

# ITCS 6156/8156 Spring 2024 Machine Learning

## Convolutional Neural Networks

Instructor: Hongfei Xue

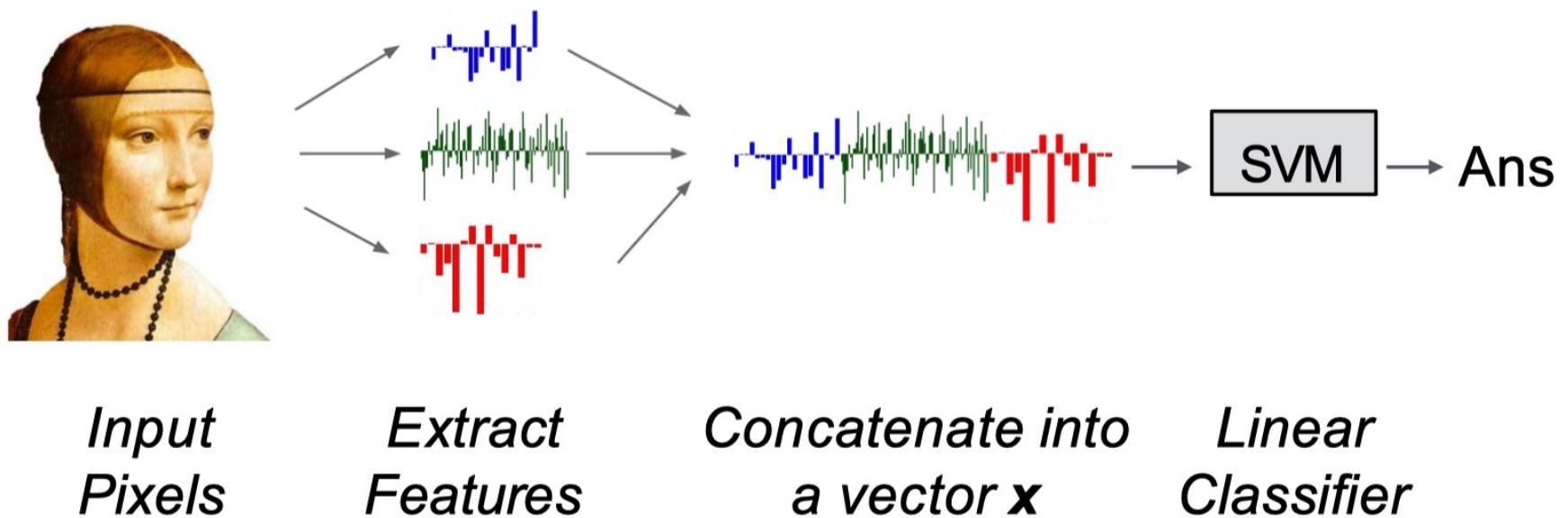
Email: [hongfei.xue@charlotte.edu](mailto:hongfei.xue@charlotte.edu)

Class Meeting: Mon & Wed, 4:00 PM – 5:15 PM, Denny 109



Some content in the slides is based on DeepMind's lecture

# Before Deep Learning



# Using Deep Learning

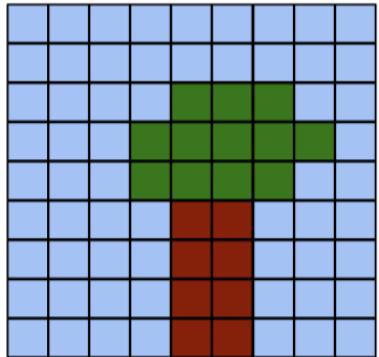


*Input  
Pixels*

*Perform everything with a big neural  
network, trained end-to-end*

- Traditional machine learning models might struggle or require extensive **feature engineering** in these scenarios. Deep learning models can **automatically learn complex representations and feature hierarchies**, leading to superior performance on tasks like image recognition, natural language processing, and speech recognition.

# Neural Networks for Images

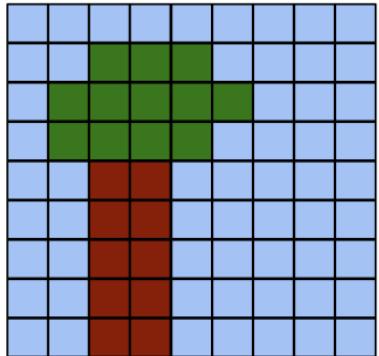


A digital image is a 2D **grid of pixels**.

A neural network expects a **vector of numbers** as input.

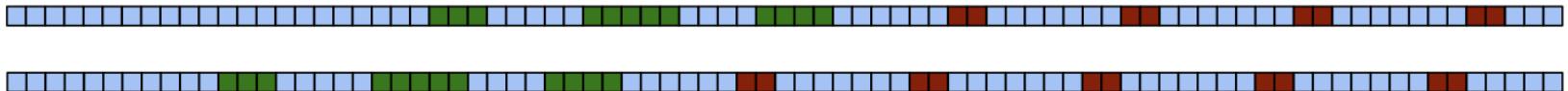


# Neural Networks for Images



A digital image is a 2D **grid of pixels**.

A neural network expects a **vector of numbers** as input.



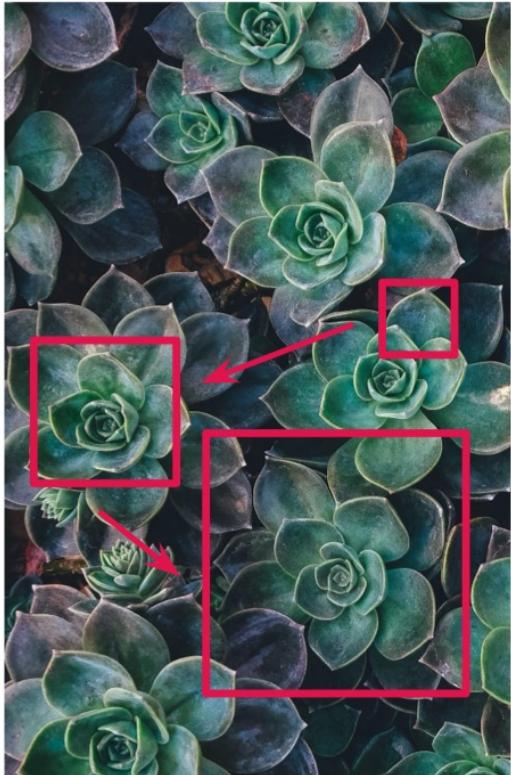
# Locality and translation invariance



**Locality:** nearby pixels are more strongly correlated

**Translation invariance:** meaningful patterns can occur anywhere in the image

# Taking advantage of topological structure



**Weight sharing:** use the same network parameters to detect local patterns at many locations in the image

**Hierarchy:** local low-level features are composed into larger, more abstract features



edges and textures

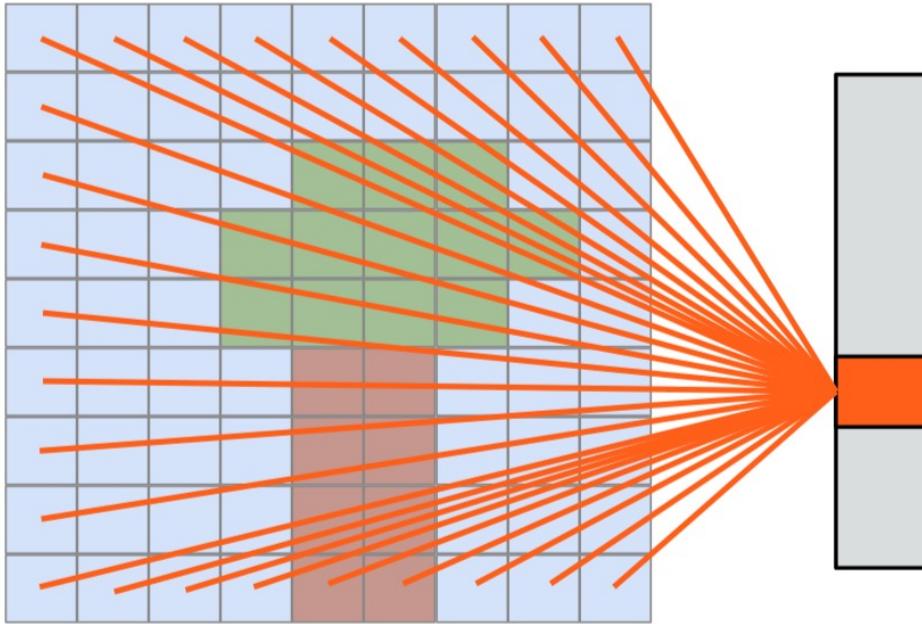


object parts



objects

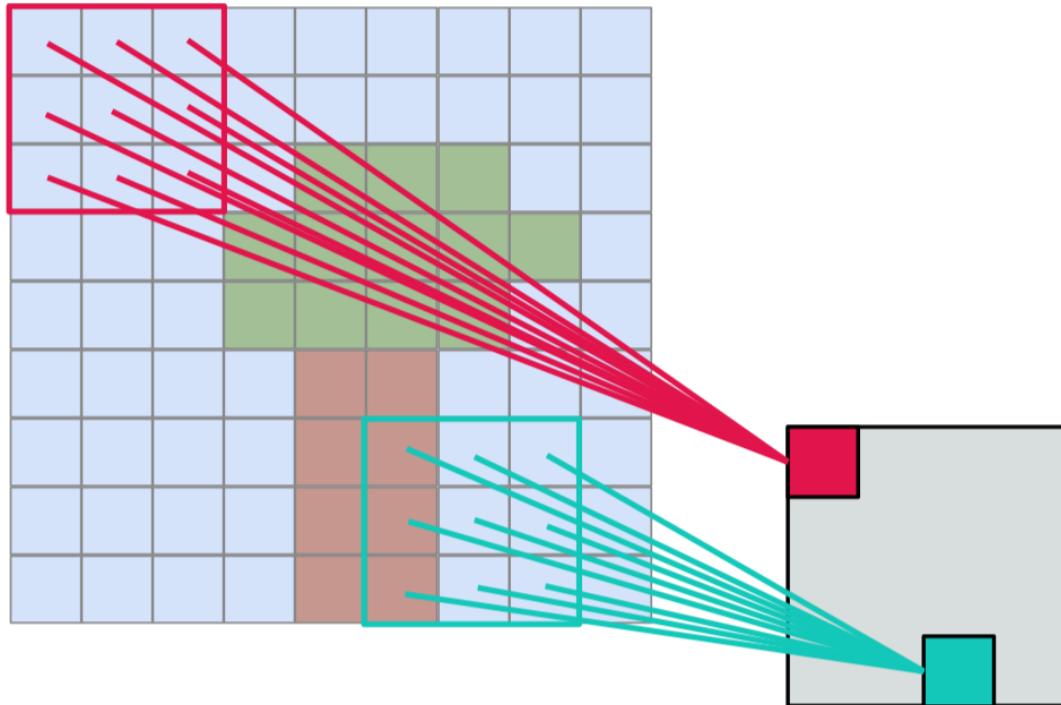
# From fully connected to locally connected



fully-connected unit

$$y = \sum_{i \in \text{image}} \mathbf{w}_i \mathbf{x}_i + b$$

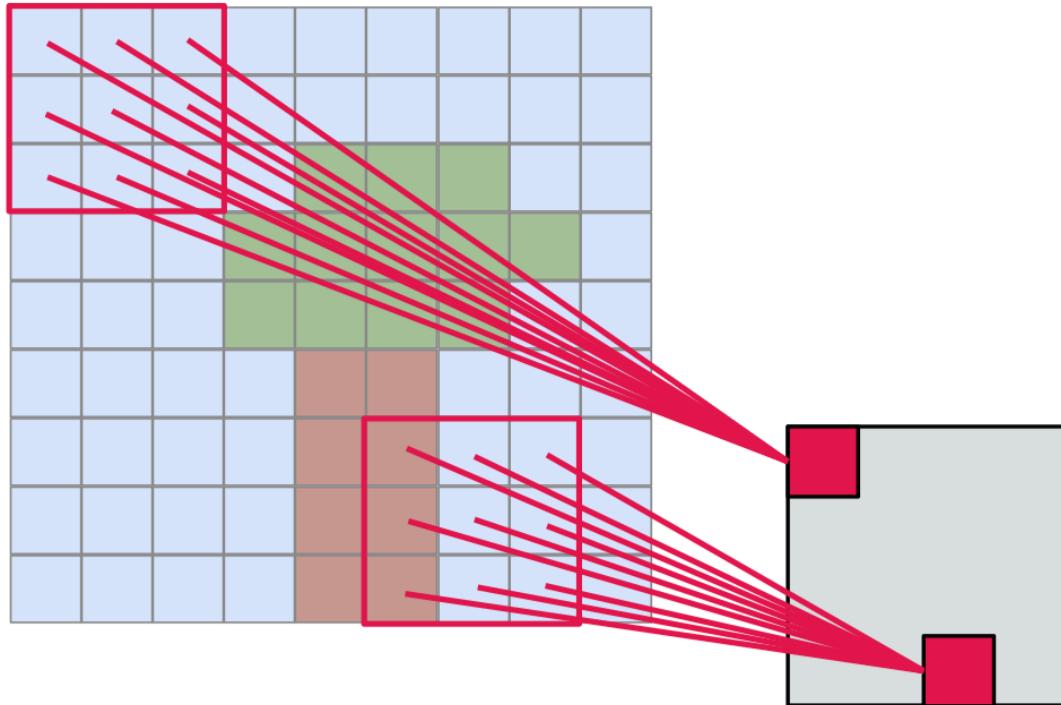
# From fully connected to locally connected



$$y = \sum_{i \in 3 \times 3} \mathbf{w}_i \mathbf{x}_i + b$$

locally-connected units  
3×3 receptive field

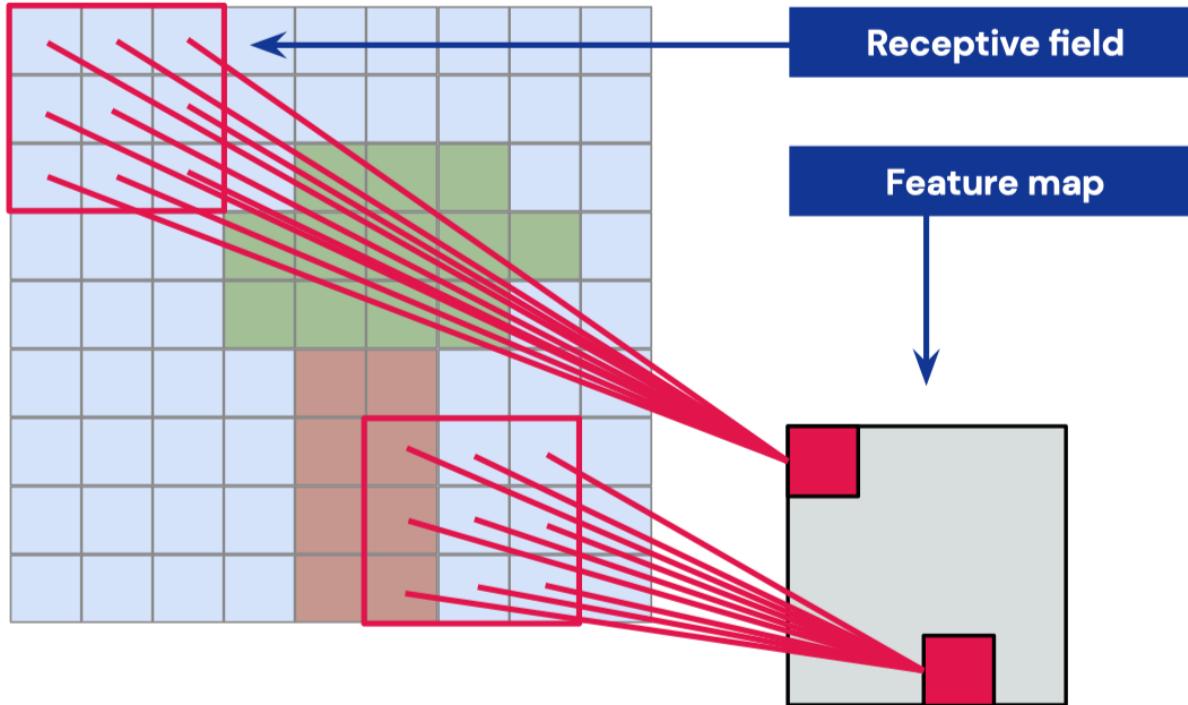
# From fully connected to locally connected



$$y = \mathbf{w} * \mathbf{x} + b$$

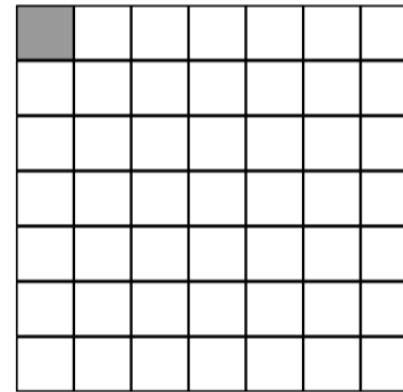
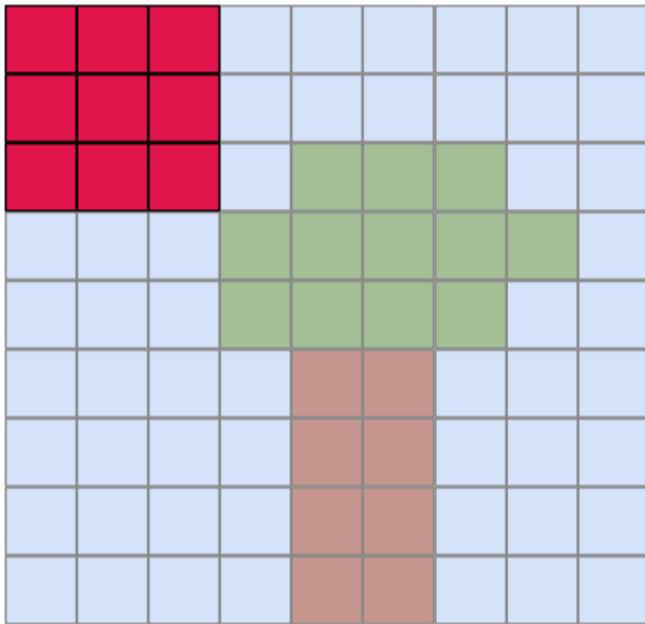
convolutional units  
3×3 receptive field

# From fully connected to locally connected



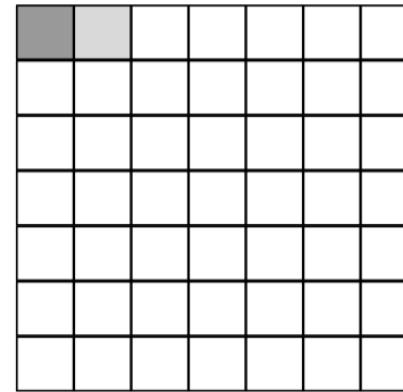
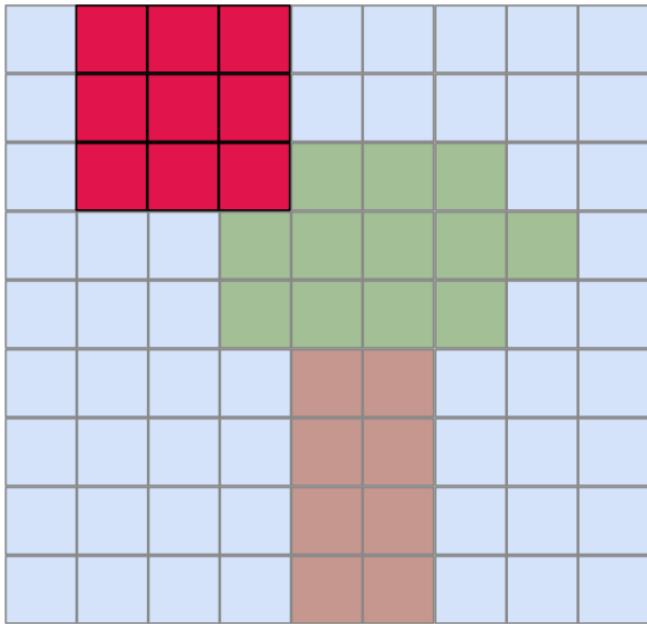
- **Receptive field:** the size of the region in the input that produces the feature. For a given layer in a CNN, the receptive field of a neuron is defined as the size of the region in the input space that affects the neuron's activation.
- **Feature map:** a feature map is a two-dimensional activation map that represents the response of a specific filter applied to the input data or to the output of a previous layer.

# Implementation: the convolution operation



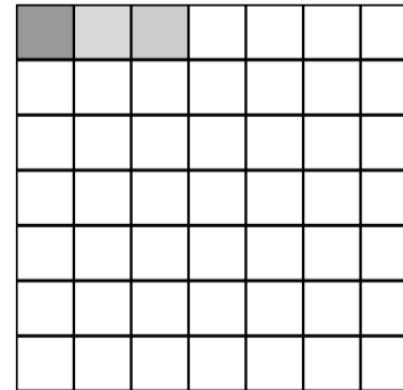
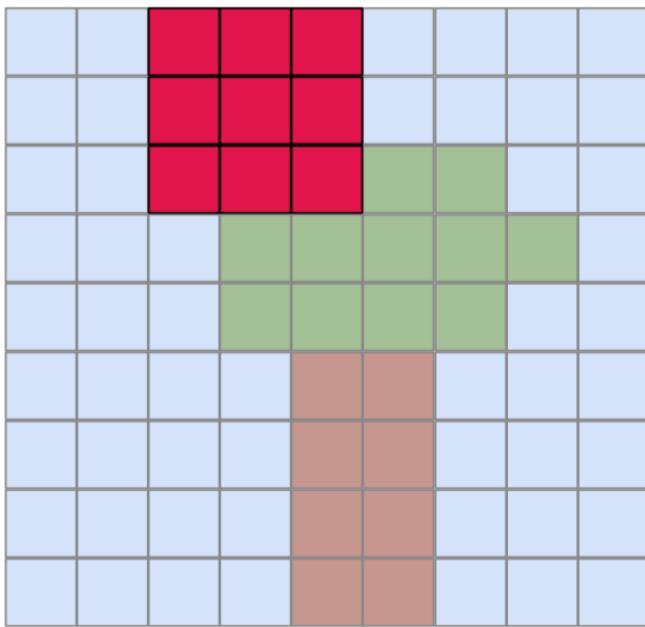
The **kernel** slides across the image and produces an output value at each position

# Implementation: the convolution operation



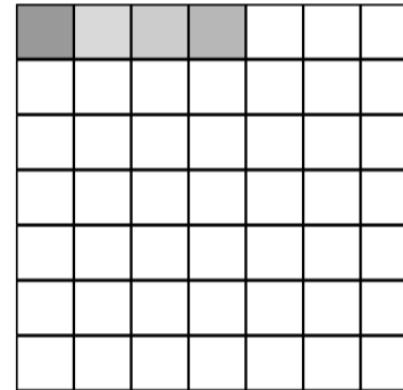
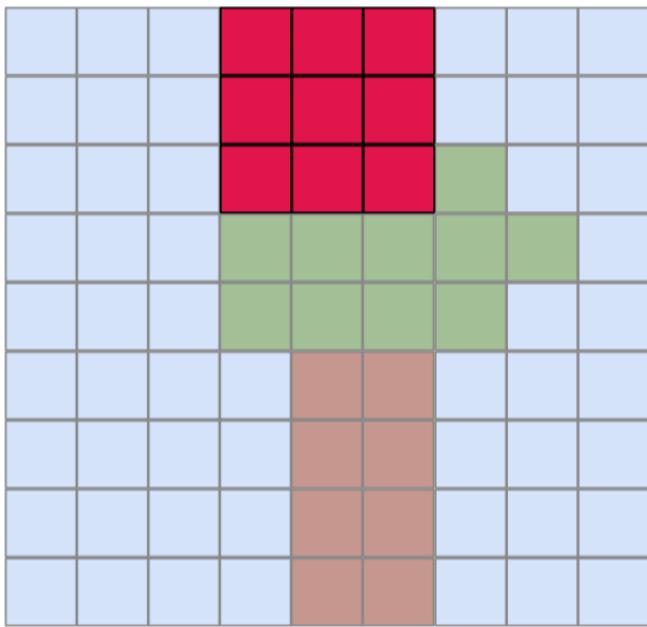
The **kernel** slides across the image and produces an output value at each position

# Implementation: the convolution operation



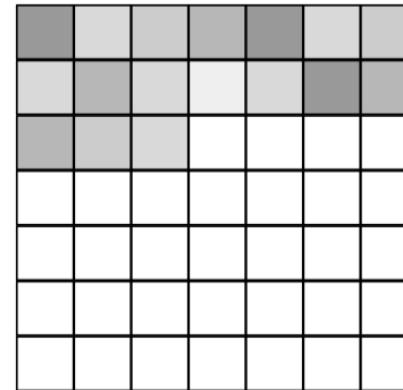
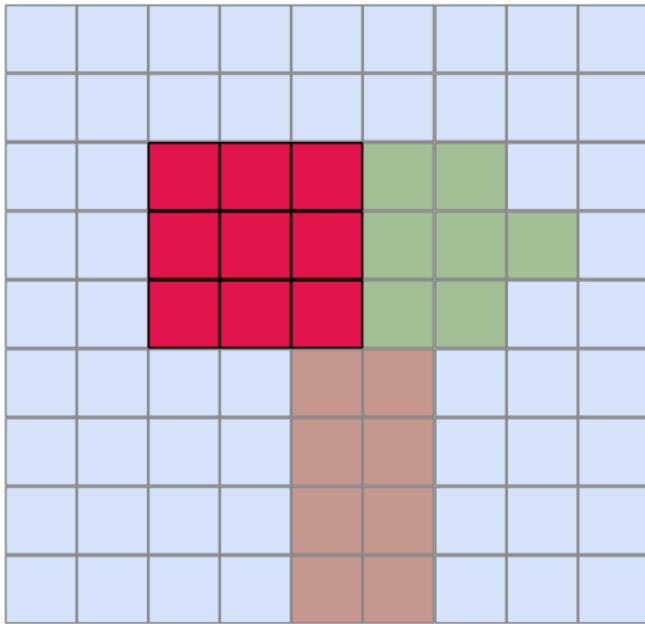
The **kernel** slides across the image and produces an output value at each position

# Implementation: the convolution operation



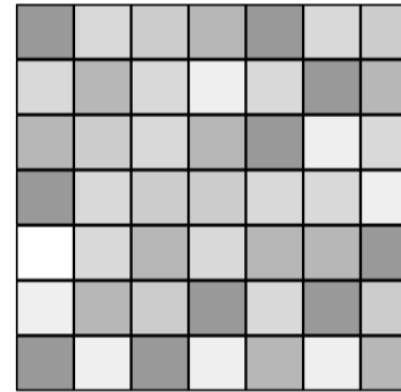
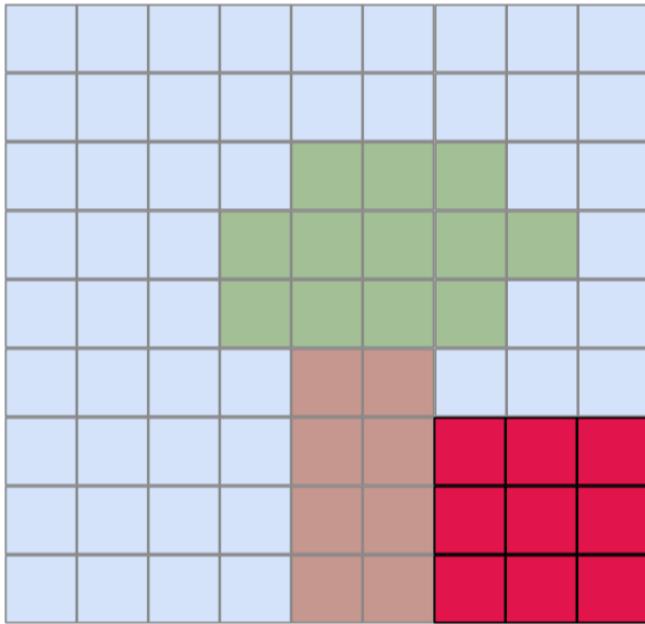
The **kernel** slides across the image and produces an output value at each position

# Implementation: the convolution operation



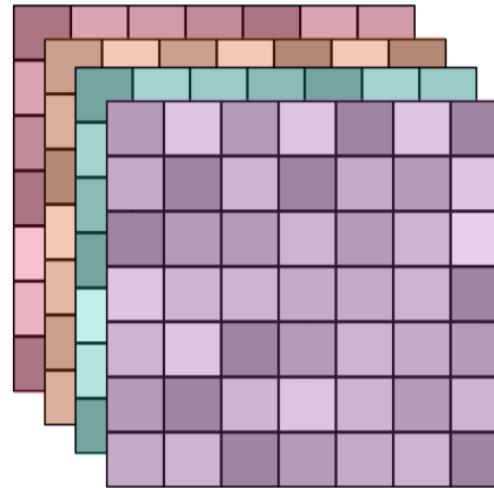
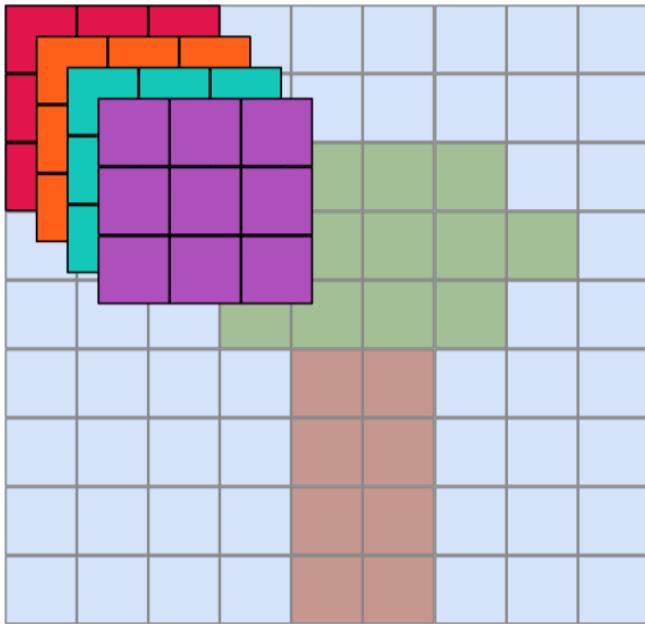
The **kernel** slides across the image and produces an output value at each position

# Implementation: the convolution operation



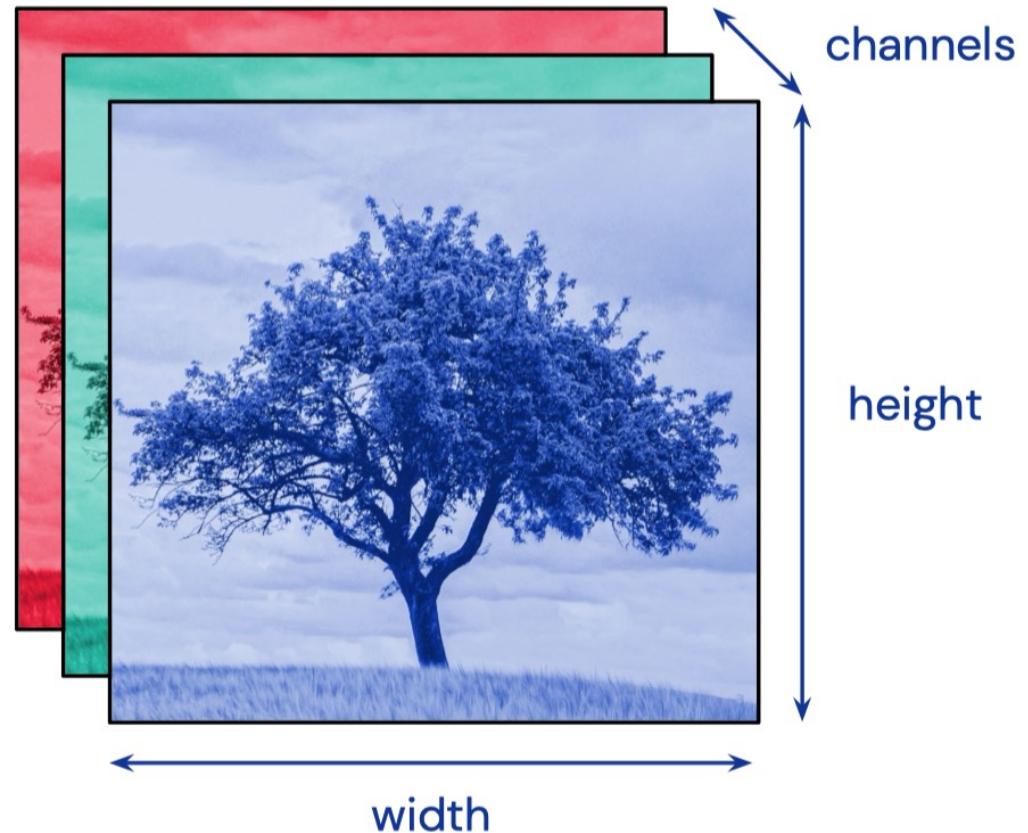
The **kernel** slides across the image and produces an output value at each position

# Implementation: the convolution operation

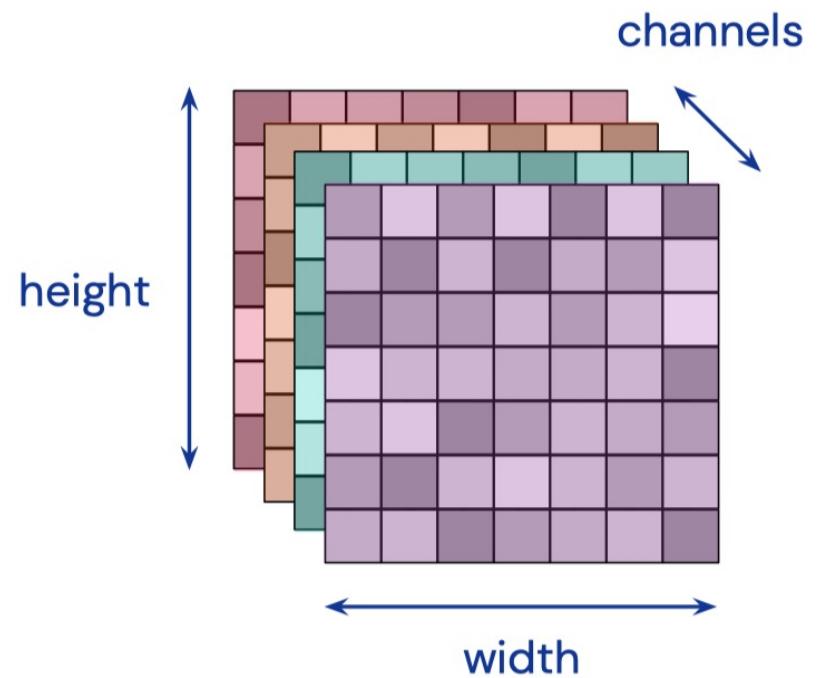


We convolve multiple kernels and obtain  
multiple feature maps or **channels**

# Inputs and outputs are tensors

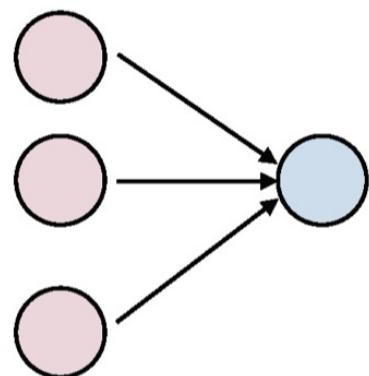


# Inputs and outputs are tensors

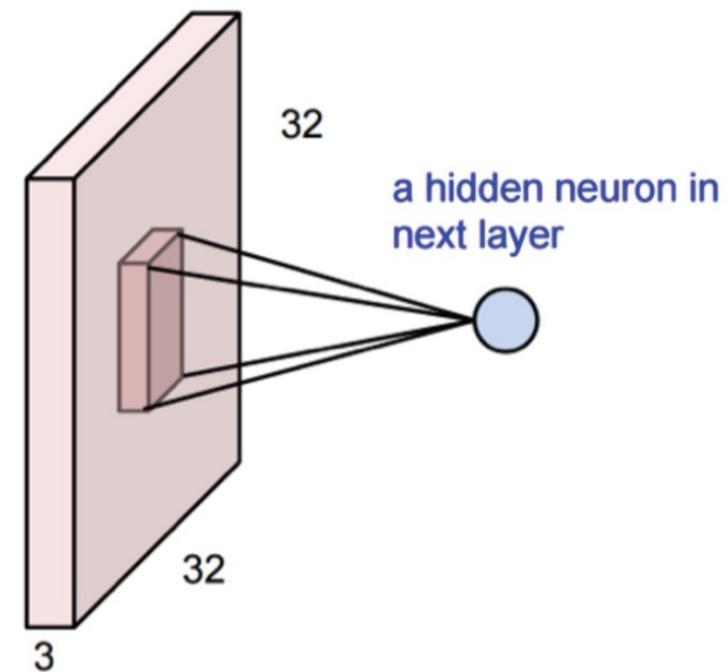


# 3D Activations

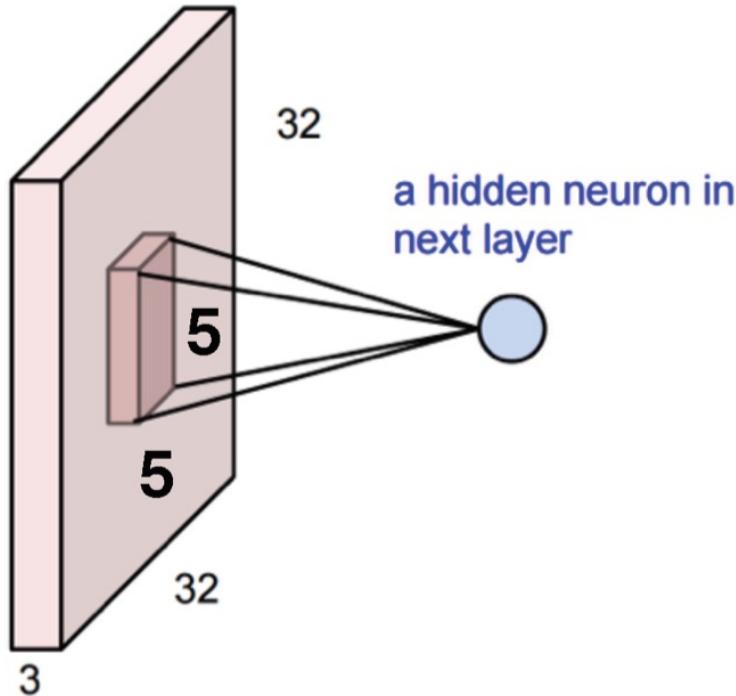
**1D Activations:**



**3D Activations:**

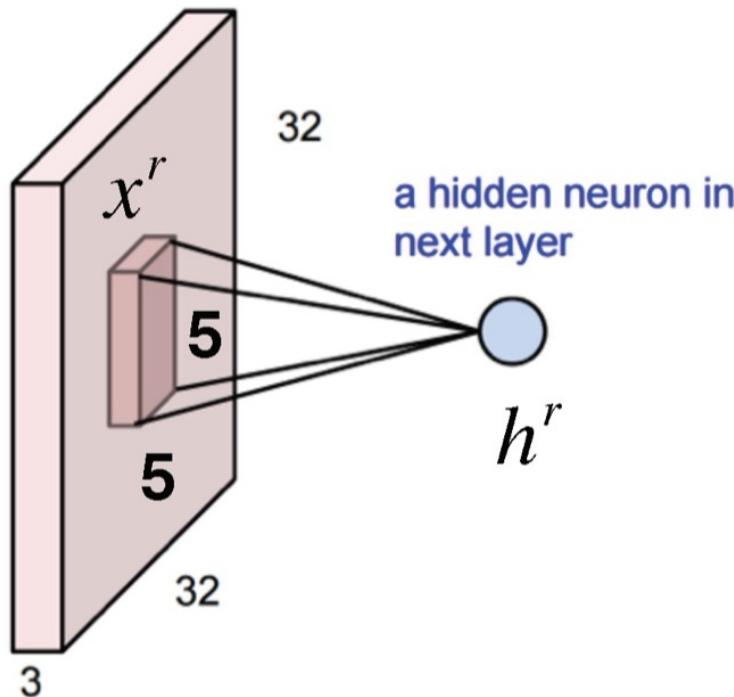


# 3D Activations



- The input is  $3 \times 32 \times 32$
- This neuron depends on a  $3 \times 5 \times 5$  chunk of the input
- The neuron also has a  $3 \times 5 \times 5$  set of weights and a bias (scalar)

# 3D Activations



Example: consider the region of the input “ $x^r$ ”

With output neuron  $h^r$

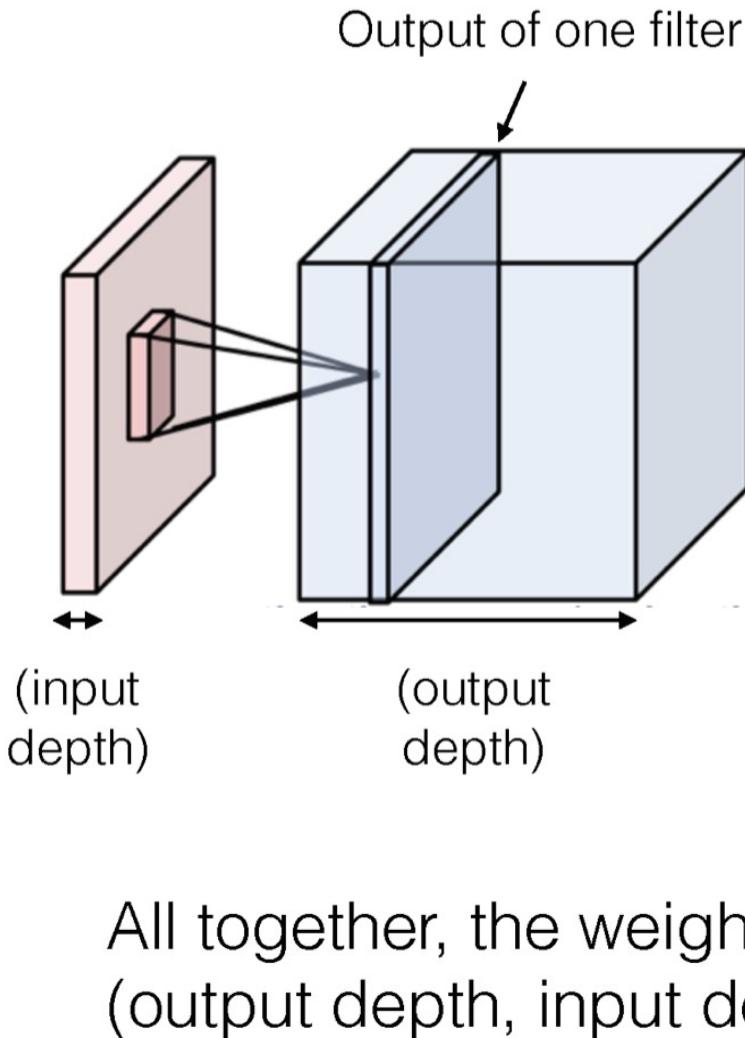
Then the output is:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$



Sum over 3 axes

# 3D Activations



One set of weights gives  
one slice in the output

To get a 3D output of depth  $D$ ,  
use  $D$  different filters

In practice, ConvNets use  
many filters (~64 to 1024)

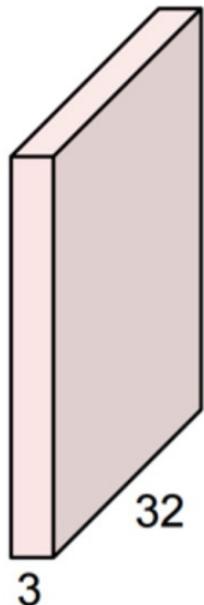
Number of parameters of  
one convolutional layer  
without bias.

All together, the weights are **4** dimensional:  
(output depth, input depth, kernel height, kernel width)

# Recap

## Convolution Layer

32x32x3 image



5x5x3 filter

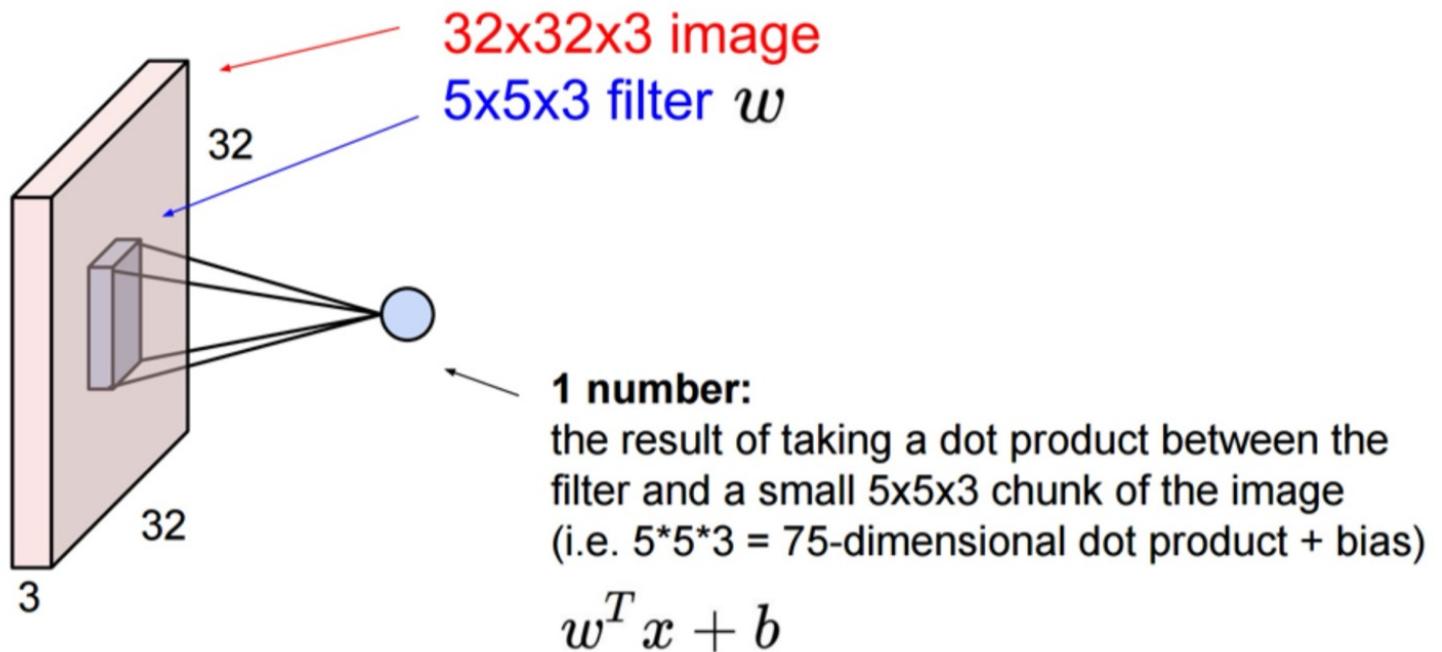


Filters always extend the full depth of the input volume

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

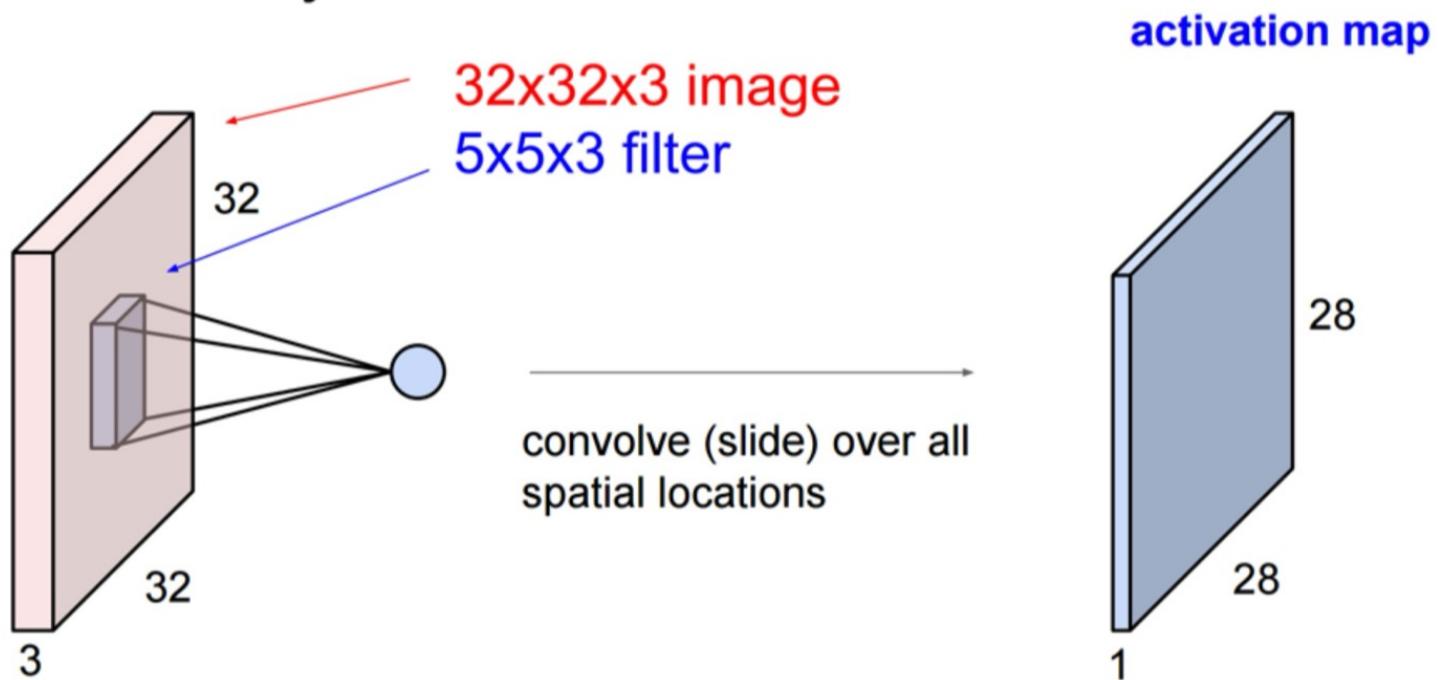
# Recap

## Convolution Layer



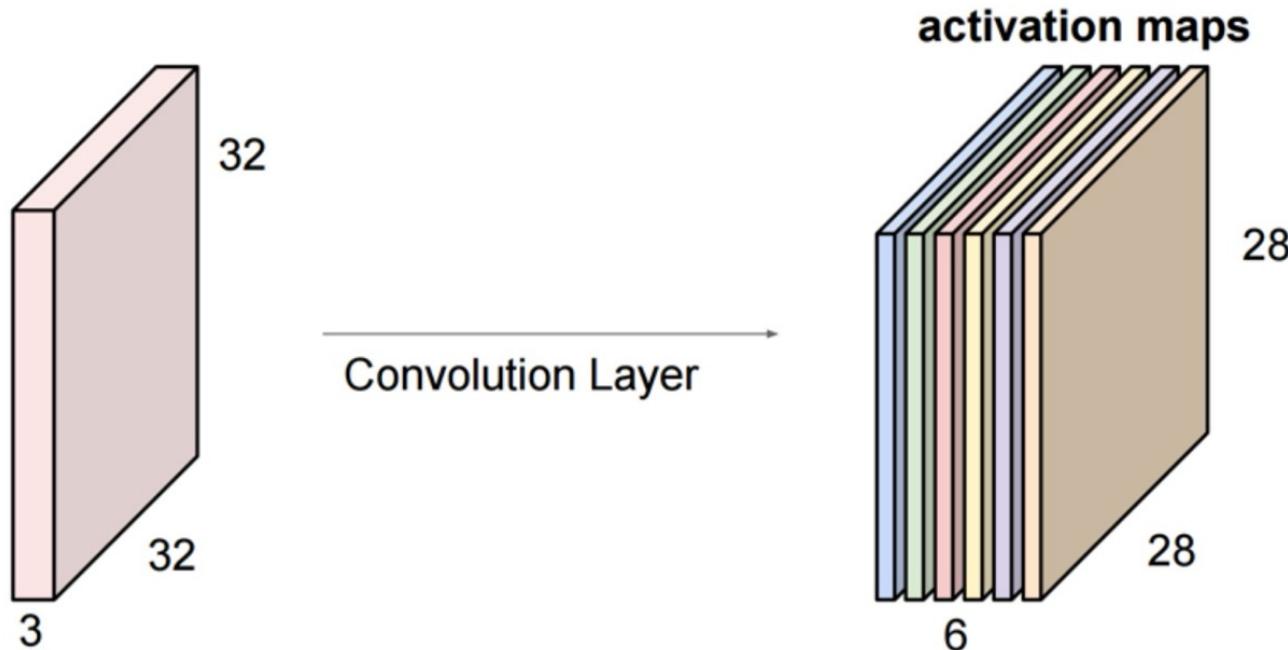
# Recap

## Convolution Layer



# Recap

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



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

- To measure the complexity of the network:

- Number of parameters:

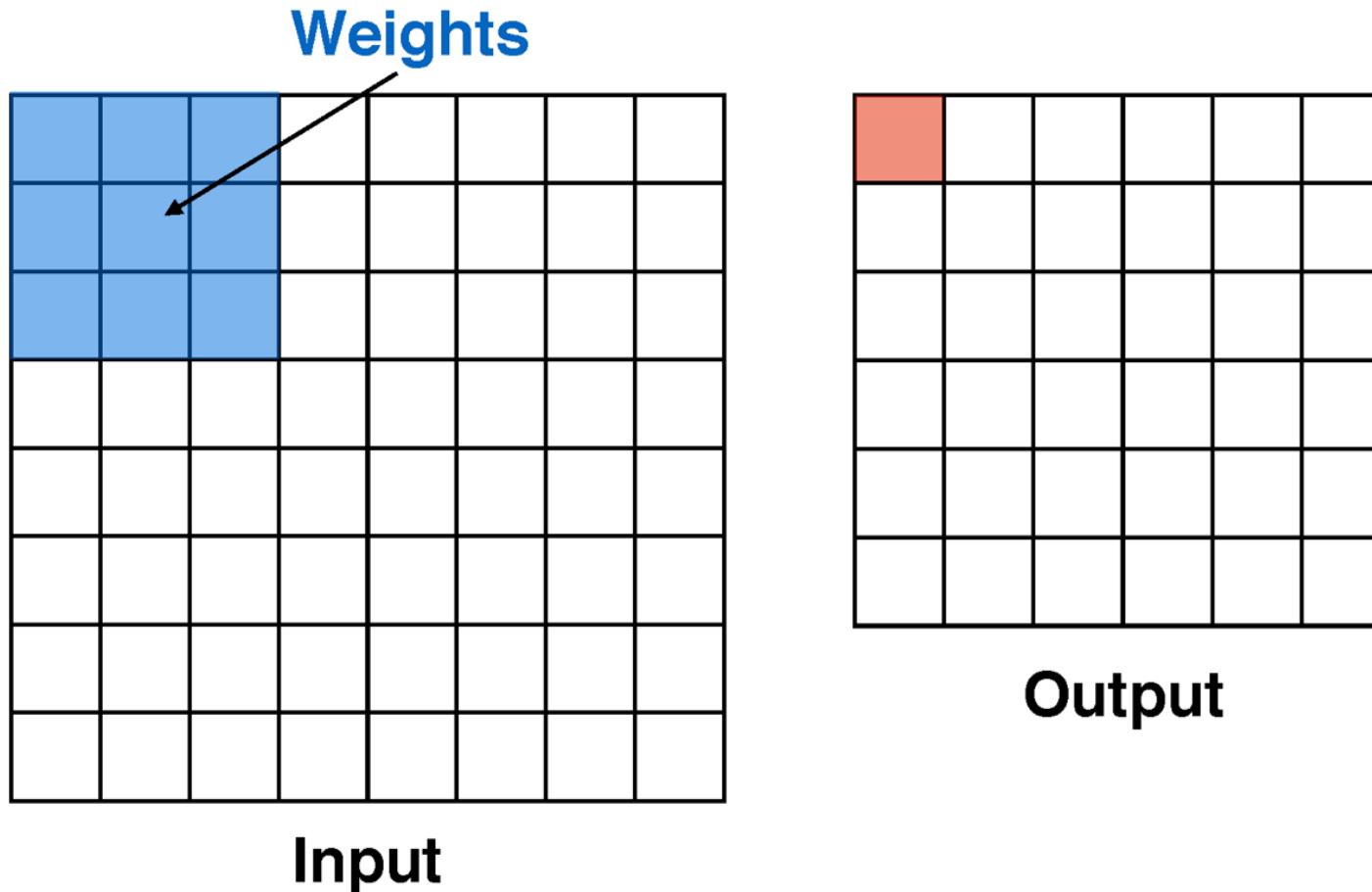
$$3 * 5 * 5 * 6 + 6 = 456$$

- Number of multiplications:

$$3 * 5 * 5 * 6 * 28 * 28 = 352,800$$

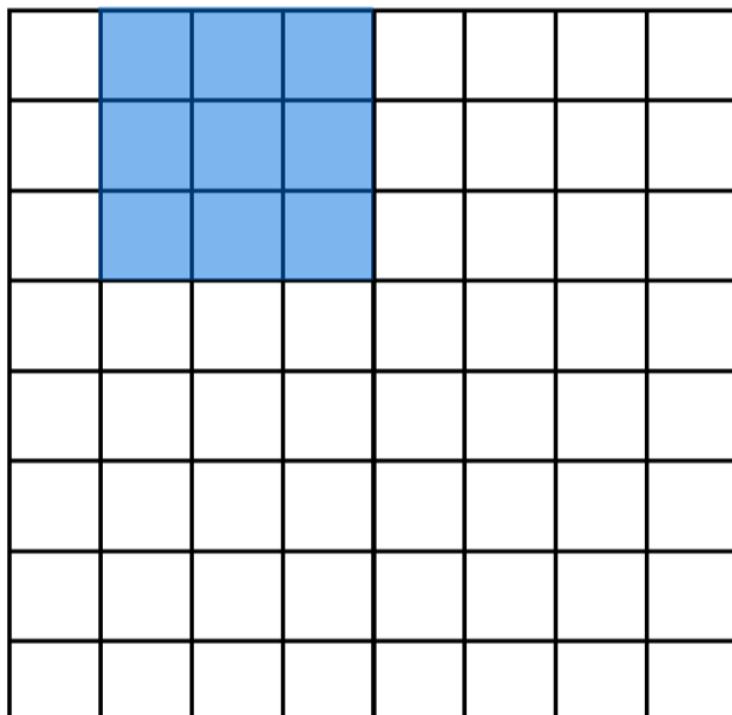
# Convolution operation: stride

During convolution, the weights “slide” along the input to generate each output

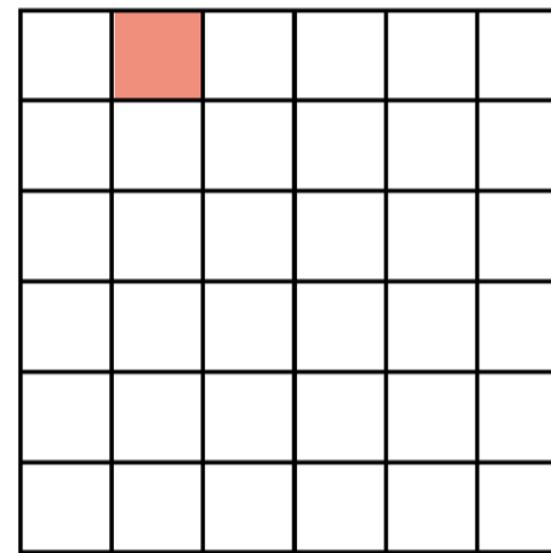


# Convolution operation: stride

During convolution, the weights “slide” along the input to generate each output



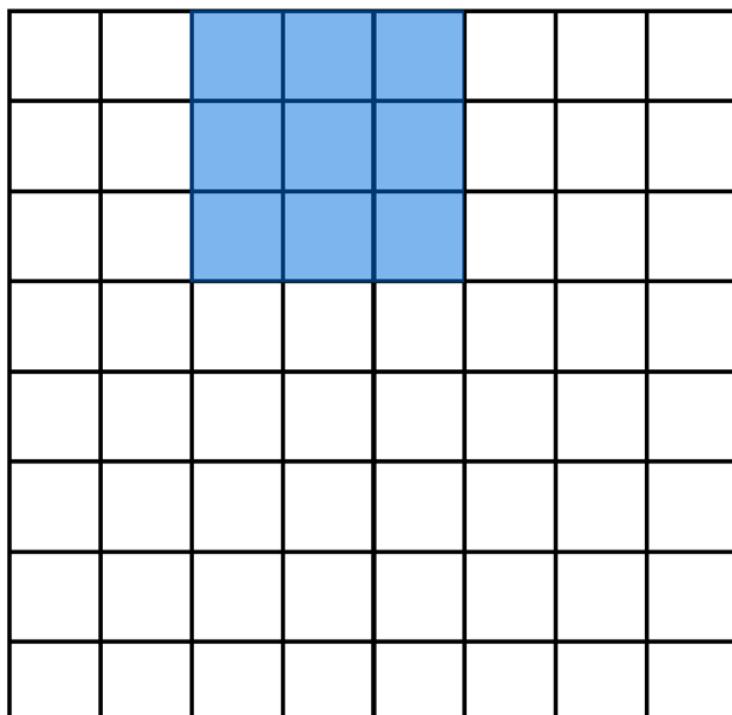
**Input**



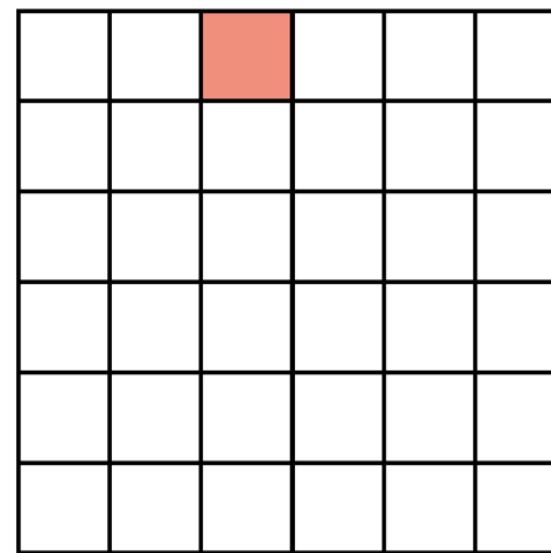
**Output**

# Convolution operation: stride

During convolution, the weights “slide” along the input to generate each output



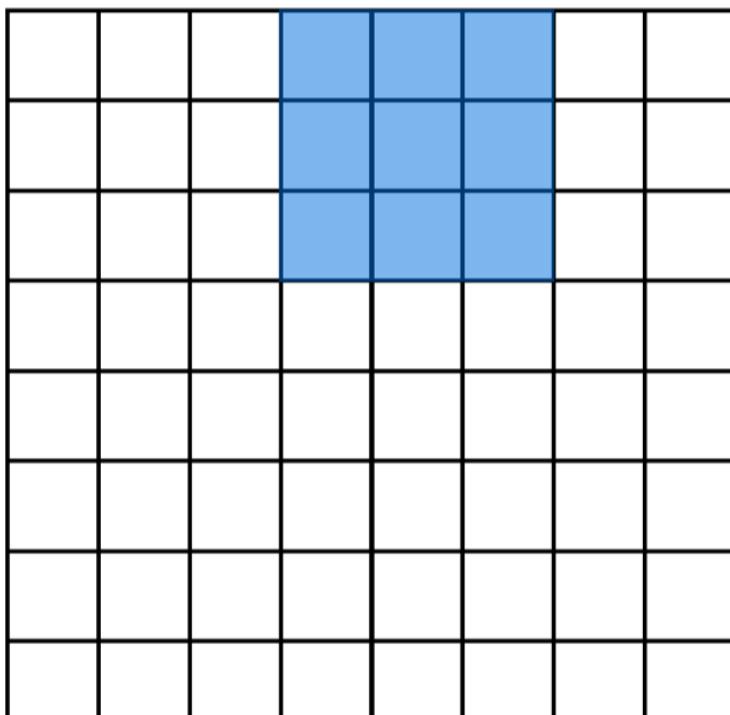
**Input**



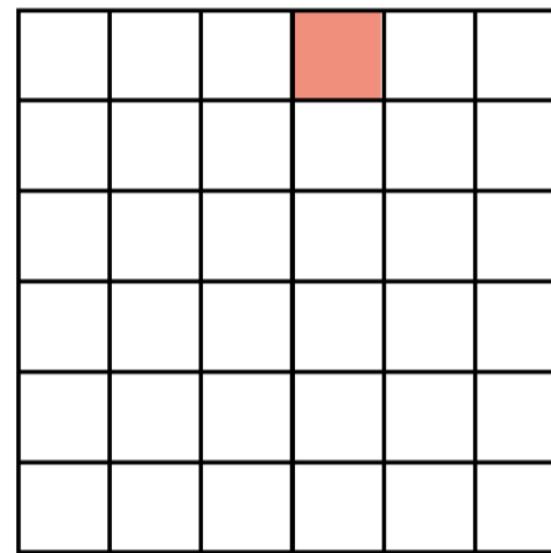
**Output**

# Convolution operation: stride

During convolution, the weights “slide” along the input to generate each output



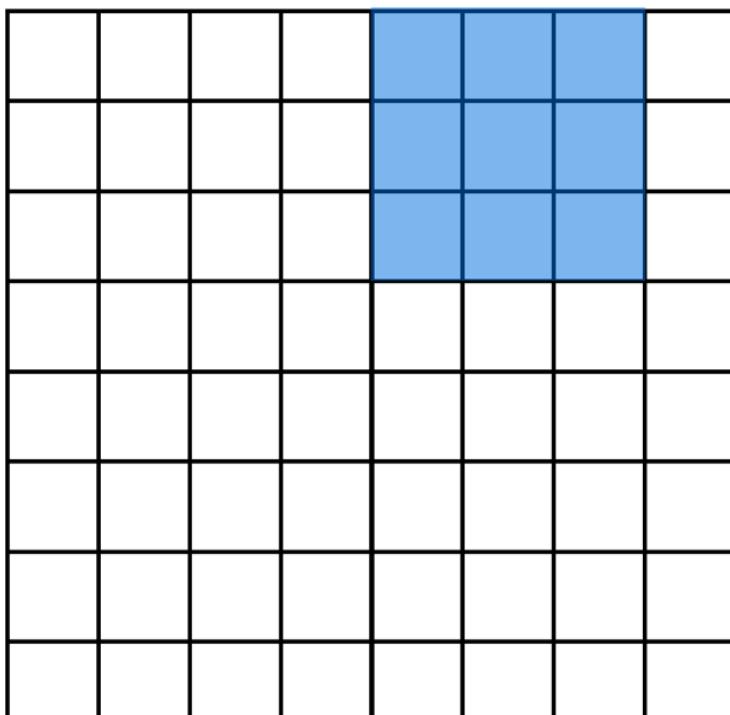
**Input**



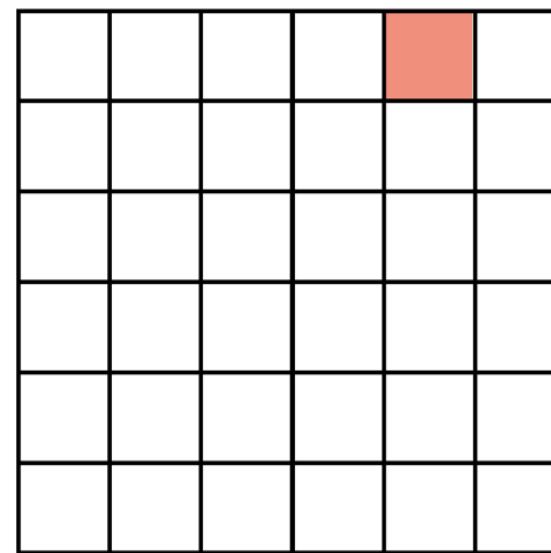
**Output**

# Convolution operation: stride

During convolution, the weights “slide” along the input to generate each output



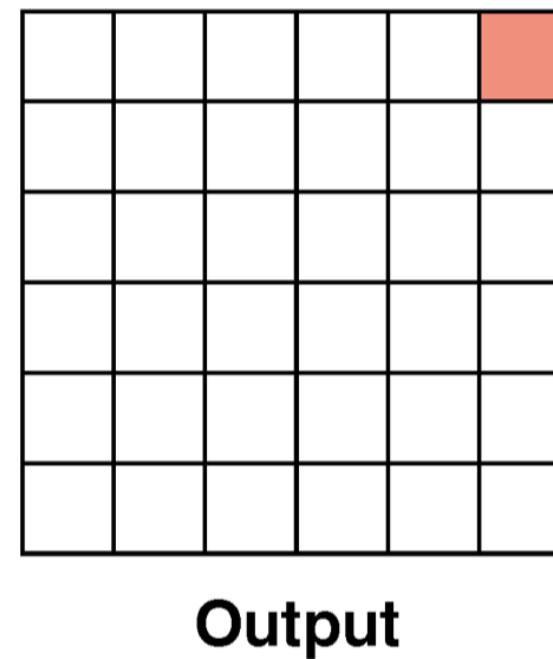
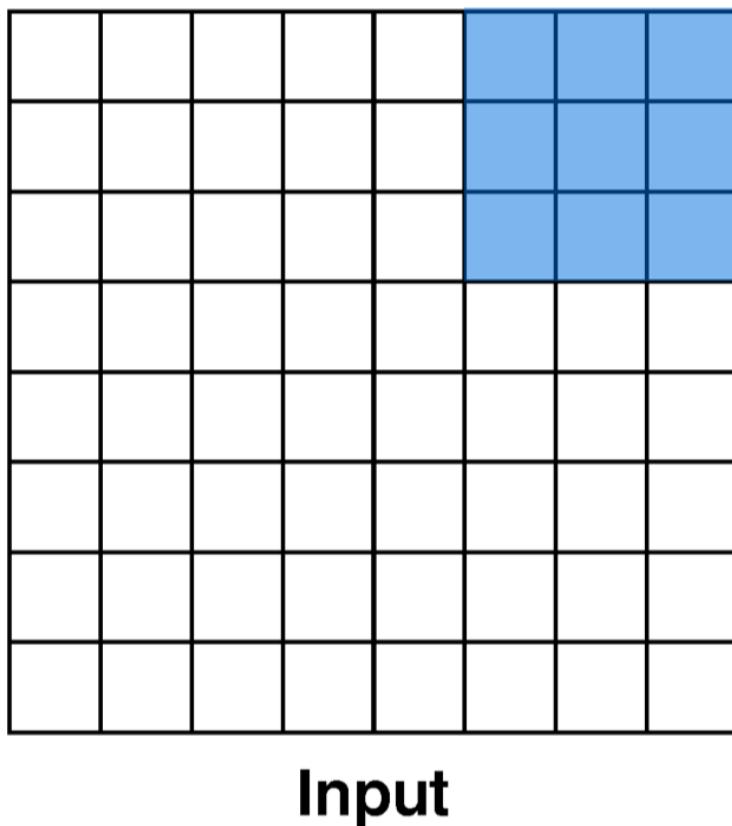
**Input**



**Output**

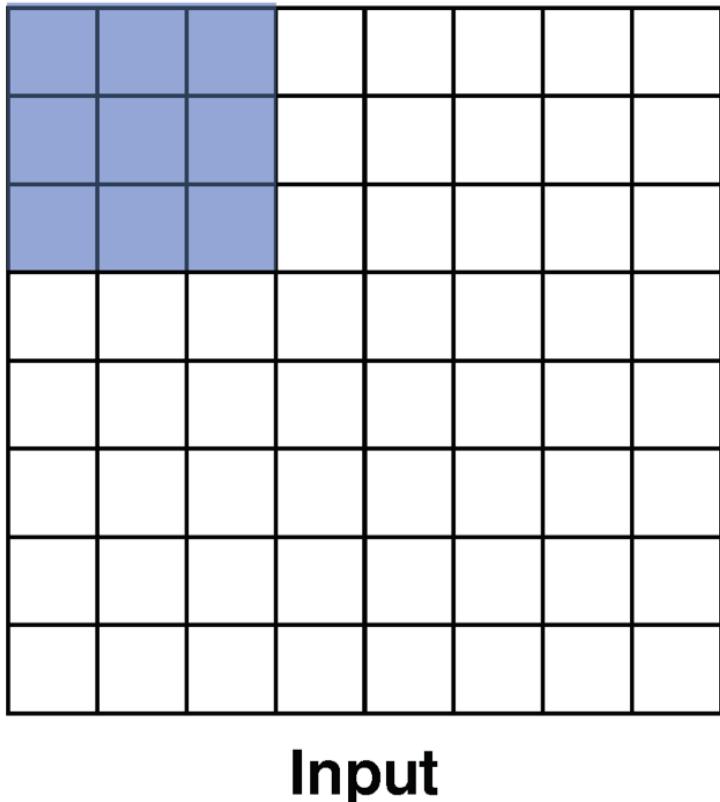
# Convolution operation: stride

During convolution, the weights “slide” along the input to generate each output



# Convolution operation: stride

During convolution, the weights “slide” along the input to generate each output



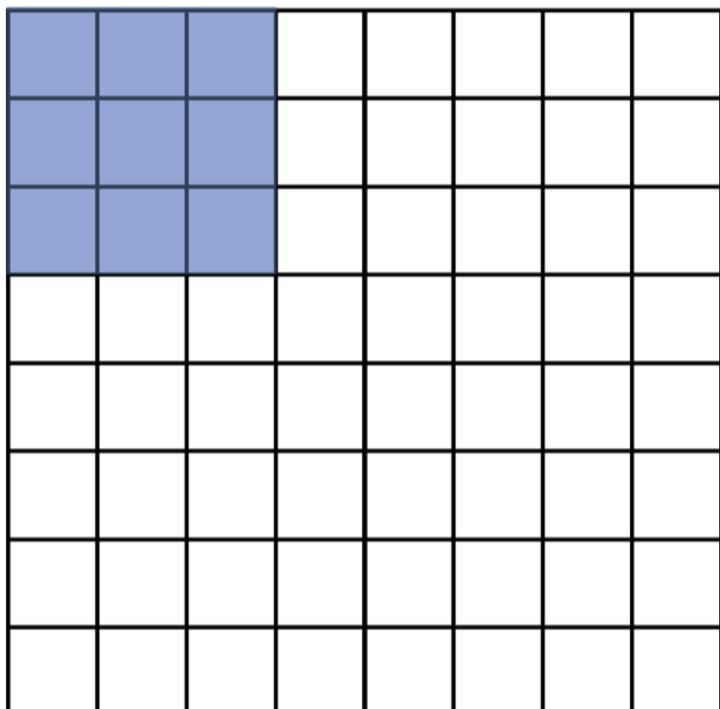
Recall that at each position,  
we are doing a **3D** sum:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$

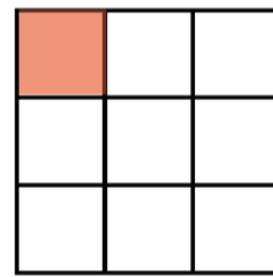
(channel, row, column)

# Convolution operation: stride

But we can also convolve with a **stride**, e.g. stride = 2



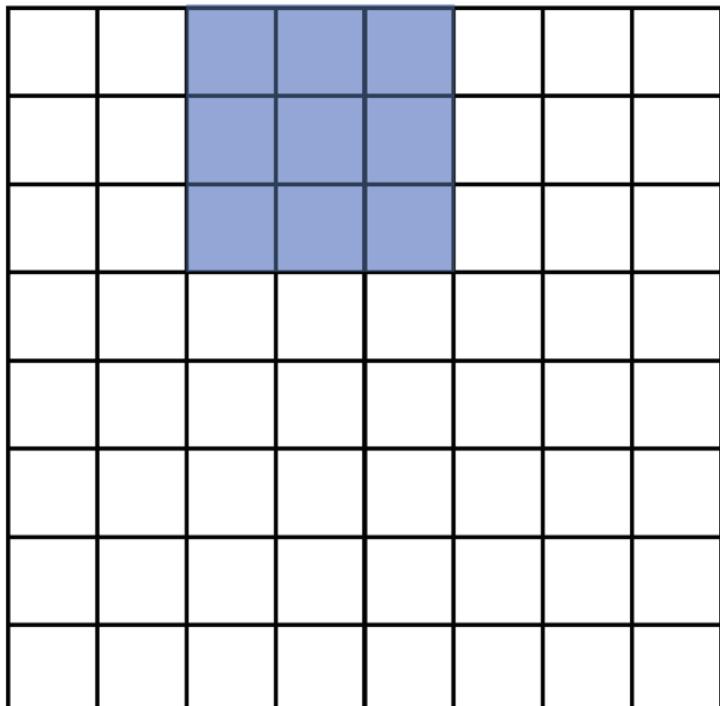
**Input**



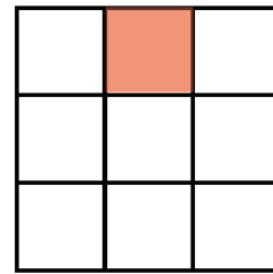
**Output**

# Convolution operation: stride

But we can also convolve with a **stride**, e.g. stride = 2



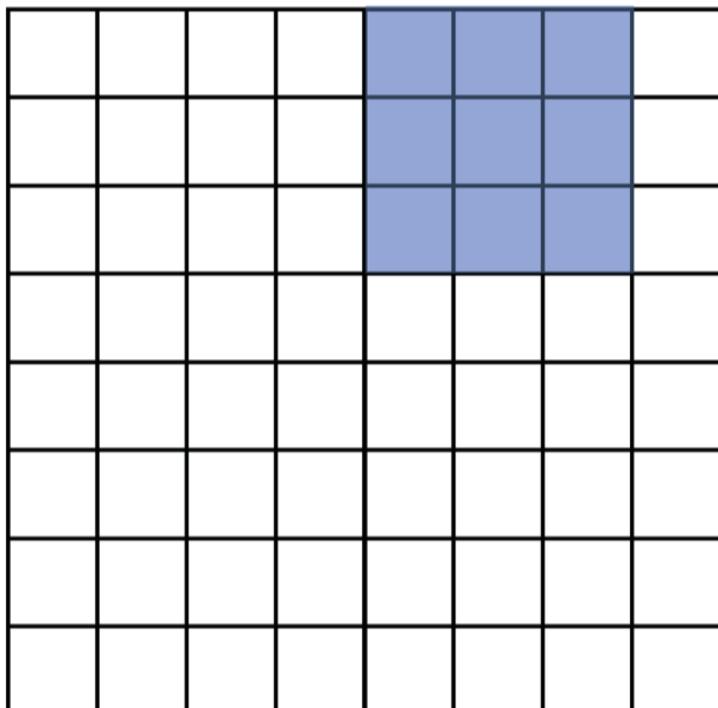
**Input**



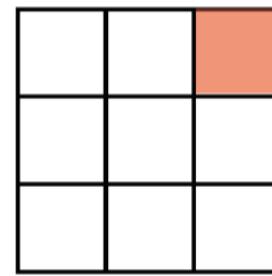
**Output**

# Convolution operation: stride

But we can also convolve with a **stride**, e.g. stride = 2



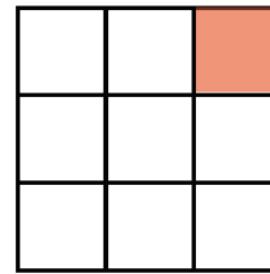
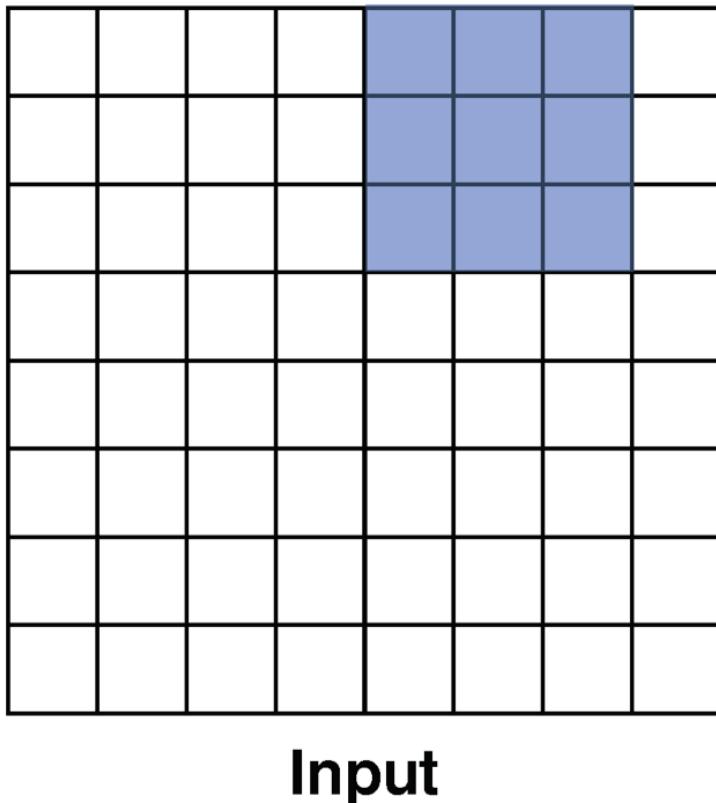
**Input**



**Output**

# Convolution operation: stride

But we can also convolve with a **stride**, e.g. stride = 2

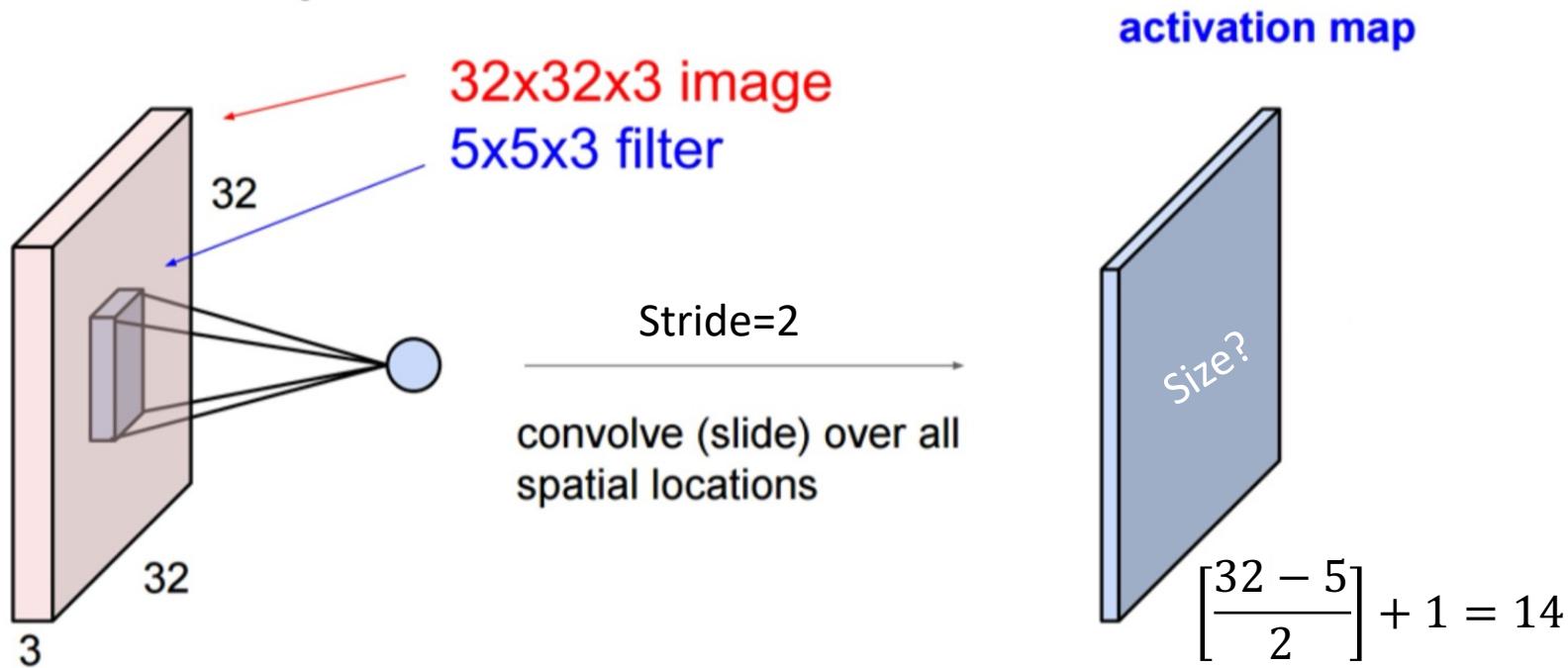


**Output**

- Notice that with certain strides, we may not be able to cover all of the input
- The output is also half the size of the input

# Convolution operation: stride

## Convolution Layer



# Convolution operation: padding

We can also pad the input with zeros.

Here, **pad = 1, stride = 2**

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

**Input**


**Output**

# Convolution operation: padding

We can also pad the input with zeros.

Here, **pad = 1, stride = 2**

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

**Input**


**Output**

# Convolution operation: padding

We can also pad the input with zeros.

Here, **pad = 1, stride = 2**

0	0	0	0	0	0	0	0	0	0
0									0
0									0
0									0
0									0
0									0
0									0
0									0
0	0	0	0	0	0	0	0	0	0

**Input**


**Output**

# Convolution operation: padding

We can also pad the input with zeros.

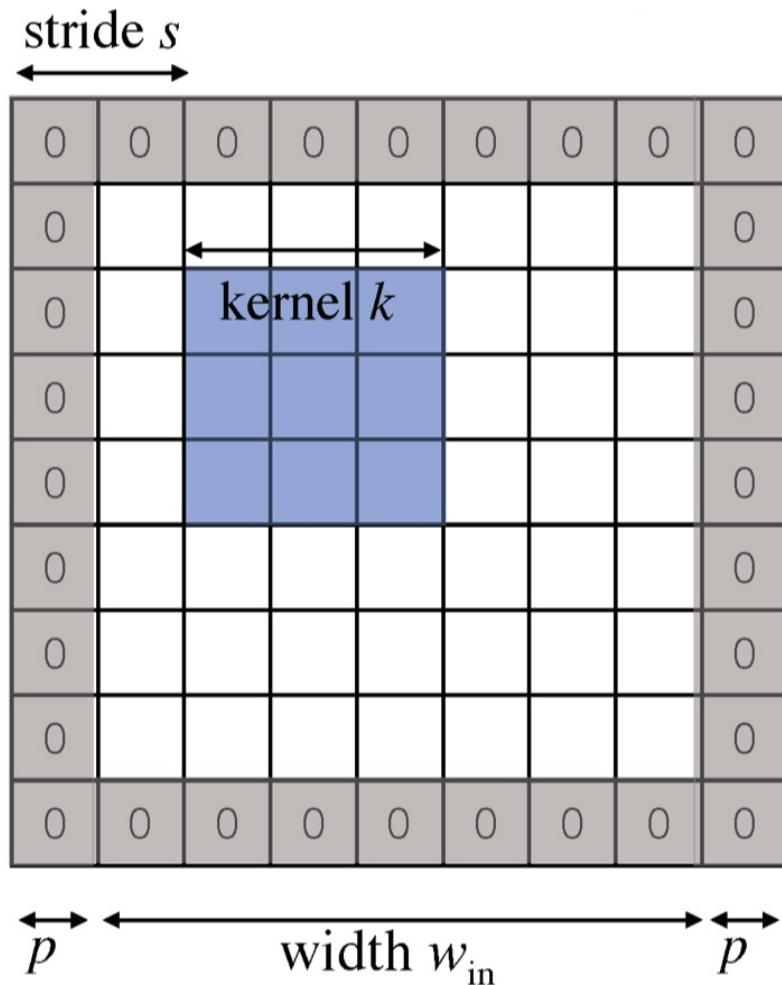
Here, **pad = 1, stride = 2**

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

**Input**


**Output**

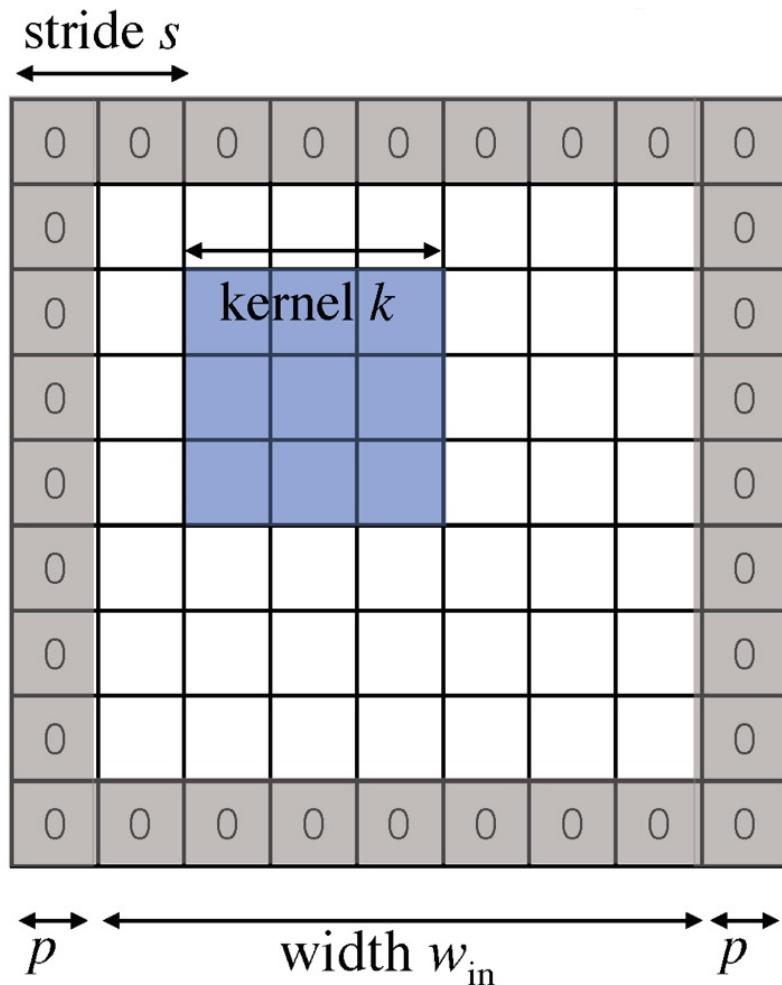
# Convolution: output size



In general, the output has size:

$$w_{\text{out}} = \left\lfloor \frac{w_{\text{in}} + 2p - k}{s} \right\rfloor + 1$$

# Convolution: output size

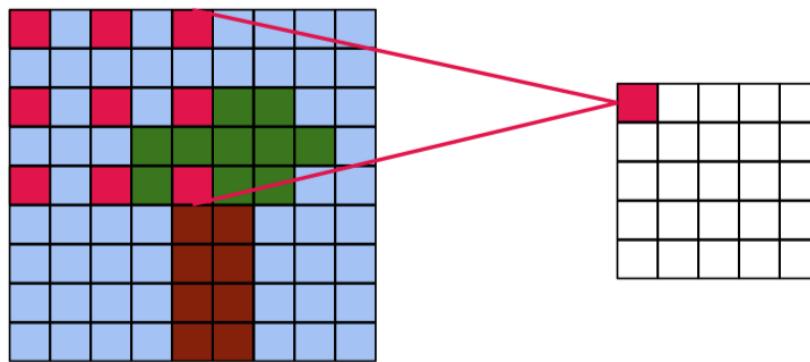


**Example:**  $k=3$ ,  $s=1$ ,  $p=1$

$$\begin{aligned}w_{\text{out}} &= \left\lfloor \frac{w_{\text{in}} + 2p - k}{s} \right\rfloor + 1 \\&= \left\lfloor \frac{w_{\text{in}} + 2 - 3}{1} \right\rfloor + 1 \\&= w_{\text{in}}\end{aligned}$$

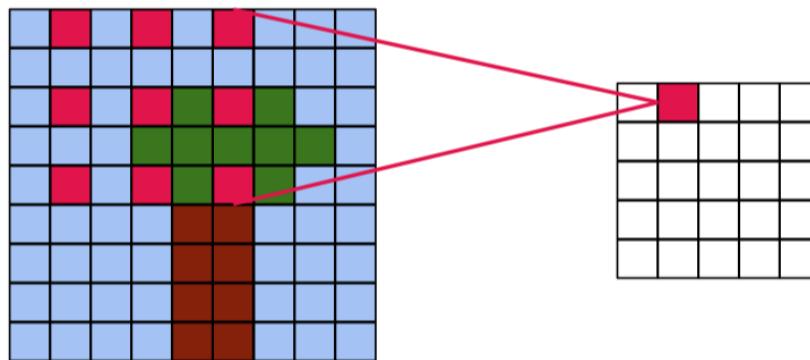
VGGNet [Simonyan 2014]  
uses filters of this shape

# Convolution operation: dilation



**Dilated convolution:** kernel is spread out, step > 1 between kernel elements

# Convolution operation: dilation

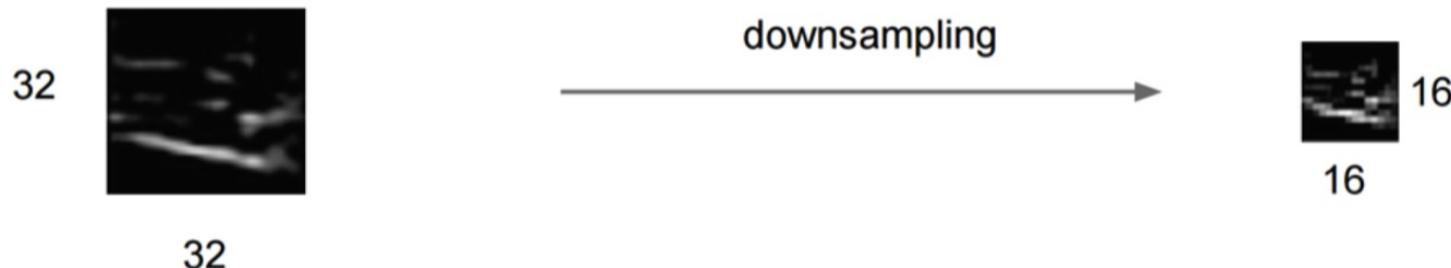


**Dilated convolution:** kernel is spread out, step > 1 between kernel elements

# Pooling

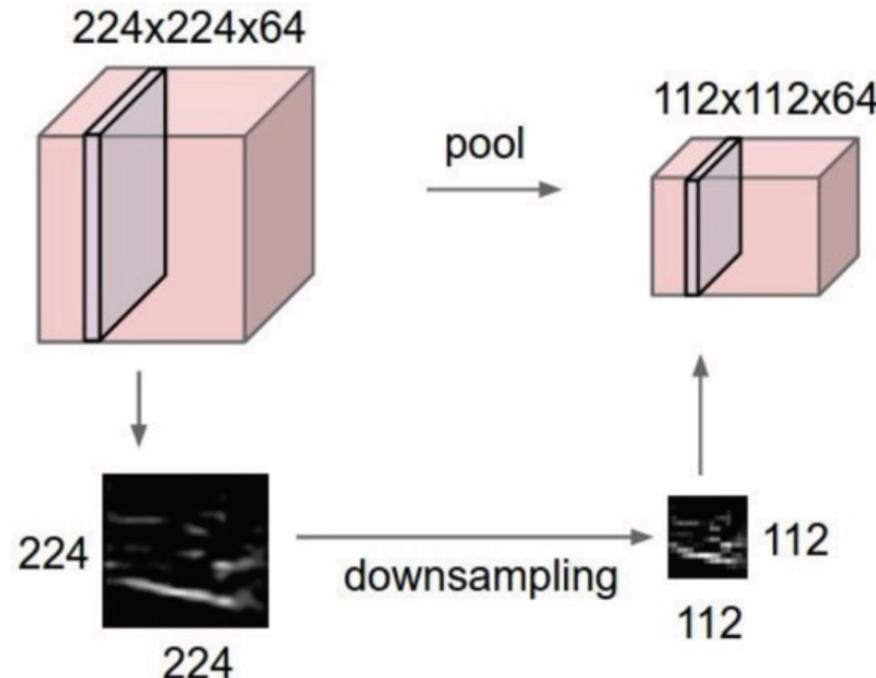
For most ConvNets, **convolution** is often followed by **pooling**:

- Creates a smaller representation while retaining the most important information
- The “max” operation is the most common
- Why might “avg” be a poor choice?

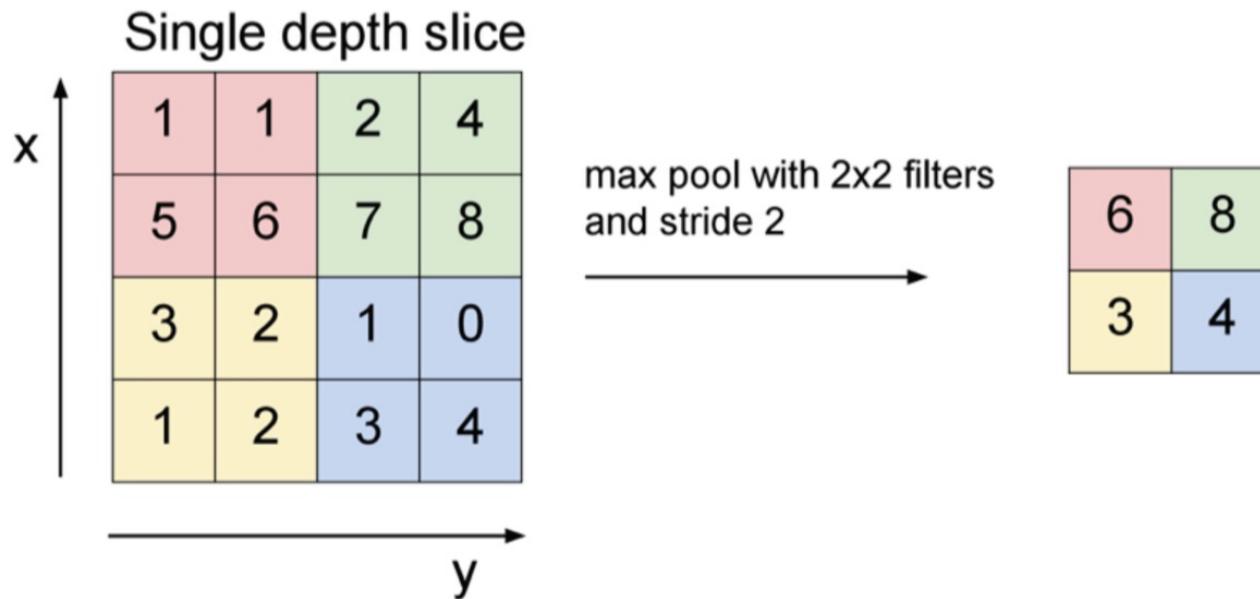


# Pooling

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



# Max Pooling



What's the backprop rule for max pooling?

- In the forward pass, store the index that took the max
- The backprop gradient is the input gradient at that index

# Computational building blocks of convnets



**fully connected**

**input**

**convolution**

**nonlinearity**

**pooling**

# Questions?