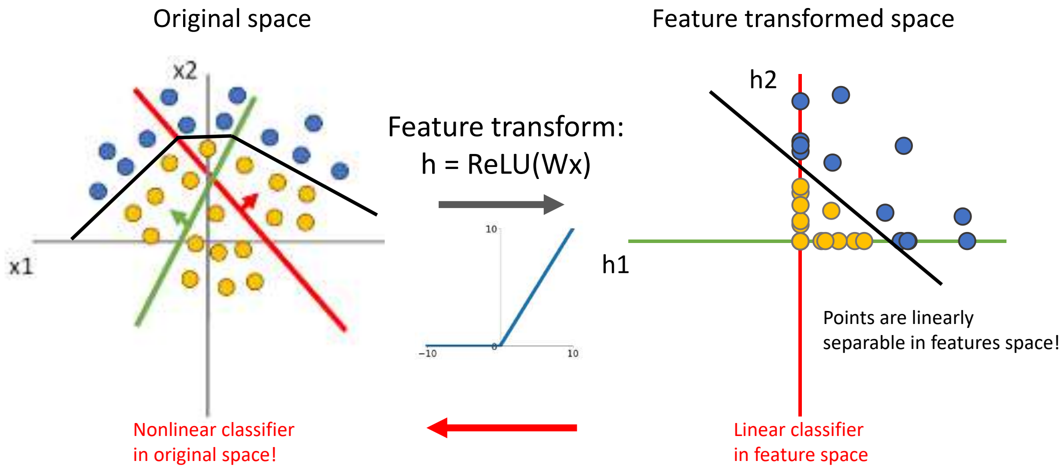
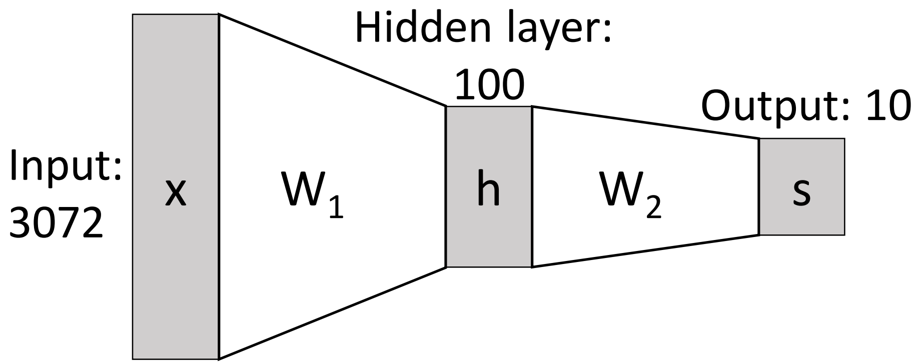
Convolutional Networks

Notes based on
CS231n, Stanford University, and
EECS 498-007 / 598-005, University of Michigan

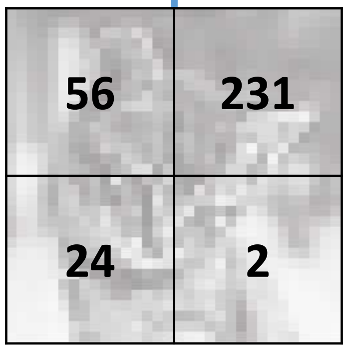
$s = f(x,W) = Wx$ $\qquad h = \mathrm{ReLU}(Wx)$



$$s = W_2 \max(0, W_1 x)$$



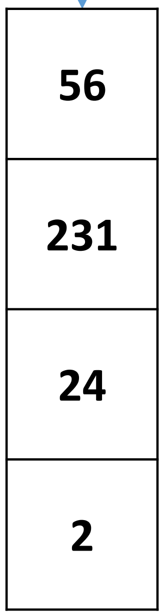
Stretch pixels into a column



Input image
(2, 2)

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

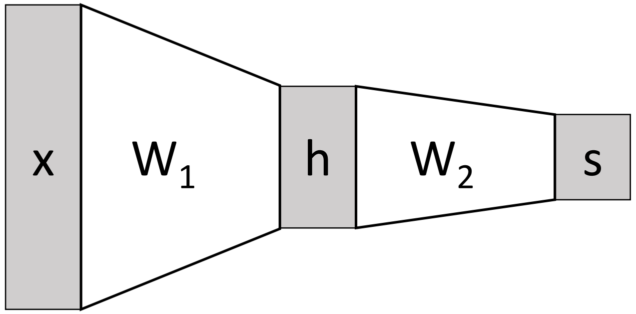
Solution: Define new computational nodes that operate on images!



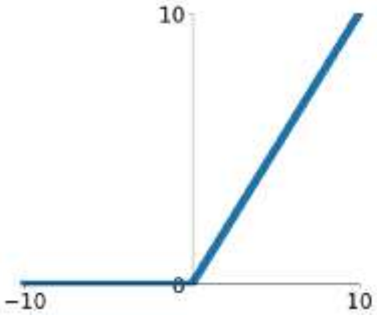
Input
(4, 1)

Components of a Full-Connected Network

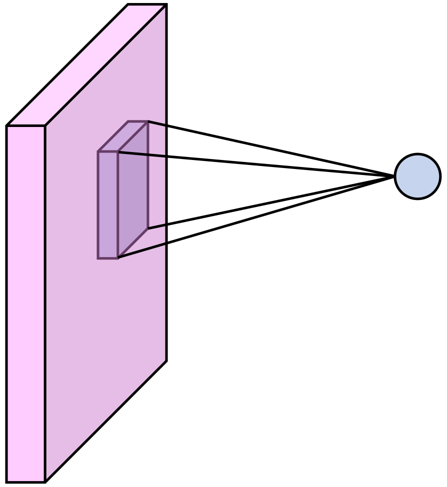
Fully-Connected Layers



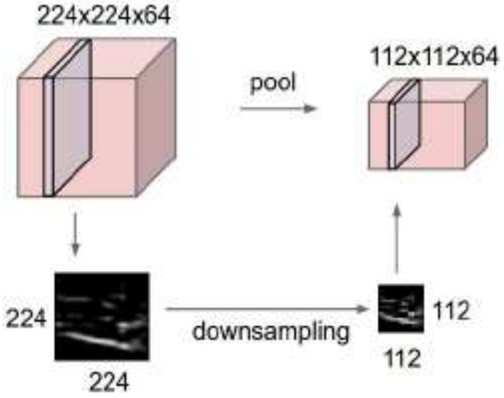
Activation Function



Convolution Layers



Pooling Layers

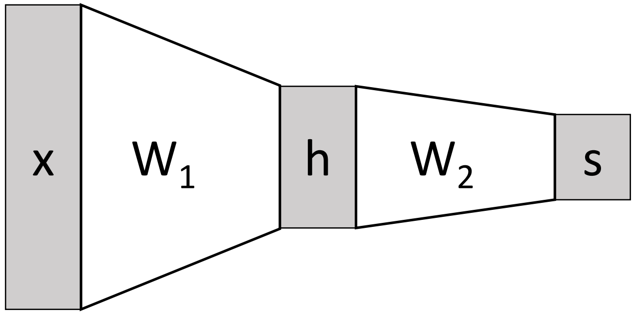


Normalization

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

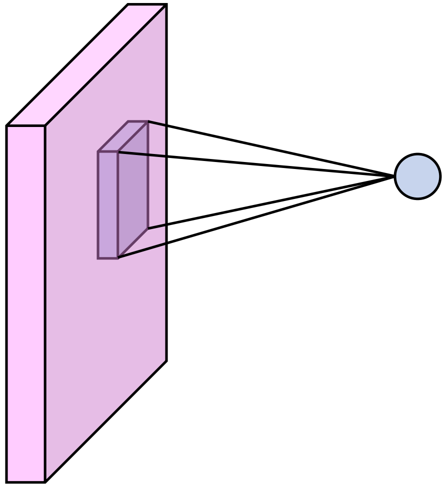
Components of a Full-Connected Network

Fully-Connected Layers

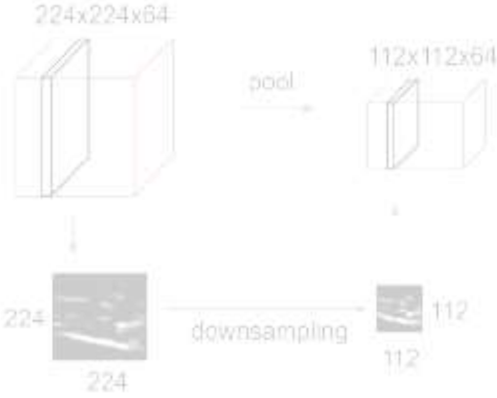


Activation Function

Convolution Layers



Pooling Layers

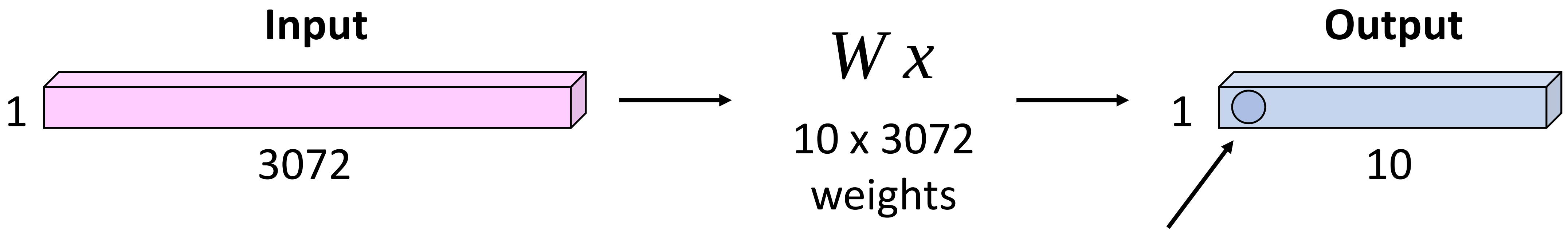


Normalization

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

Fully-Connected Layer

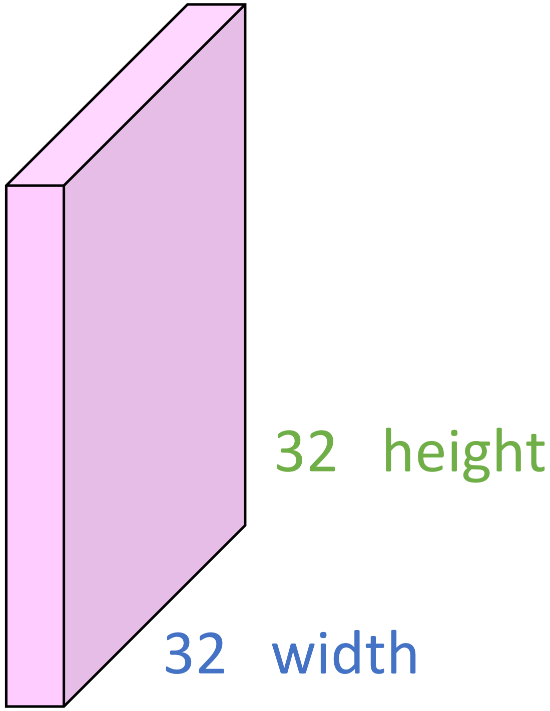
32x32x3 image -> stretch to 3072 x 1



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

Convolution Layer

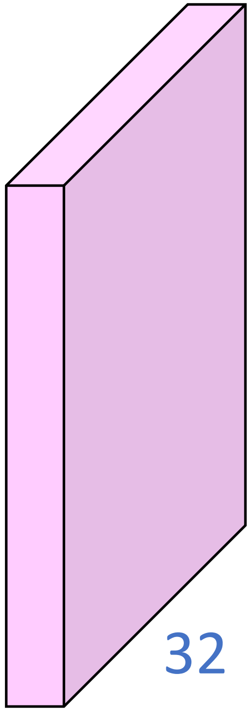
3x32x32 image: preserve spatial structure



Convolution Layer

Filters always extend the full depth of the input volume

3x32x32 image

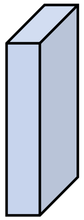


32 height

32 width

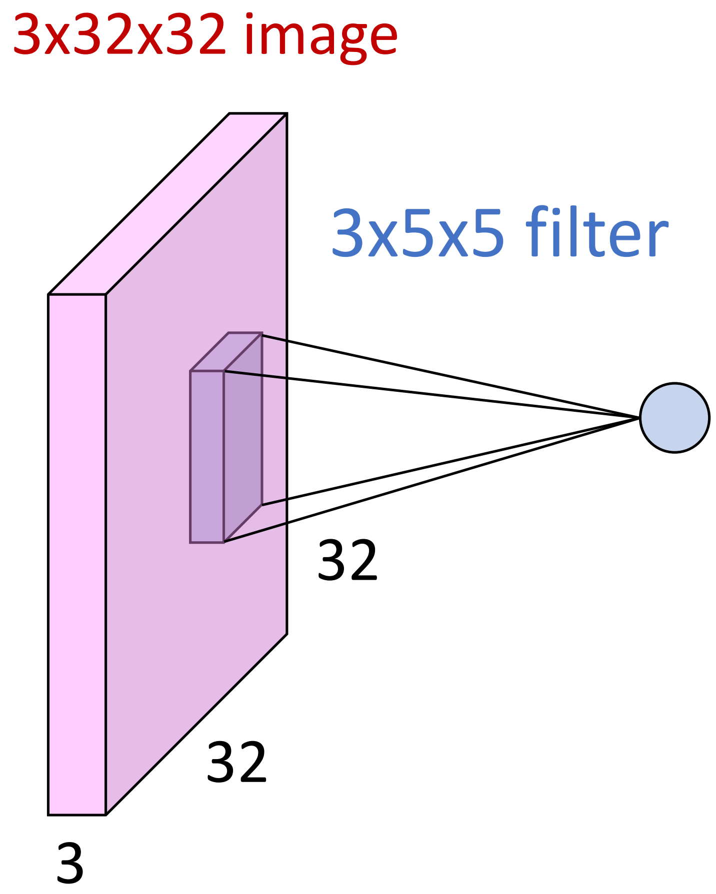
3 depth / channels

3x5x5 filter



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

Convolution Layer

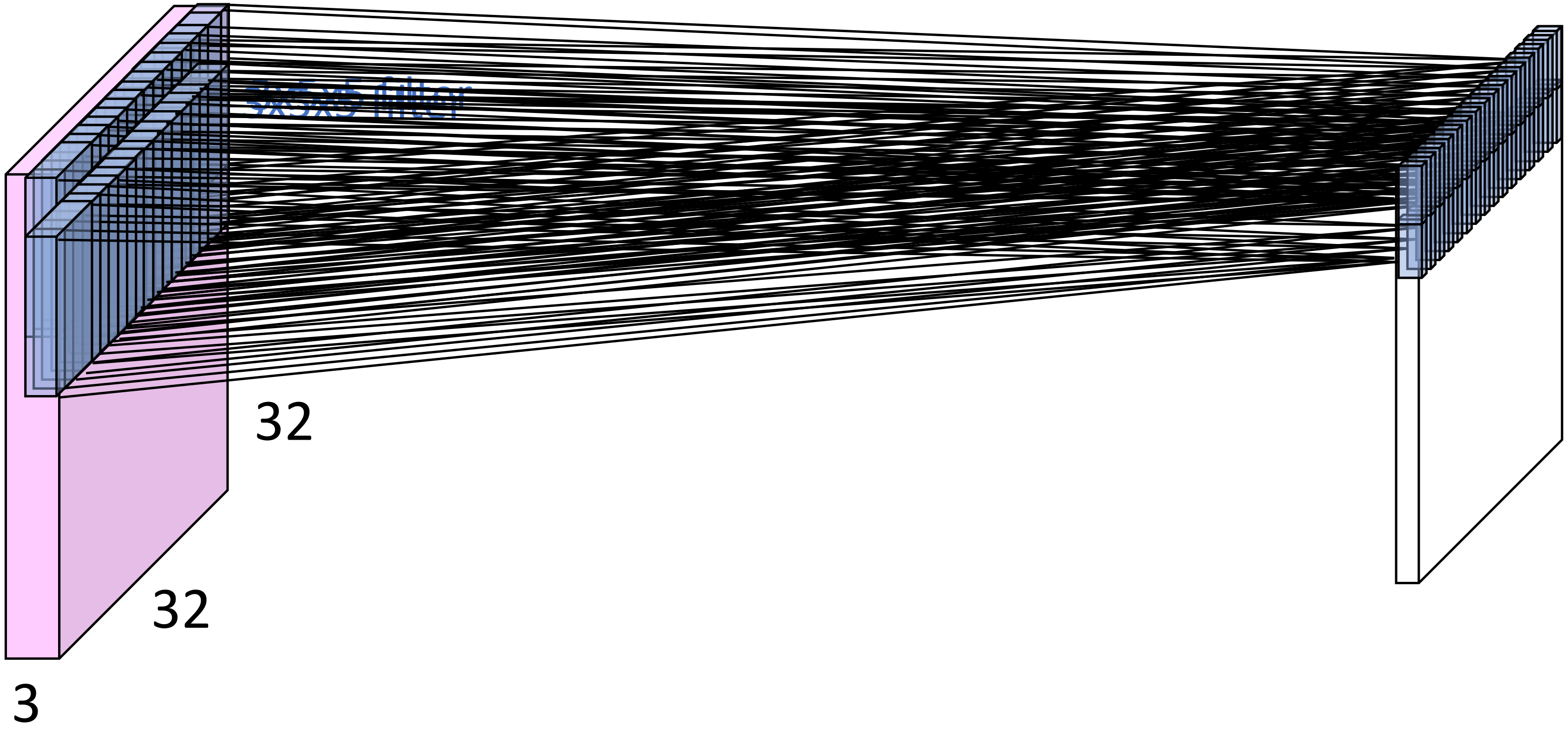


1 number:
the result of taking a dot product between the filter and a small 3x5x5 chunk of the image
(i.e. 3*5*5 = 75-dimensional dot product + bias)

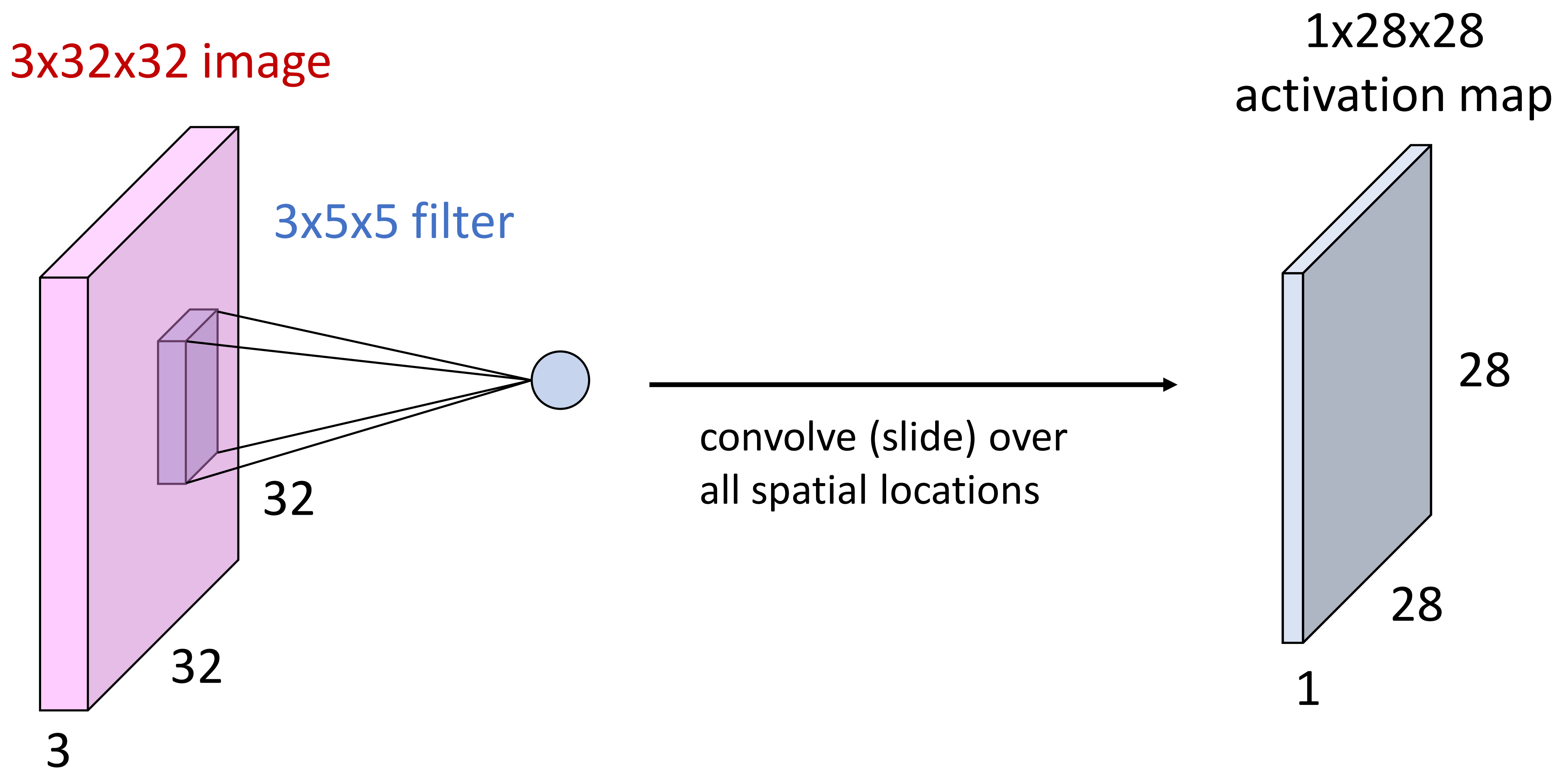
$$w^T x + b$$

Convolution Layer

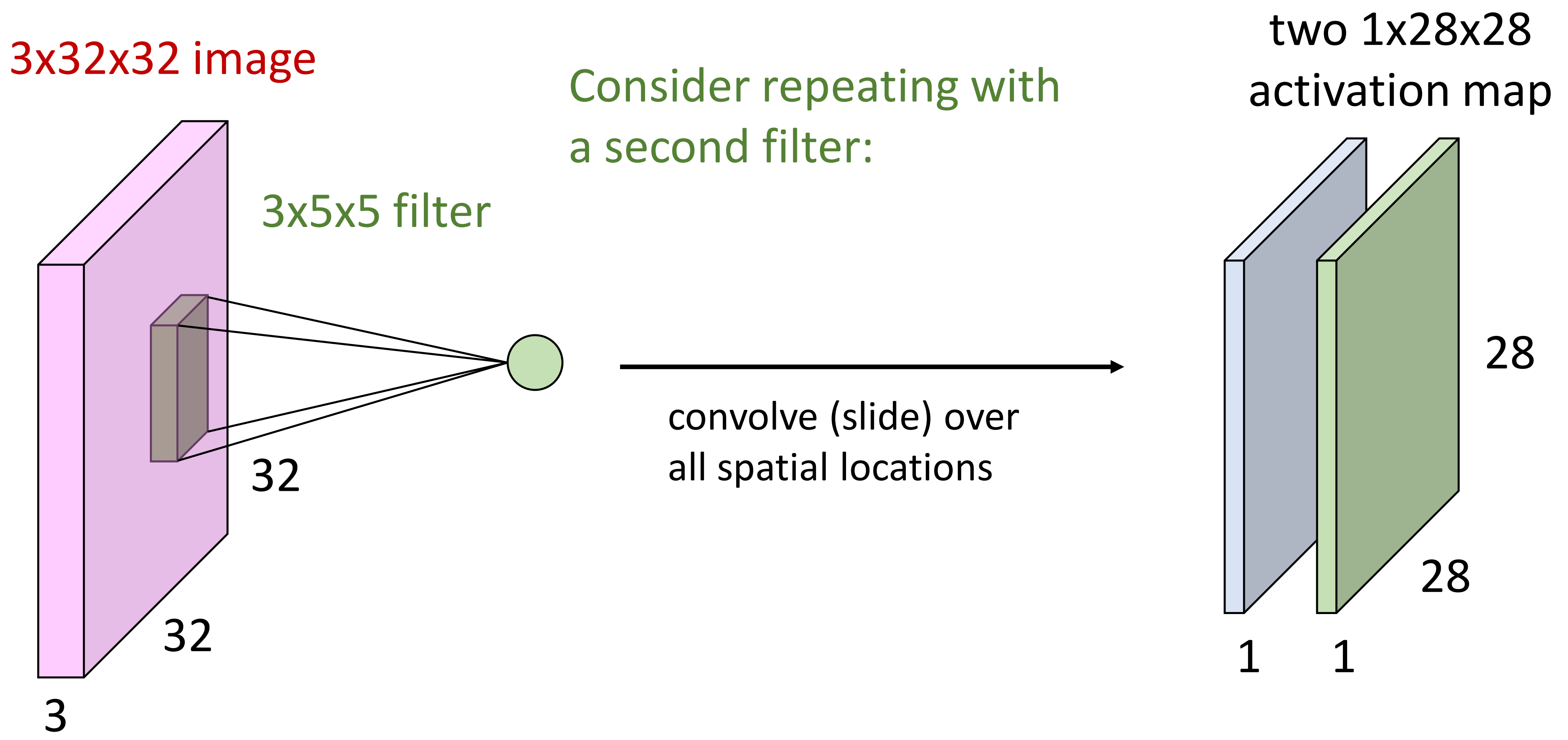
3x32x32 image



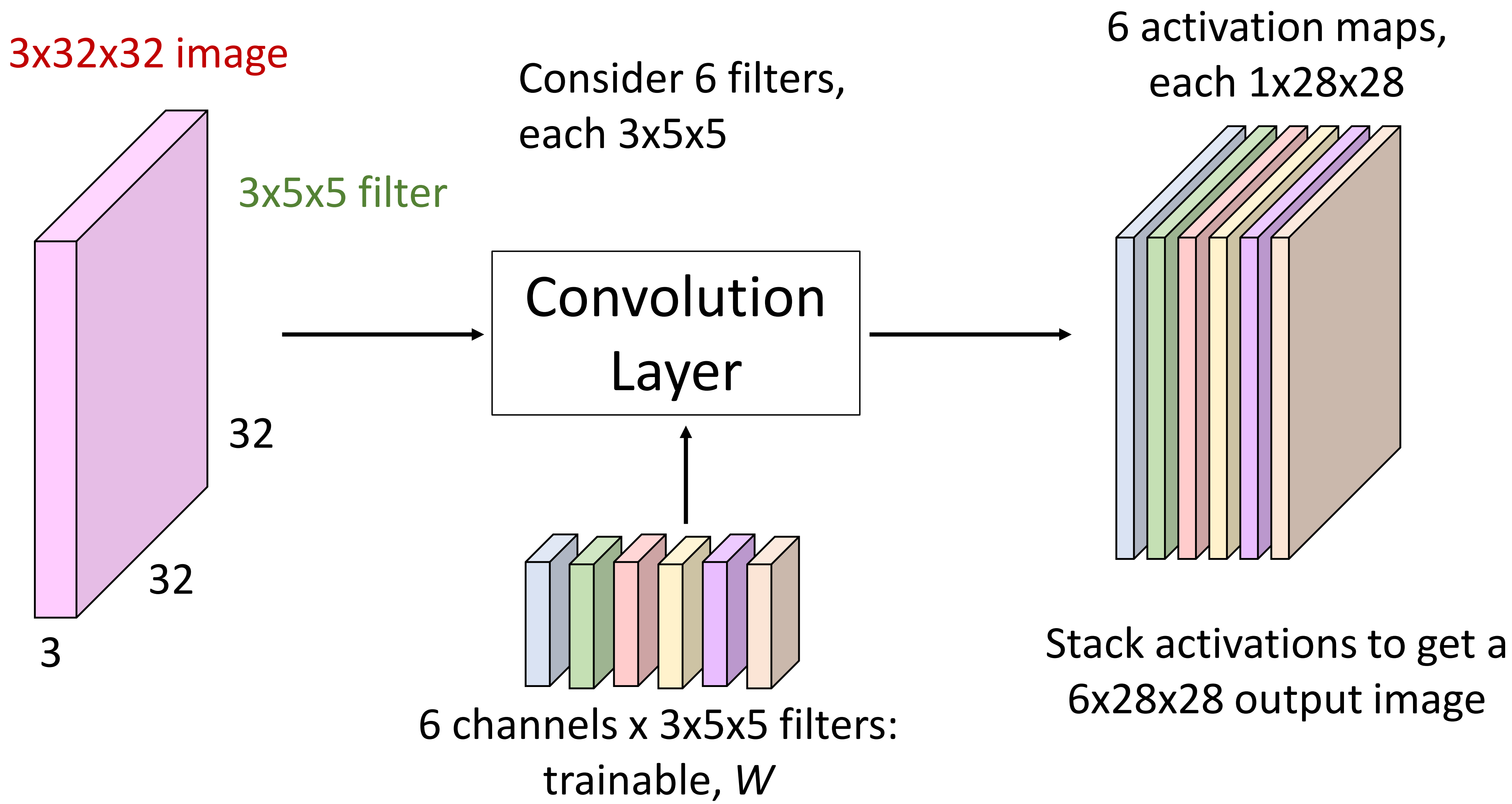
Convolution Layer



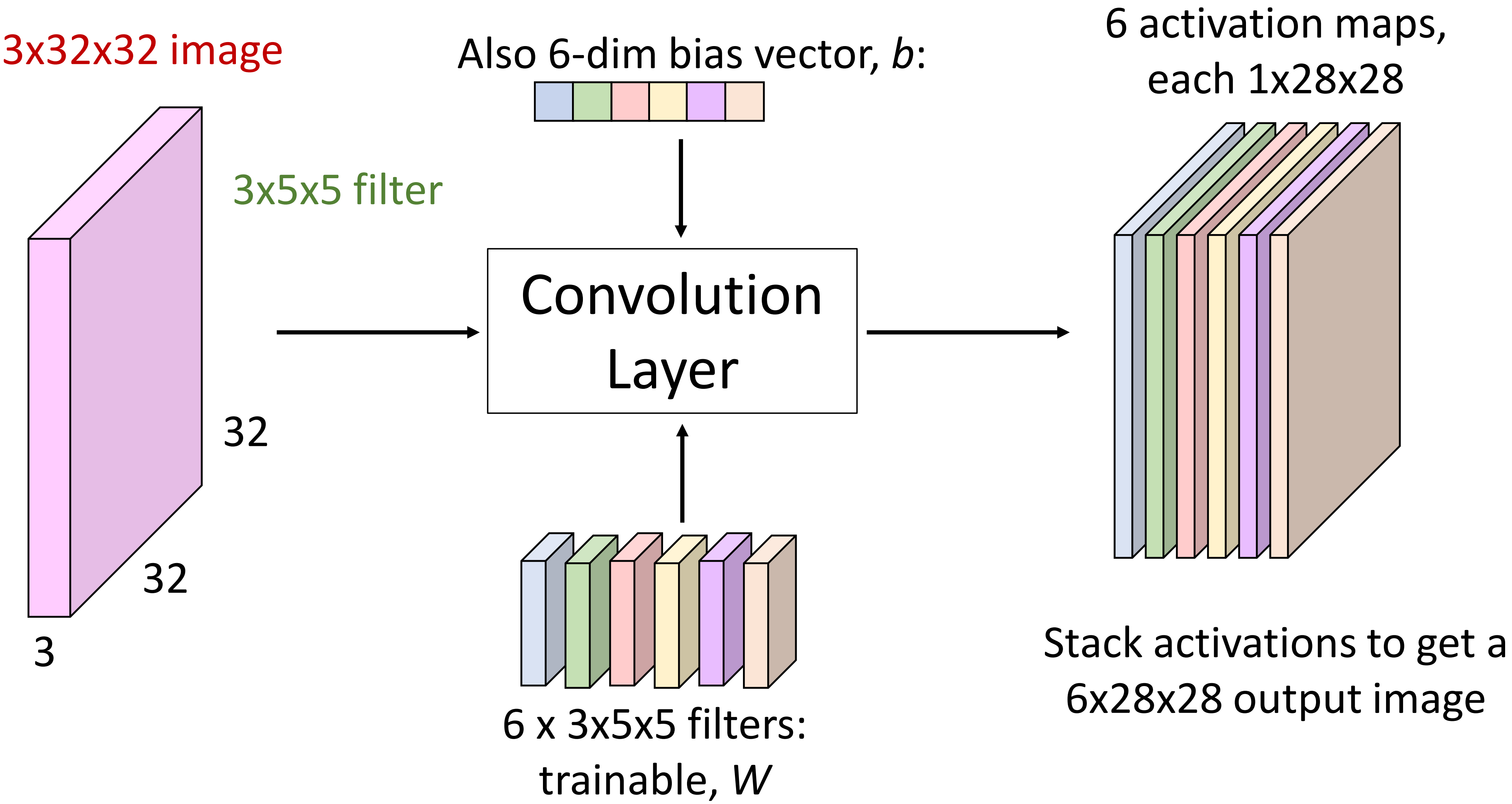
Convolution Layer



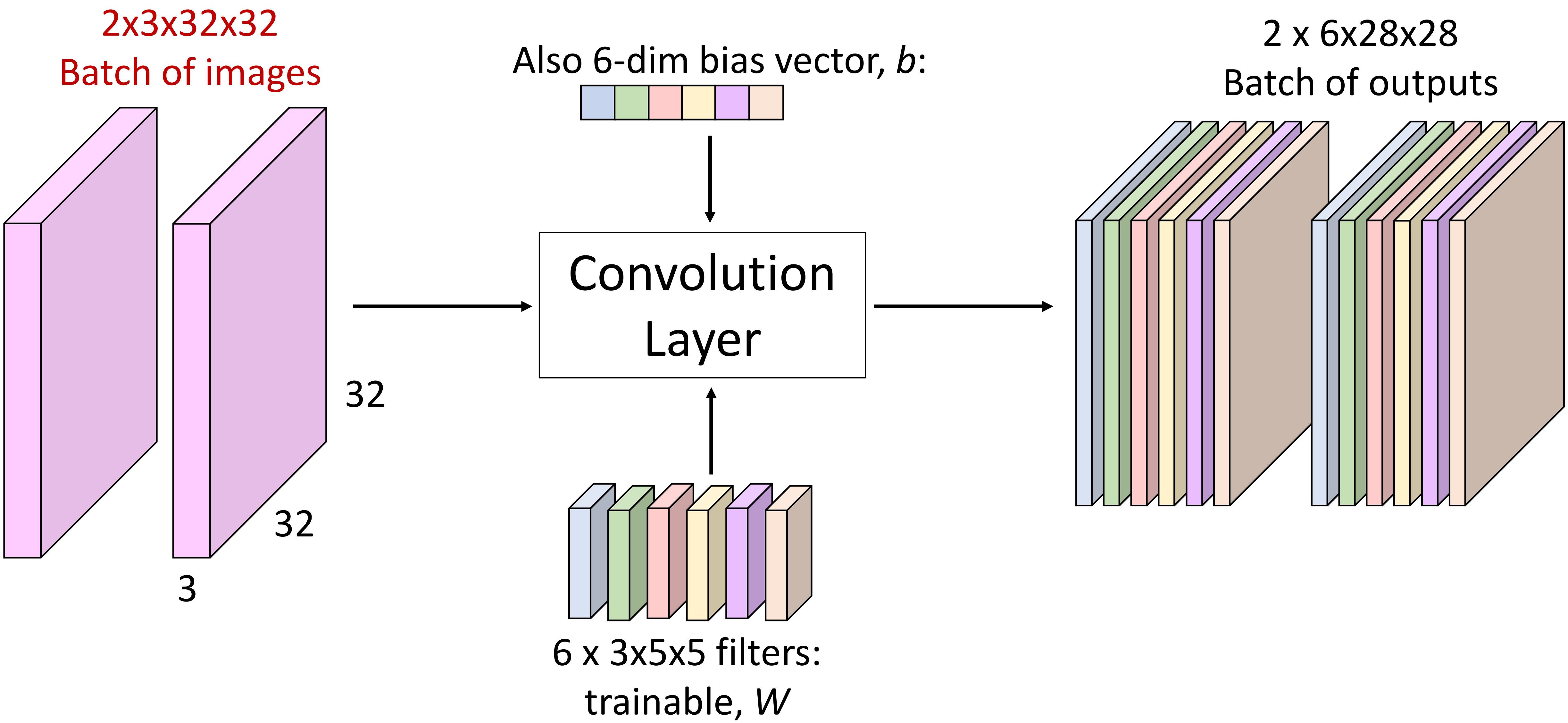
Convolution Layer



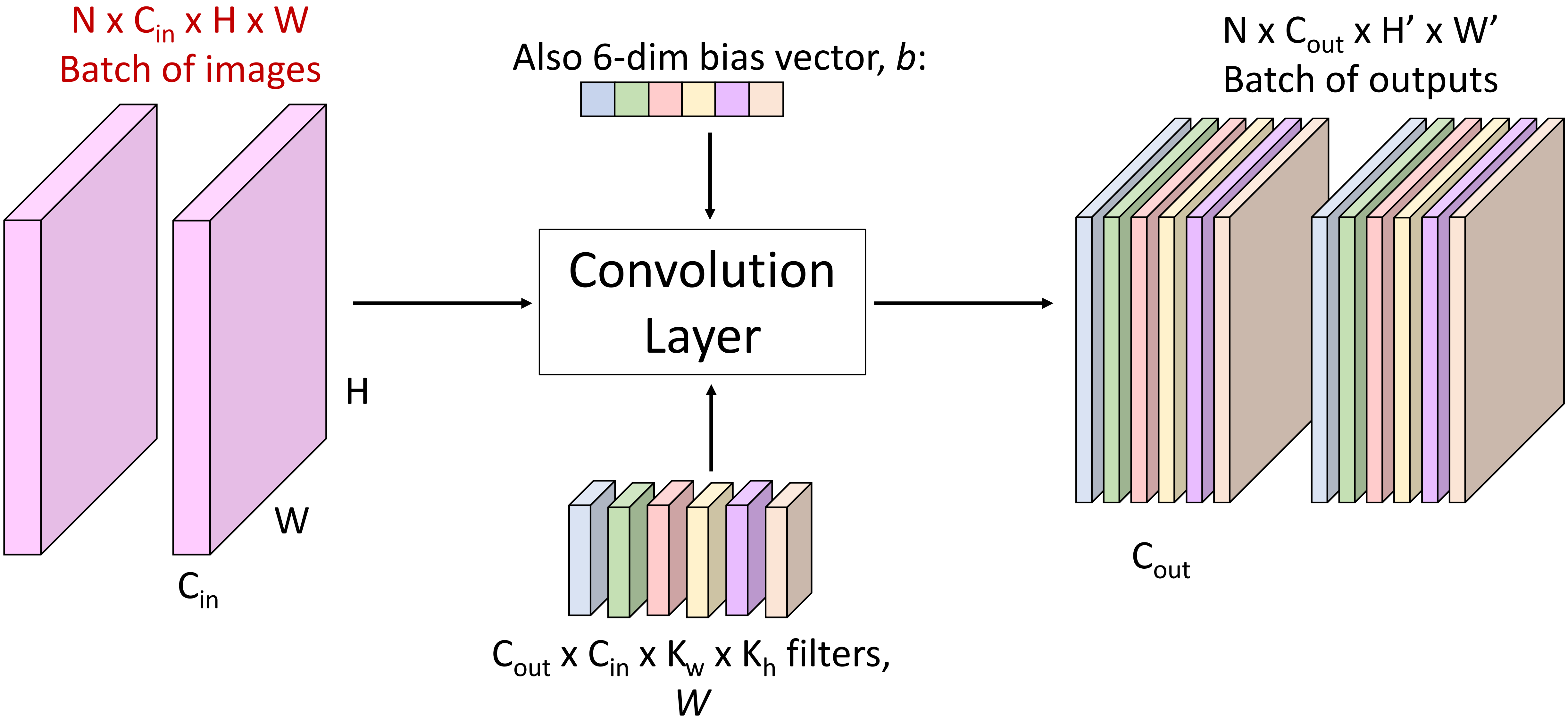
Convolution Layer



Convolution Layer



Convolution Layer

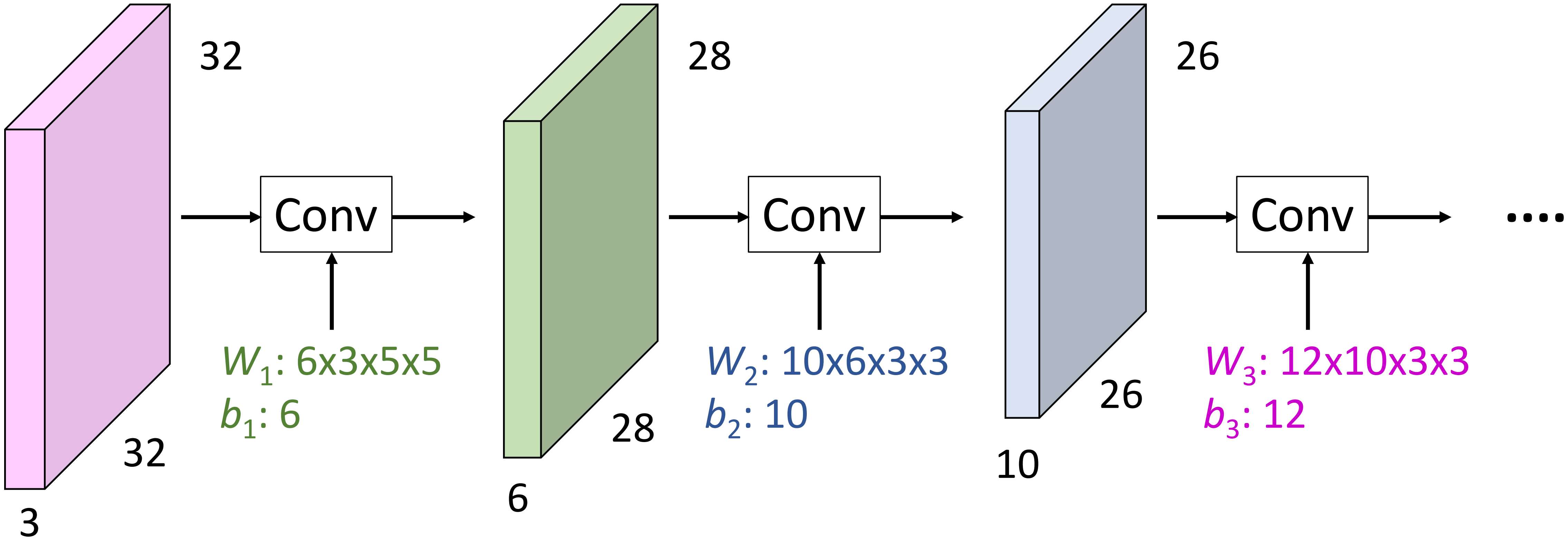


Stacking Convolutions

Q: What happens if we stack two convolution layers?

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

A: We get another convolution!



Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

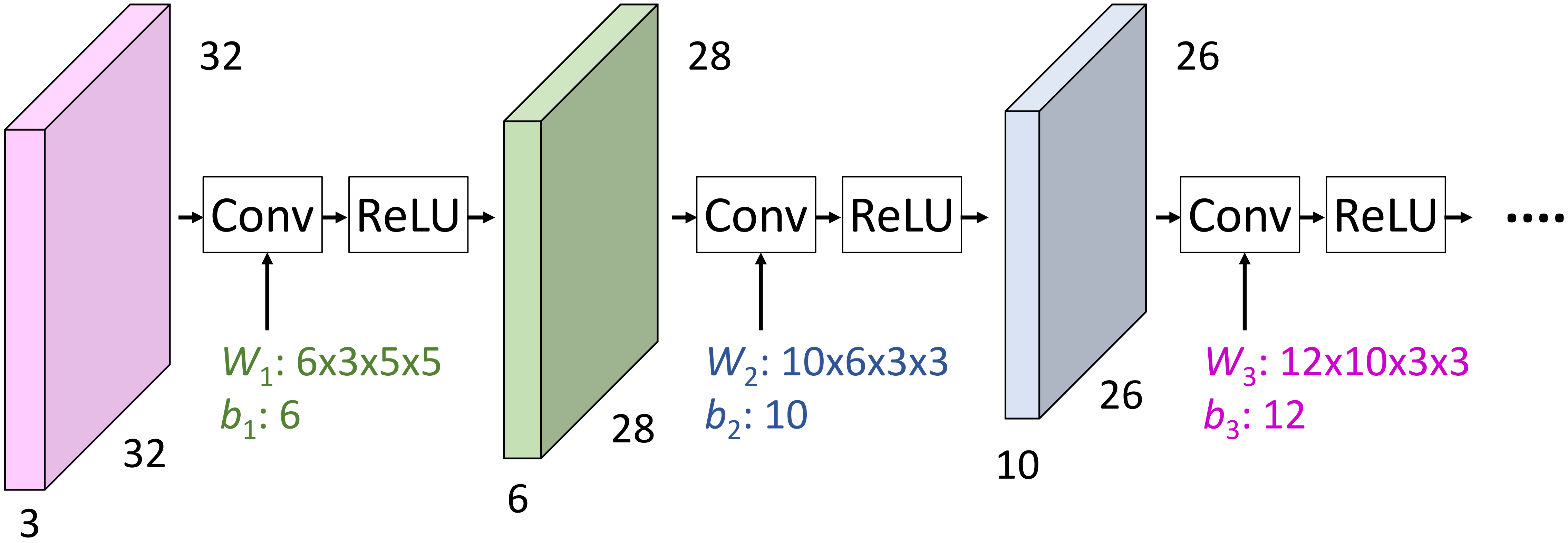
Second hidden layer:
N x 10 x 26 x 26

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!

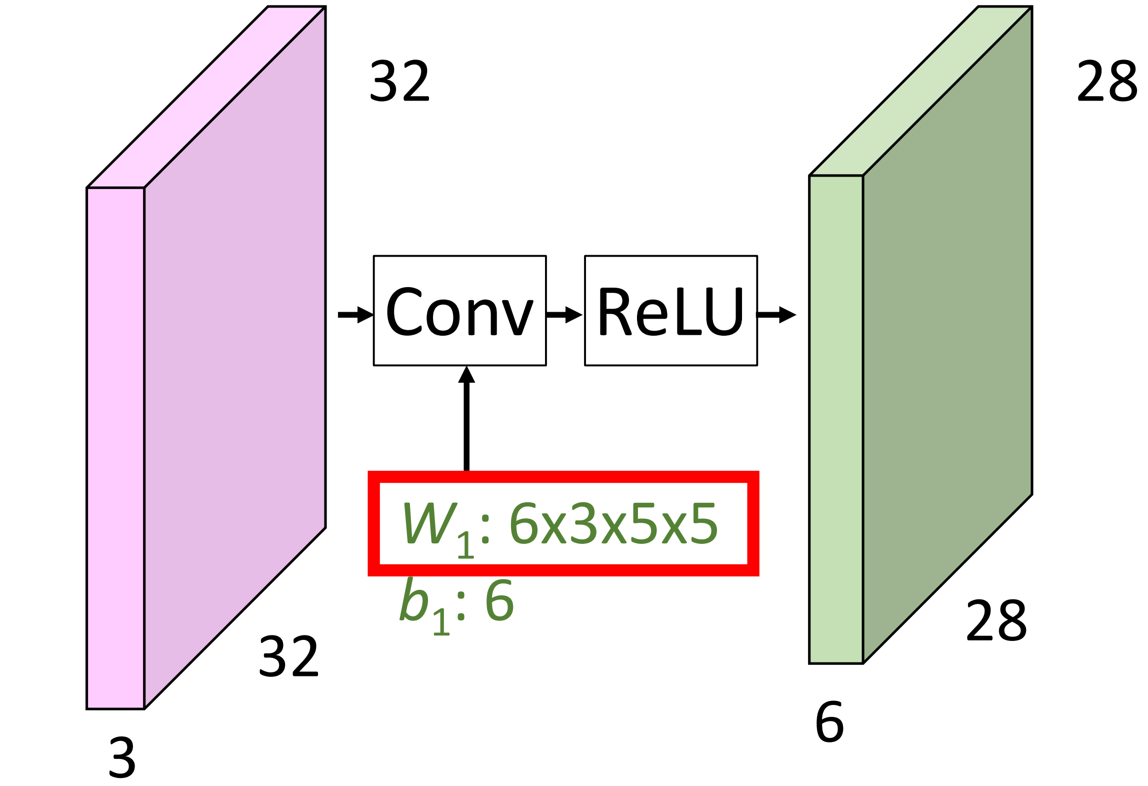


Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

Second hidden layer:
N x 10 x 26 x 26

What do convolutional filters learn?

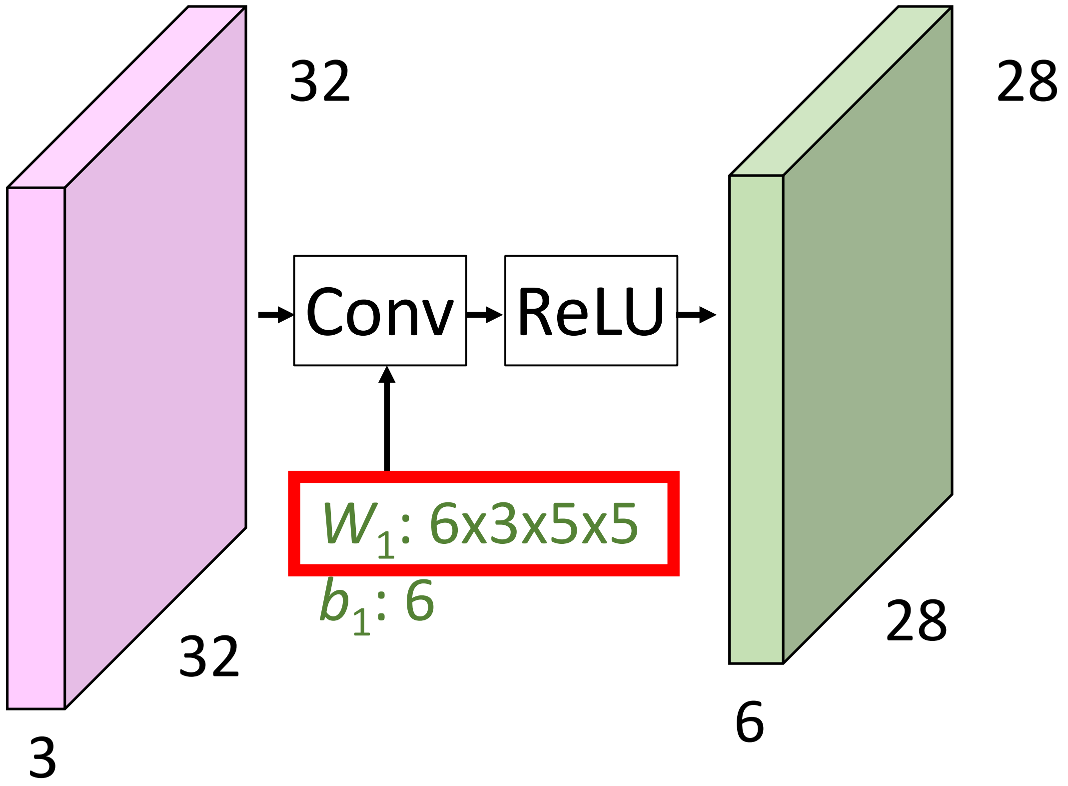


Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

Previously:
Linear classifier: One template per class

What do convolutional filters learn?

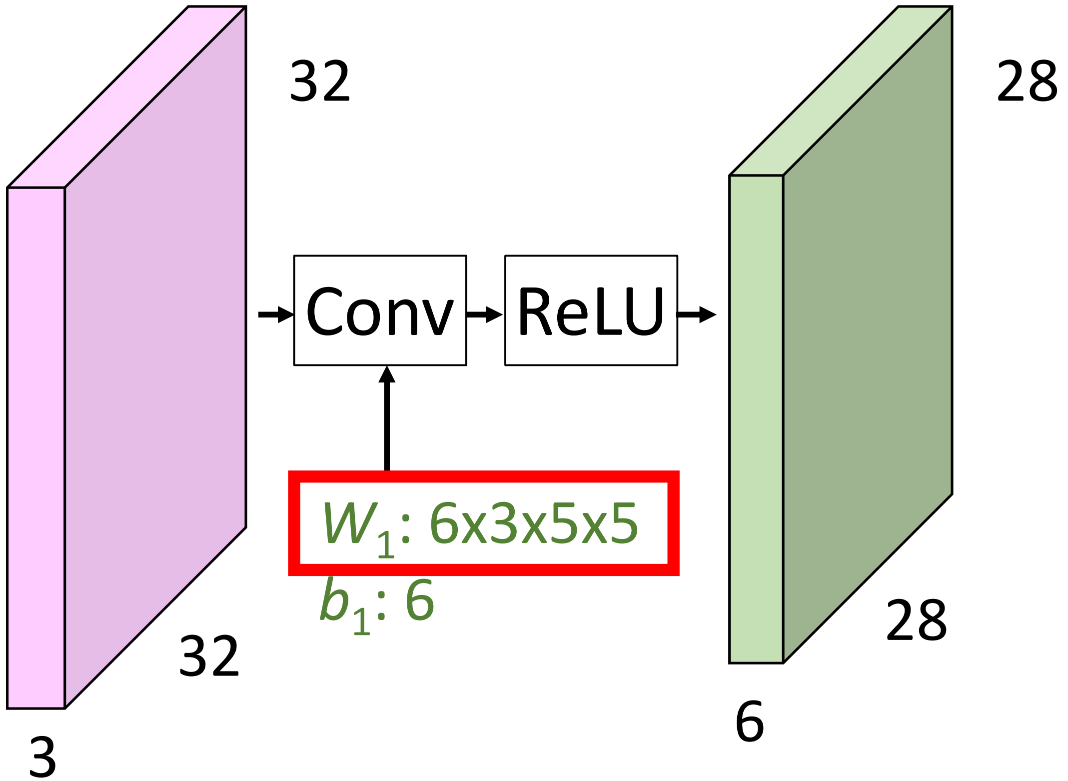


Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

Previously:
MLP: Bank of whole-image templates

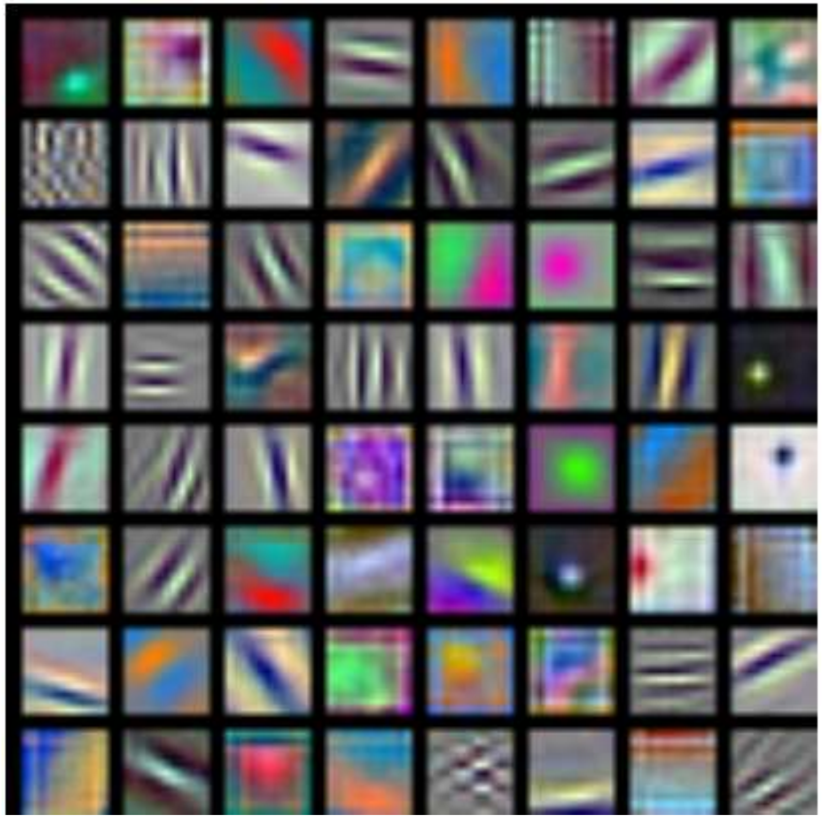
What do convolutional filters learn?



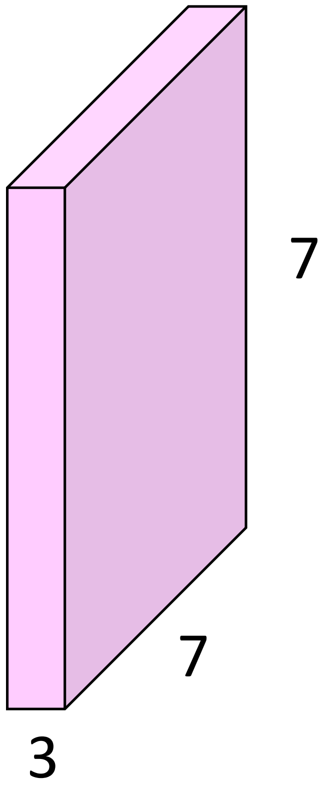
Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

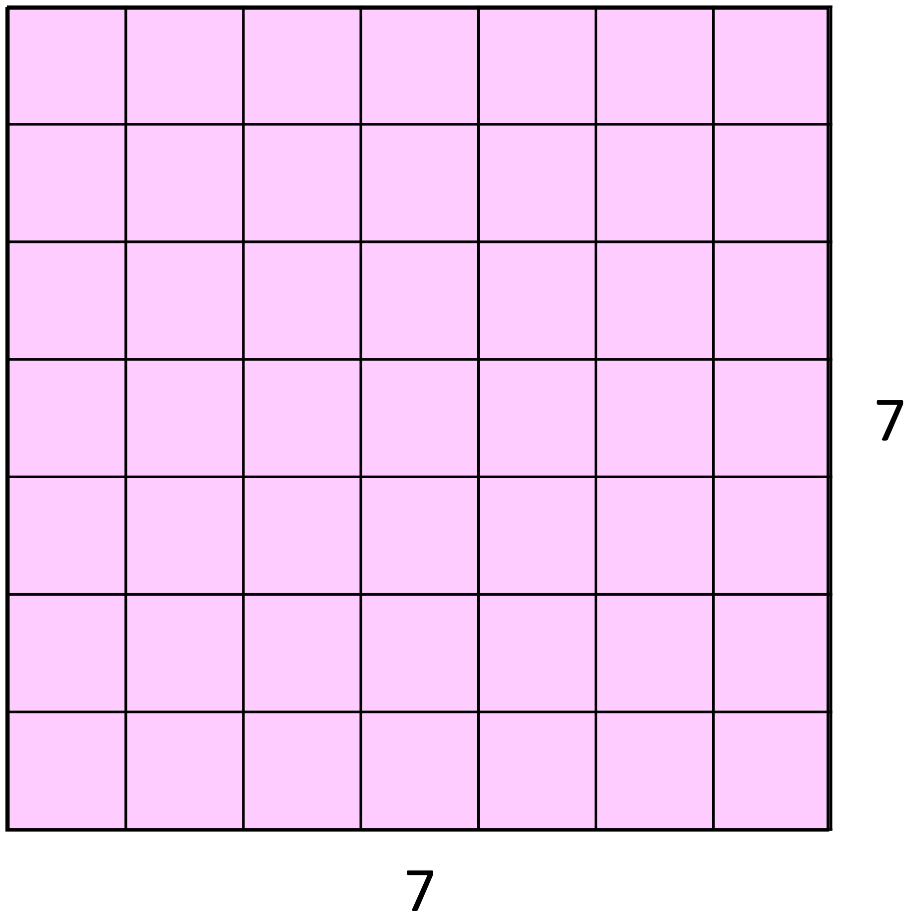
First-layer conv filters: local image templates
(Often learns oriented edges, opposing colors)



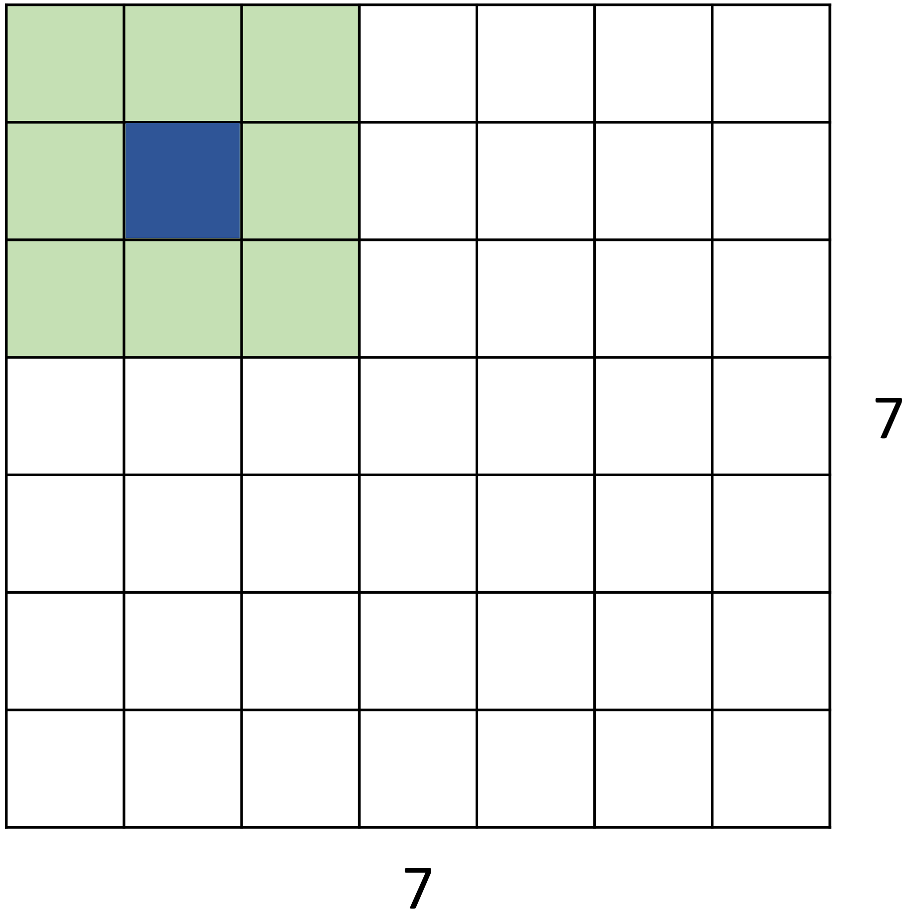
AlexNet: 64 filters, each 3x11x11



Input:
N x 3 x 7 x 7


7
7

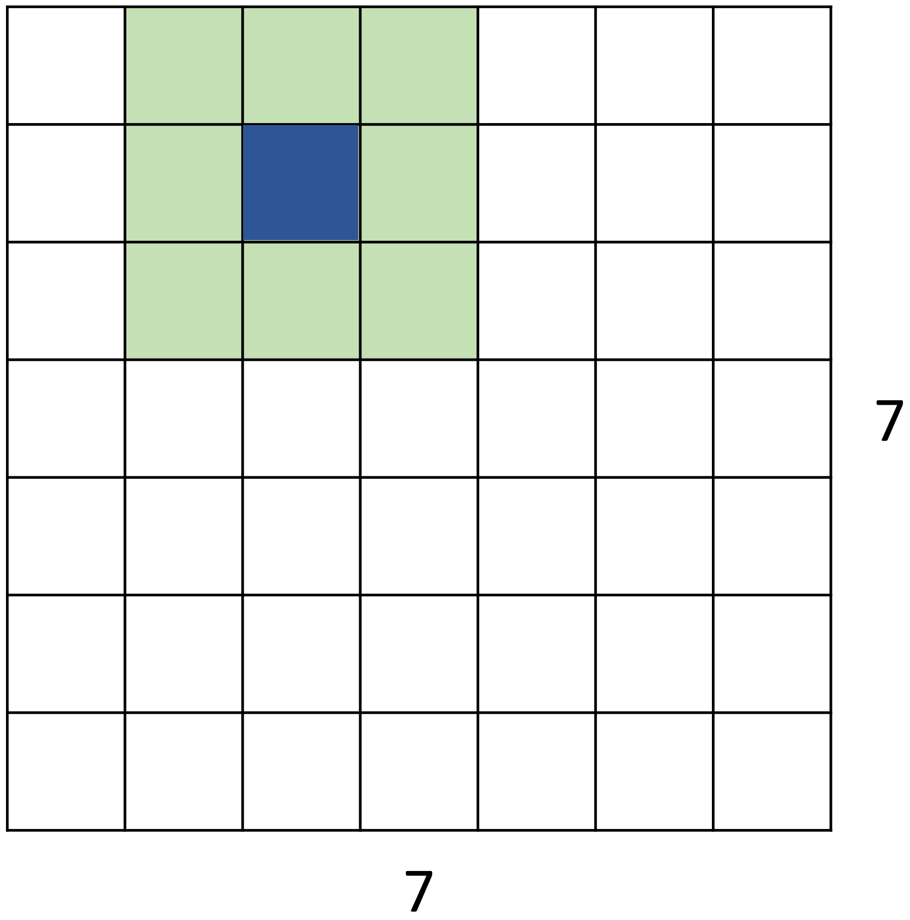
A closer look at spatial dimensions



Input: 7x7

Filter: 3x3

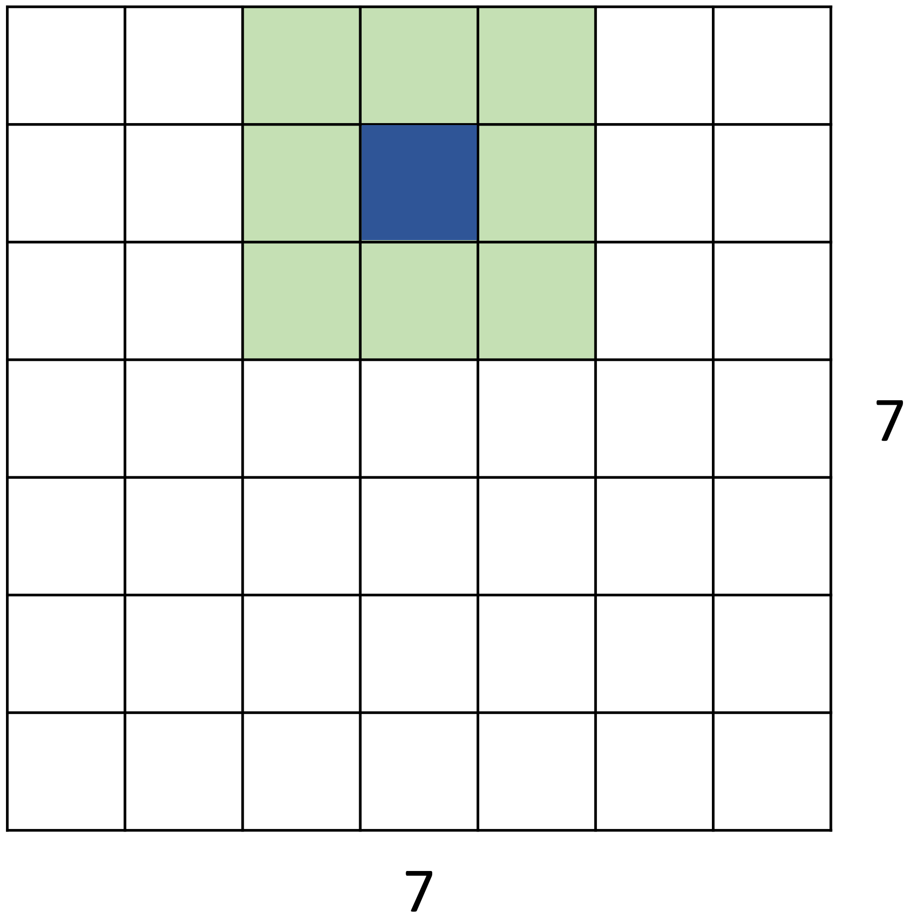
A closer look at spatial dimensions



Input: 7x7

Filter: 3x3

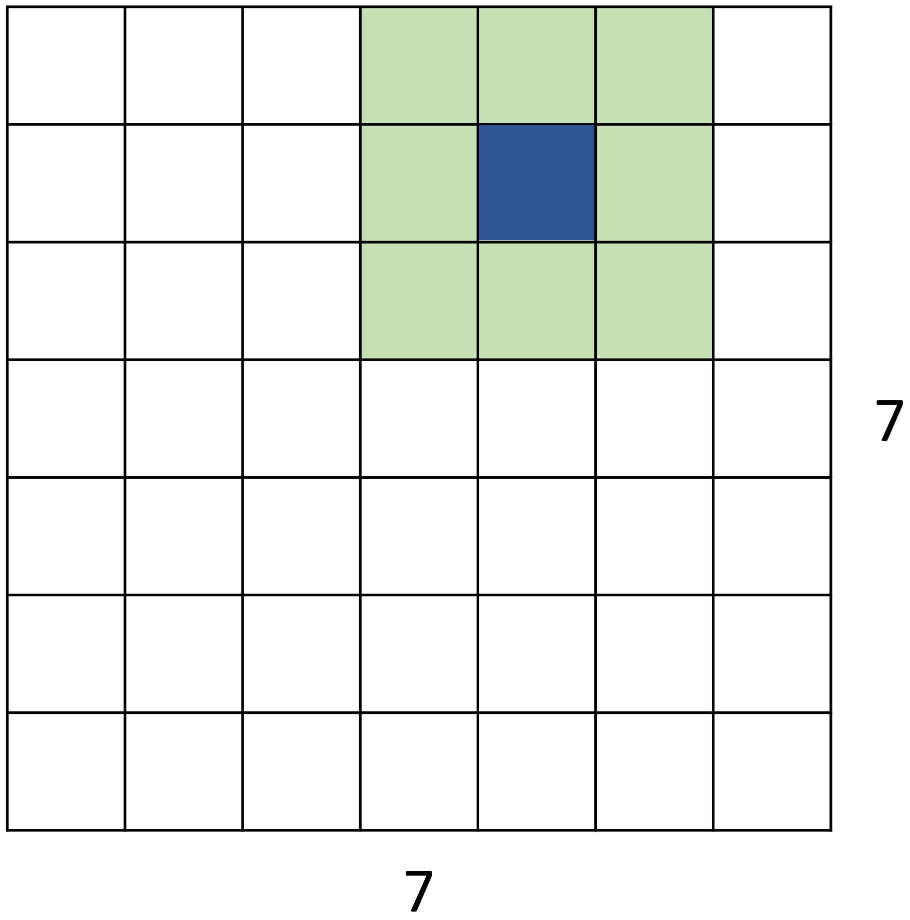
A closer look at spatial dimensions



Input: 7x7

Filter: 3x3

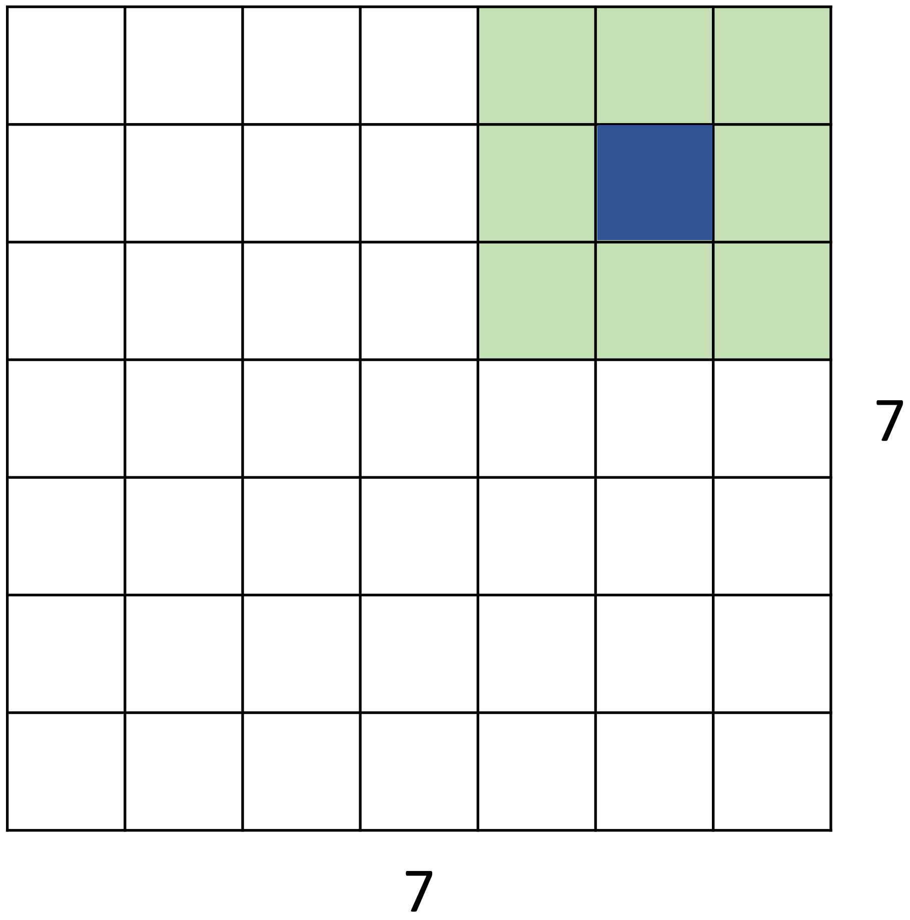
A closer look at spatial dimensions



Input: 7x7

Filter: 3x3

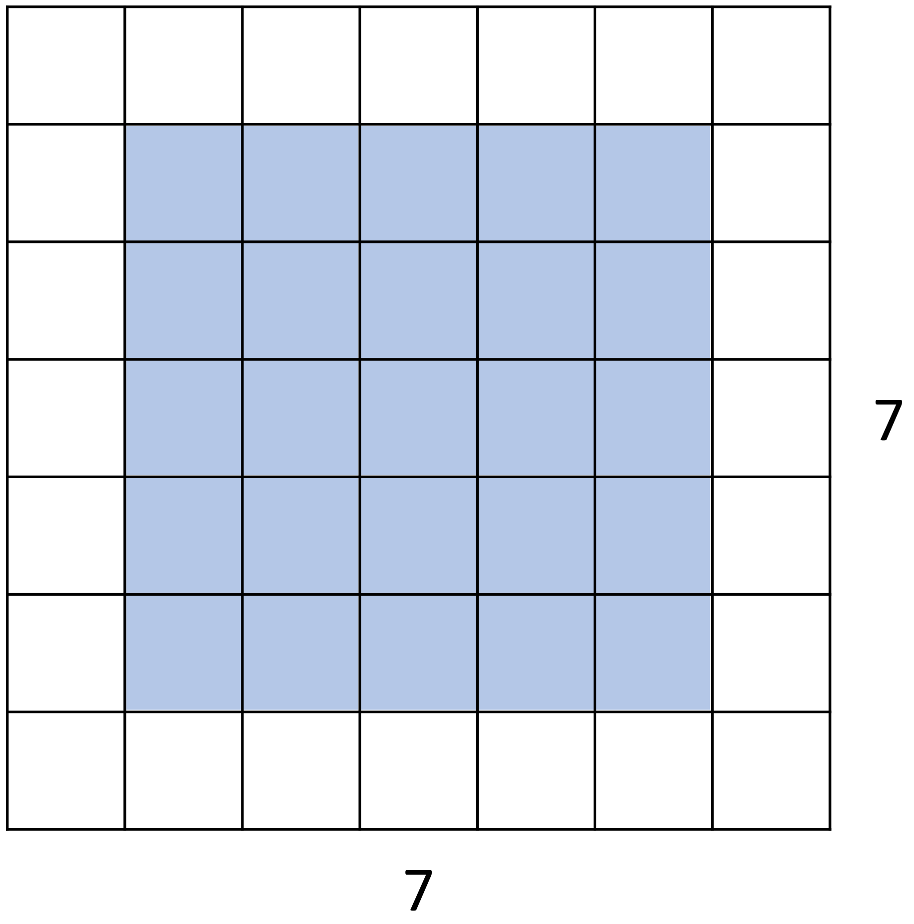
A closer look at spatial dimensions



Input: 7x7

Filter: 3x3

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

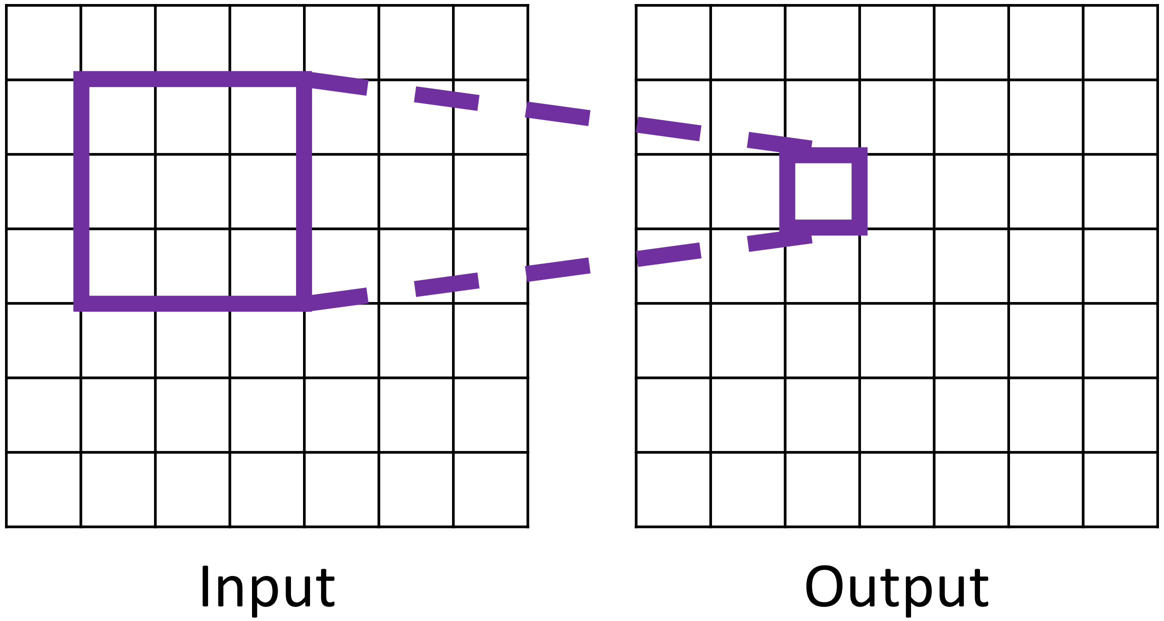
Input: 7x7
Filter: 3x3
Output: 5x5

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

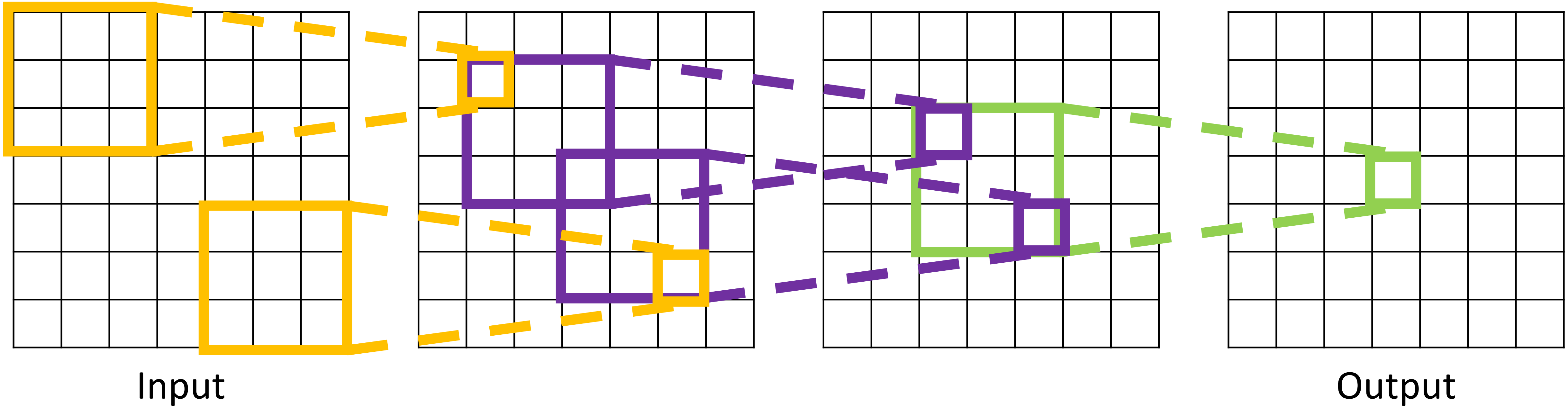
For convolution with kernel size K, each element in the output depends on a K x K **receptive field** in the input



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)

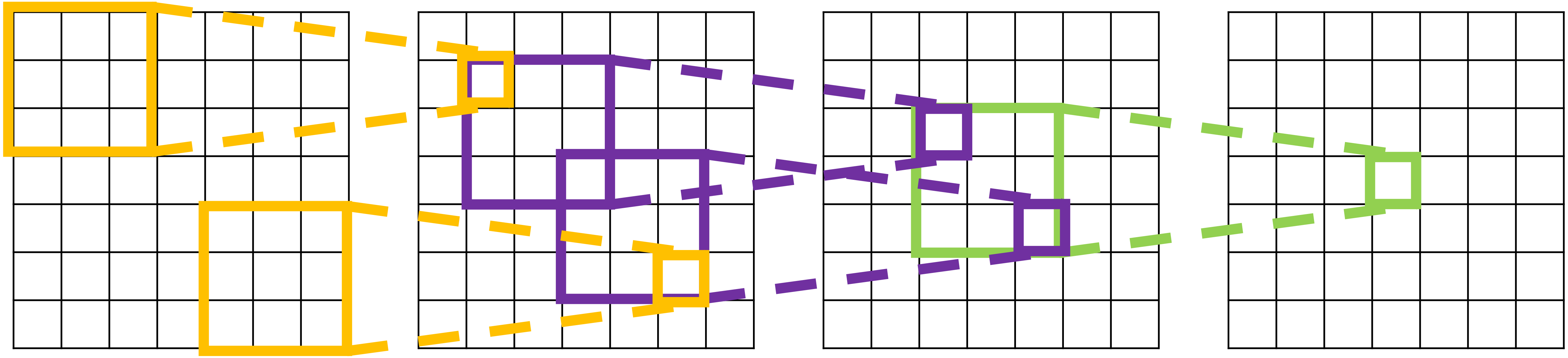


Be careful – ”receptive field in the input” vs “receptive field in the previous layer”
Hopefully clear from context

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)



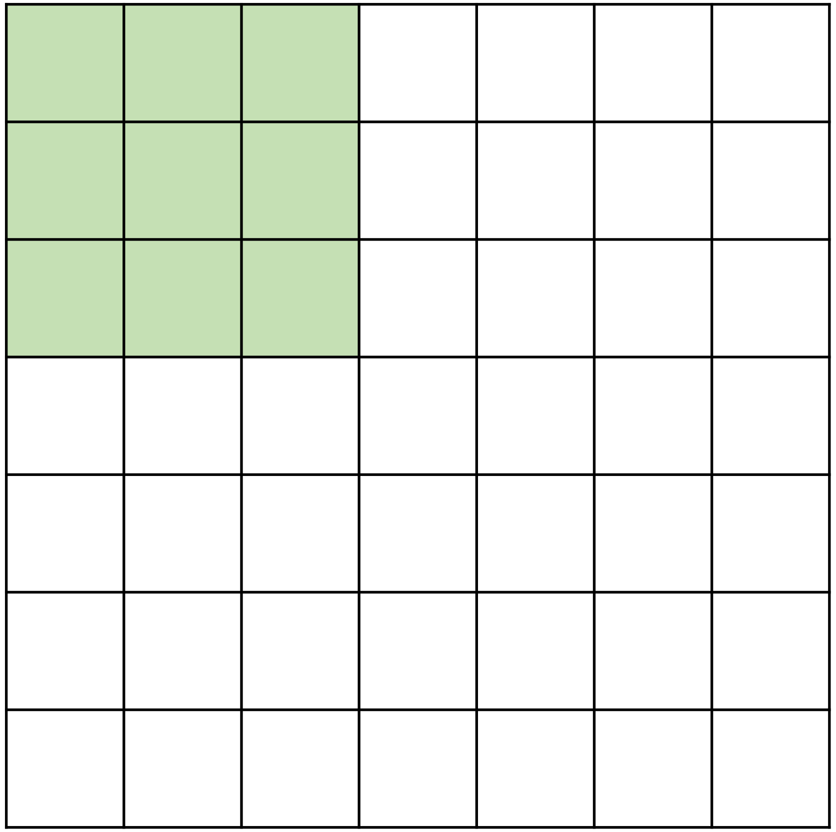
Input

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

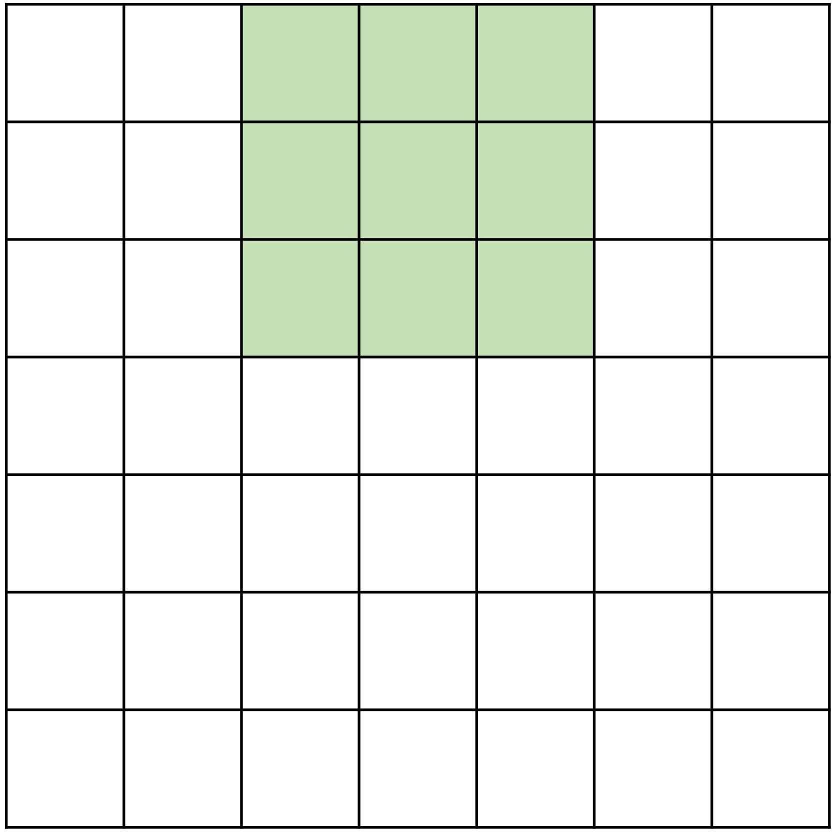


Input: 7x7

Filter: 3x3

Stride: 2

Strided Convolution

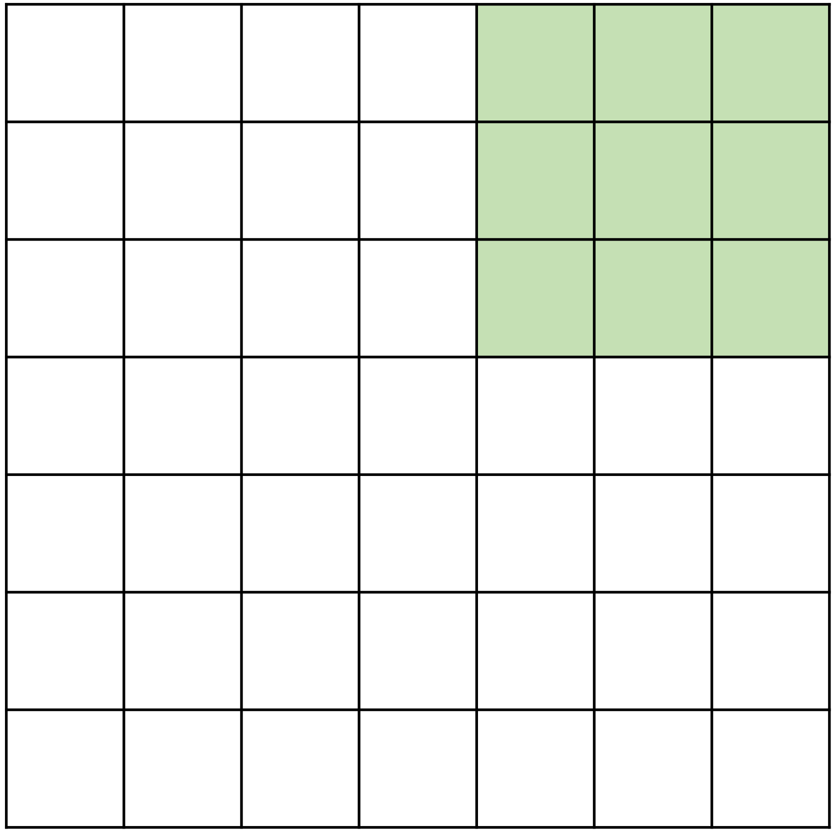


Input: 7x7

Filter: 3x3

Stride: 2

Strided Convolution



Input: 7x7

Filter: 3x3

Stride: 2

Output: 3x3

In general:

Input: W

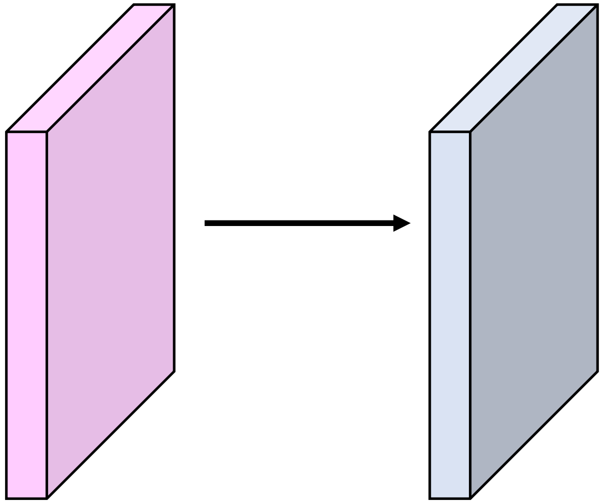
Filter: K

Padding: P

Stride: S

Output: $[\ (W - K + 2P) / S\ ] + 1$

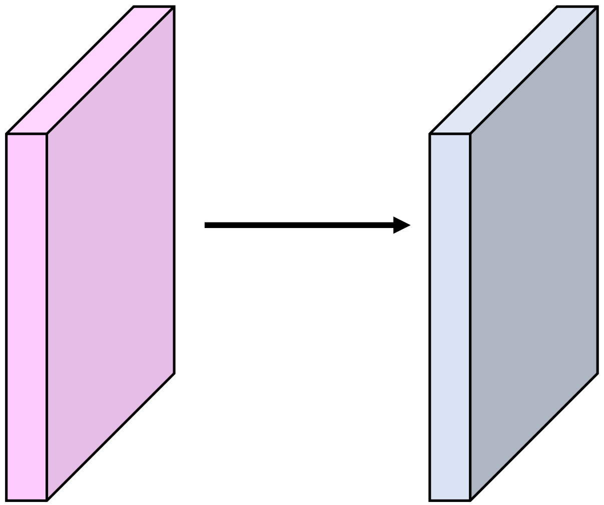
Convolution Example



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:

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

10 x 32 x 32

In general:

Input: W

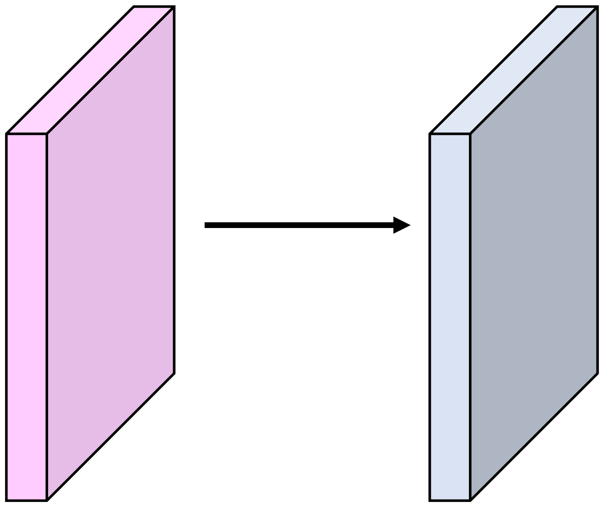
Filter: K

Padding: P

Stride: S

Output: [(W – K + 2P) / S] + 1

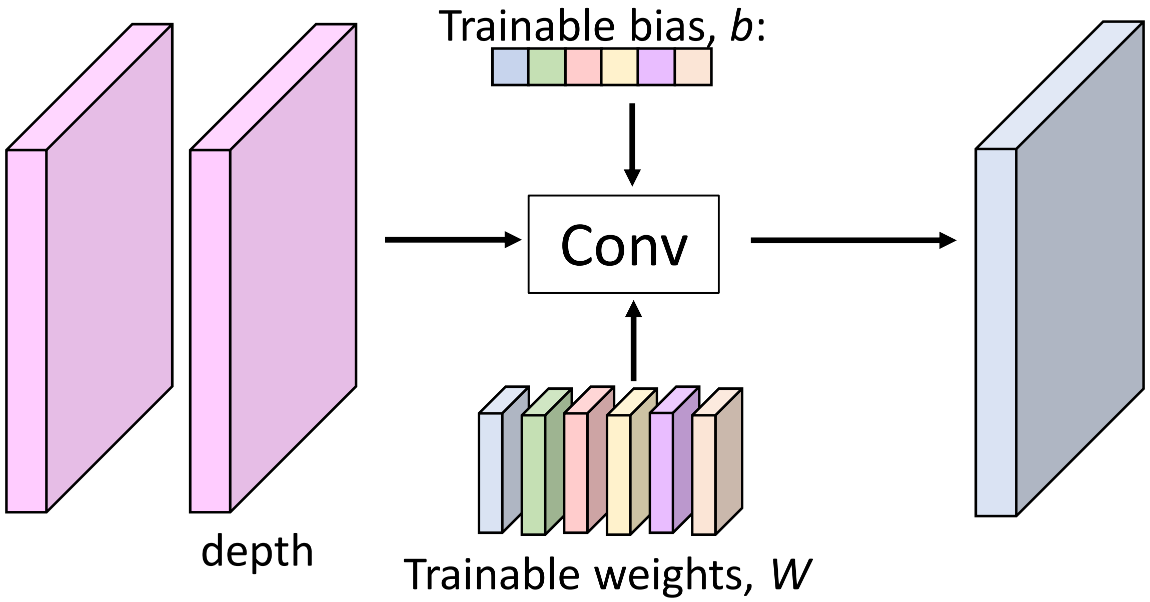
Convolution Example



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

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

Convolution Example

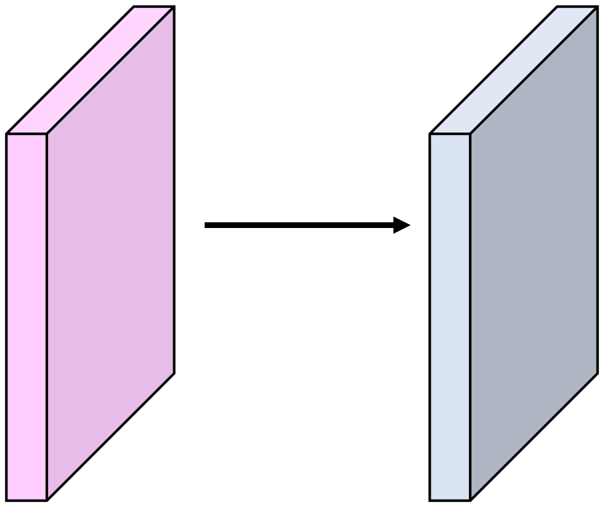


Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

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**

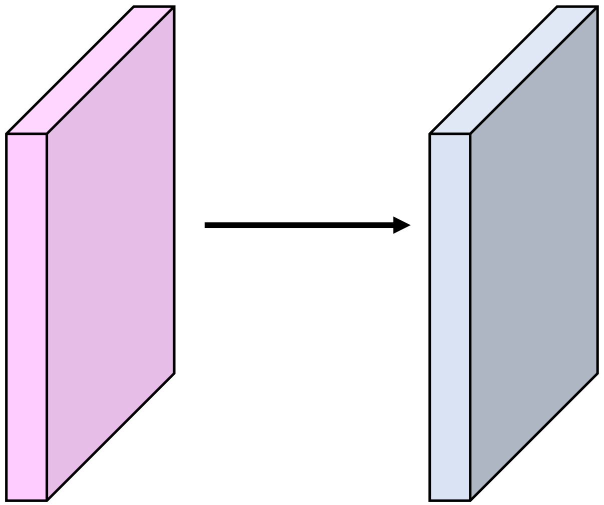
Convolution Example



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

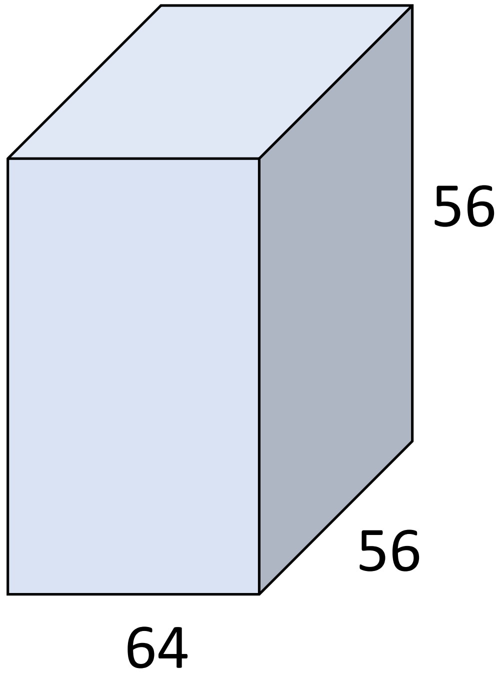


Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

Output volume size: **10 x 32 x 32**
Number of learnable parameters: 760
Number of multiply-add operations: **768,000**

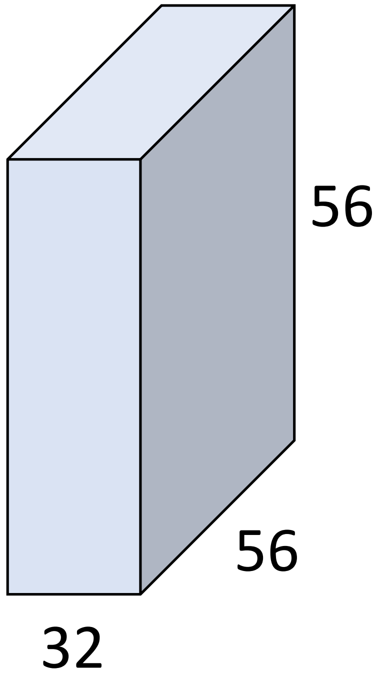
10*32*32 = 10,240 outputs;
each output is the inner product of two **3**x**5x5** tensors (75 elems);
total = 75*10240 = **768K**

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

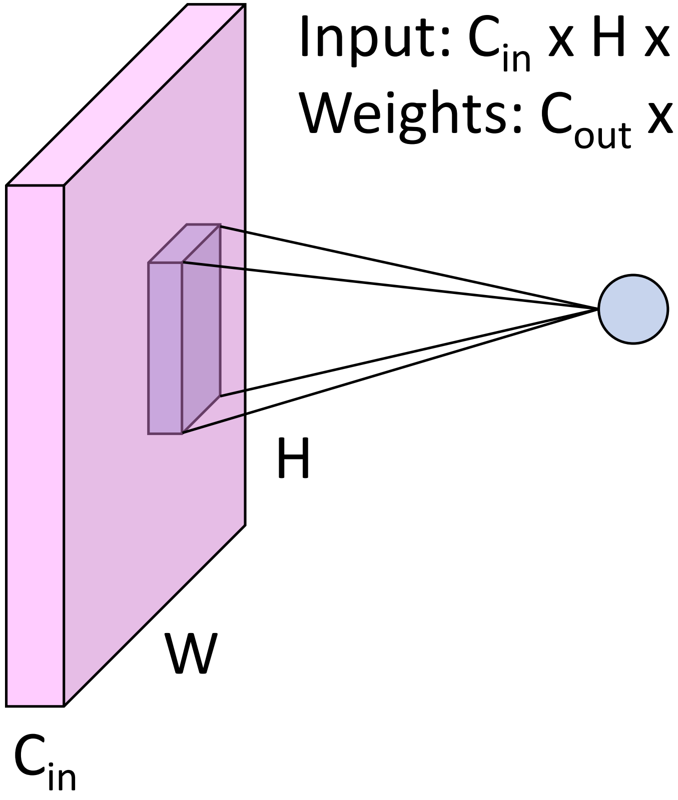
Weight matrix: C_{out} x C_{in} x K_H x K_W giving C_{out} filters of size C_{in} x K_H x K_W
Bias vector: C_{out}
Output size: C_{out} x H' x W' where:
- H' = [(H – K + 2P) / S] + 1
- W' = [(W – K + 2P) / S] + 1

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)

Other types of convolution

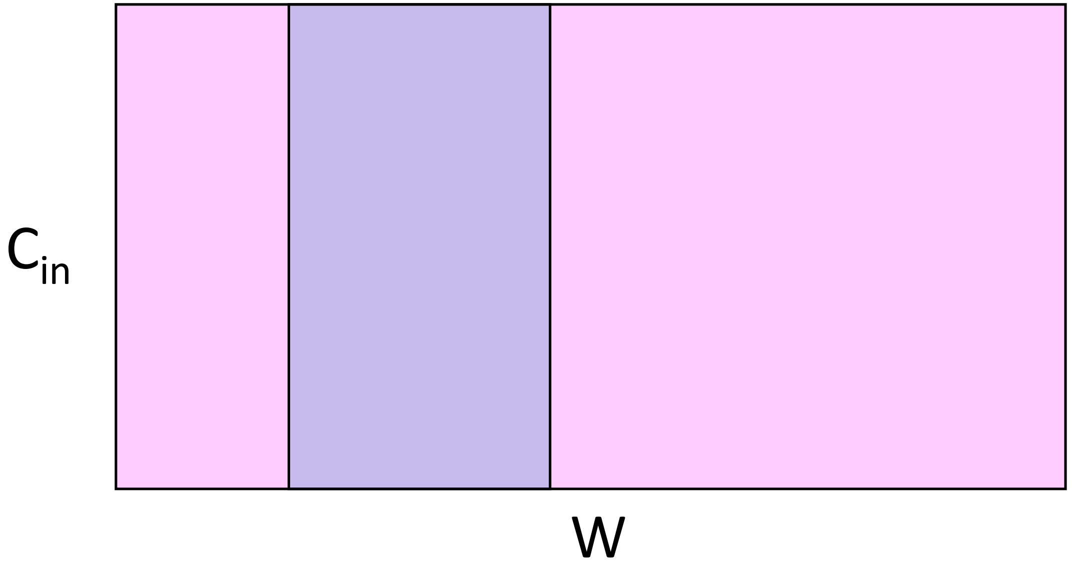
So far: 2D Convolution

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



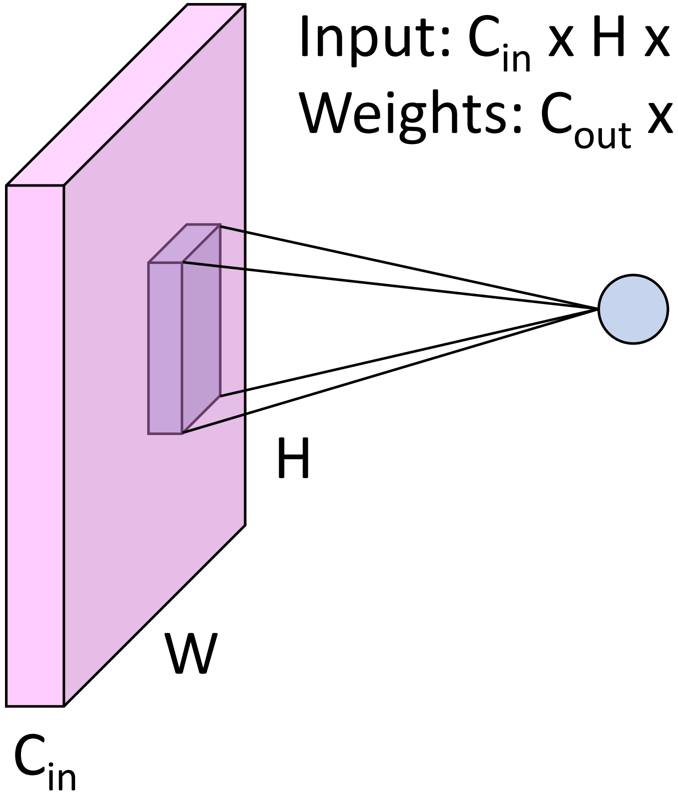
1D Convolution

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



Other types of convolution

So far: 2D Convolution

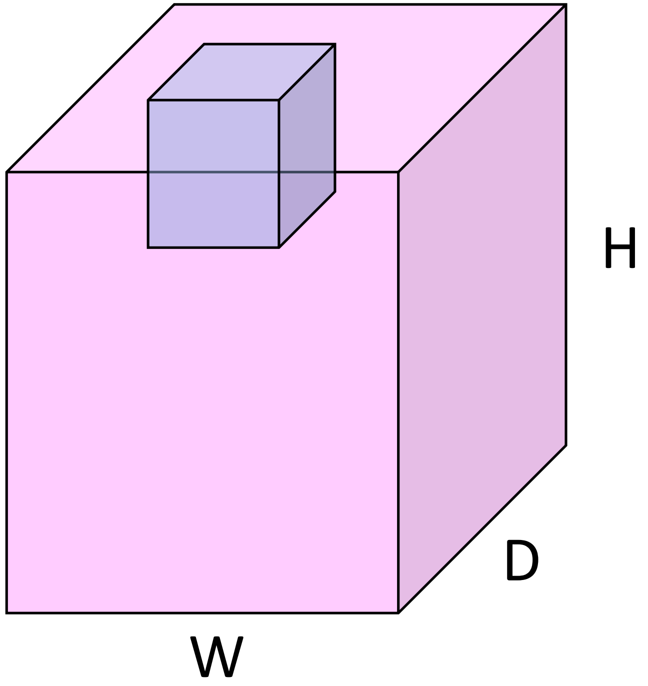


Input: C_{in} x H x W
Weights: C_{out} x C_{in} x K x K

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



Tensorflow Keras Convolutional Layers

Conv2D

```
tf.keras.layers.Conv2D(
    filters, kernel_size, strides=(1, 1), padding='valid', data_format=None,
    dilation_rate=(1, 1), groups=1, activation=None, use_bias=True
)
```

`filters`	Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
`kernel_size`	An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
`strides`	An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
`padding`	one of `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.

Tensorflow Keras Convolutional Layers

Conv2D

```
tf.keras.layers.Conv2D(
    filters, kernel_size, strides=(1, 1), padding='valid', data_format=None,
    dilation_rate=(1, 1), groups=1, activation=None, use_bias=True
)
```

Conv1D

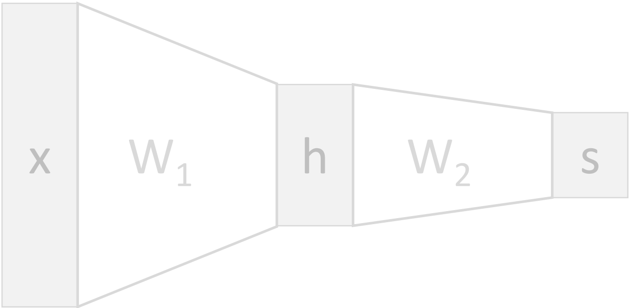
```
tf.keras.layers.Conv1D(
    filters, kernel_size, strides=1, padding='valid', data_format='channels_last',
    dilation_rate=1, groups=1, activation=None, use_bias=True,
)
```

Conv3D

```
tf.keras.layers.Conv3D(
    filters, kernel_size, strides=(1, 1, 1), padding='valid', data_format=None,
    dilation_rate=(1, 1, 1), groups=1, activation=None, use_bias=True
)
```

Components of a Full-Connected Network

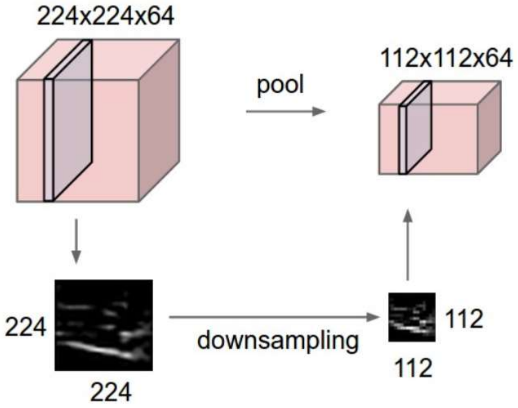
Fully-Connected Layers



Activation Function

Convolution Layers

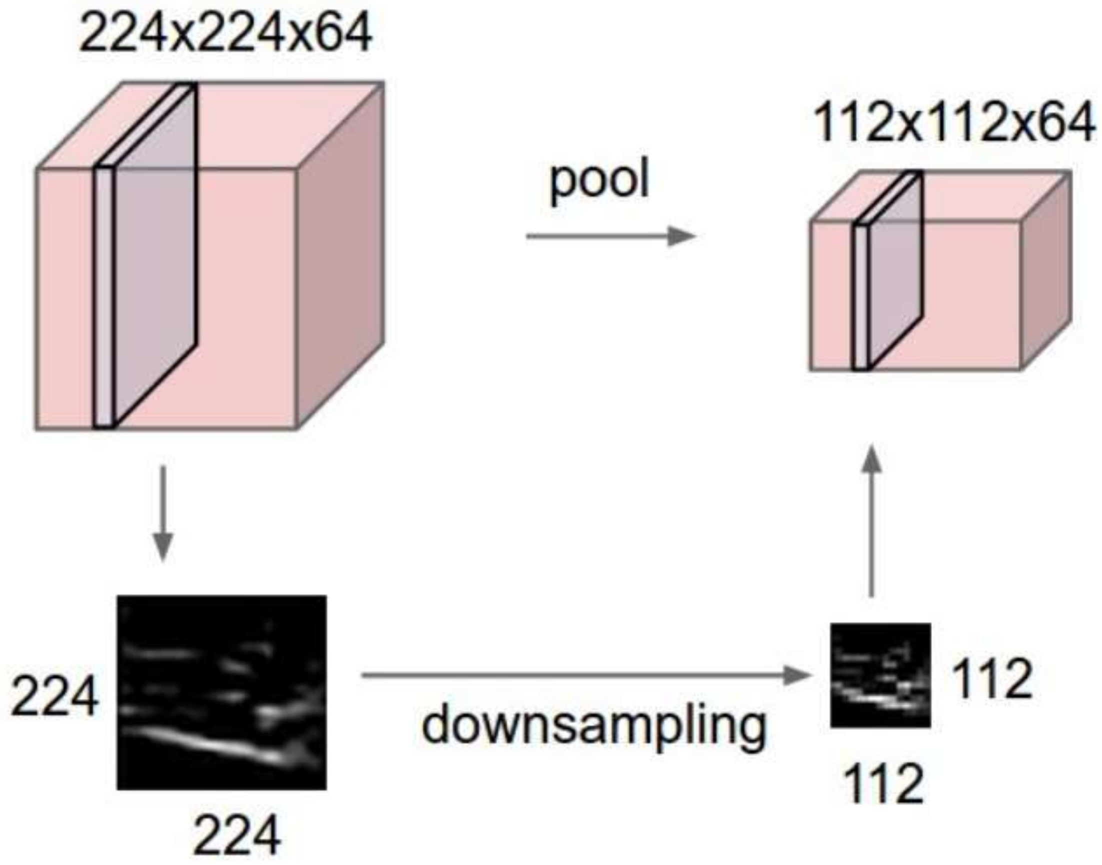
Pooling Layers



Normalization

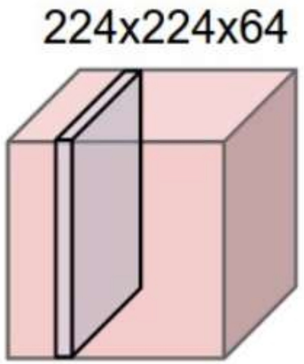
$$\hat{x}_{i,j} = \frac{\hat{x}_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Pooling Layers: Another way to downsample



Hyperparameters:
Kernel Size
Stride
Pooling function

Max Pooling



Single depth slice

x

1	1	2	4
5	**6**	7	**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!

Pooling Summary

Input: C x H x W
Hyperparameters:
- Kernel size: K
- Stride: S
- Pooling function (max, avg)

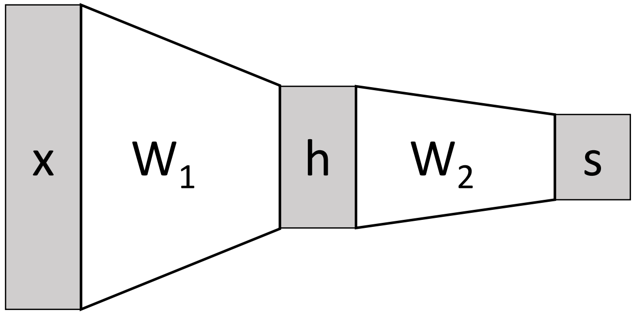
Output: C x H’ x W’ where
- H’ = [(H – K) / S] + 1
- W’ = [(W – K) / S] + 1

Learnable parameters: None!

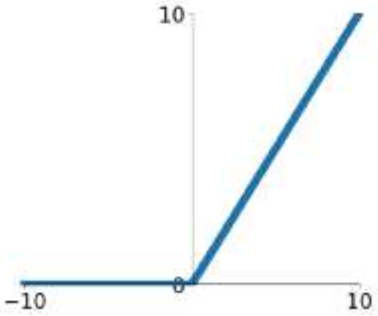
Common settings:
max, K = 2, S = 2
max, K = 3, S = 2 (AlexNet)

Components of a Full-Connected Network

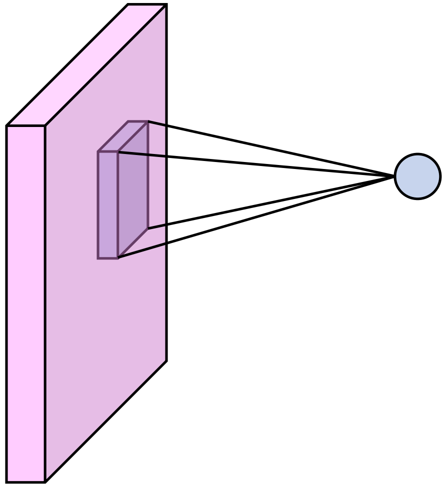
Fully-Connected Layers



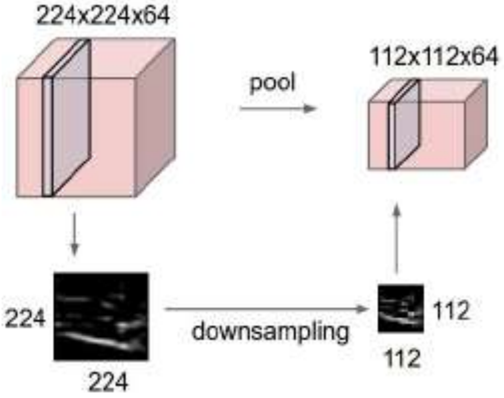
Activation Function



Convolution Layers



Pooling Layers



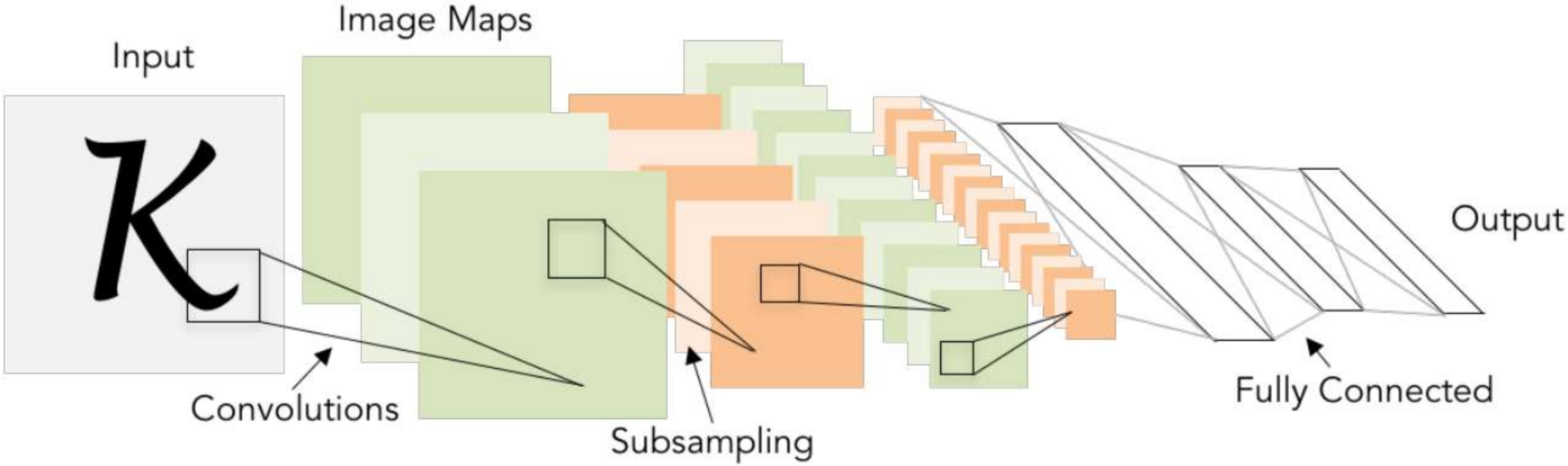
Normalization

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

Convolutional Networks

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

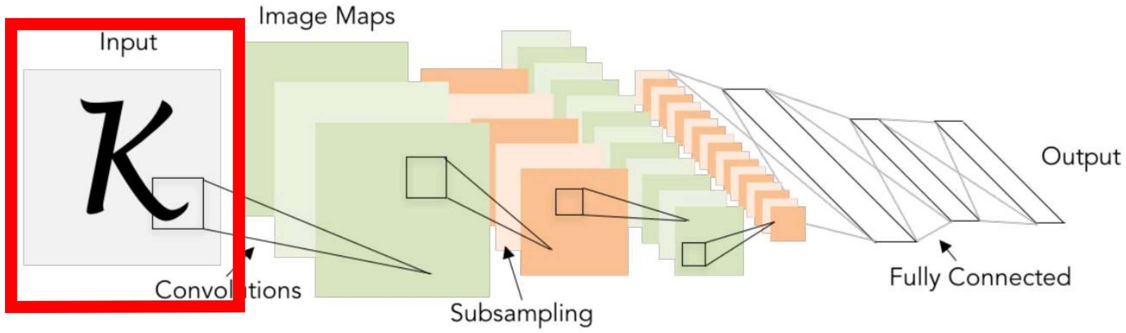
Example: LeNet-5 to train on MNIST



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

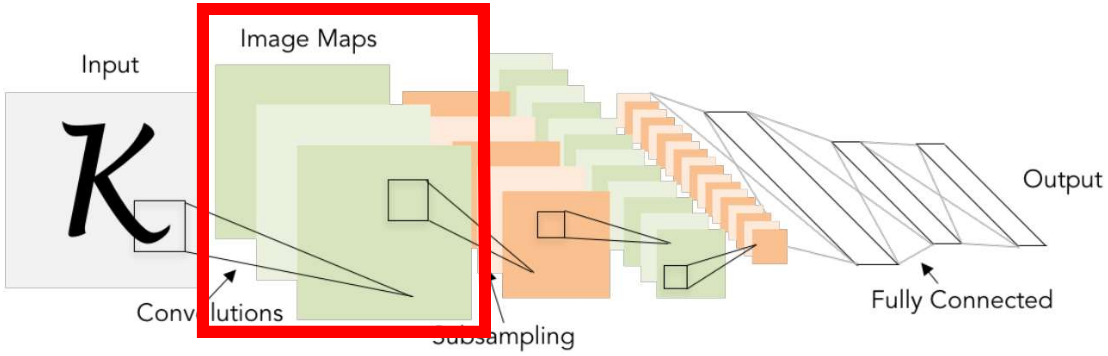
Example: LeNet-5

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 (Cout=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

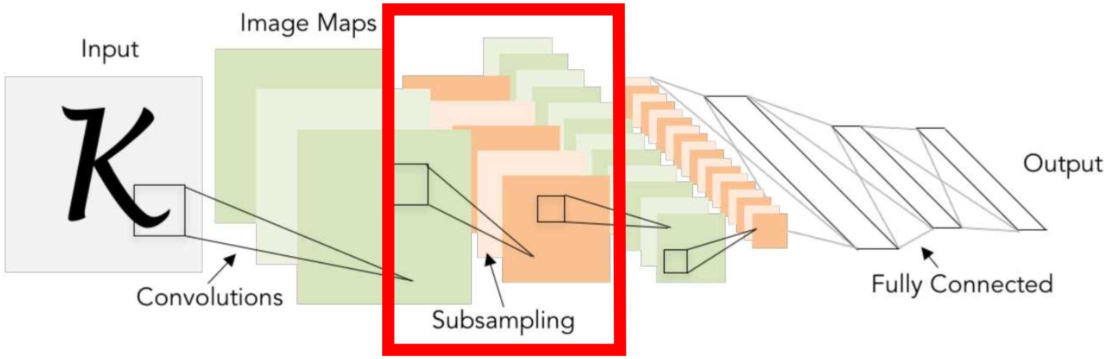
Example: LeNet-5



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 (Cout=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

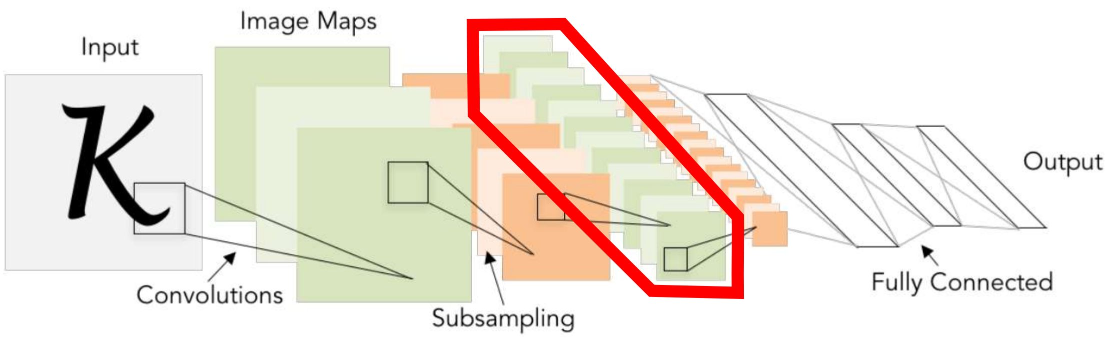
Example: LeNet-5



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 (Cout=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

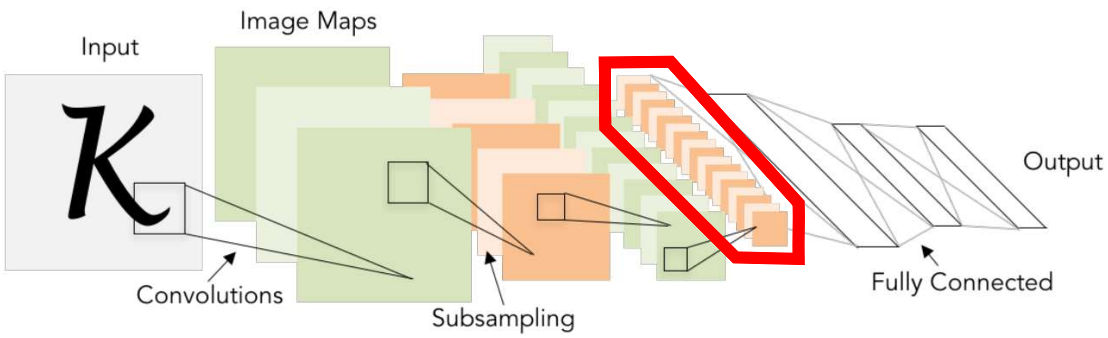
Example: LeNet-5



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 (Cout=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

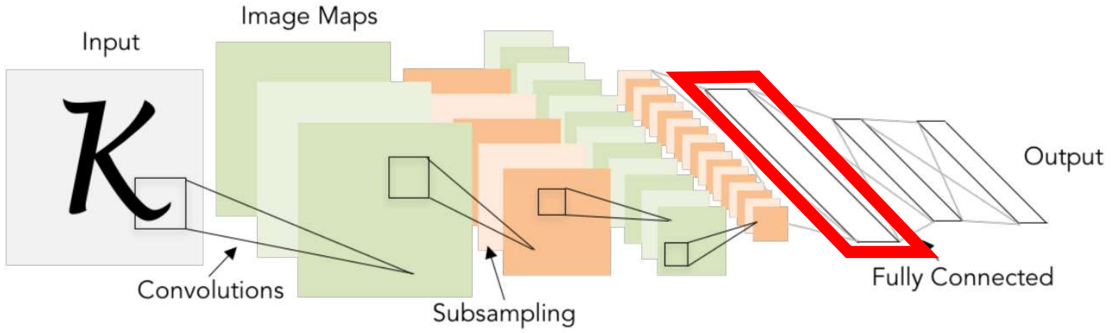
Example: LeNet-5



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 (Cout=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

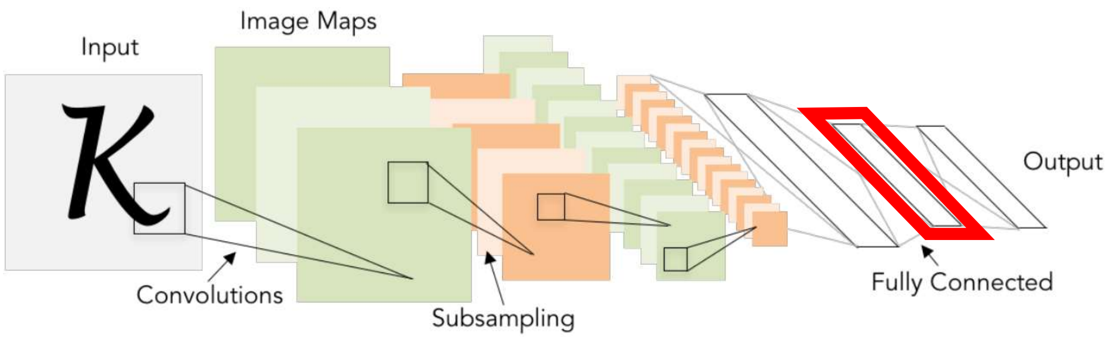
Example: LeNet-5



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 (Cout=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

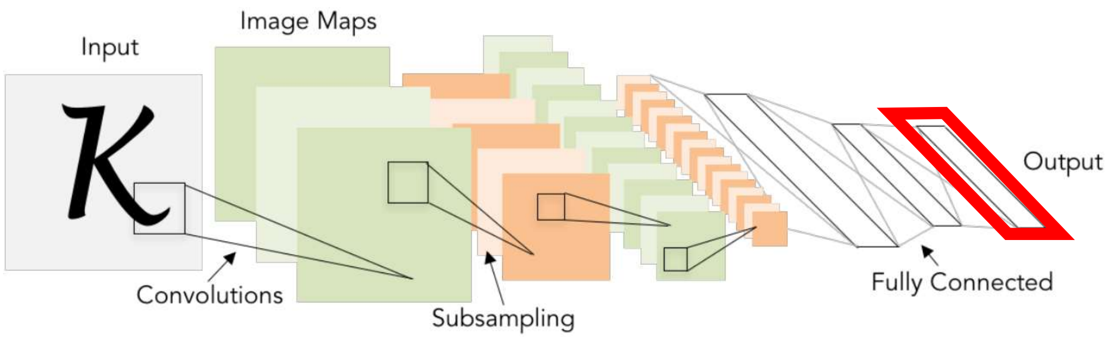
Example: LeNet-5



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 (Cout=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

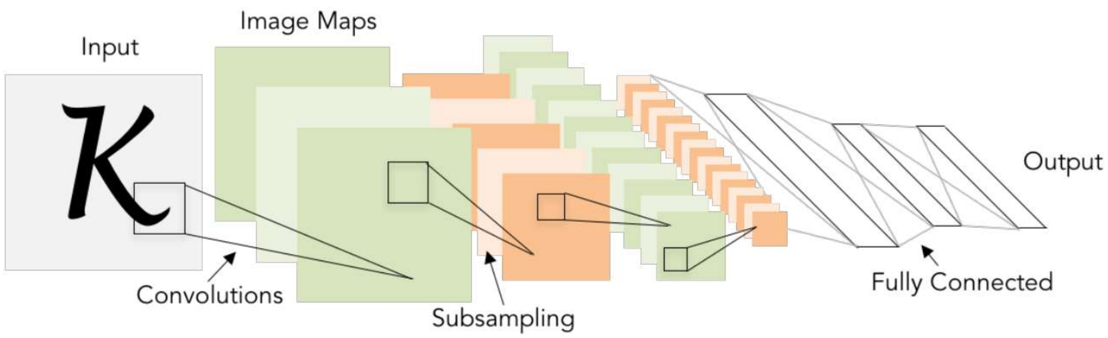


Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (Cout=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)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

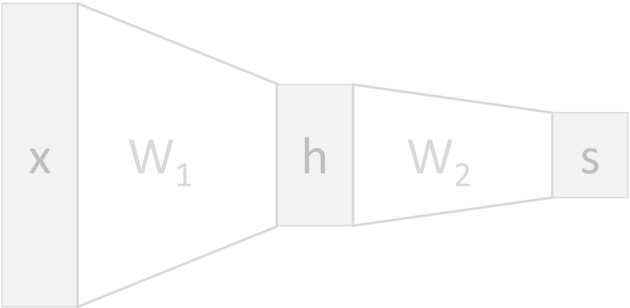
Number of channels **increases**
(total "volume" is preserved!)

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

Problem: Deep Networks very hard to train!

Components of a Full-Connected Network

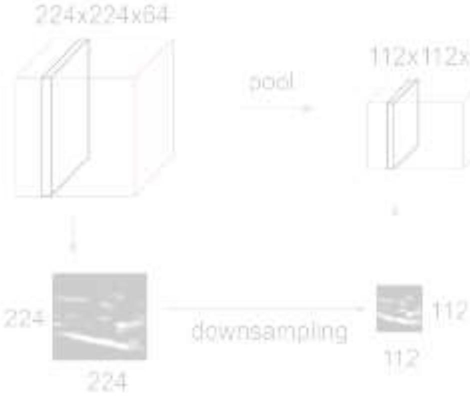
Fully-Connected Layers



Activation Function

Convolution Layers

Pooling Layers



Normalization

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

Batch Normalization

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

Why? Improves optimization

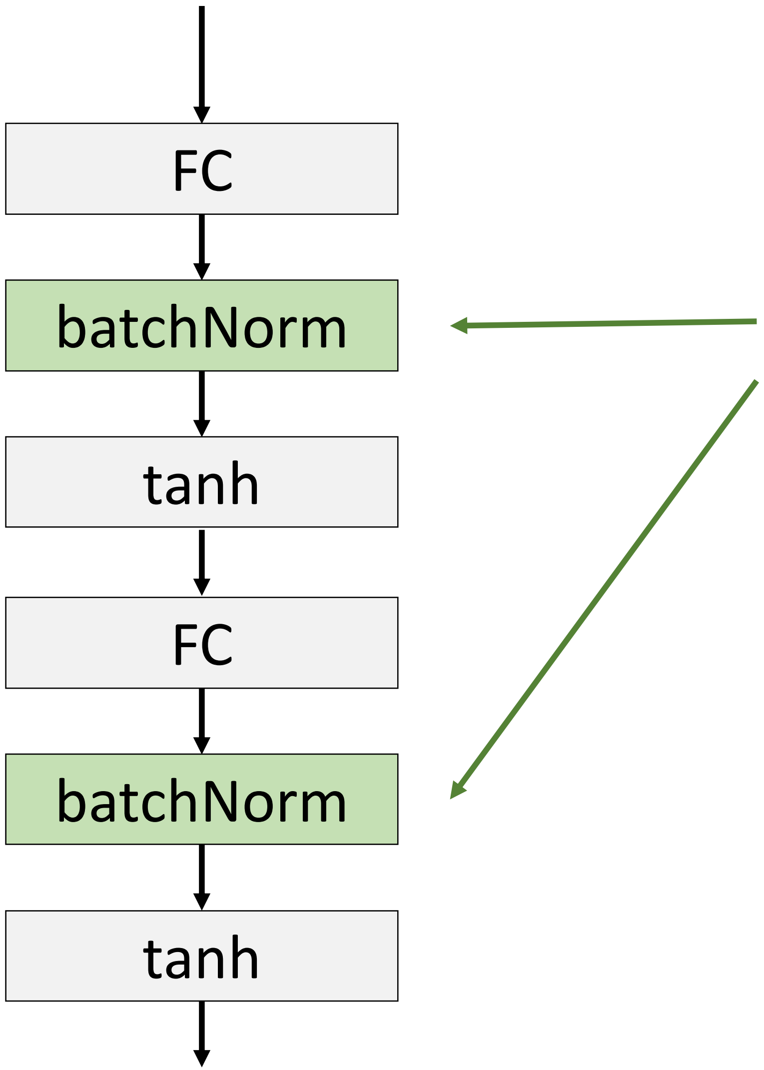
We can normalize a batch of activations like this:

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

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

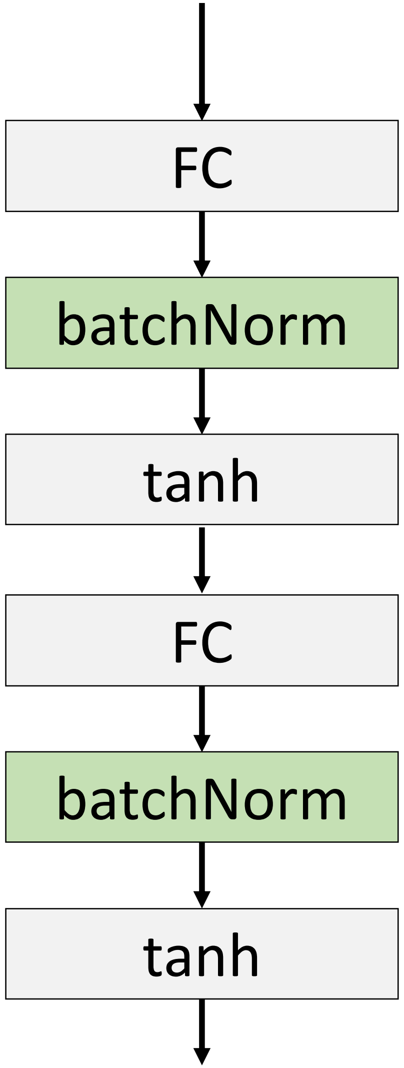


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

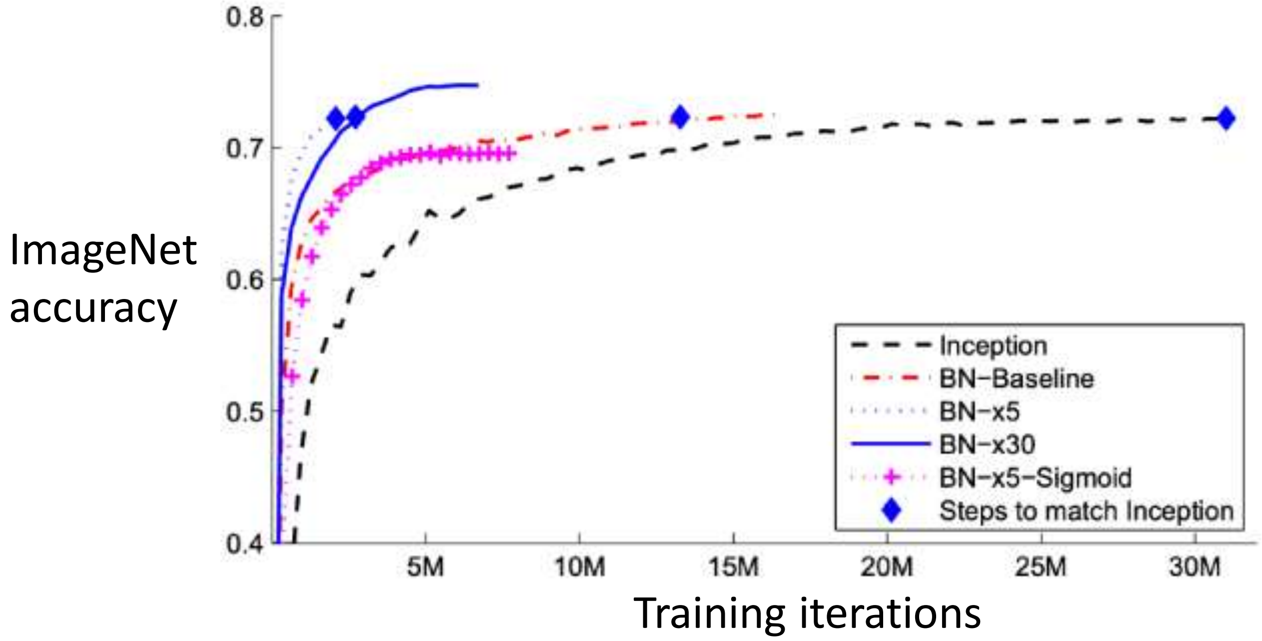
$$\hat{x}_{i,j} = \frac{\hat{x}_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

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

Batch Normalization

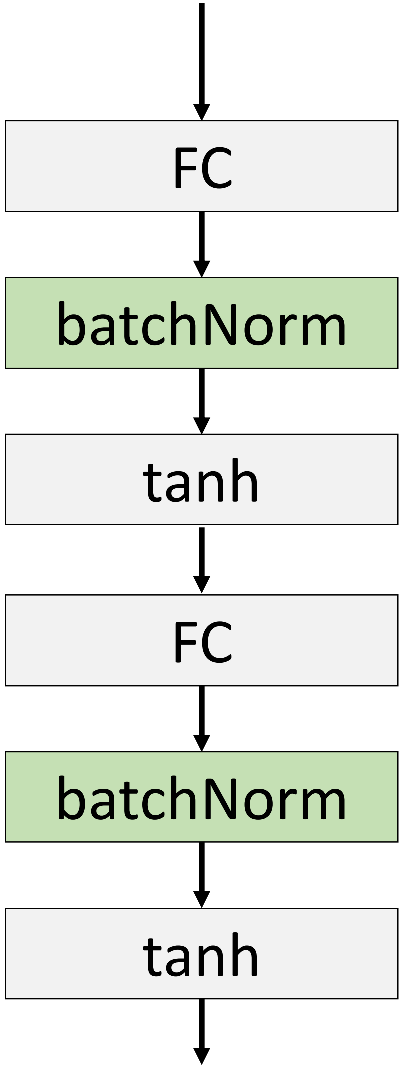


- Makes deep networks much easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!



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

Batch Normalization

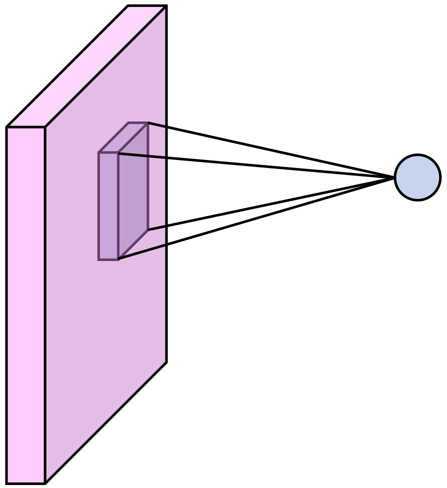



```
tf.keras.layers.BatchNormalization(
    axis=-1,
    momentum=0.99,
    epsilon=0.001,
    center=True,
    scale=True,
    beta_initializer="zeros",
    gamma_initializer="ones",
    moving_mean_initializer="zeros",
    moving_variance_initializer="ones“
)
```

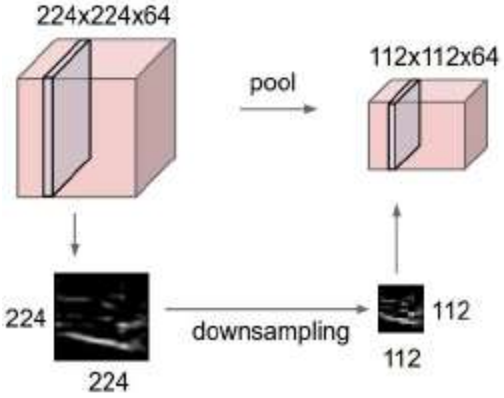
Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015

Summary: Components of a Full-Connected Network

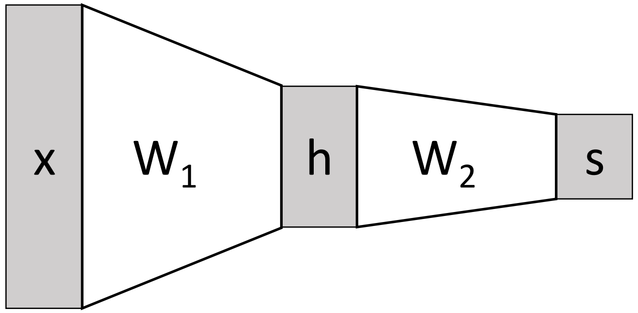
Convolution Layers



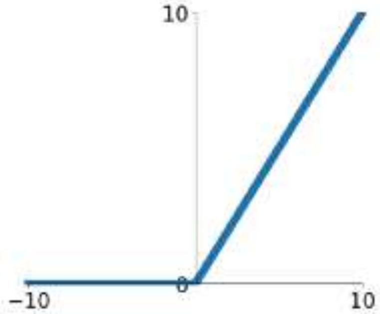
Pooling Layers



Fully-Connected Layers



Activation Function

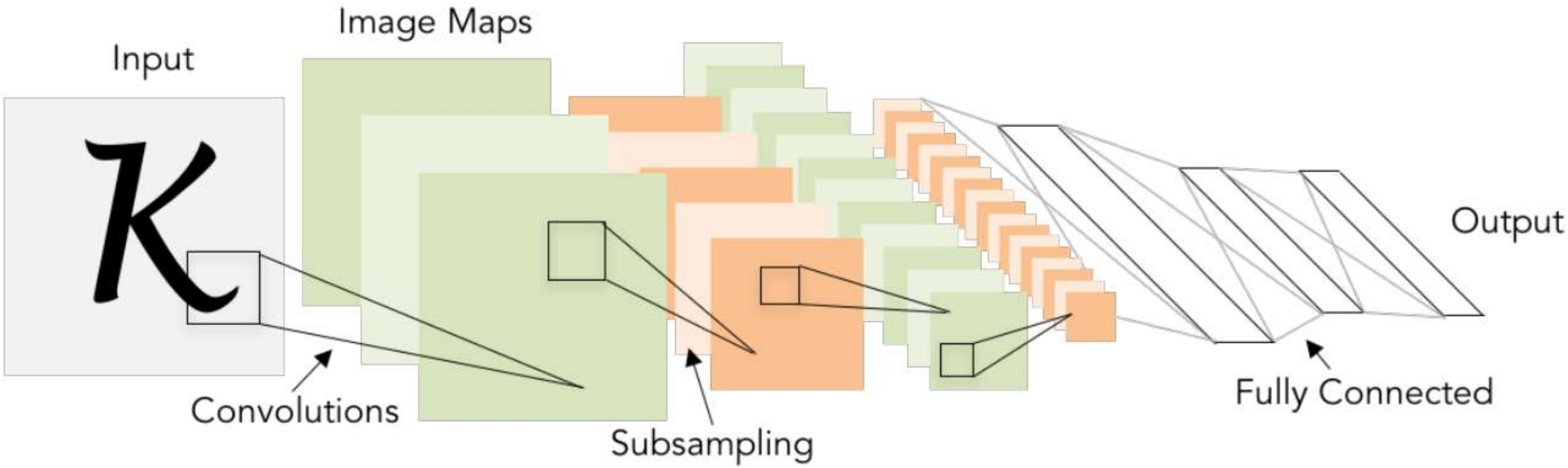


Normalization

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

Summary: Components of a Full-Connected Network

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



Next:

CNN Architectures