{\begin{bmatrix} 0 & .25 \\ .5 & 1 \end{bmatrix}}_{k} =$$

Slide credit: Russ Salakhutdinov

Discrete Convolution

- The convolution of an image x with kernel k is computed as follows:

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r-p,r-q}$$

- Example:

1 x 0 + 0.5 x 80 + 0.25 x 20 + 0 x 40 = 45

$$\underbrace{\begin{bmatrix} 1 & .5 \\ .25 & 0 \end{bmatrix}}_{\tilde{k}} \qquad \underbrace{\begin{bmatrix} 0 & 80 & 40 \\ 20 & 40 & 0 \\ 0 & 0 & 40 \end{bmatrix}}_{x} * \underbrace{\begin{bmatrix} 0 & .25 \\ .5 & 1 \end{bmatrix}}_{k} = \begin{bmatrix} 45 & \\ & \end{bmatrix}$$

Slide credit: Russ Salakhutdinov

Discrete Convolution

- The convolution of an image x with kernel k is computed as follows:

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r-p,r-q}$$

- Example:

1 x 80 + 0.5 x 40 + 0.25 x 40 + 0 x 0 = 110

$$\underset{\tilde{k}}{\begin{bmatrix} 1 & .5 \\ .25 & 0 \end{bmatrix}}$$

$$\underset{x}{\begin{bmatrix} 0 & 80 & 40 \\ 20 & 40 & 0 \\ 0 & 0 & 40 \end{bmatrix}} * \underset{k}{\begin{bmatrix} 0 & .25 \\ .5 & 1 \end{bmatrix}} = \begin{bmatrix} 45 & 110 \\ & \end{bmatrix}$$

Slide credit: Russ Salakhutdinov

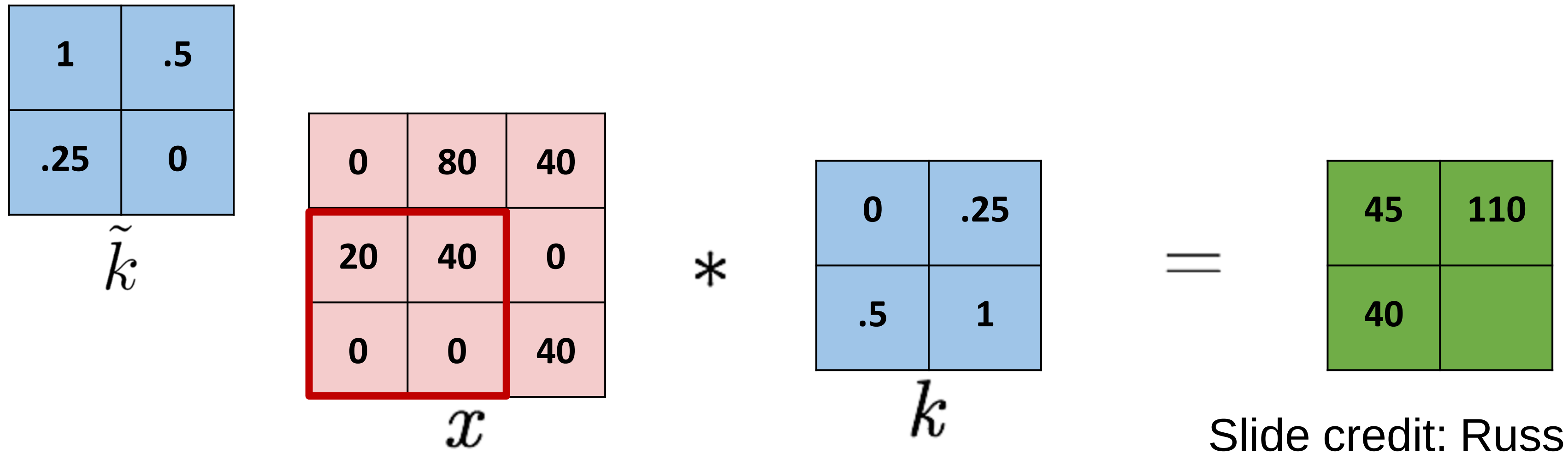
Discrete Convolution

- The convolution of an image x with kernel k is computed as follows:

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r-p,r-q}$$

- Example:

1 x 20 + 0.5 x 40 + 0.25 x 0 + 0 x 0 = 40



Slide credit: Russ Salakhutdinov

Discrete Convolution

- The convolution of an image x with kernel k is computed as follows:

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r-p,r-q}$$

- Example:

1 x 40 + 0.5 x 0 + 0.25 x 0 + 0 x 40 = 40

$$\underset{\tilde{k}}{\begin{bmatrix} 1 & .5 \\ .25 & 0 \end{bmatrix}} \qquad \underset{x}{\begin{bmatrix} 0 & 80 & 40 \\ 20 & 40 & 0 \\ 0 & 0 & 40 \end{bmatrix}} * \underset{k}{\begin{bmatrix} 0 & .25 \\ .5 & 1 \end{bmatrix}} = \begin{bmatrix} 45 & 110 \\ 40 & 40 \end{bmatrix}$$

Slide credit: Russ Salakhutdinov

(Discrete) Convolution in CNNs

- The convolution of an image x with kernel k is computed as follows:

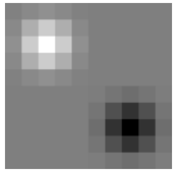
$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r-p,r-q}$$

- Convolution comes from the context of signal processing, where we flip kernels because it has several nice mathematical properties.
- In CNNs, kernels/filters are **learnable**, so flipping does not matter; assume that we learn flipped kernels/filters.

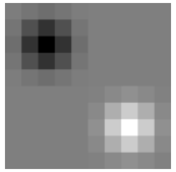
(Discrete) Convolution vs. Cross-Correlation

- What we have seen is in fact cross-correlation, NOT convolution.

Cross-Correlation
(Slide filter over image)



Convolution
(Flip filter, then slide it)



- Convolution comes from the context of signal processing, where we flip filters because it has several nice mathematical properties.
- In CNNs, filters are **learnable**, so flipping does not matter; assume that we learn flipped filters.

(Discrete) Convolution vs. Cross-Correlation

- The convolution of an image x with kernel k is computed as follows:

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r-p,r-q}$$

- C.f. the cross-correlation of an image x with kernel k is computed as:

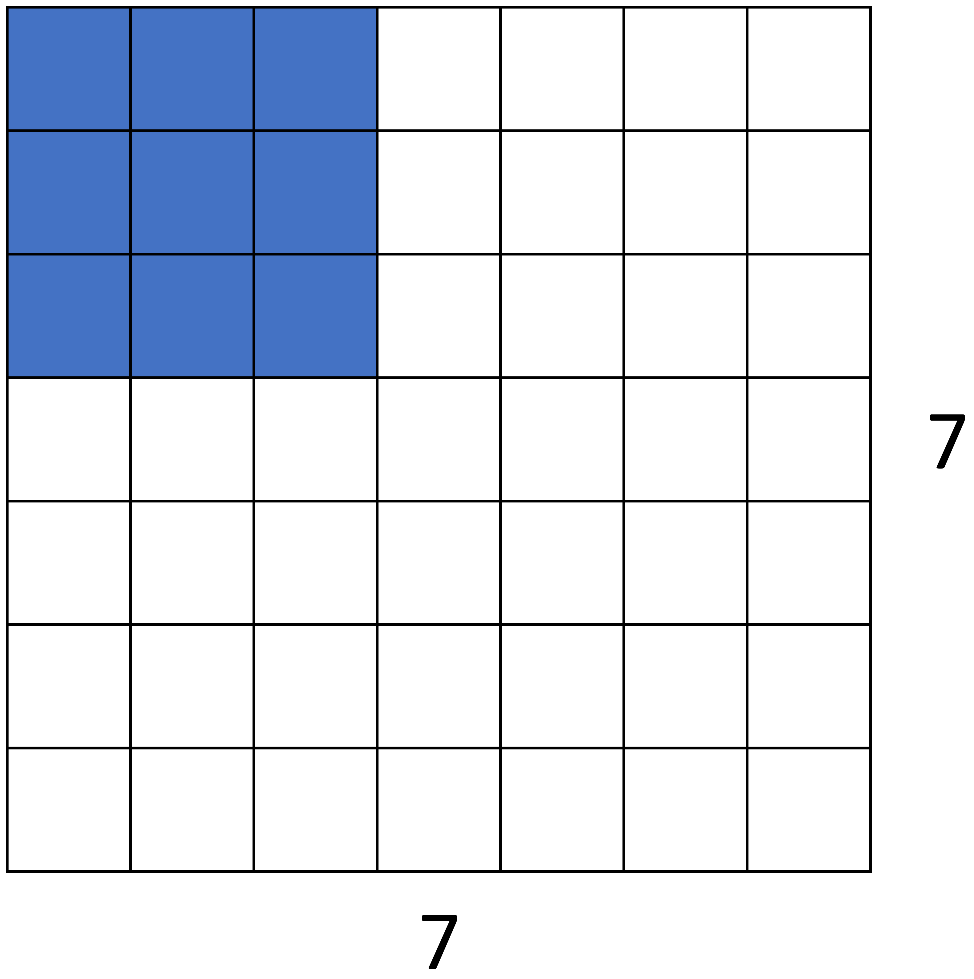
$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r+p,r+q}$$

- Example:

$$\underset{x}{\begin{bmatrix} 0 & 80 & 40 \\ 20 & 40 & 0 \\ 0 & 0 & 40 \end{bmatrix}} * \underset{k}{\begin{bmatrix} 0 & .25 \\ .5 & 1 \end{bmatrix}} = \begin{bmatrix} 70 & 30 \\ 10 & 40 \end{bmatrix}$$

Slide credit: Russ Salakhutdinov

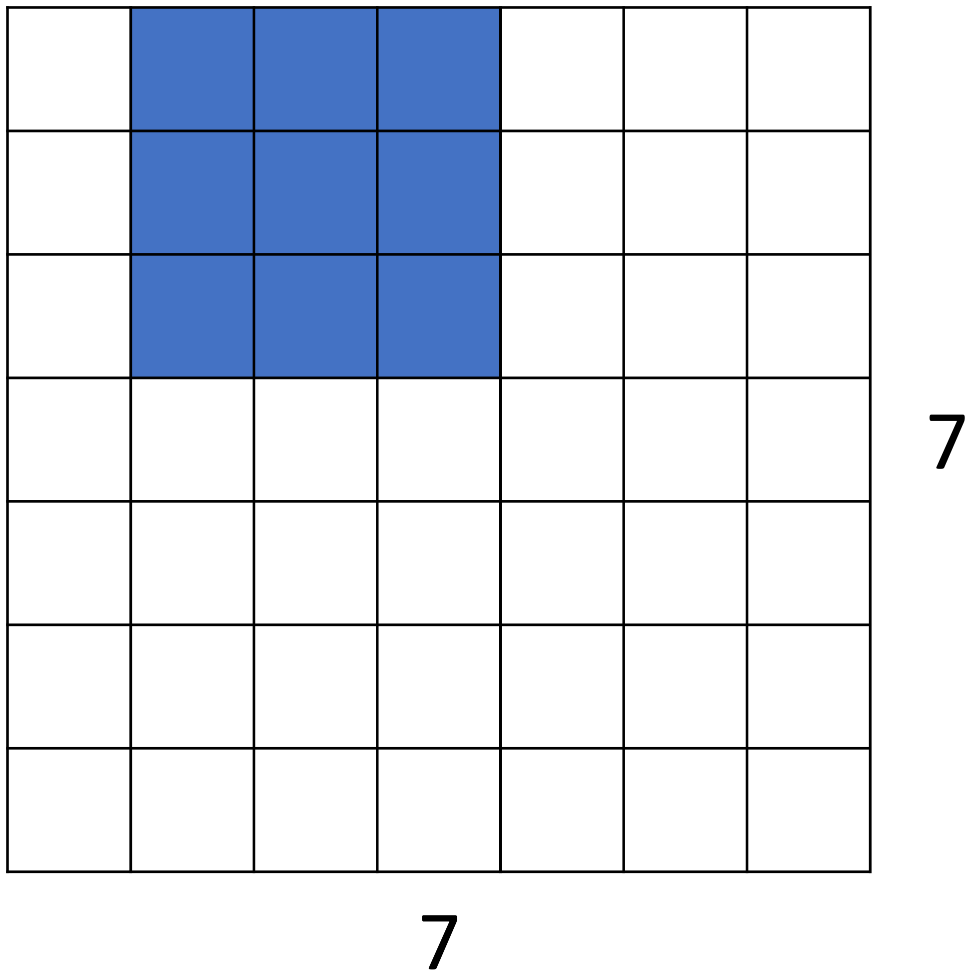
A closer look at spatial dimensions



Input: 7x7

Filter: 3x3

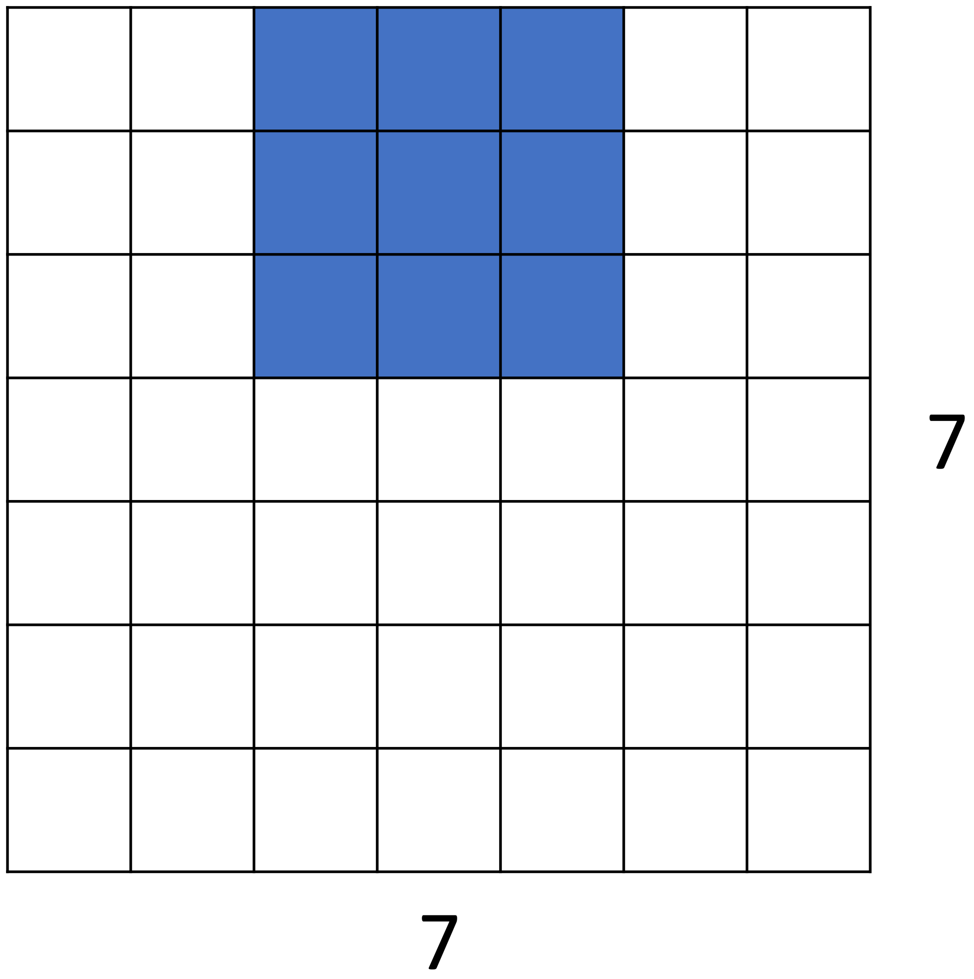
A closer look at spatial dimensions



Input: 7x7

Filter: 3x3

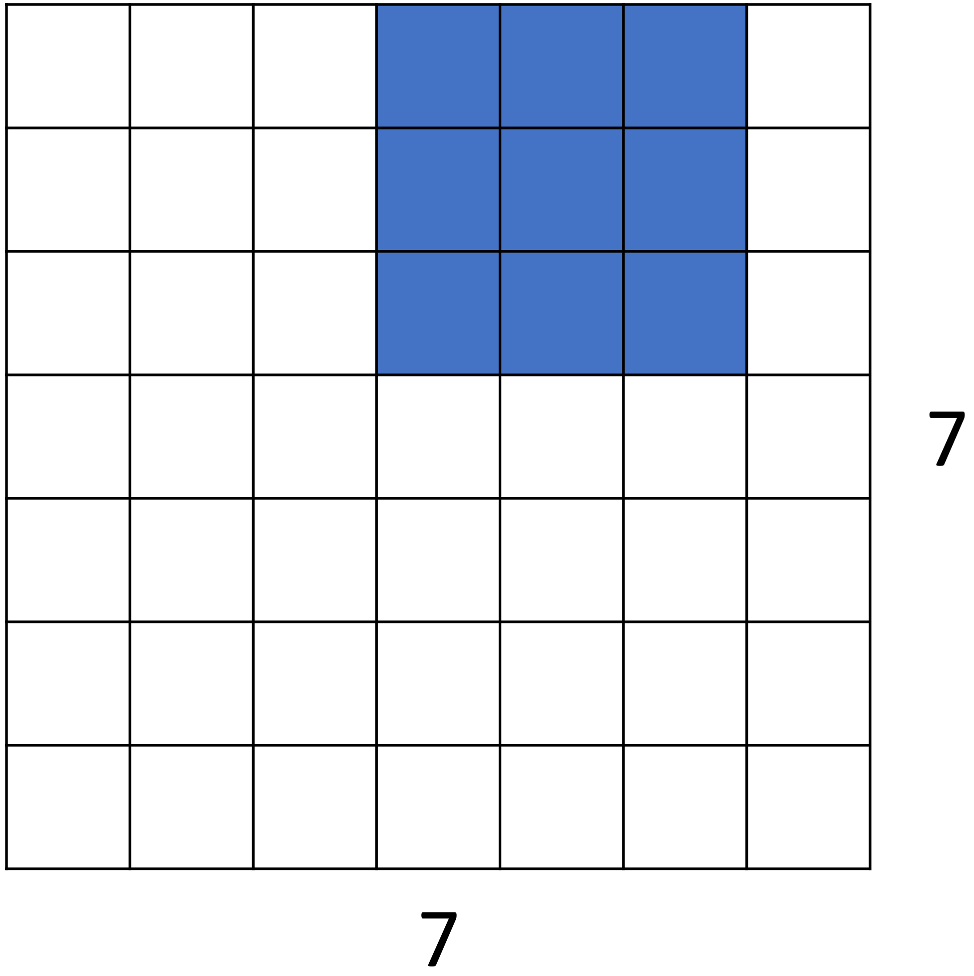
A closer look at spatial dimensions



Input: 7x7

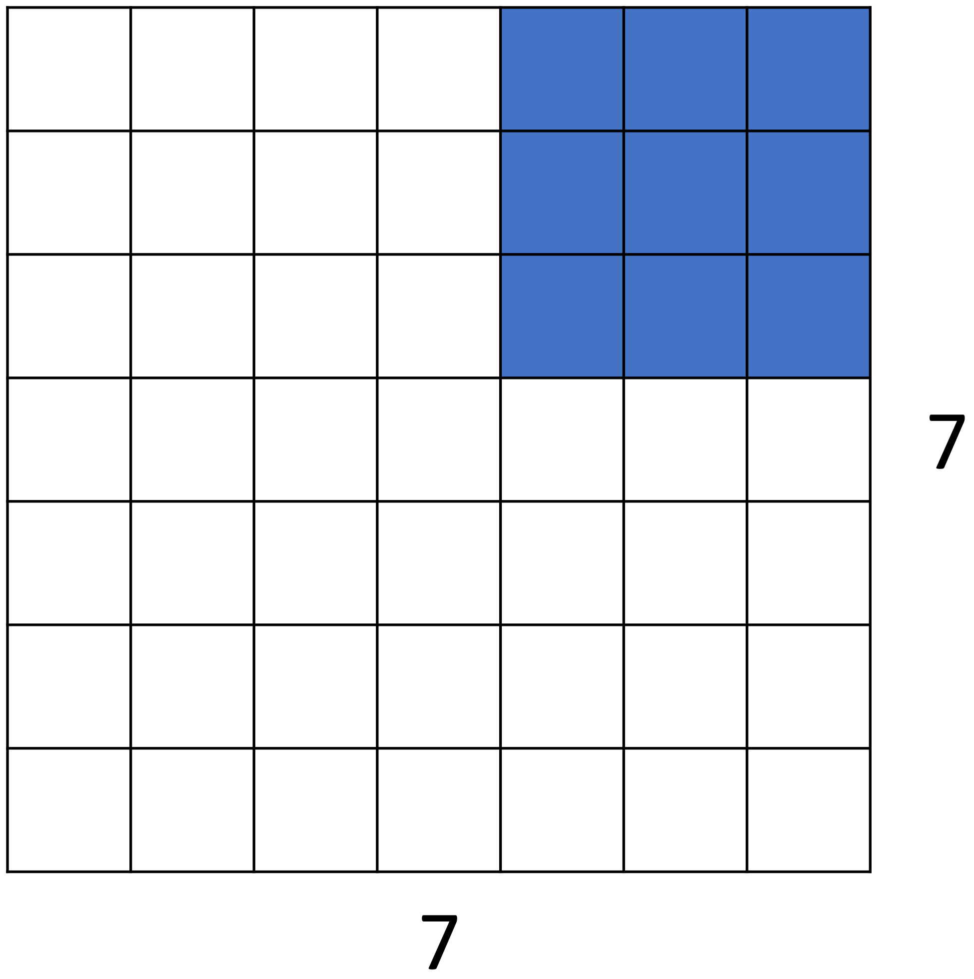
Filter: 3x3

A closer look at spatial dimensions



Input: 7x7
Filter: 3x3

A closer look at spatial dimensions



Input: 7x7
Filter: 3x3
Output: 5x5

In general:
Input: W
Filter: K
Output: W – K + 1

Problem: Feature maps “shrink” with each layer!

A closer look at spatial dimensions

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

Input: 7x7
Filter: 3x3
Output: 5x5

In general:
Input: W
Filter: K
Output: W – K + 1

Problem: Feature maps "shrink" with each layer!

Solution: **padding**
Add zeros around the input

A closer look at spatial dimensions

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

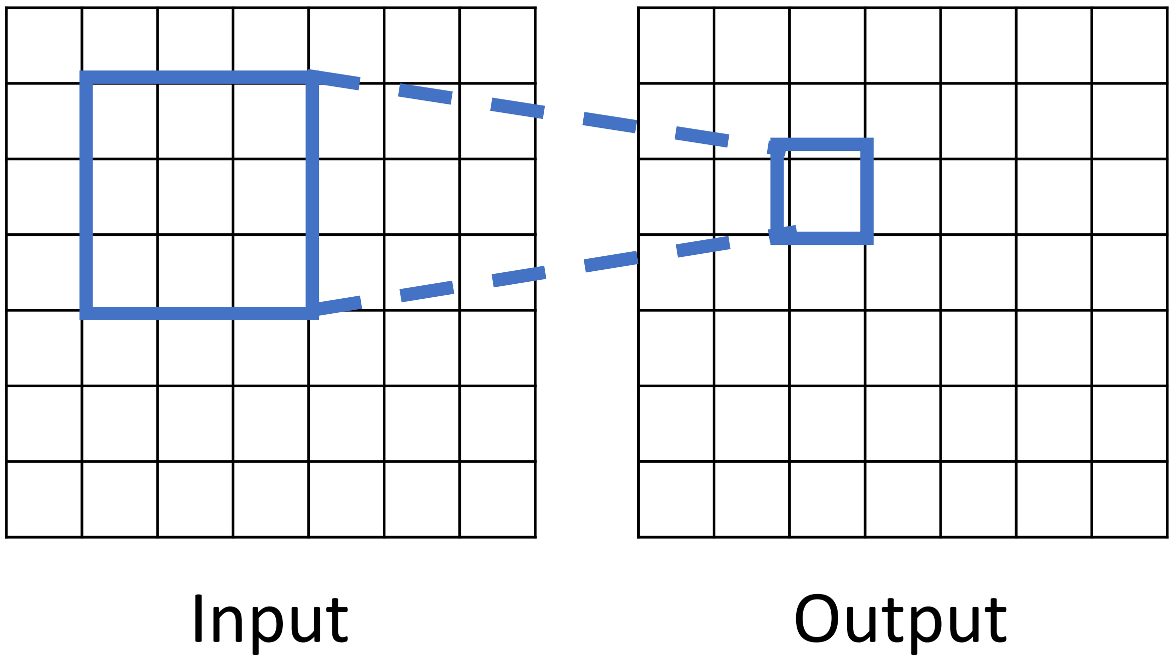
Input: 7x7
Filter: 3x3
Output: 5x5

In general:
Input: W
Filter: K
Padding: P
Output: W – K + 1 + 2P

Very common:
Set P = (K – 1) / 2 to make output have same size as input!

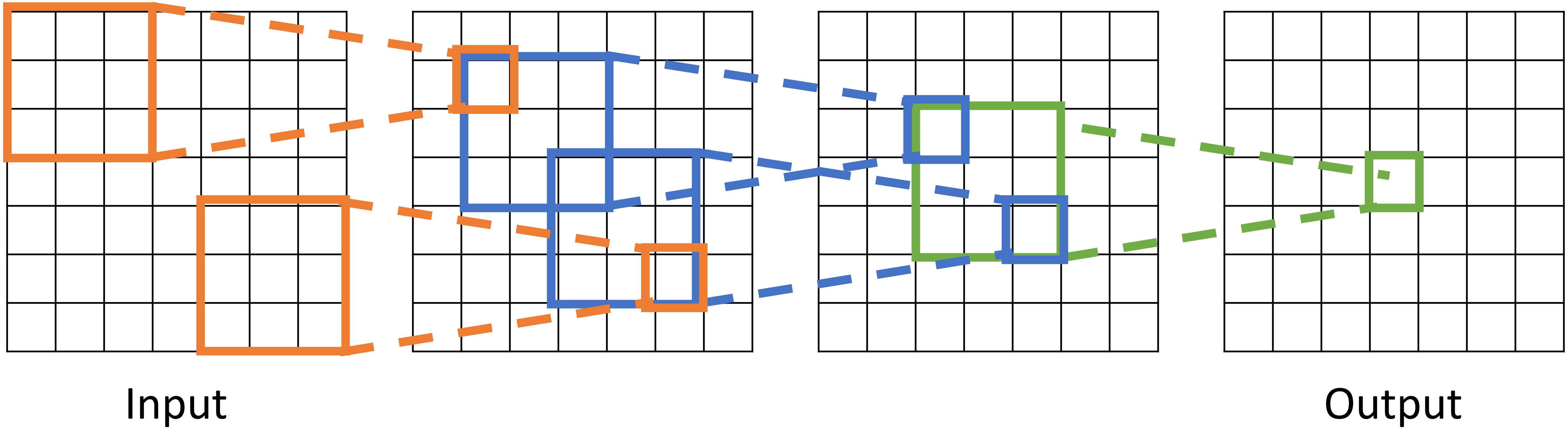
Receptive Fields

For convolution with kernel size K, each element in the output depends on a K x K **receptive field** in the input



Receptive Fields

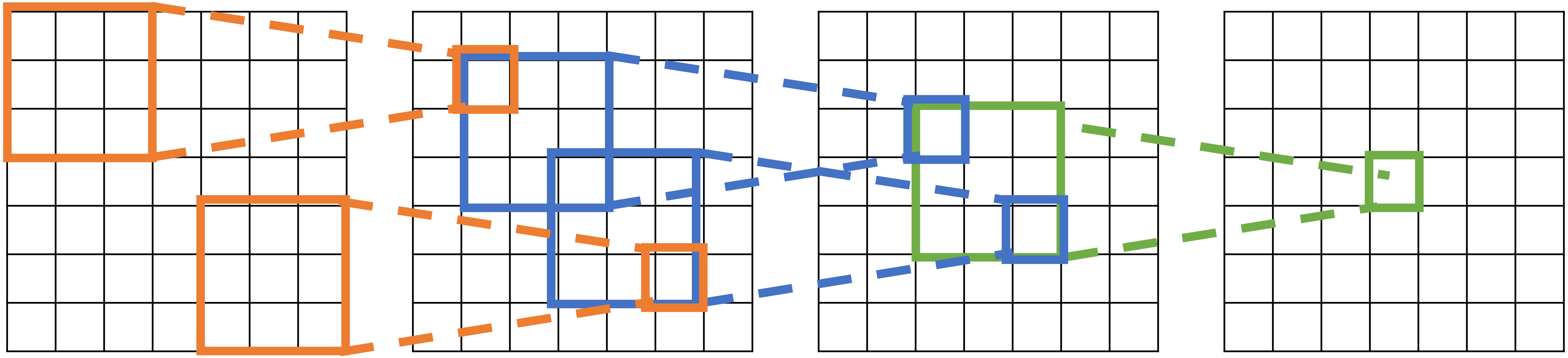
Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)



Be careful – ”receptive field in the input” vs “receptive field in the previous layer”
Hopefully clear from context!

Receptive Fields

Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)



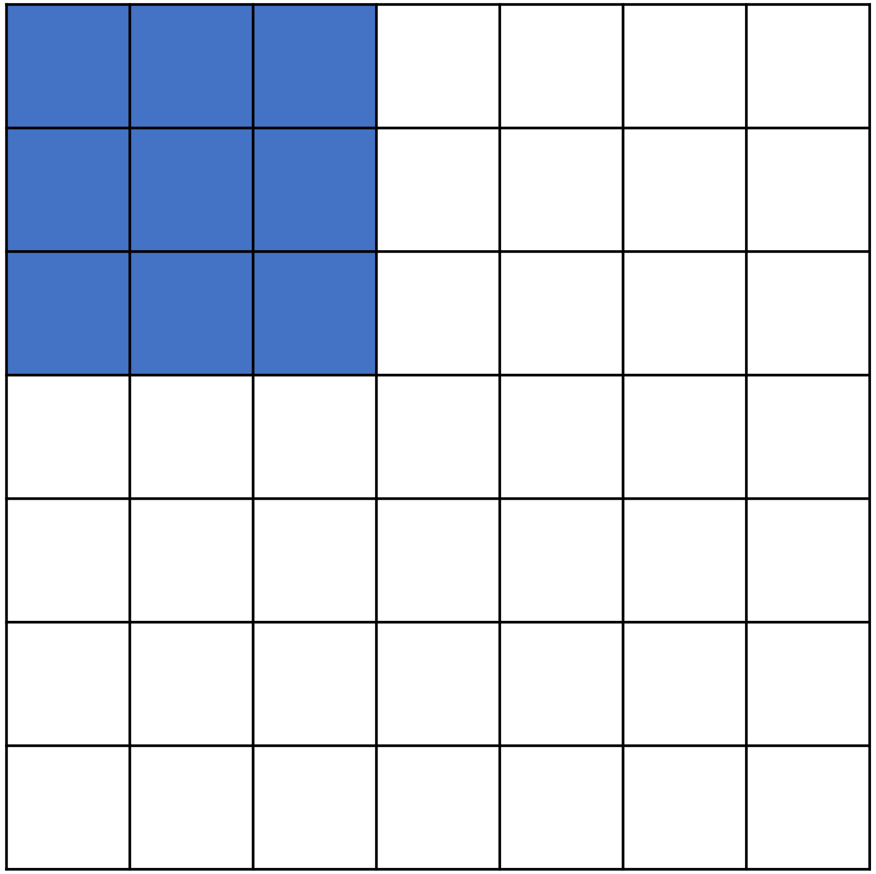
Input

Output

Problem: For large images we need many layers for each output to "see" the whole image image

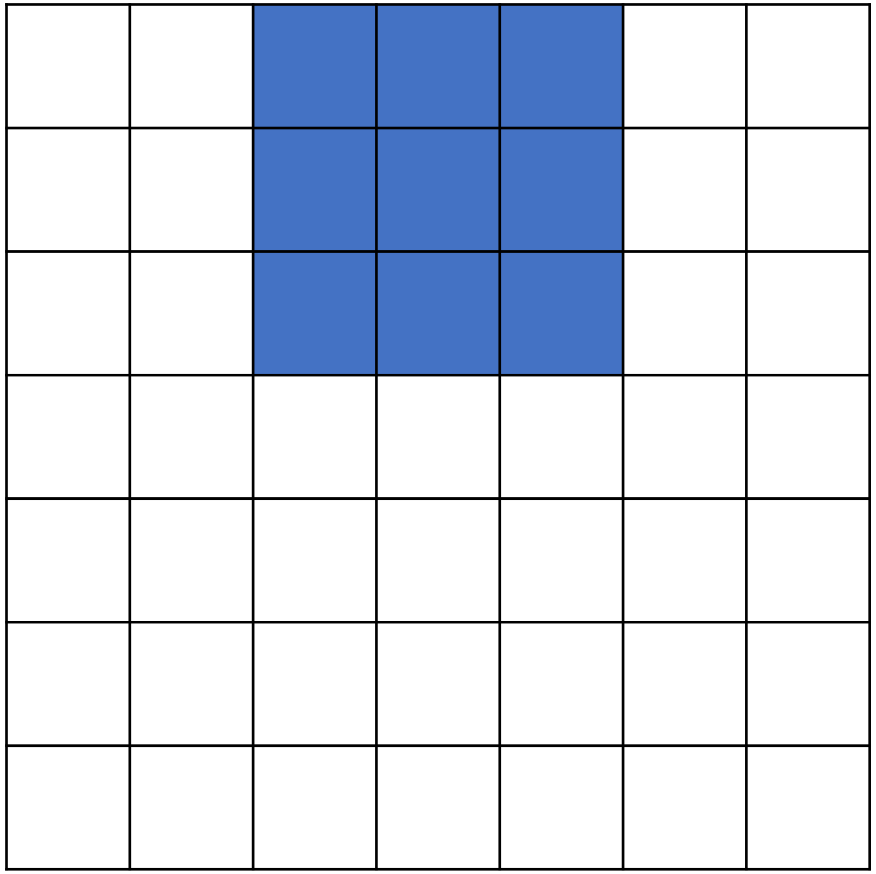
Solution: Downsample inside the network

Strided Convolution



Input: 7x7
Filter: 3x3
Stride: 2

Strided Convolution

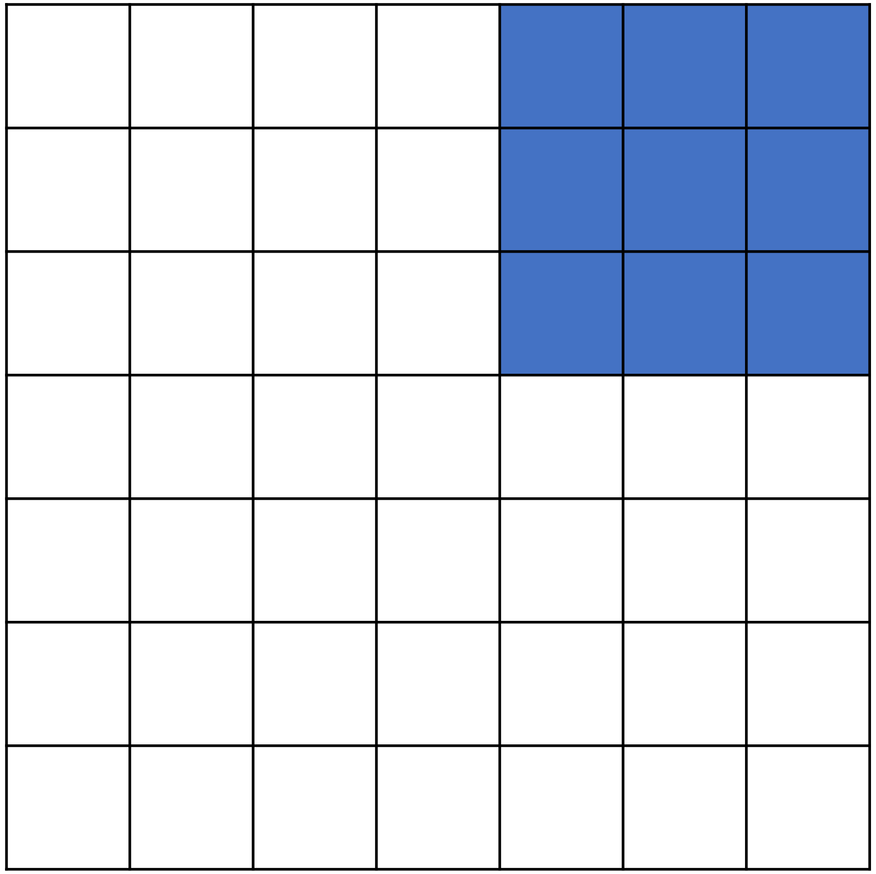


Input: 7x7

Filter: 3x3

Stride: 2

Strided Convolution



Input: 7x7

Filter: 3x3

Stride: 2

Output: 3x3

In general:

Input: W

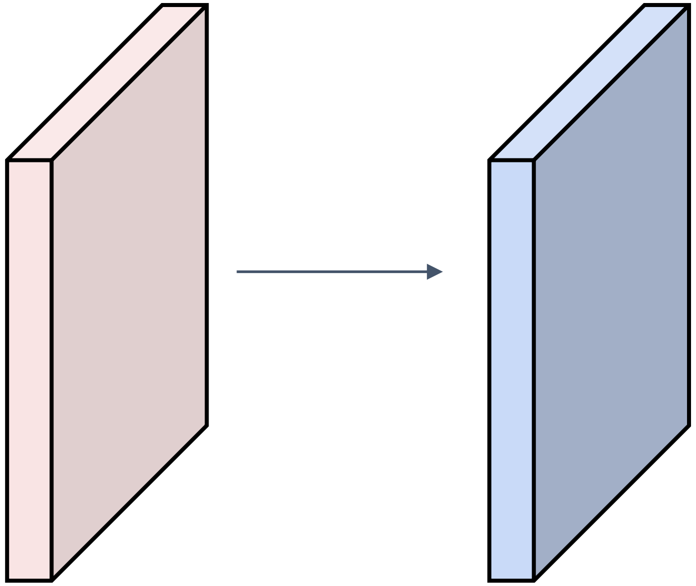
Filter: K

Padding: P

Stride: S

Output: (W – K + 2P) / S + 1

Convolution Example

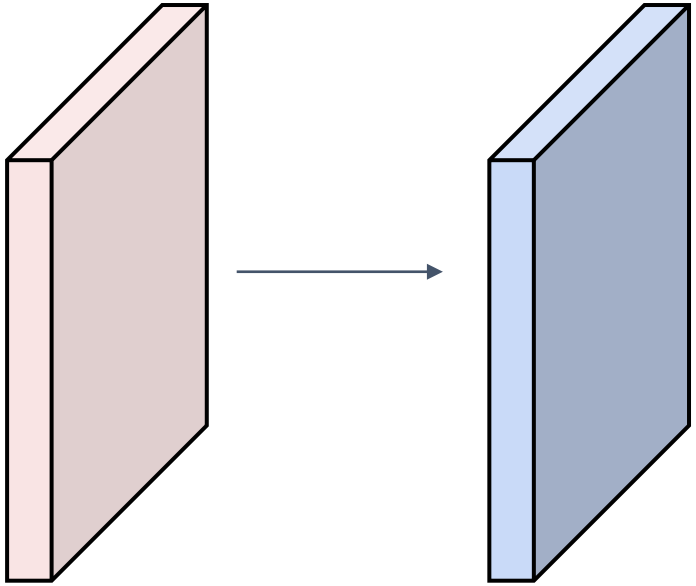


Input volume: 3 x 32 x 32

10 5x5 filters with stride 1, pad 2

Output volume size: ?

Convolution Example



Input volume: 3 x **32** x **32**

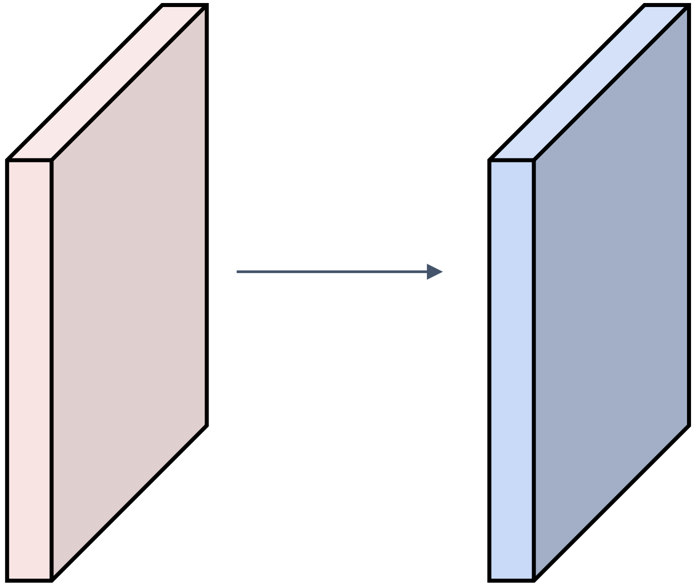
10 5x5 filters with stride **1**, pad **2**

Output volume size:

(32+2*2-5)/1+1 = 32 spatially, so

10 x 32 x 32

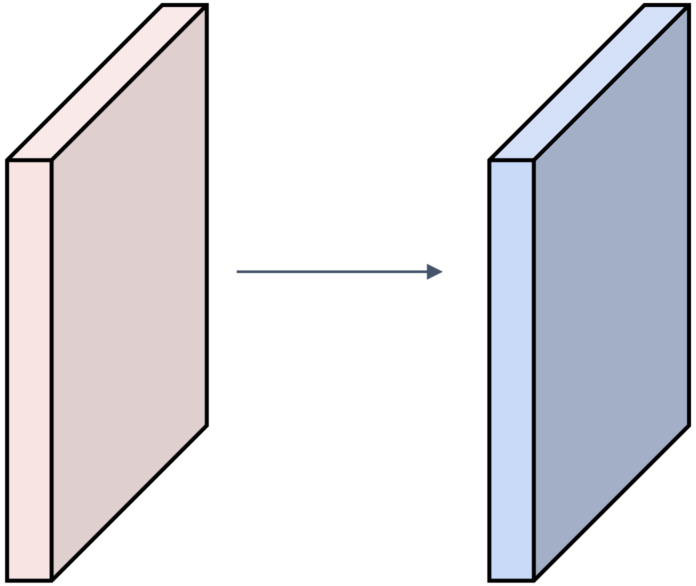
Convolution Example



Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

Output volume size: 10 x 32 x 32
Number of learnable parameters: ?

Convolution Example



Input volume: 3 x 32 x 32

10 5x5 filters with stride 1, pad 2

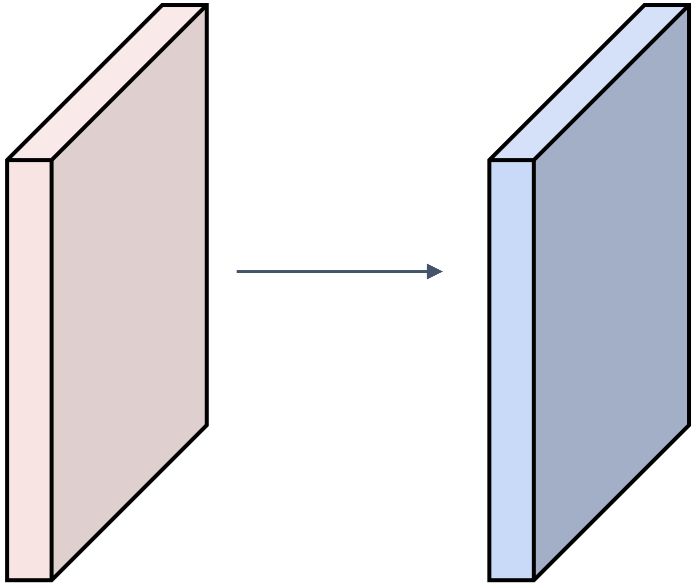
Output volume size: 10 x 32 x 32

Number of learnable parameters: **760**

Parameters per filter: 3*5*5 + 1 (for bias) = **76**

10 filters, so total is 10 * **76** = **760**

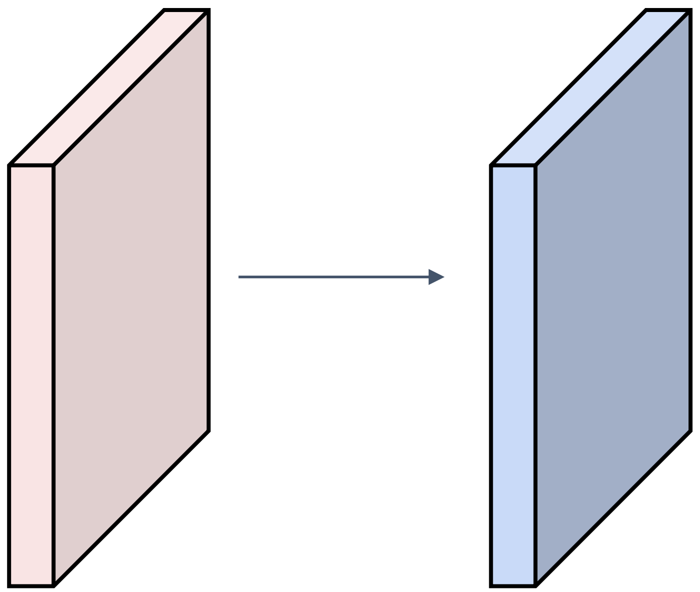
Convolution Example



Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

Output volume size: 10 x 32 x 32
Number of learnable parameters: 760
Number of multiply-add operations: ?

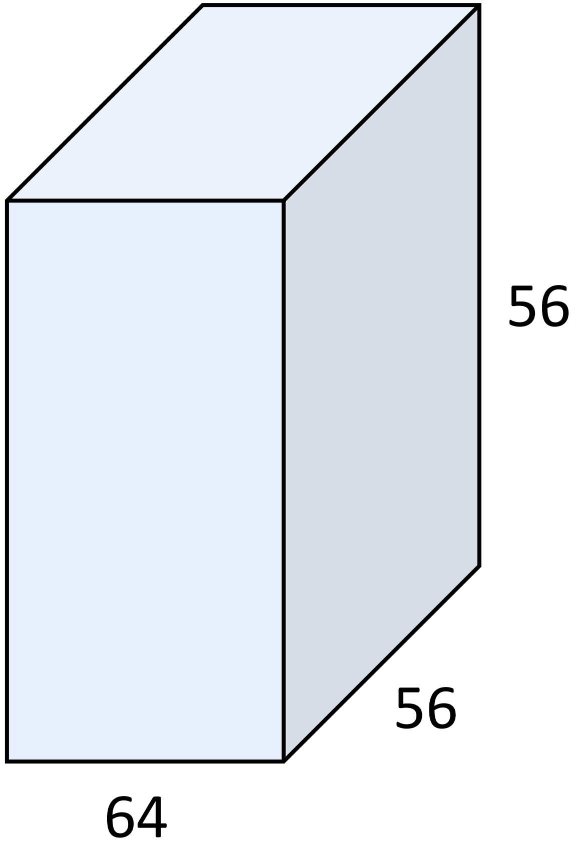
Convolution Example



Input volume: **3** x 32 x 32
10 **5x5** filters with stride 1, pad 2

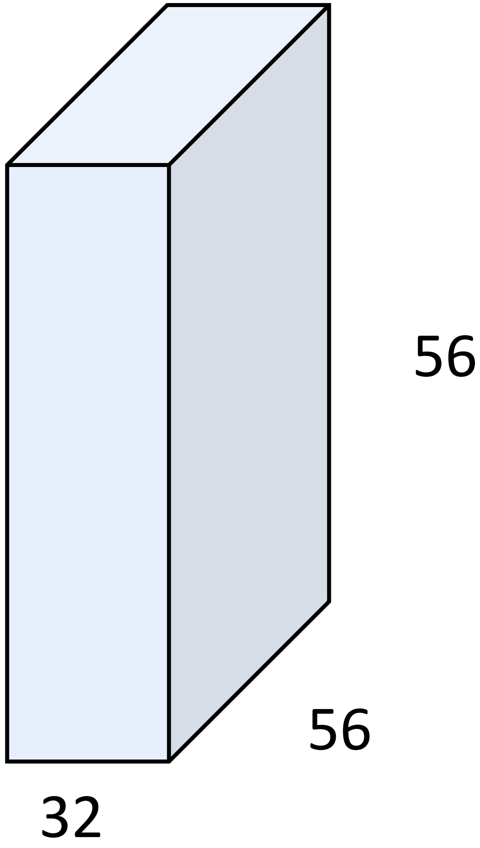
Output volume size: **10 x 32 x 32**
Number of learnable parameters: 760
Number of multiply-add operations: **768,000**
10*32*32 = 10,240 outputs; each output is the inner product of two **3**x**5x5** tensors (75 elems); total = 75*10240 = **768K**

Example: 1x1 Convolution



1x1 CONV
with 32 filters

(each filter has size 1x1x64, and performs a 64-dimensional dot product)



Stacking 1x1 conv layers gives MLP operating on each input position

Lin et al, "Network in Network", ICLR 2014

Convolution Summary

Input: C_{in} x H x W

Hyperparameters:

- **Kernel size**: K_H x K_W
- **Number filters**: C_{out}
- **Padding**: P
- **Stride**: S

Weight matrix: C_{out} x C_{in} x K_H x K_W giving C_{out} filters of size C_{in} x K_H x K_W

Bias vector: C_{out}

Output: C_{out} x H' x W' where:

- $H' = (H - K + 2P) / S + 1$
- $W' = (W - K + 2P) / S + 1$

Common settings:

$K_H = K_W$ (Small square filters)

$P = (K - 1) / 2$ ("Same" padding)

C_{in}, C_{out} = 32, 64, 128, 256 (powers of 2)

K = 3, P = 1, S = 1 (3x3 conv)

K = 5, P = 2, S = 1 (5x5 conv)

K = 1, P = 0, S = 1 (1x1 conv)

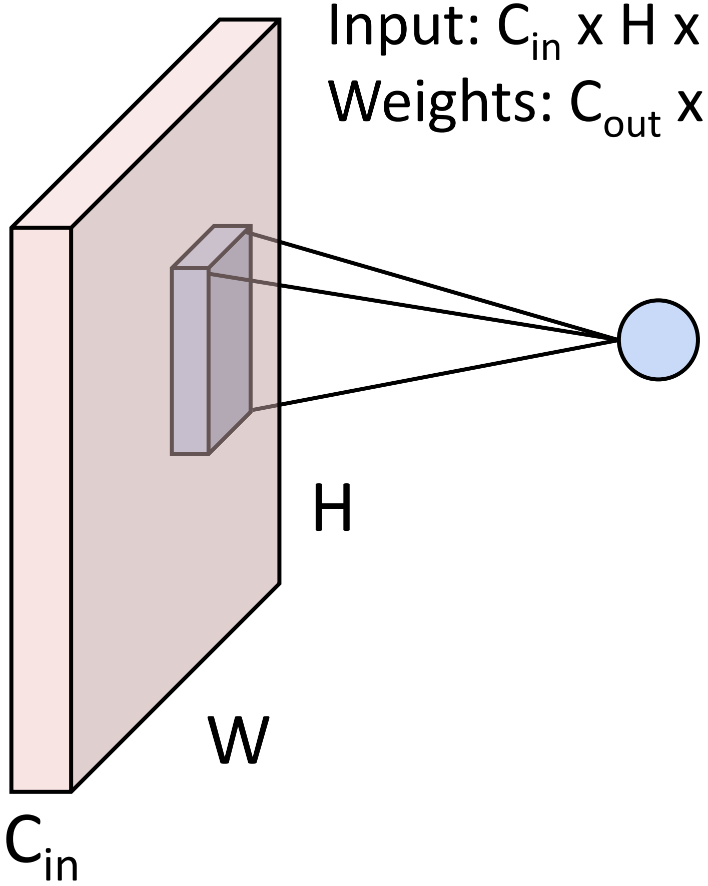
K = 3, P = 1, S = 2 (Downsample by 2)

K = 7, P = 3, S = 2 (7x7 conv, downsample)

Other types of convolution

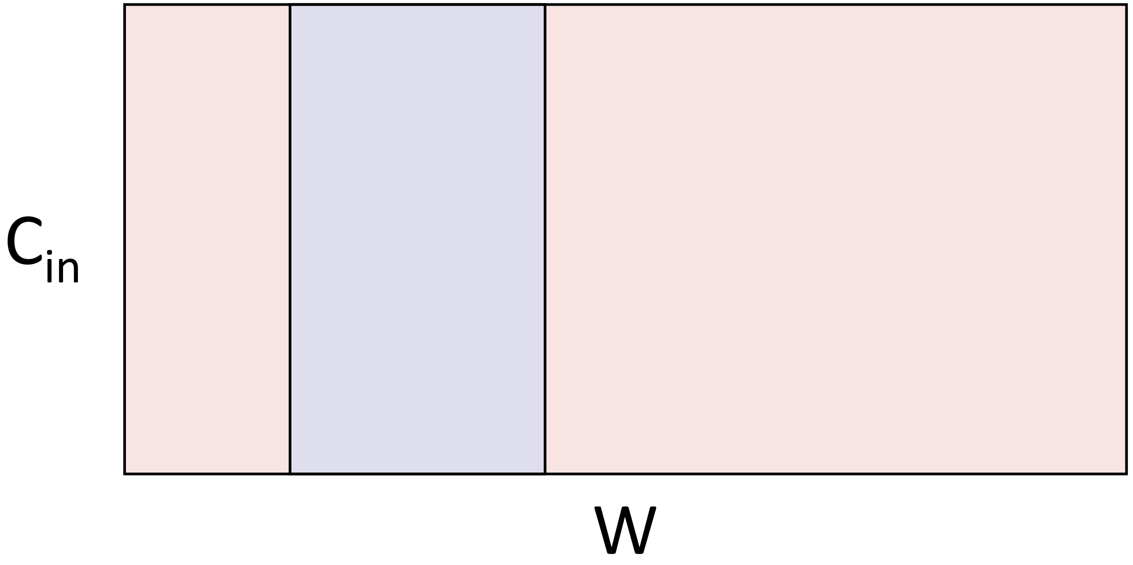
So far: 2D Convolution

Input: C_{in} x H x W
Weights: C_{out} x C_{in} x K x K



1D Convolution

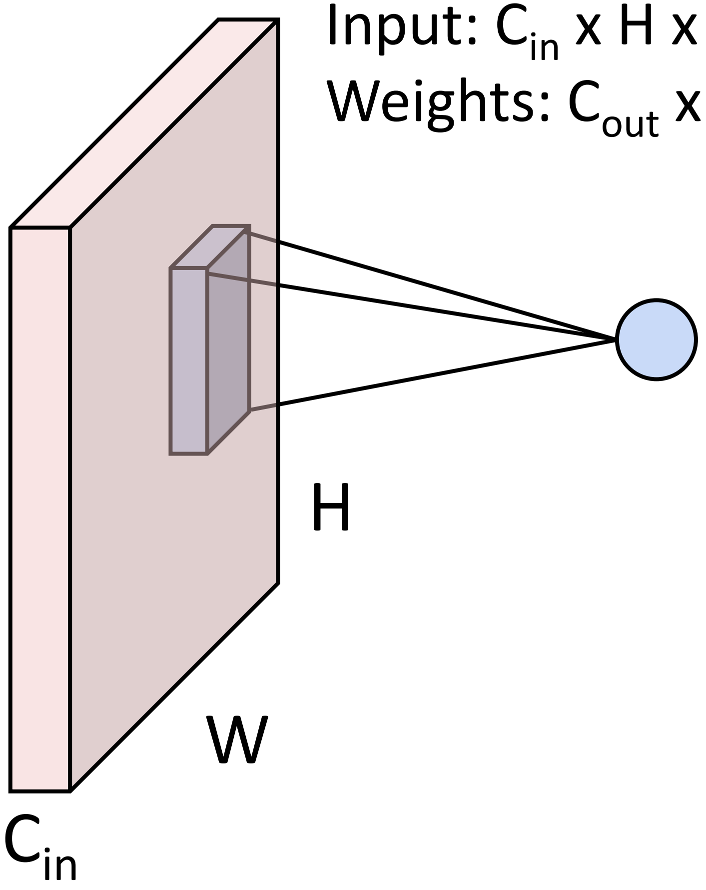
Input: C_{in} x W
Weights: C_{out} x C_{in} x K



Other types of convolution

So far: 2D Convolution

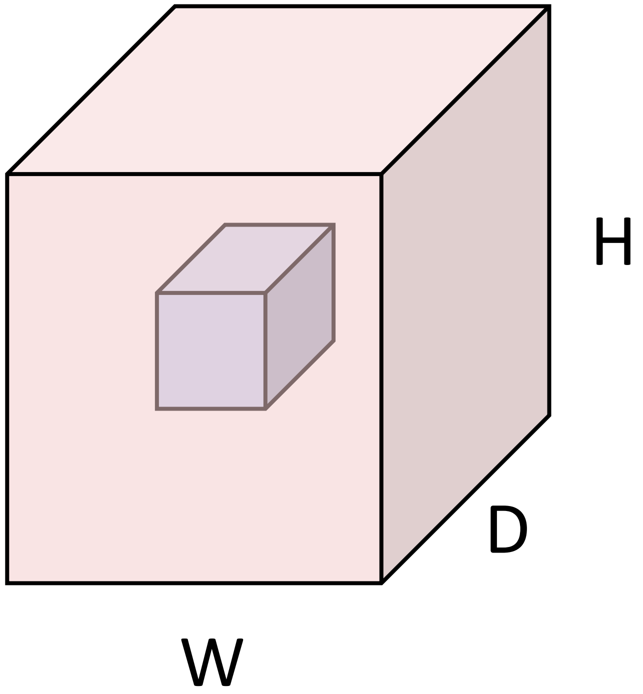
Input: C_{in} x H x W
Weights: C_{out} x C_{in} x K x K



3D Convolution

Input: C_{in} x H x W x D
Weights: C_{out} x C_{in} x K x K x K

C_{in}-dim vector
at each point
in the volume



PyTorch Convolution Layer

Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

PyTorch Convolution Layers

Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

Conv1d

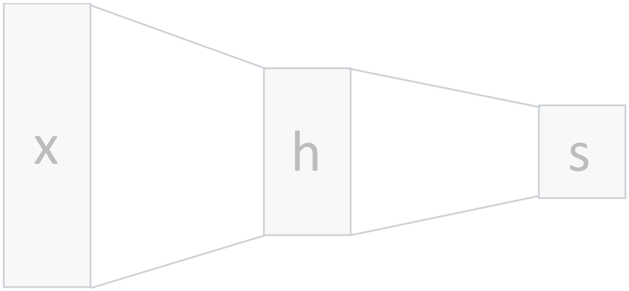
CLASS `torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

Conv3d

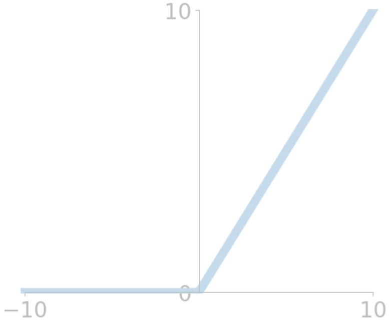
CLASS `torch.nn.Conv3d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

Components of a Convolutional Network

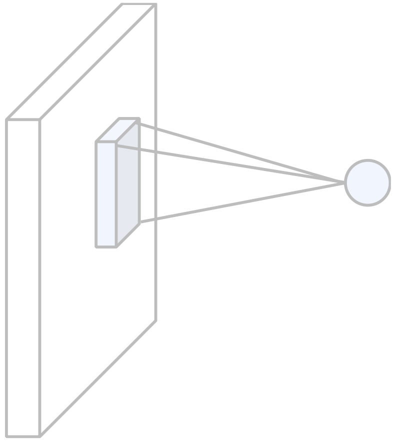
Fully-Connected Layers



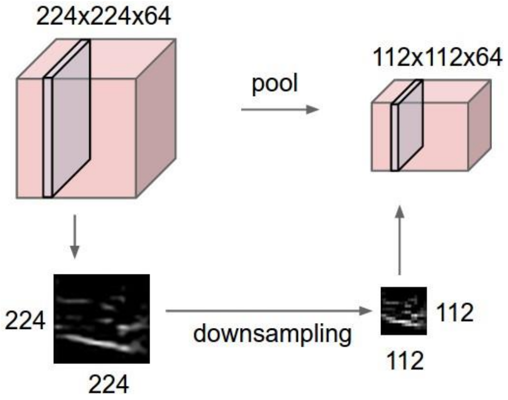
Activation Function



Convolution Layers



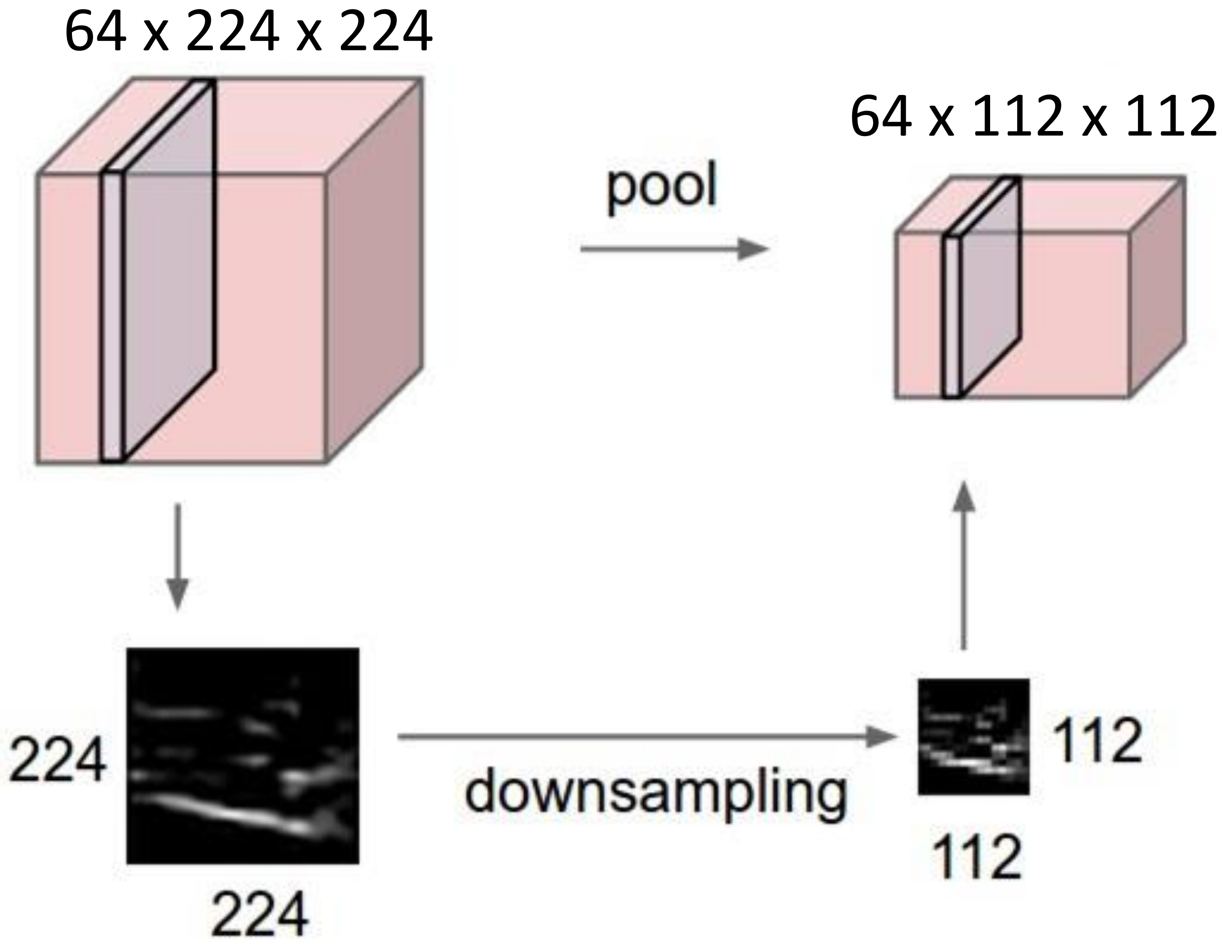
Pooling Layers



Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

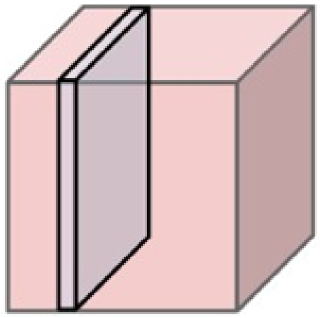
Pooling Layers: Another way to downsample



Hyperparameters:
Kernel Size
Stride
Pooling function

Max Pooling

64 x 224 x 224



Single depth slice

x

2	3	2	4
5	**6**	6	**8**
3	2	1	0
1	2	3	**4**

y

Max pooling with 2x2
kernel size and stride 2

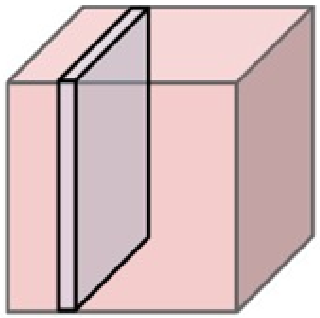
6	8
3	4

Introduces **invariance** to small spatial shifts

No learnable parameters!

Average Pooling

64 x 224 x 224



Single depth slice

x

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

y

Avg pooling with 2x2 kernel size and stride 2

4	5
2	2

Introduces **invariance** to small spatial shifts

No learnable parameters!

Pooling Summary

Input: C x H x W

Hyperparameters:

- Kernel size: K
- Stride: S
- Pooling function (max, avg)

Output: C x H' x W' where

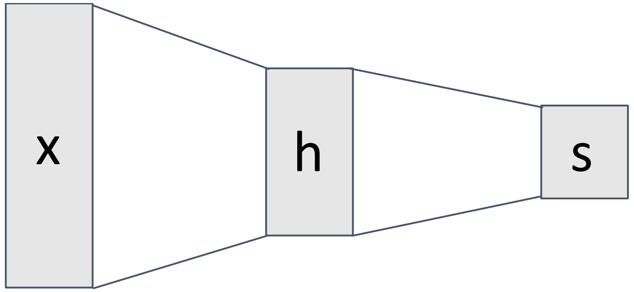
- H' = (H – K) / S + 1
- W' = (W – K) / S + 1

Learnable parameters: None!

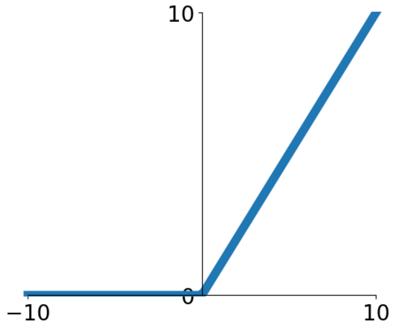
Common settings:
max, K = 2, S = 2
max, K = 3, S = 2 (AlexNet)

Components of a Convolutional Network

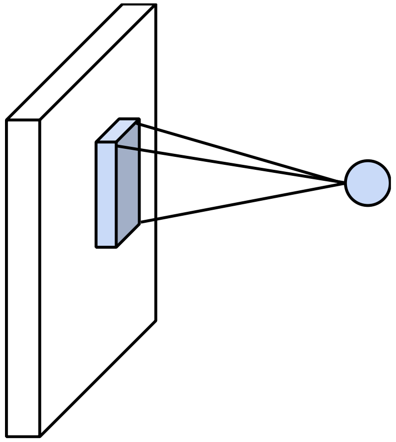
Fully-Connected Layers



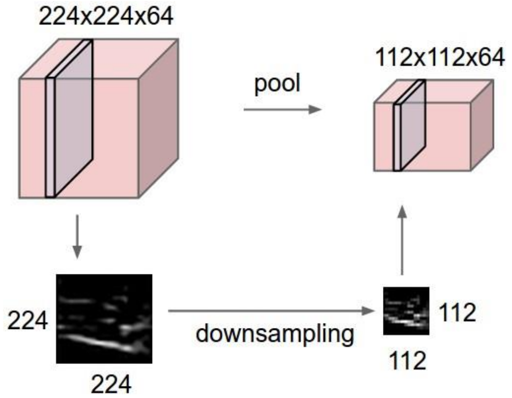
Activation Function



Convolution Layers



Pooling Layers



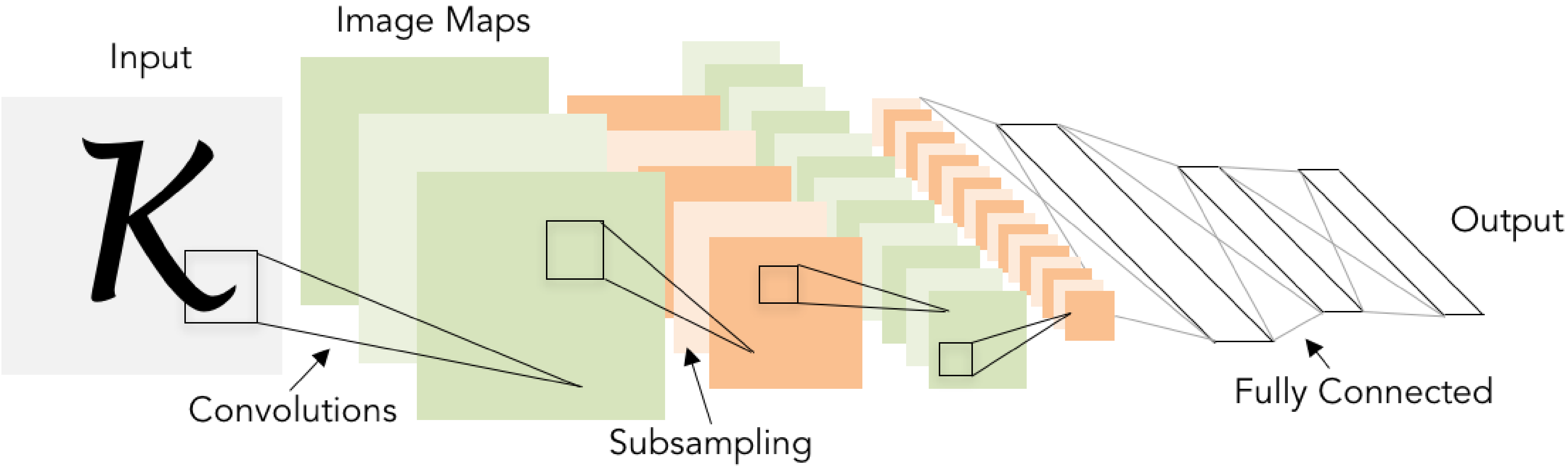
Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Convolutional Networks

Classic architecture: [Conv, ReLU, Pool] x N, flatten, [FC, ReLU] x N, FC

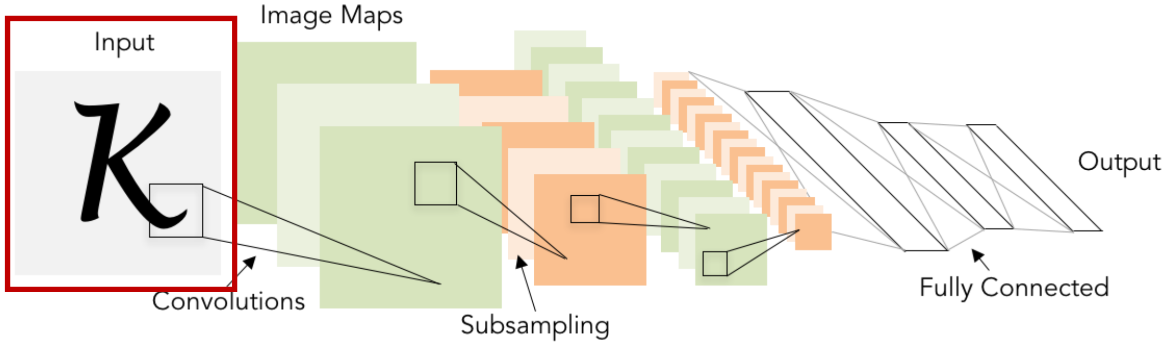
Example: LeNet-5



Lecun et al, "Gradient-based learning applied to document recognition", 1998

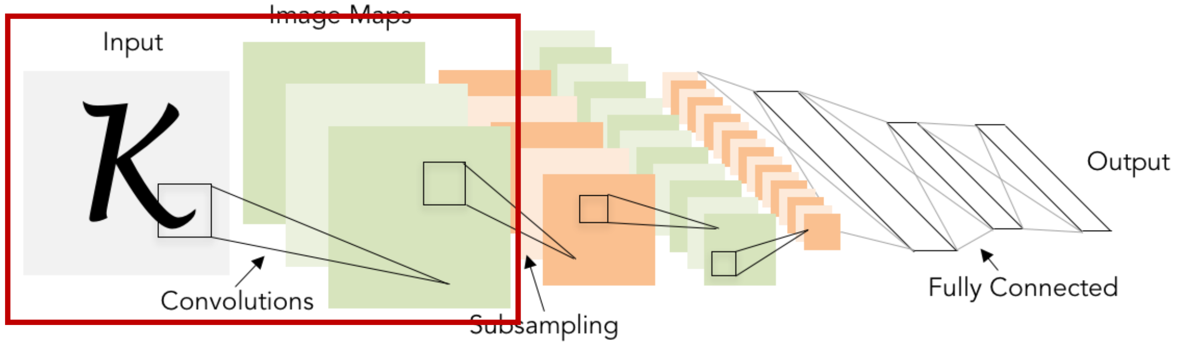
Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	



Lecun et al, "Gradient-based learning applied to document recognition", 1998

Example: LeNet-5

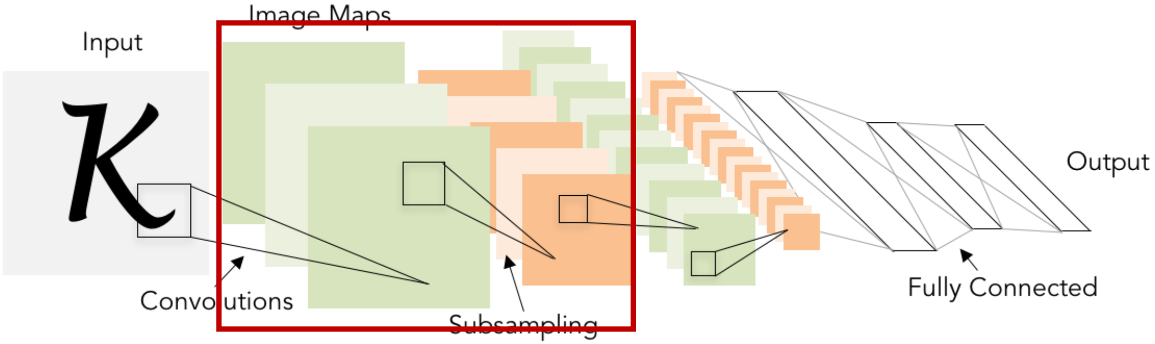


Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	

Lecun et al, “Gradient-based learning applied to document recognition”, 1998

Example: LeNet-5

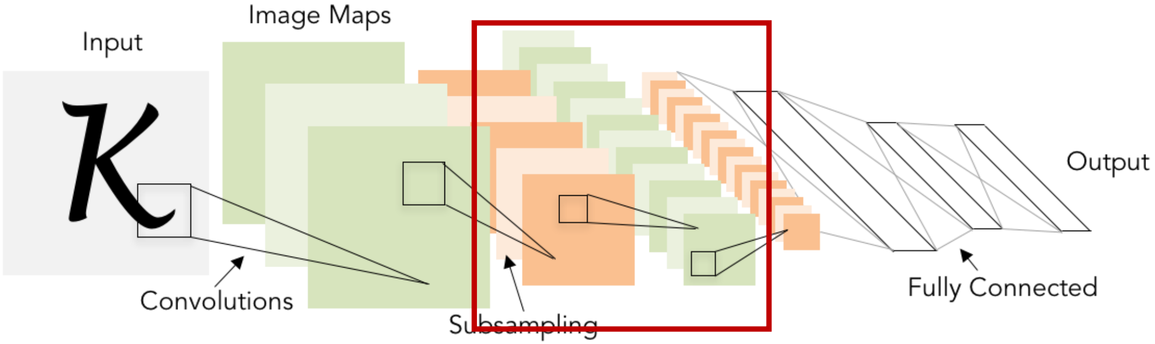
Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	



Lecun et al, “Gradient-based learning applied to document recognition”, 1998

Example: LeNet-5

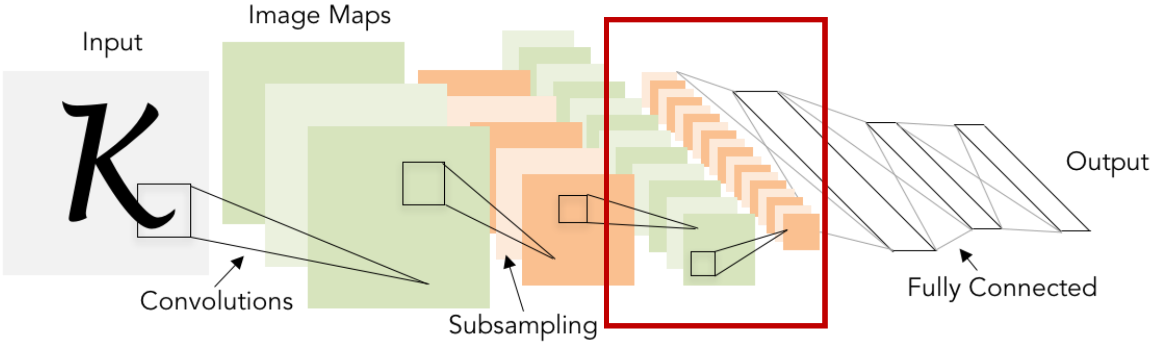
Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	



Lecun et al, “Gradient-based learning applied to document recognition”, 1998

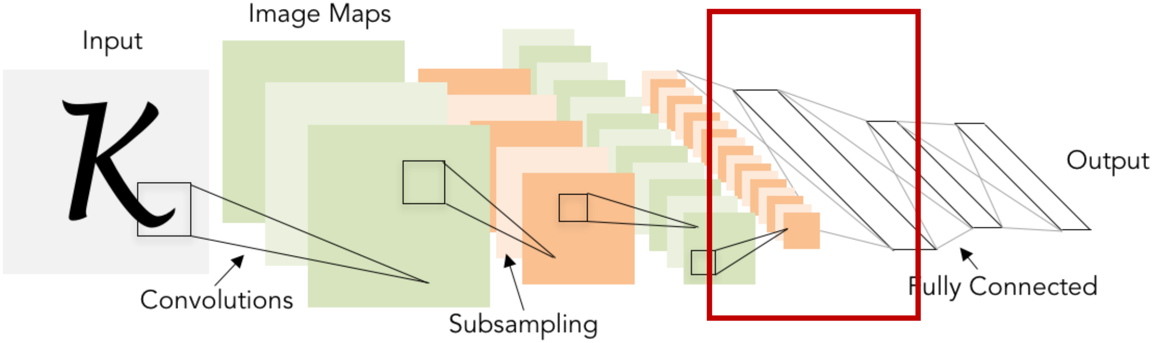
Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	



Lecun et al, “Gradient-based learning applied to document recognition”, 1998

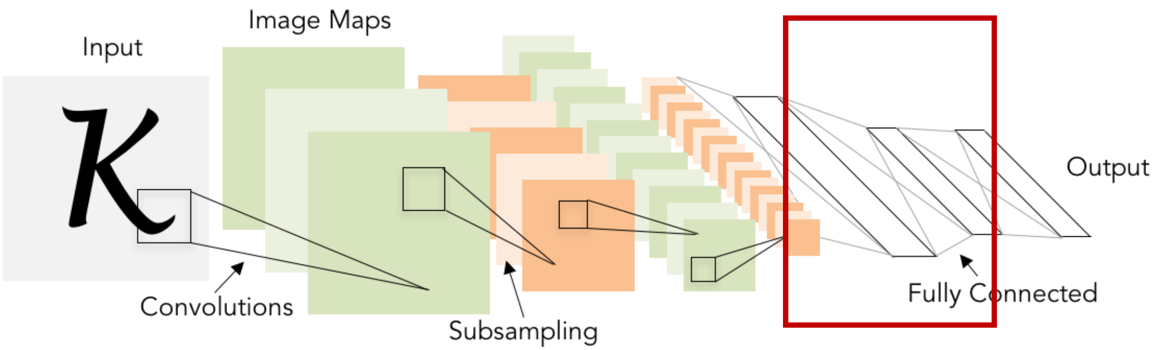
Example: LeNet-5



Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	

Lecun et al, “Gradient-based learning applied to document recognition”, 1998

Example: LeNet-5

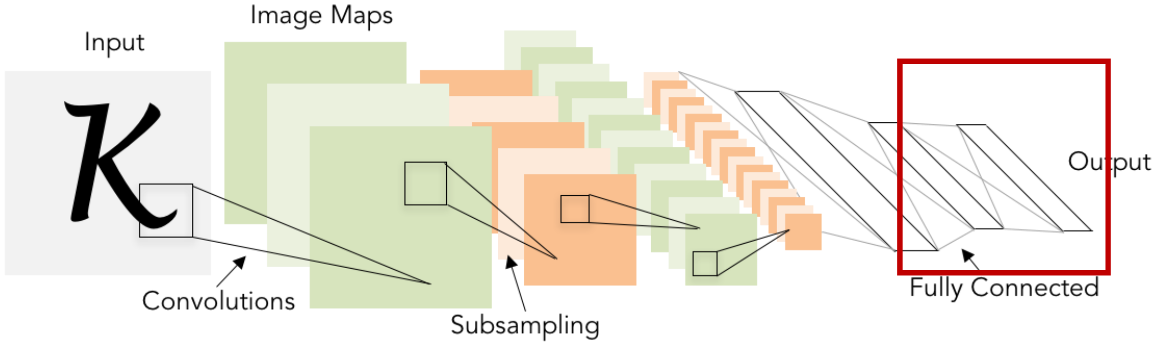


Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	

Lecun et al, “Gradient-based learning applied to document recognition”, 1998

Example: LeNet-5

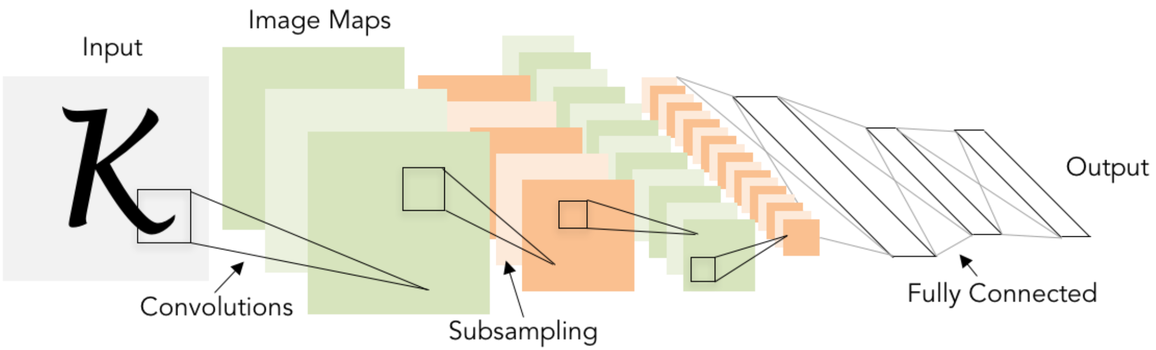
Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	
Linear (500 -> 10)	10	500 x 10



Lecun et al, “Gradient-based learning applied to document recognition”, 1998

Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	
Linear (500 -> 10)	10	500 x 10



As we go through the network:

Spatial size **decreases**
(using pooling or strided conv)

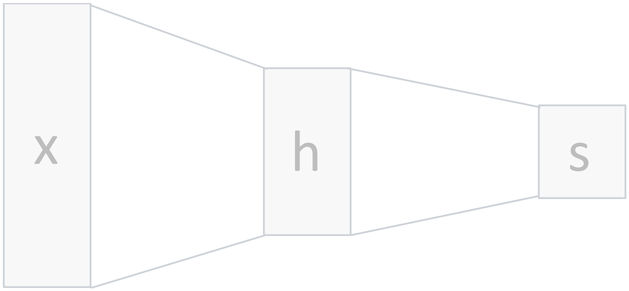
Number of channels **increases**
(total "volume" is preserved!)

Lecun et al, "Gradient-based learning applied to document recognition", 1998

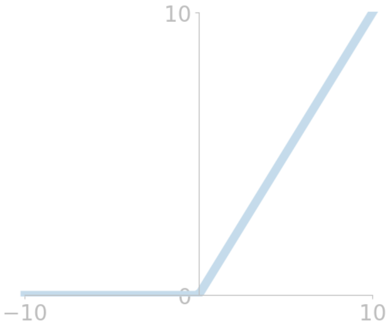
Problem: Deep Networks very hard to train!

Components of a Convolutional Network

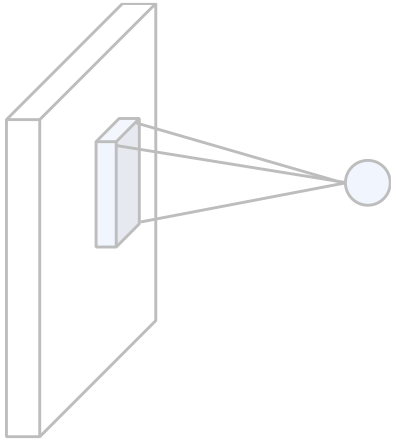
Fully-Connected Layers



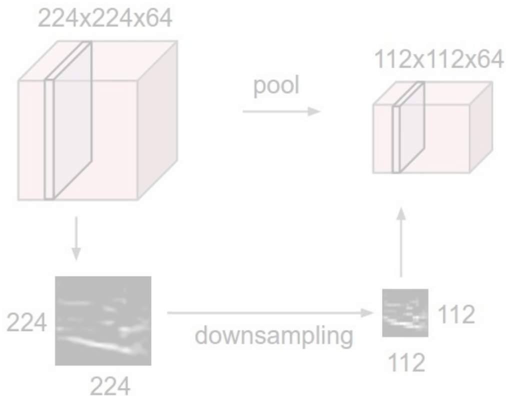
Activation Function



Convolution Layers



Pooling Layers



Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Batch Normalization

Consider a single layer $y = Wx$

The following could lead to tough optimization:
- Inputs x are not *centered around zero* (need large bias)
- Inputs x have different scaling per-element
  (entries in W will need to vary a lot)

Idea: force inputs to be "nicely scaled" at each layer!

Batch Normalization

Idea: "Normalize" the outputs of a layer so they have zero mean and unit variance

Why? Helps reduce "internal covariate shift", improves optimization

We can normalize a batch of activations like this:

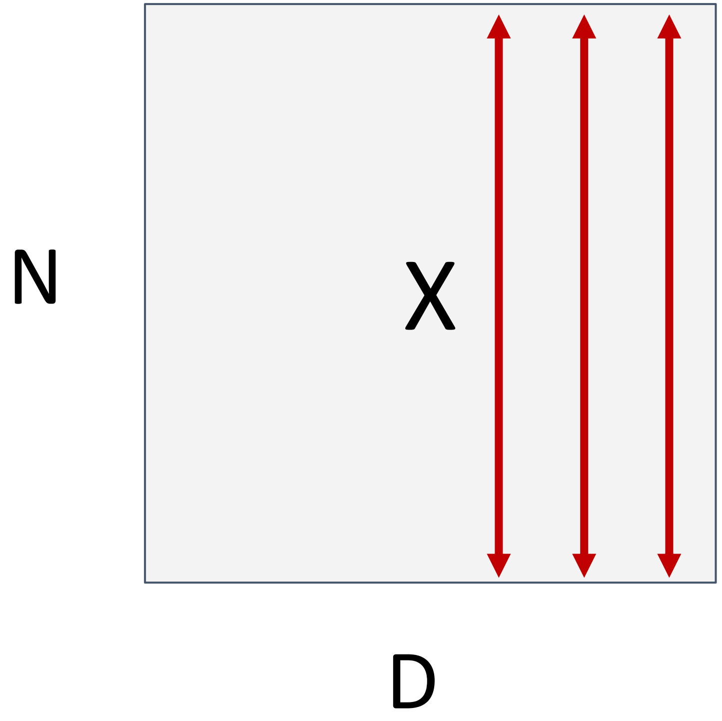
$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

This is a **differentiable function**, so we can use it as an operator in our networks and backprop through it!

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} \left(x_{i,j} - \mu_j\right)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint?

Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015

Batch Normalization

Problem: Estimates depend on minibatch; can't do this at test-time!

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} \left(x_{i,j} - \mu_j\right)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^{D}$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$\mu_j =$ (Running) average of values seen during training

Per-channel mean, shape is D

$\sigma_j^2 =$ (Running) average of values seen during training

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

$\mu_j =$ (Running) average of values seen during training

Per-channel mean, shape is D

Learnable scale and shift parameters:

$$\gamma, \beta \in \mathbb{R}^{D}$$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$\mu_j^{test} = 0$$

For each training iteration:

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

$$\mu_j^{test} = 0.99\, \mu_j^{test} + 0.01\, \mu_j$$

(Similar for σ)

Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

During testing batchnorm becomes a linear operator! Can be fused with the previous fully-connected or conv layer

$\mu_j =$ (Running) average of values seen during training — Per-channel mean, shape is D

$\sigma_j^2 =$ (Running) average of values seen during training — Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

$$x : N \times D$$

Normalize $\downarrow$ (N)

$$\mu, \sigma : 1 \times D$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

$$x : N \times C \times H \times W$$

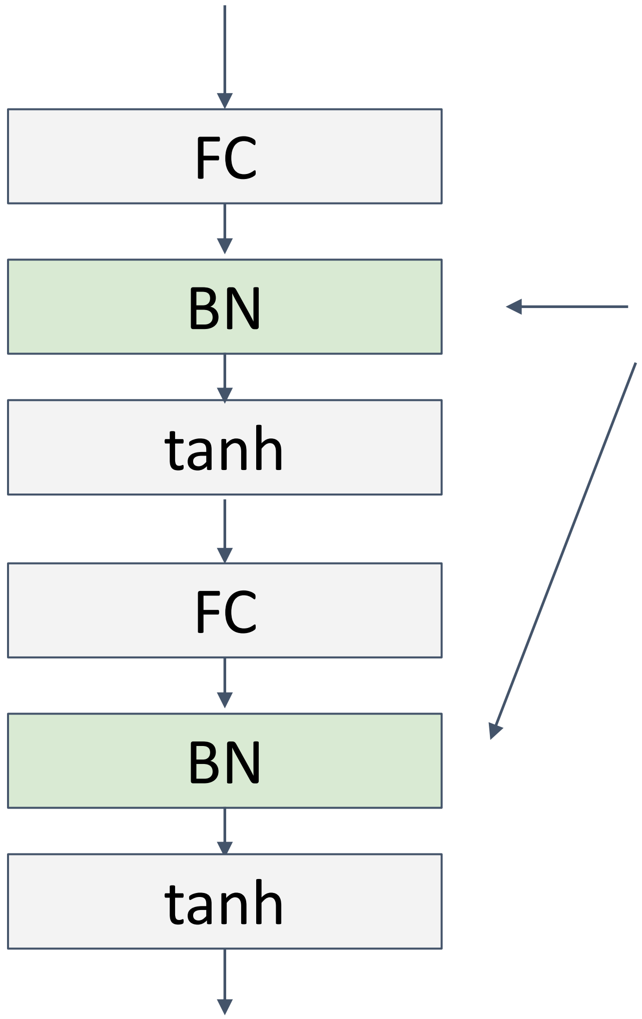
Normalize $\downarrow$ (N, H, W)

$$\mu, \sigma : 1 \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Batch Normalization

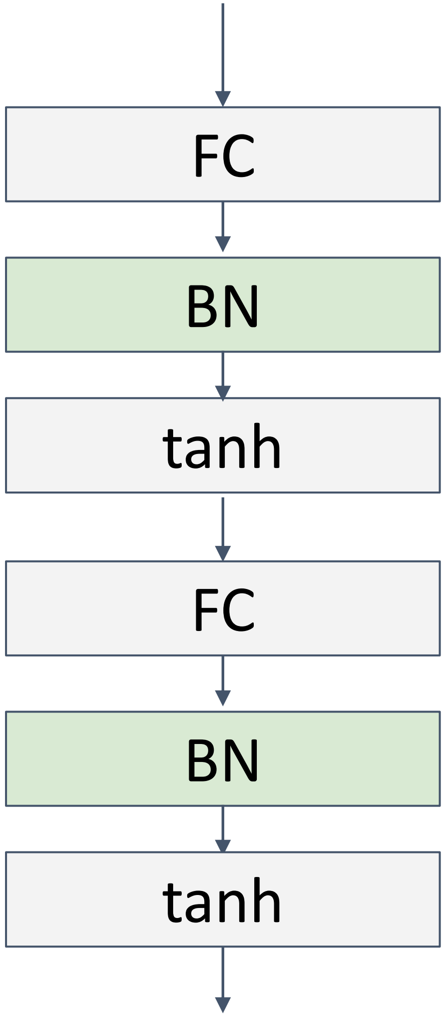


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

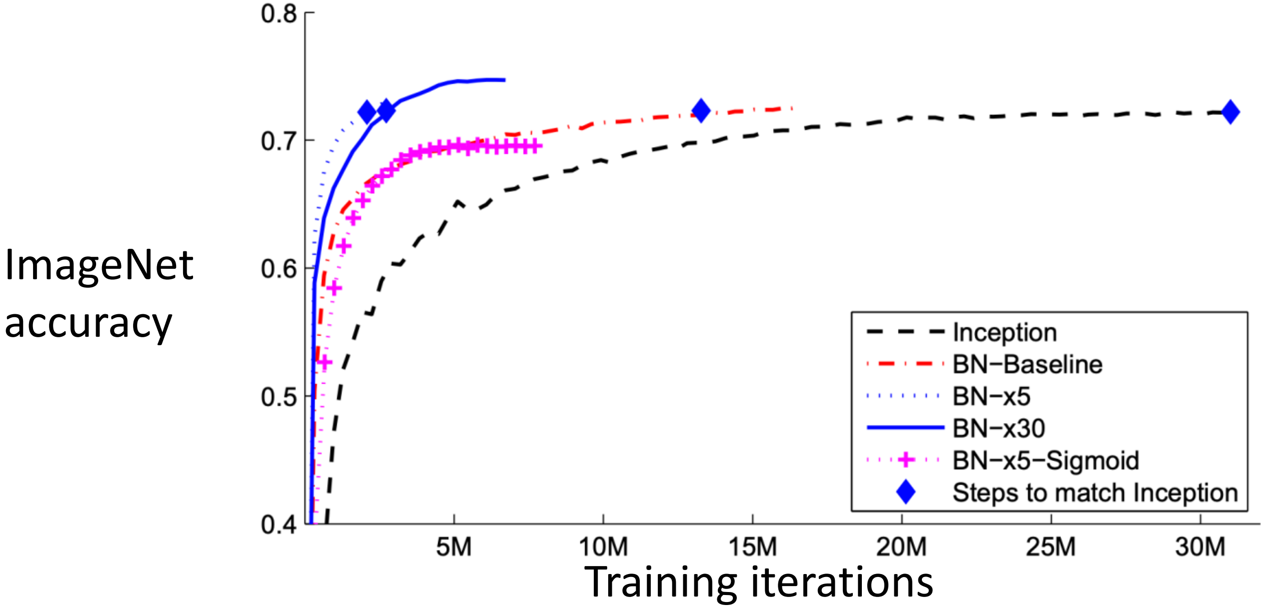
$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

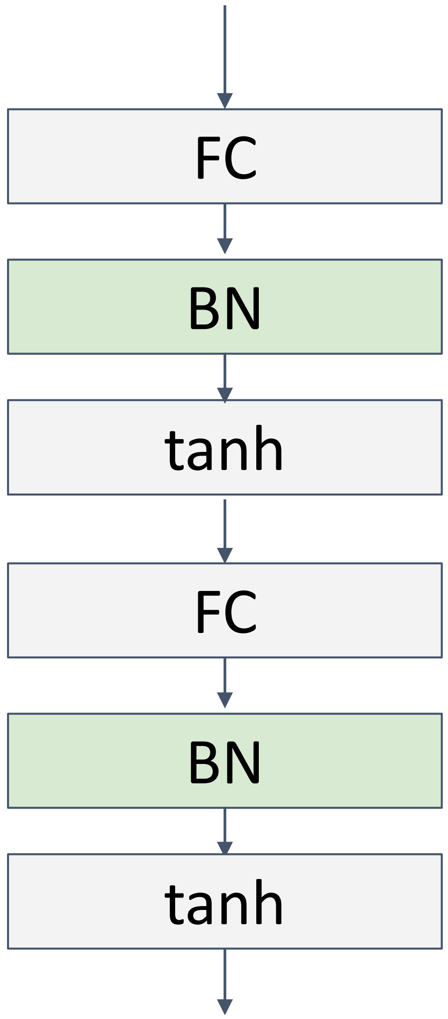


- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!



Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015

Batch Normalization



- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Not well-understood theoretically (yet)
  - Original paper: BN reduces internal covariate shift (ICS)
  - Santurkar et al.: ICS is not the reason of improvement; even BN might not reduce ICS; instead, BN makes the optimization landscape smoother
- Behaves differently during training and testing: this is a very common source of bugs!

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Santurkar et al, "How Does Batch Normalization Help Optimization?", NeurIPS 2018

Layer Normalization

Batch Normalization for **fully-connected** networks

$$x : N \times D$$

Normalize

$$\mu, \sigma \; : 1 \times D$$

$$\gamma, \beta \; : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Layer Normalization for fully-connected networks
Same behavior at train and test!
Used in RNNs, Transformers

$$x : N \times D$$

Normalize

$$\mu, \sigma \; : N \times 1$$

$$\gamma, \beta \; : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Instance Normalization

Batch Normalization for convolutional networks

$$x : N \times C \times H \times W$$

Normalize

$$\mu, \sigma : 1 \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Instance Normalization for convolutional networks

$$x : N \times C \times H \times W$$

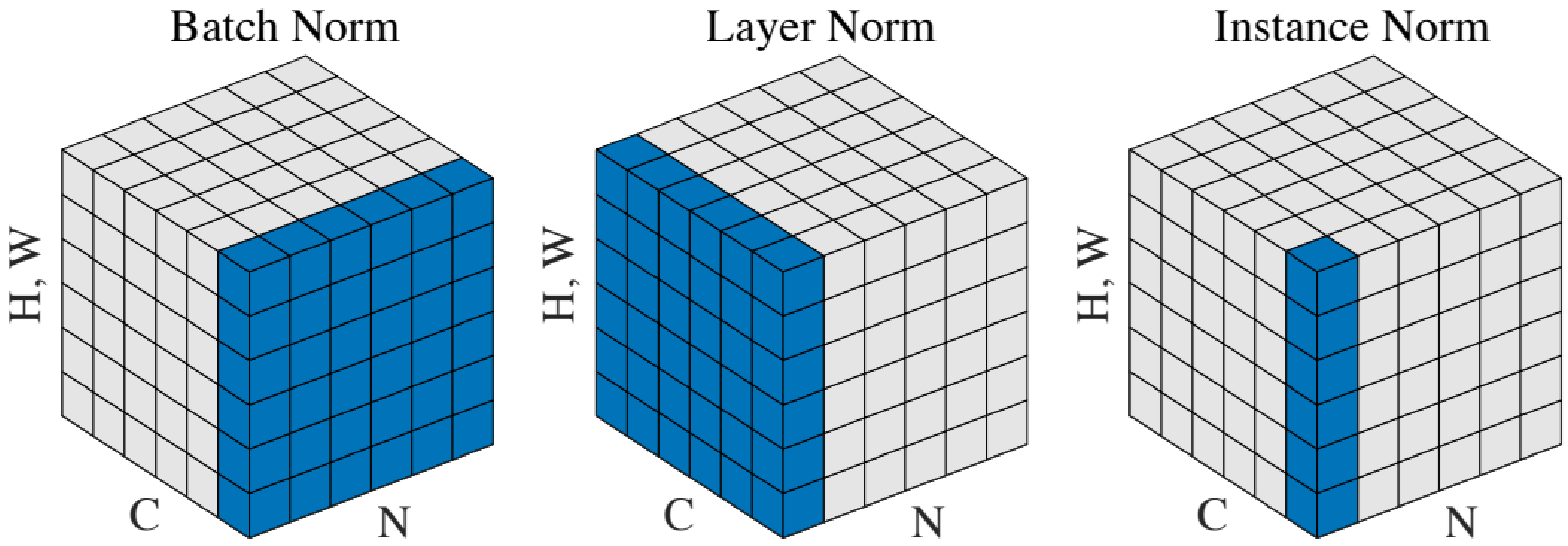
Normalize

$$\mu, \sigma : N \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

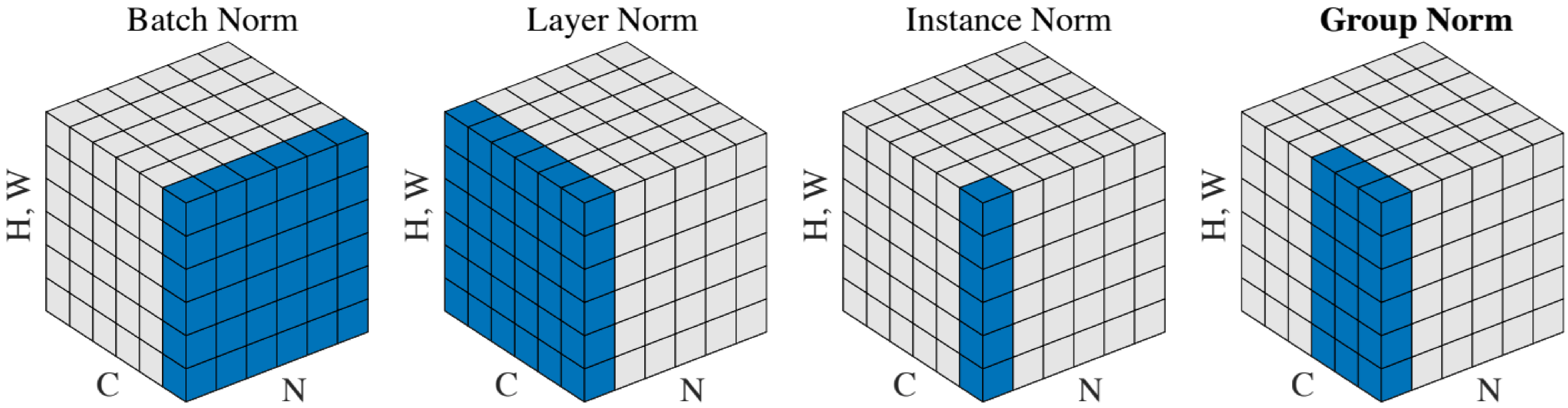
$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Comparison of Normalization Layers



Wu and He, “Group Normalization”, ECCV 2018

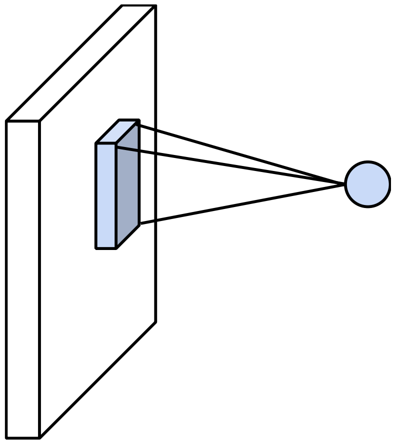
Group Normalization



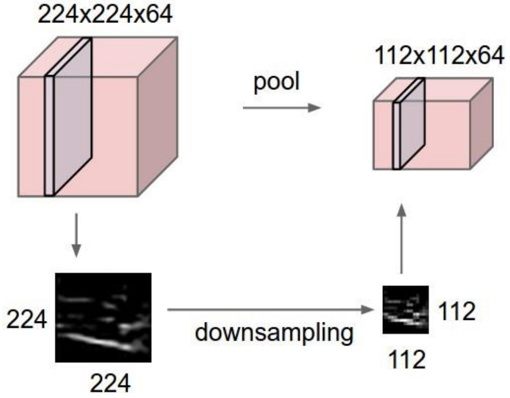
Wu and He, “Group Normalization”, ECCV 2018

Summary: Components of a Convolutional Network

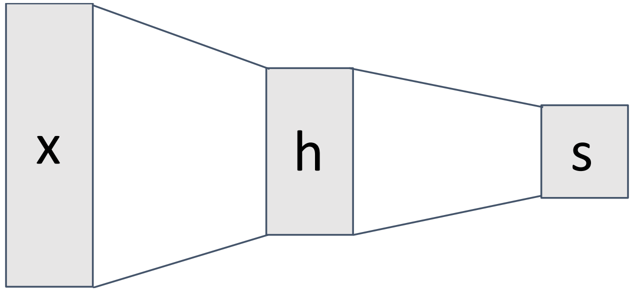
Convolution Layers



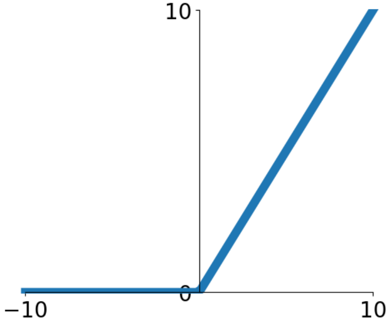
Pooling Layers



Fully-Connected Layers



Activation Function

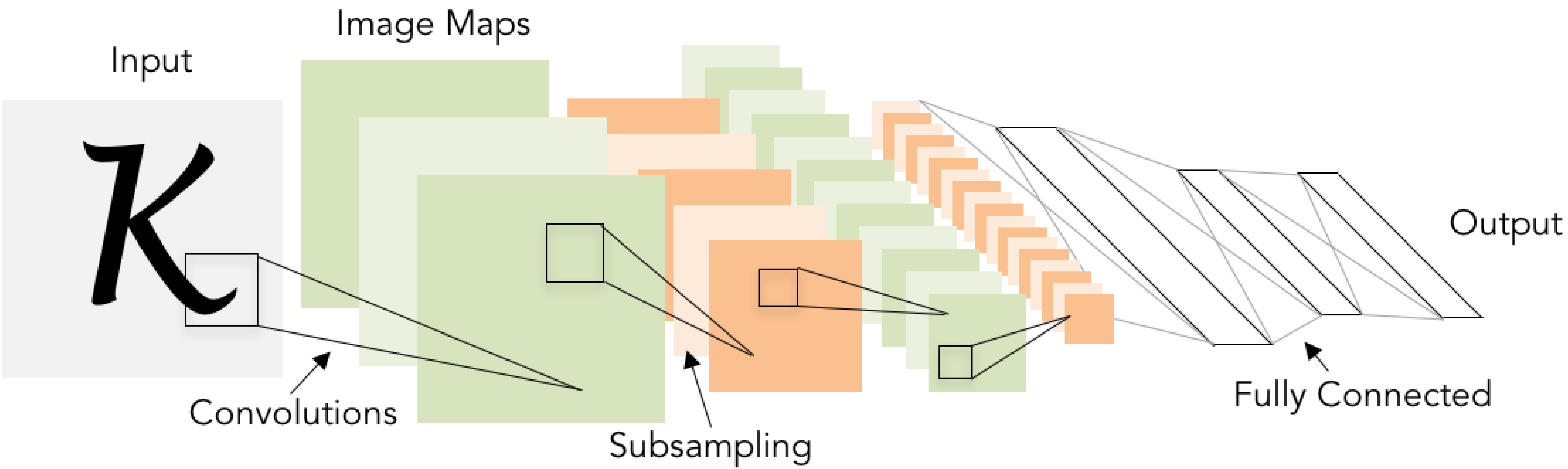


Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Summary: Components of a Convolutional Network

Problem: What is the right way to combine all these components?



Next: CNN Architectures