Convolutional Neural Networks

Layers used to build convnets

- a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function.

- We use three main types of layers to build ConvNet architectures:
  - Convolutional Layer
  - Pooling Layer
  - Fully-Connected Layer

- We will stack these layers to form a full ConvNet architecture.

Convolutional Layer

- The CONV layer's parameters consist of a set of learnable filters.
- Every filter is small spatially (along width and height) but extends through the full depth of the input volume.
- For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels).
- During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position.

Convolutional Layer

- As we slide the filter over the width and height of the input volume, we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position.

- Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network.

- Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map.

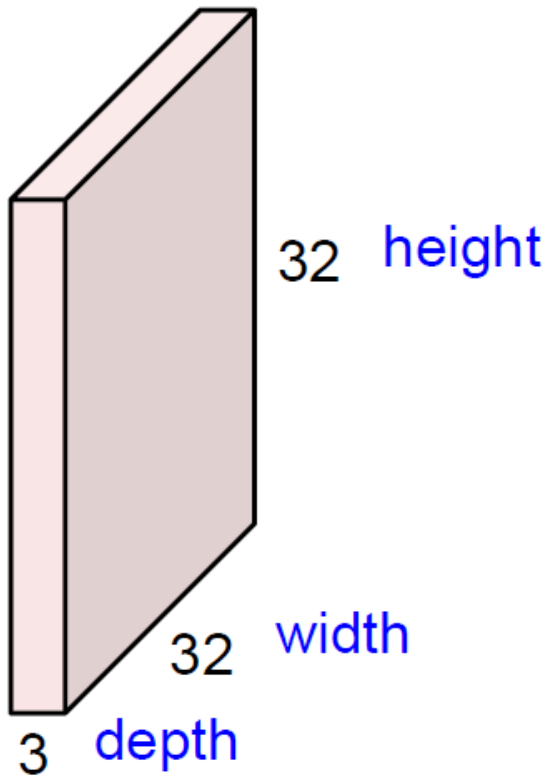
- We will stack these activation maps along the depth dimension and produce the output volume.

Convolutional layer

- **Local Connectivity**
  - When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume.
  - Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the **receptive field** of the neuron (equivalently this is the filter size).

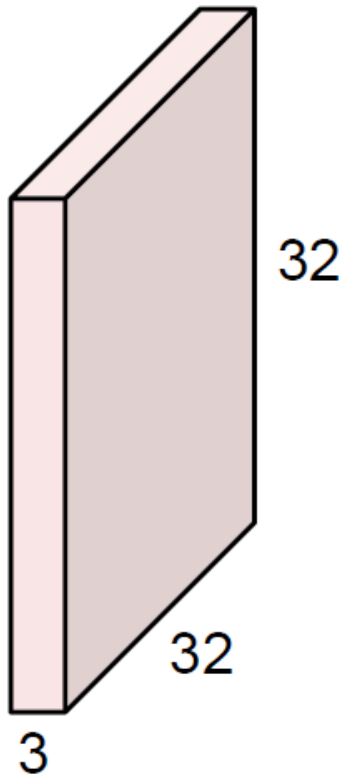
Convolution Layer

32x32x3 image -> preserve spatial structure

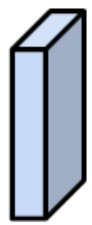


Convolution Layer

32x32x3 image



5x5x3 filter

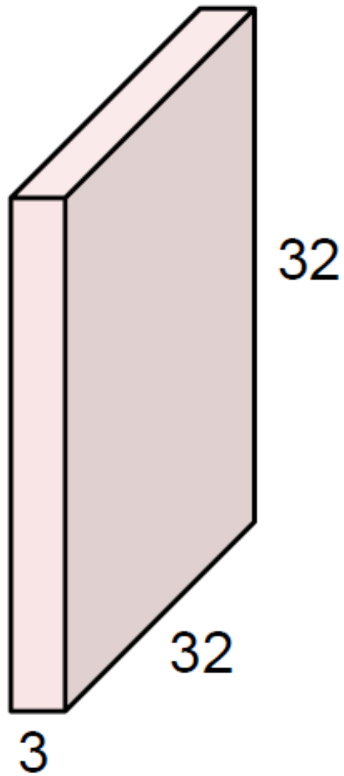


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

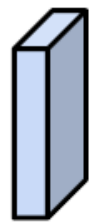
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

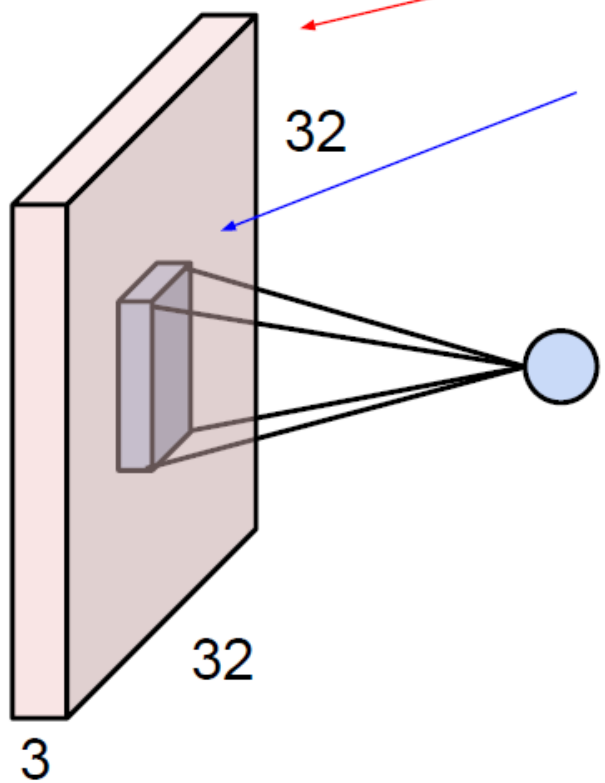


5x5x3 filter



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

Convolution Layer



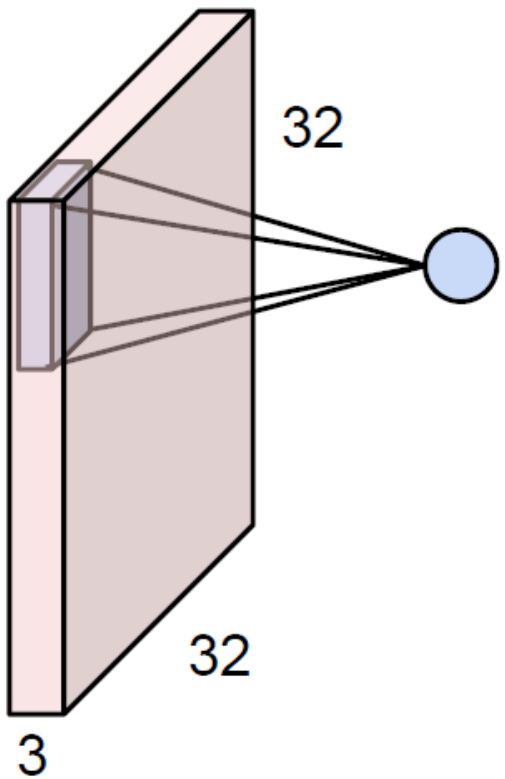
32x32x3 image

5x5x3 filter w

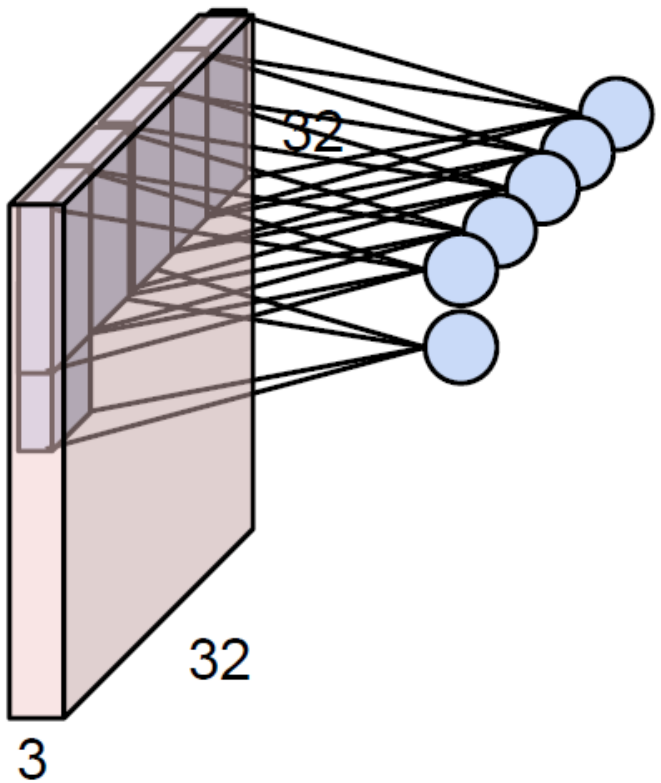
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

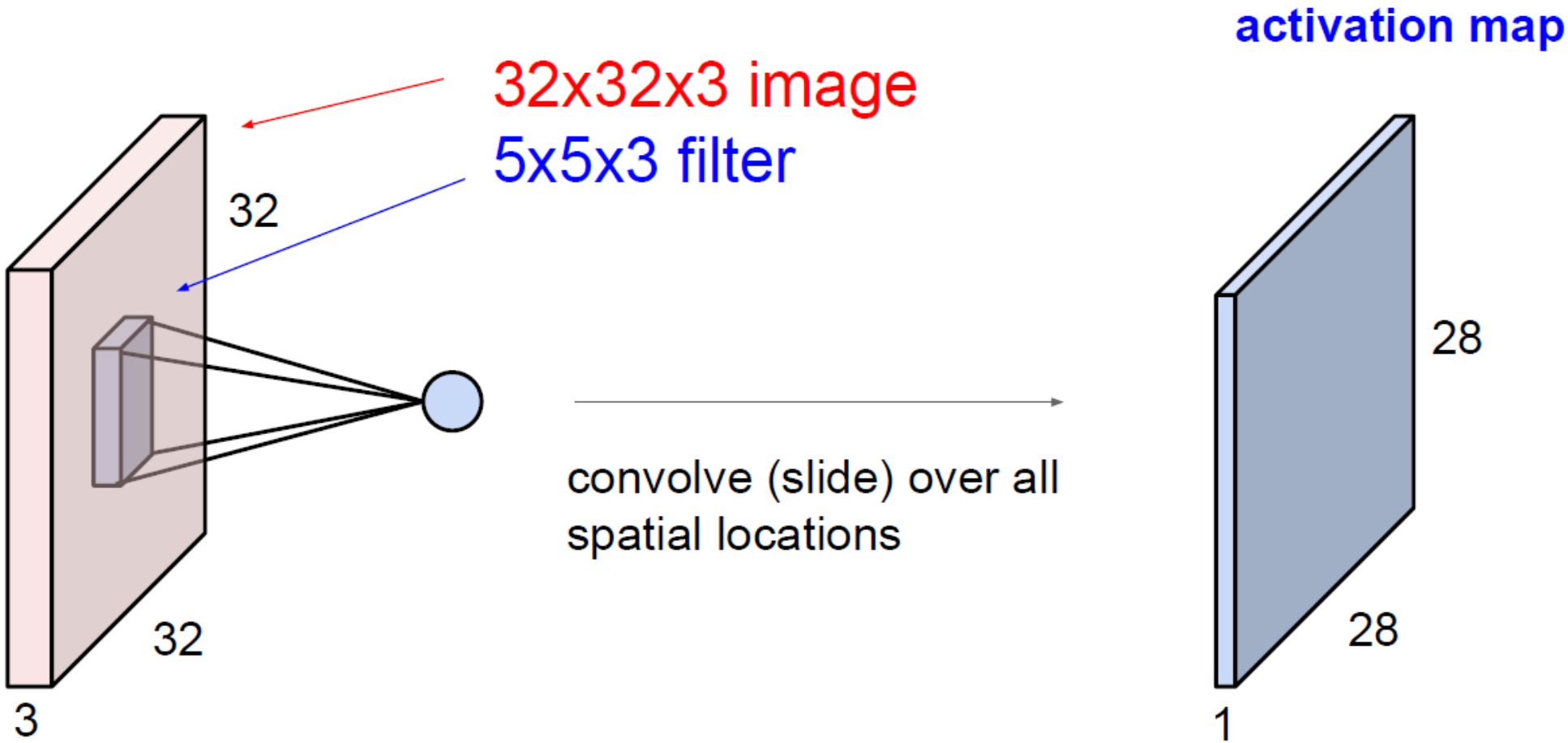
Convolution Layer



Convolution Layer

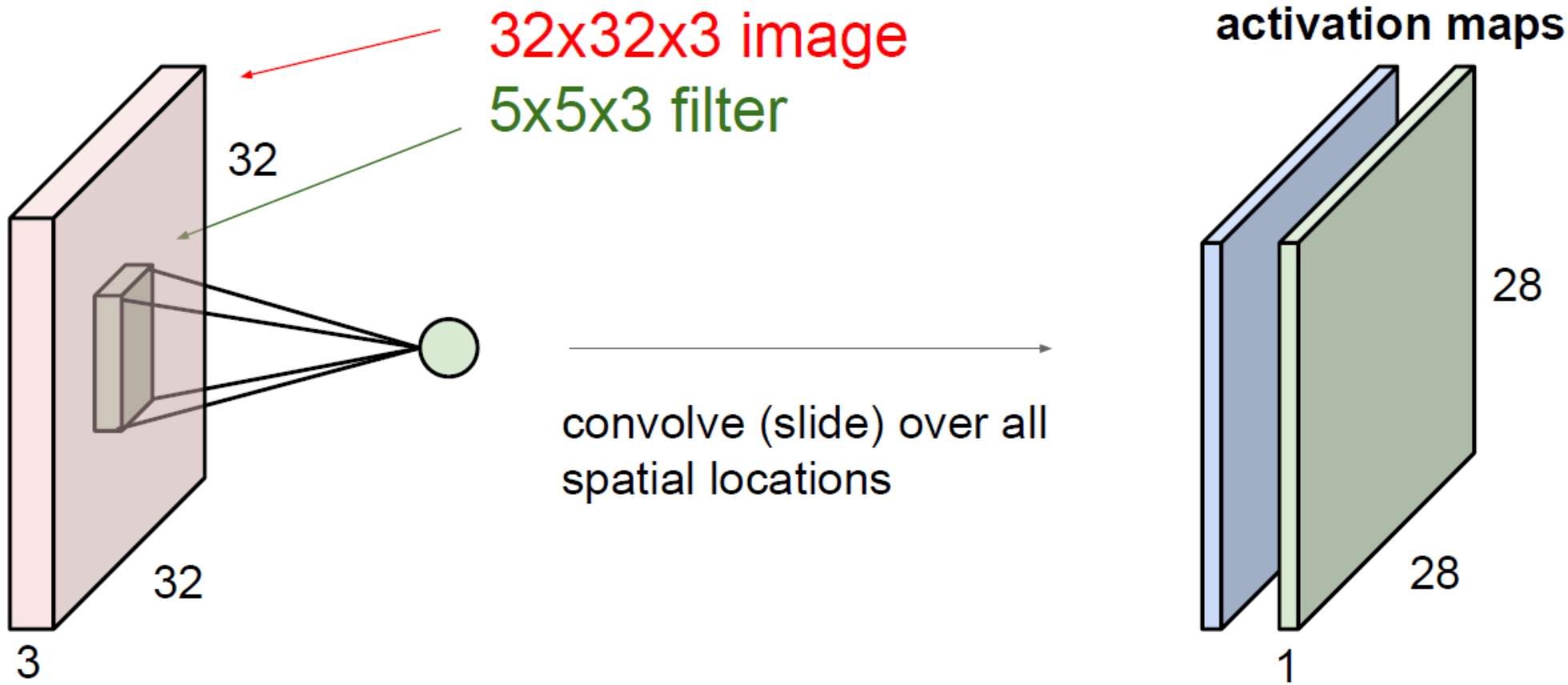


Convolution Layer

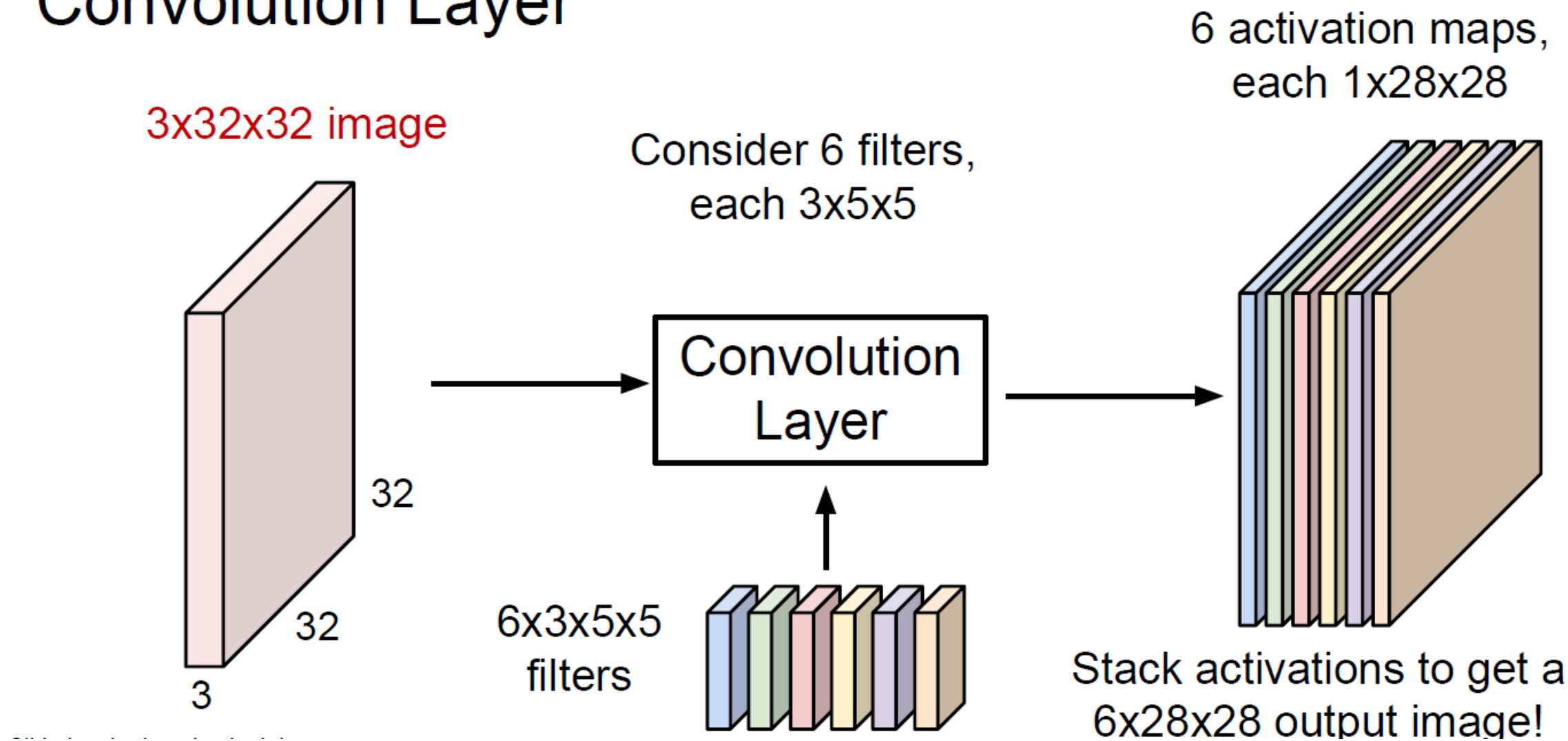


Convolution Layer

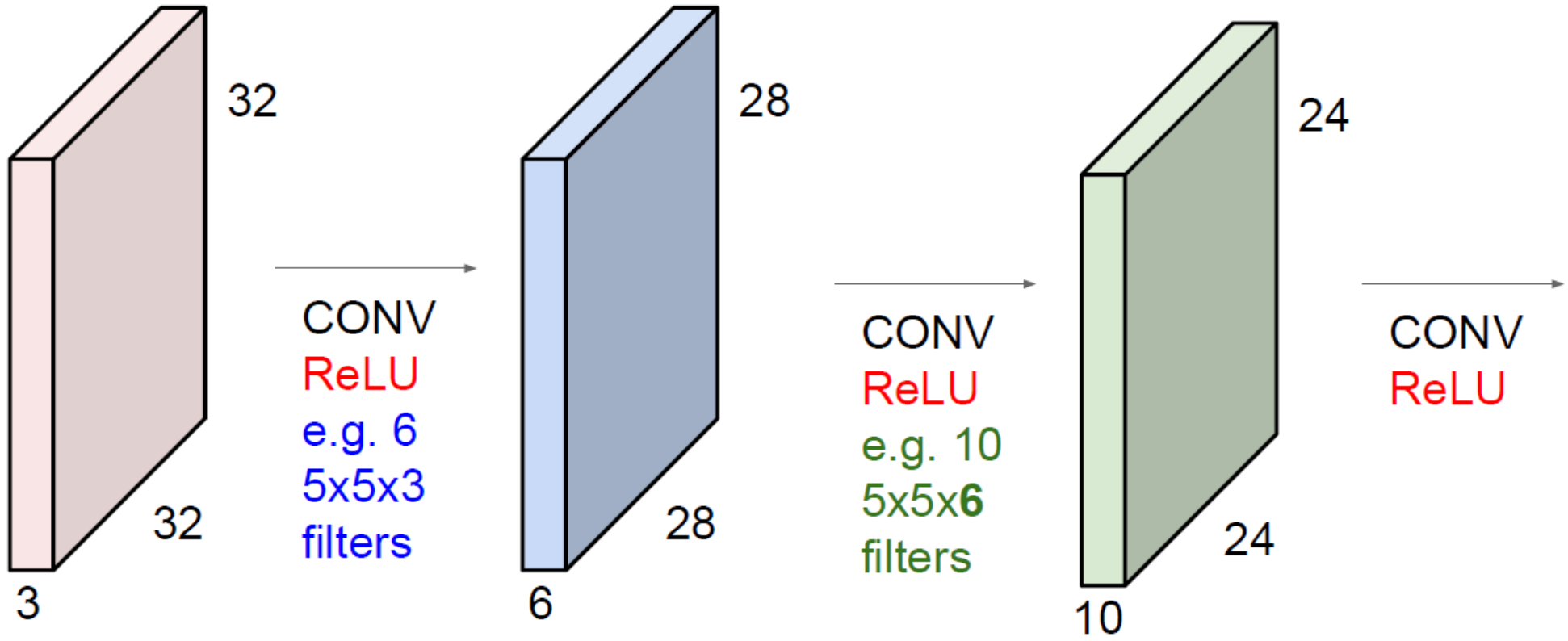
consider a second, green filter



Convolution Layer



ConvNet is a sequence of Convolution Layers, with **activation functions**

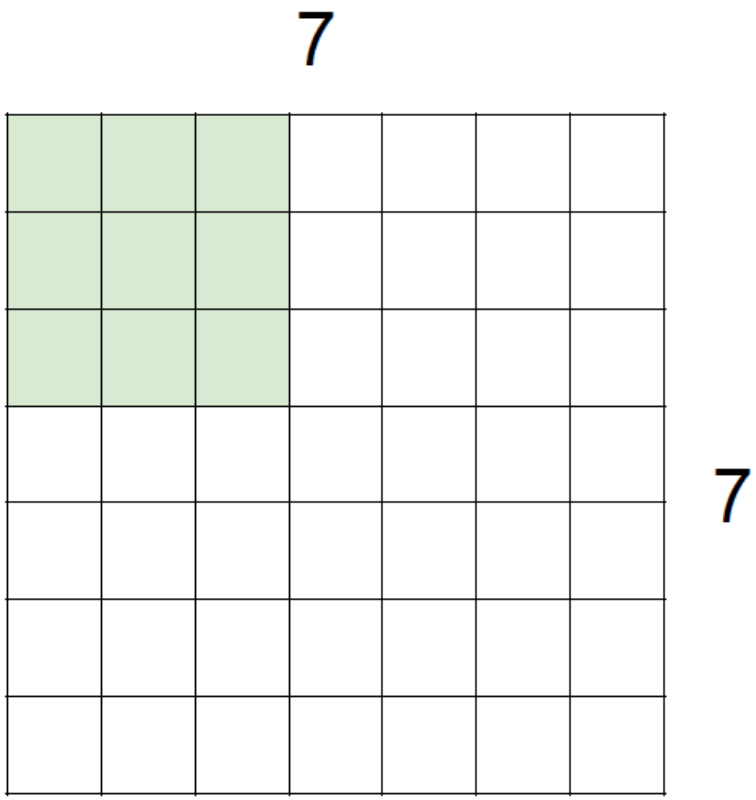


Spatial arrangement

- We have explained the connectivity of each neuron in the Conv Layer to the input volume, but we haven’t yet discussed how many neurons there are in the output volume or how they are arranged.

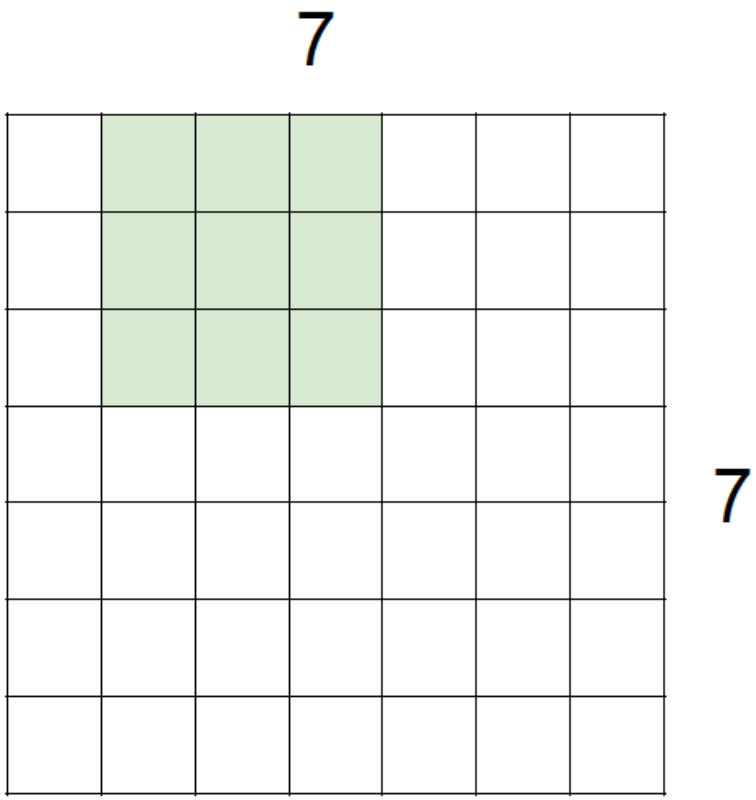
- Three hyperparameters control the size of the output volume:
  - depth
  - stride
  - zero-padding

Spatial arrangement



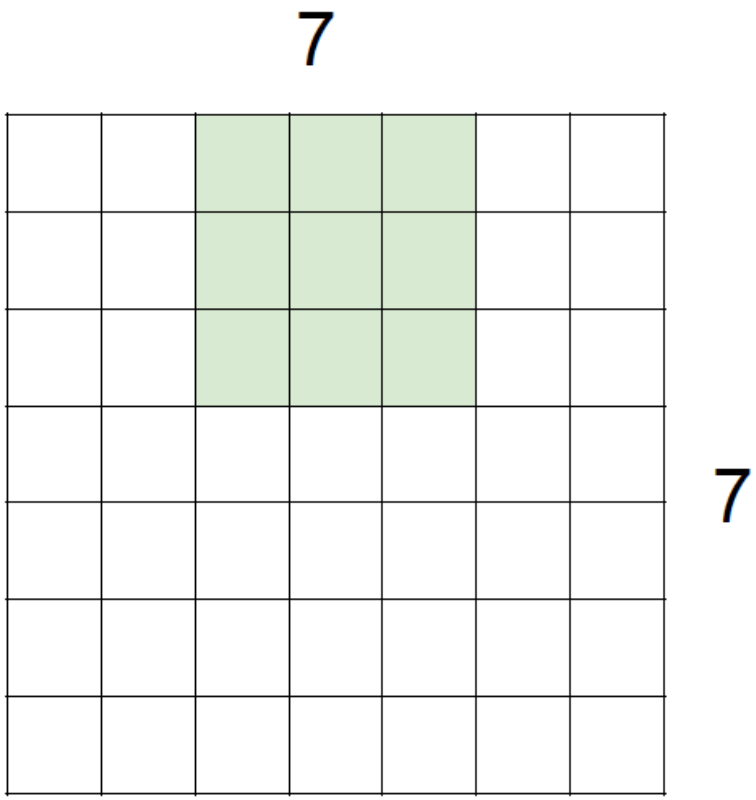
7x7 input (spatially)
assume 3x3 filter

Spatial arrangement



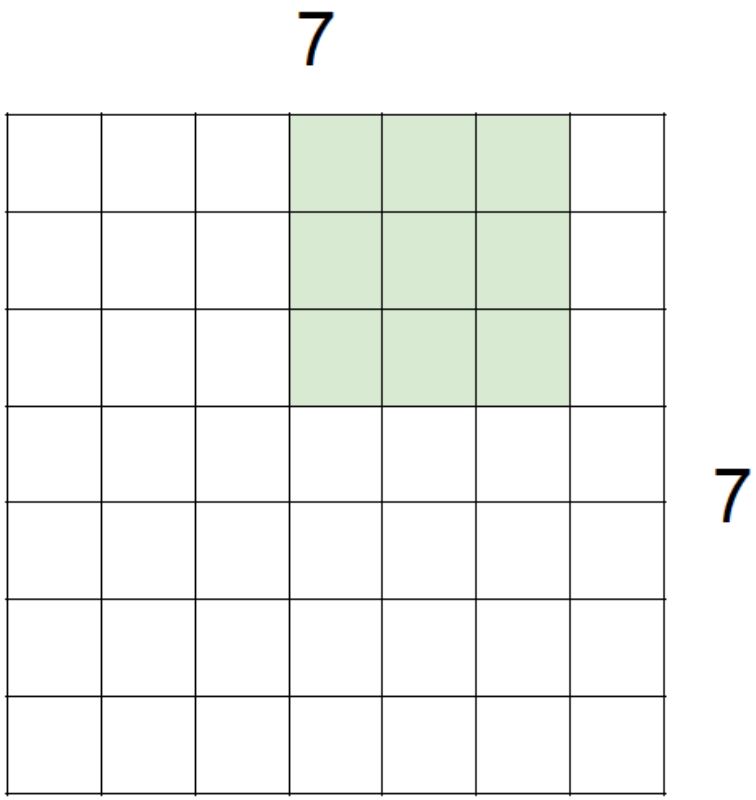
7x7 input (spatially)
assume 3x3 filter

Spatial arrangement



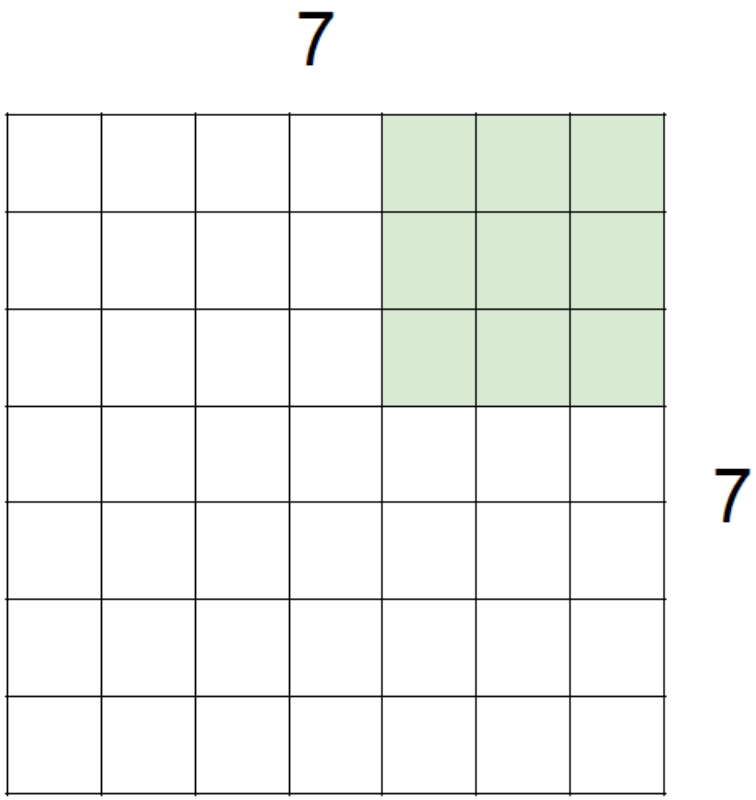
7x7 input (spatially)
assume 3x3 filter

Spatial arrangement



7x7 input (spatially)
assume 3x3 filter

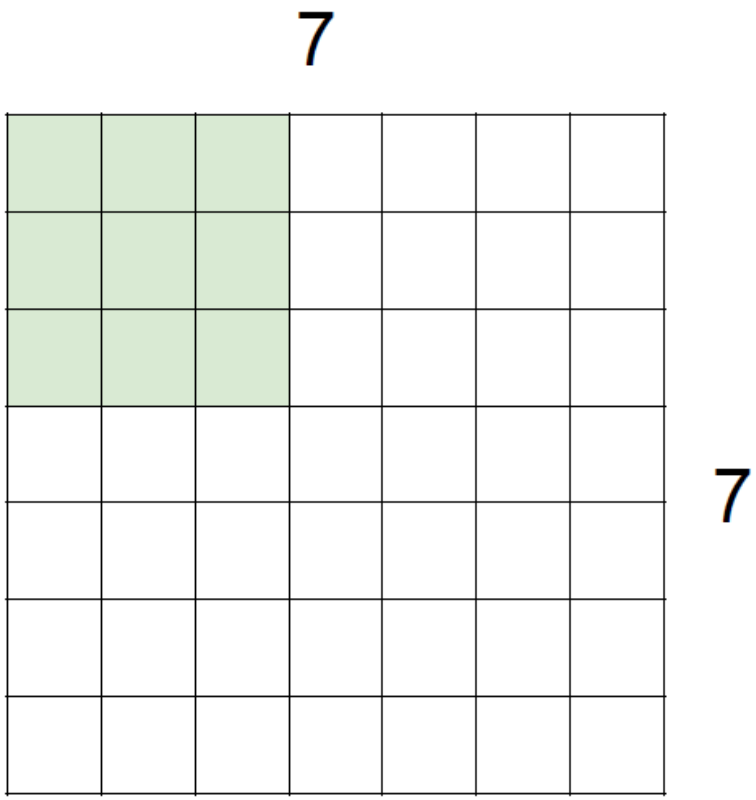
Spatial arrangement



7x7 input (spatially)
assume 3x3 filter

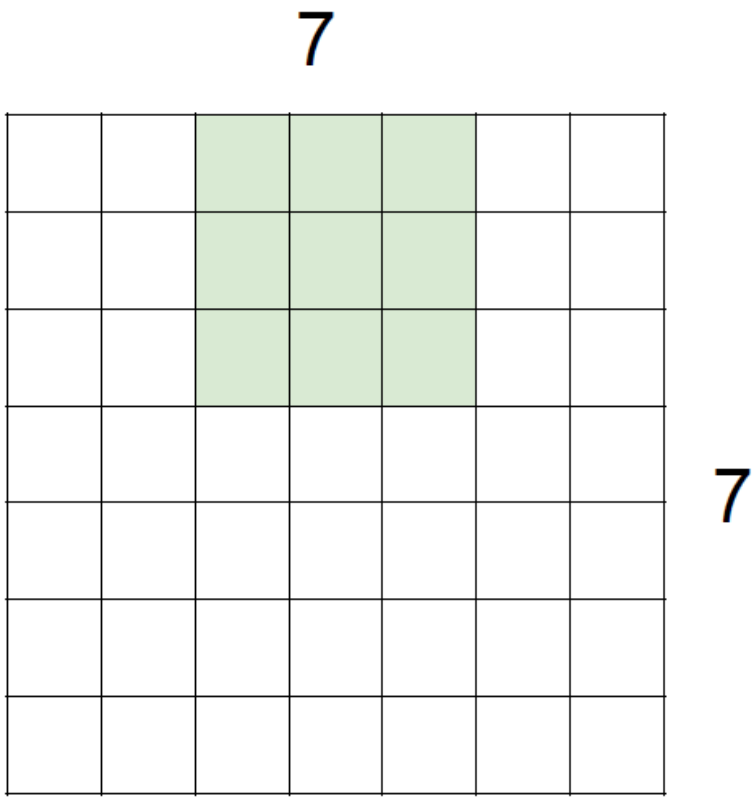
=> 5x5 output

Spatial arrangement



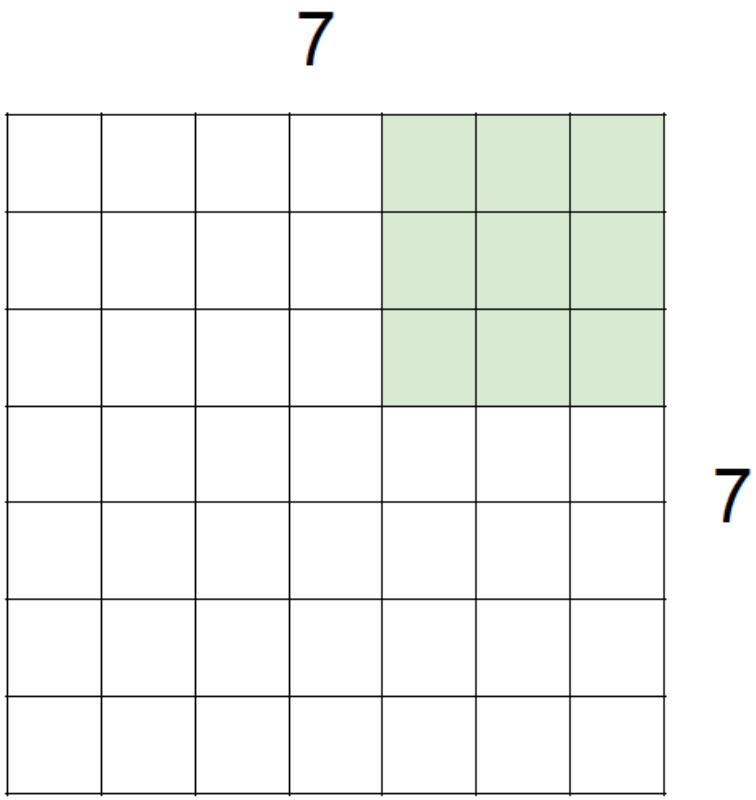
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

Spatial arrangement



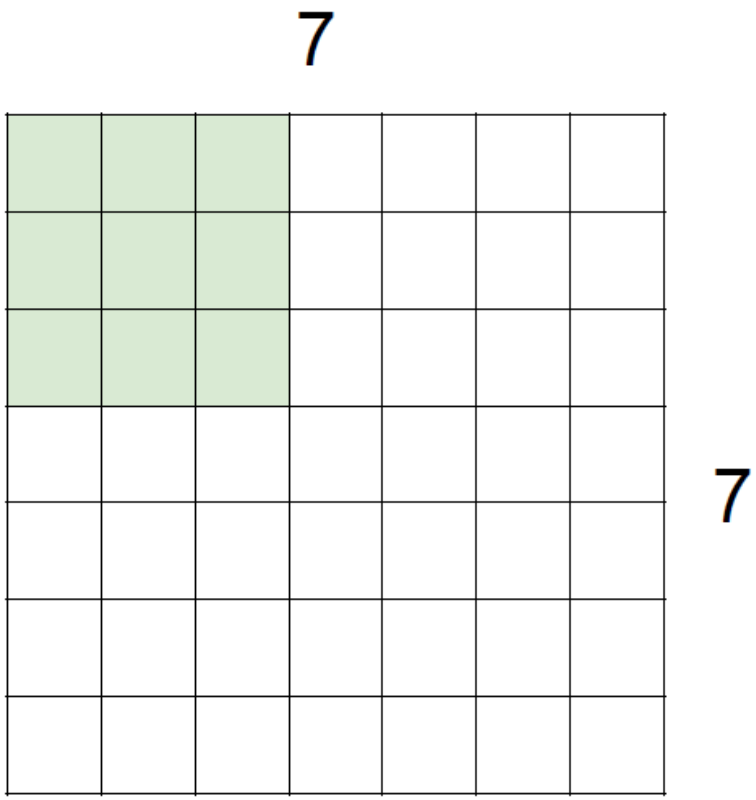
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

Spatial arrangement



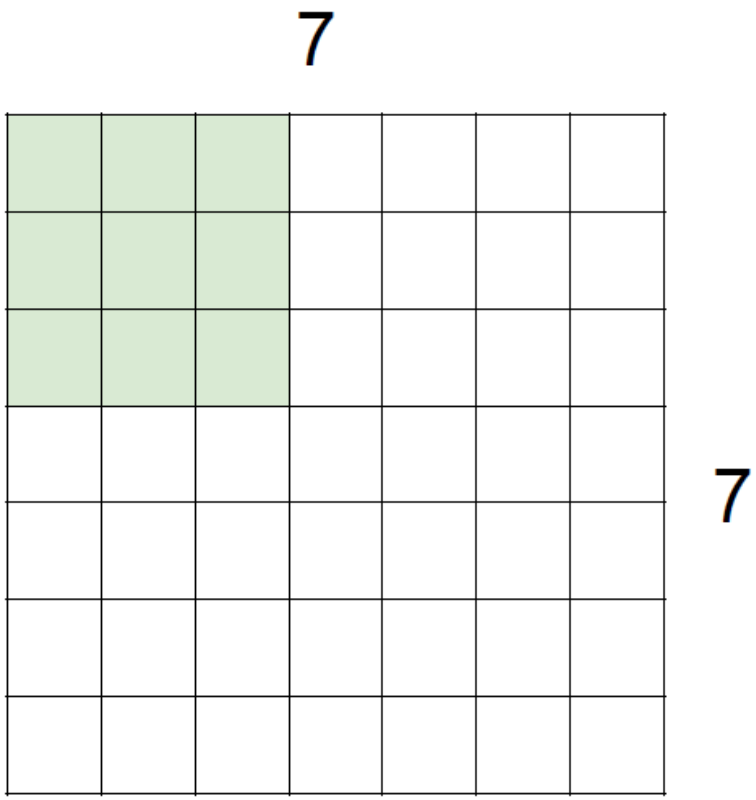
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

Spatial arrangement



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

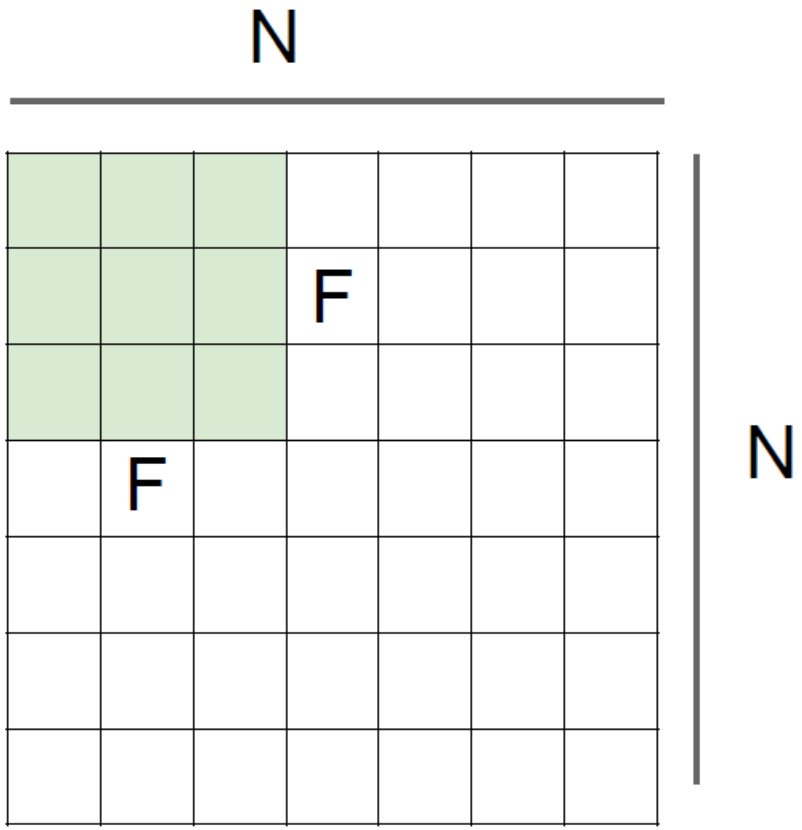
Spatial arrangement



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.

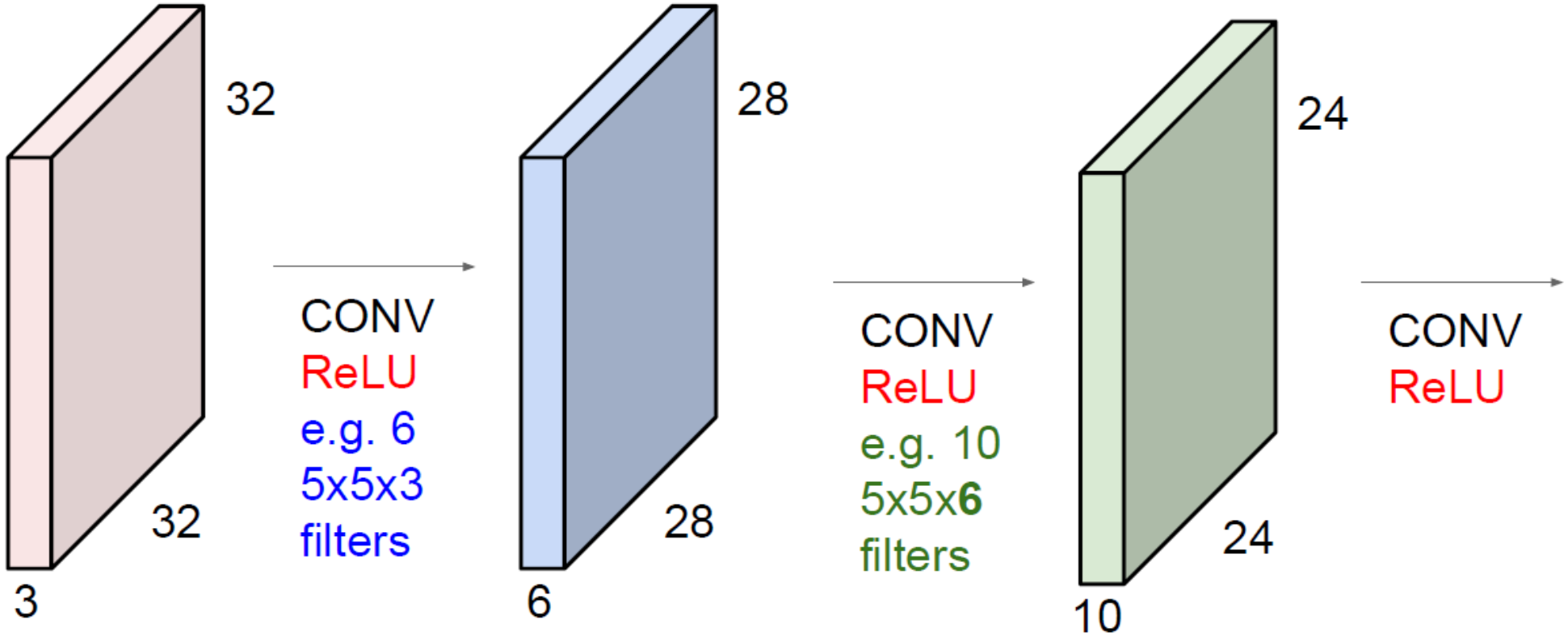
Spatial arrangement



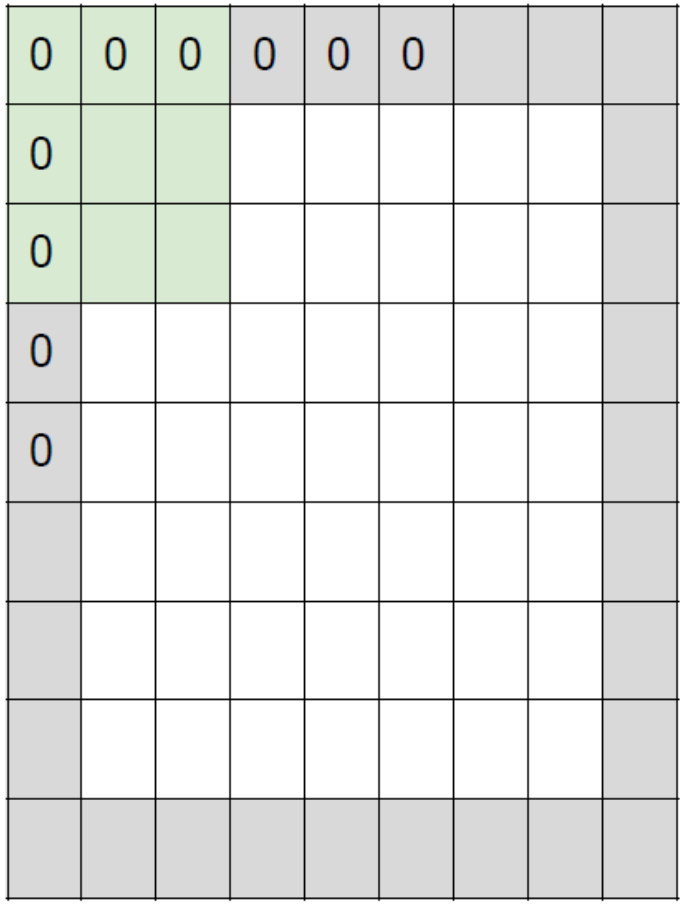
Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

Remember back to…
E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially!
(32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.



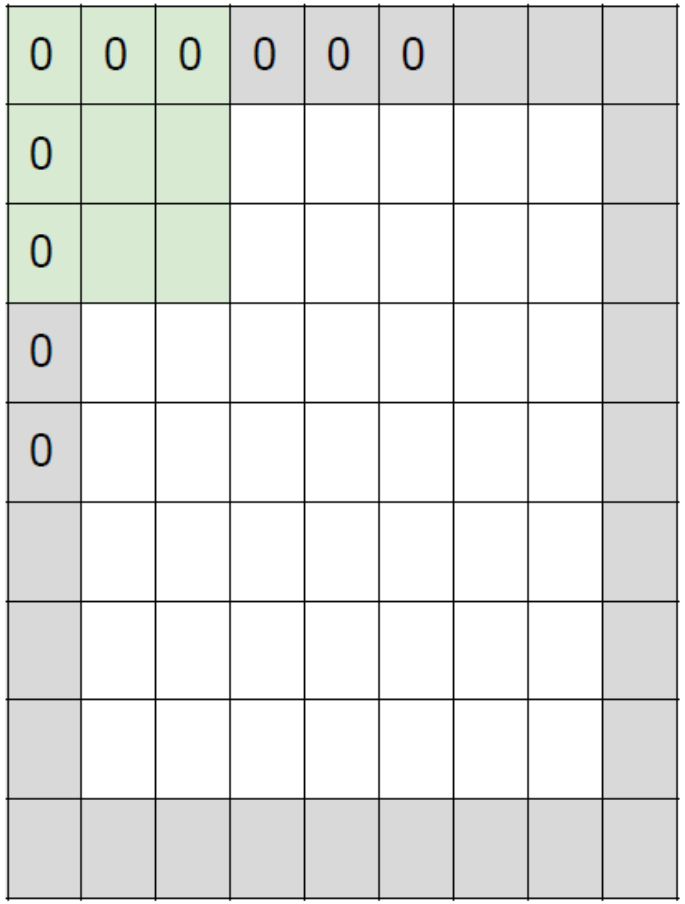
In practice: Common to zero pad the border



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

(recall:)
(N - F) / stride + 1

In practice: Common to zero pad the border

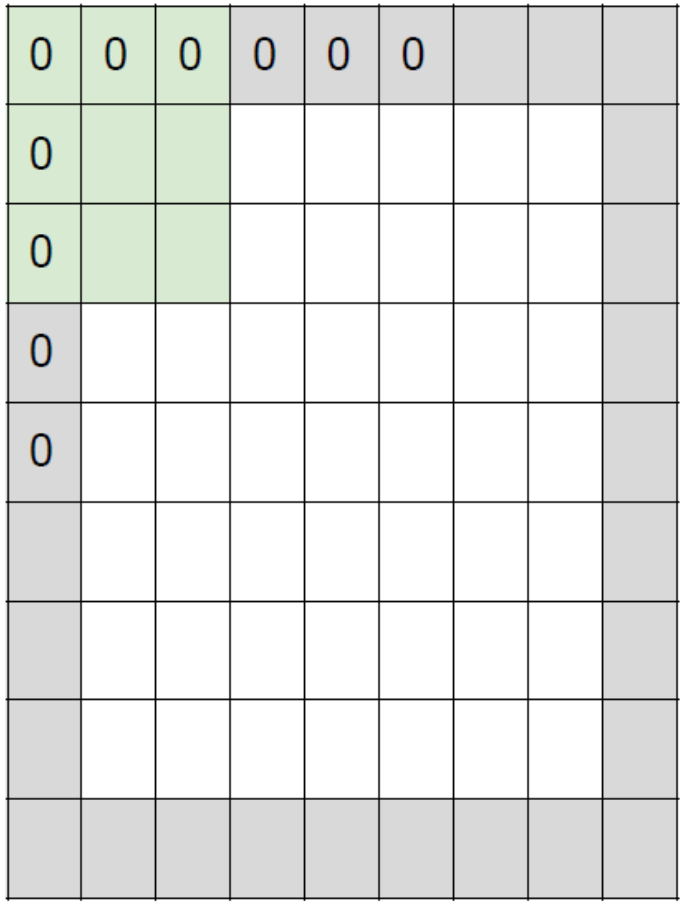


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

(recall:)
(N + 2P - F) / stride + 1

In practice: Common to zero pad the border



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

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Conv Layer: summary

- Let's assume input is W_1 x H_1 x C
- Conv layer needs 4 hyperparameters:
  - Number of filters K
  - The filter size F
  - The stride S
  - The zero padding P

- This will produce an output of W_2 x H_2 x K
- where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$

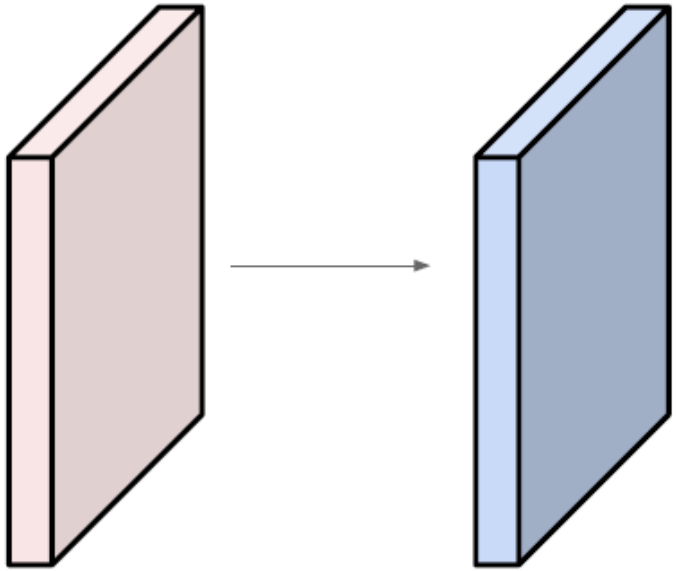
- Number of parameters: F^2CK and K biases

Examples

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Output volume size?



Examples

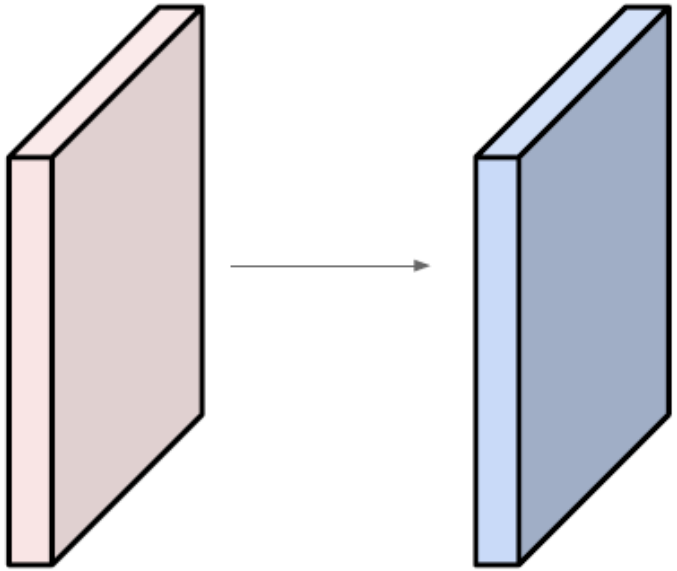
Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Output volume size?

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

32x32x10

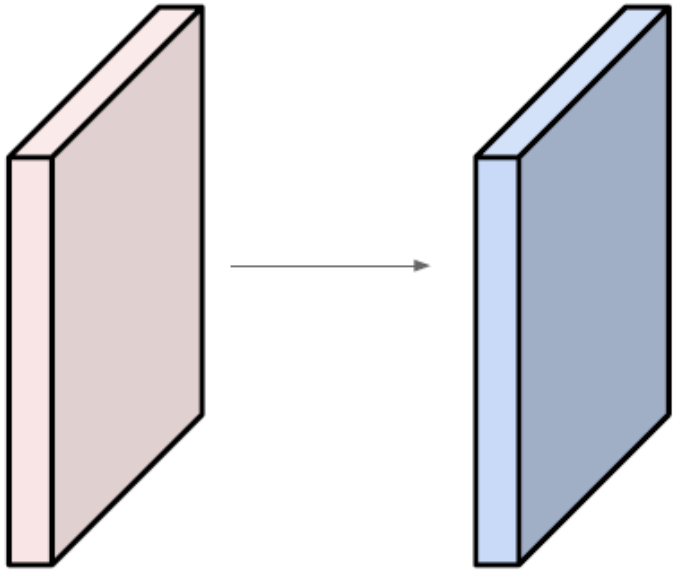


Examples

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

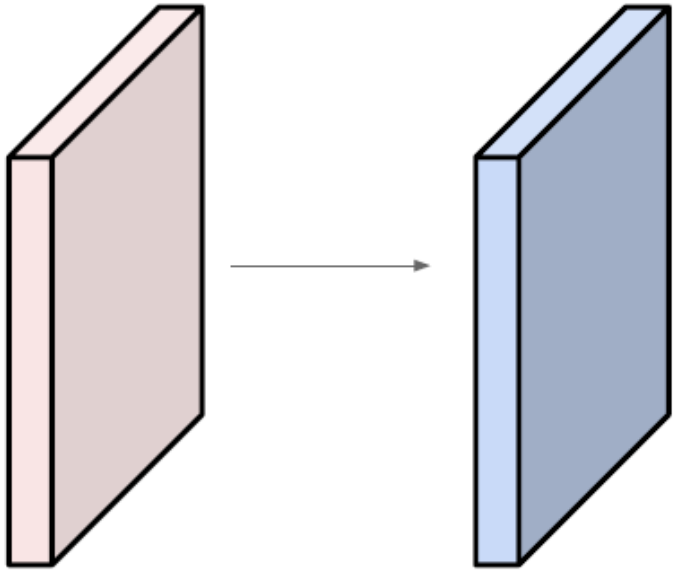
Number of parameters in this layer?



Examples

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2



Number of parameters in this layer?

each filter has 5*5*3 + 1 = 76 params (+1 for bias)

=> 76*10 = **760**

Real-world example

- The Krizhevsky et al. architecture that won the ImageNet challenge in 2012 accepted images of size [227x227x3].

- On the first Convolutional Layer, it used neurons with receptive field size F=11, stride S=4 and no zero padding P=0.

- Since (227 - 11)/4 + 1 = 55, and since the Conv layer had a depth of K=96, the Conv layer output volume had size [55x55x96]. Each of the 55*55*96 neurons in this volume was connected to a region of size [11x11x3] in the input volume.

- Moreover, all 96 neurons in each depth column are connected to the same [11x11x3] region of the input, but with different weights!!

Parameter Sharing

- Using the real-world example above, we see that there are 55*55*96 = 290,400 neurons in the first Conv Layer, and each has 11*11*3 = 363 weights and 1 bias. Together, this adds up to 290400 * 364 = 105,705,600 parameters on the first layer of the ConvNet alone. Clearly, this number is very high.

- It turns out that we can dramatically reduce the number of parameters by making one reasonable assumption: That if one feature is useful to compute at some spatial position (x,y), then it should also be useful to compute at a different position (x_2,y_2).

- With this parameter sharing scheme, the first Conv Layer in our example would now have only 96 unique set of weights (one for each depth slice), for a total of 96*11*11*3 = 34,848 unique weights, or 34,944 parameters (+96 biases).

Parameter Sharing

- **Parameter sharing** is a fundamental concept in Convolutional Neural Networks (CNNs) and refers to the idea of using the same set of weights (parameters) for multiple units in a layer.

Pooling Layer

- It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture.

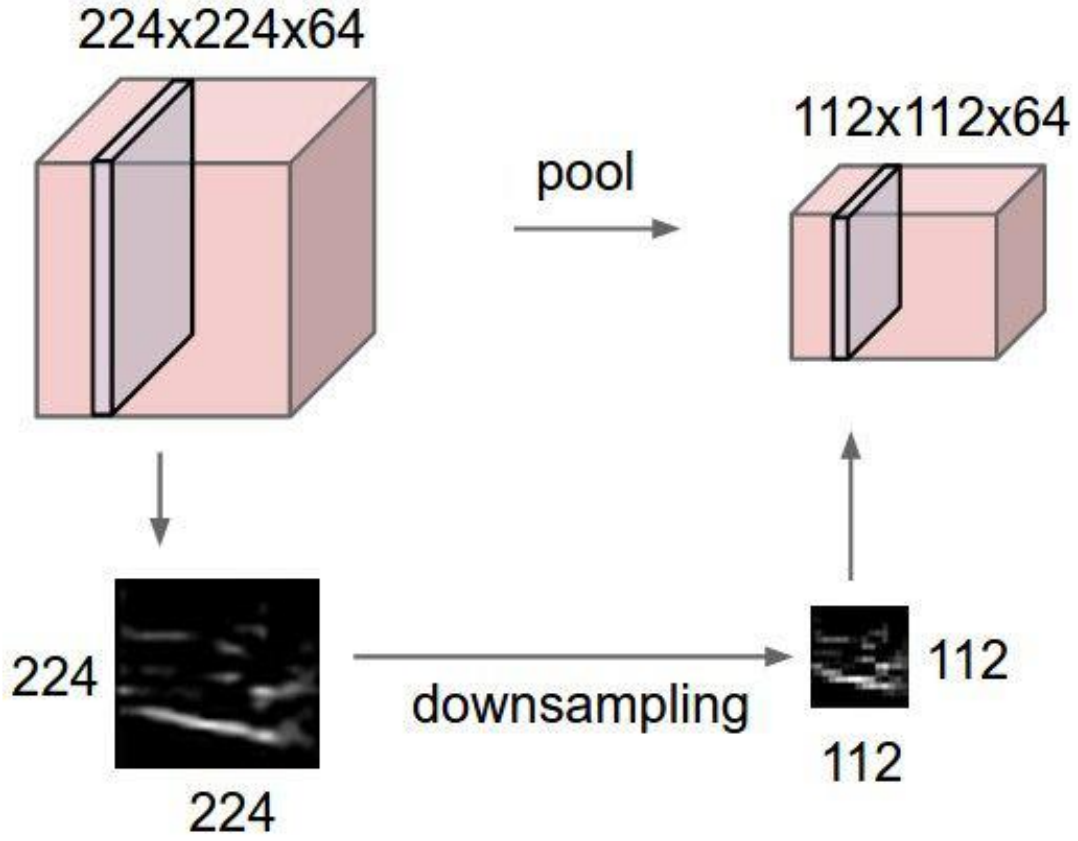
- Its function is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network.

- The Pooling Layer operates independently on every depth slice of the input and resizes it spatially.

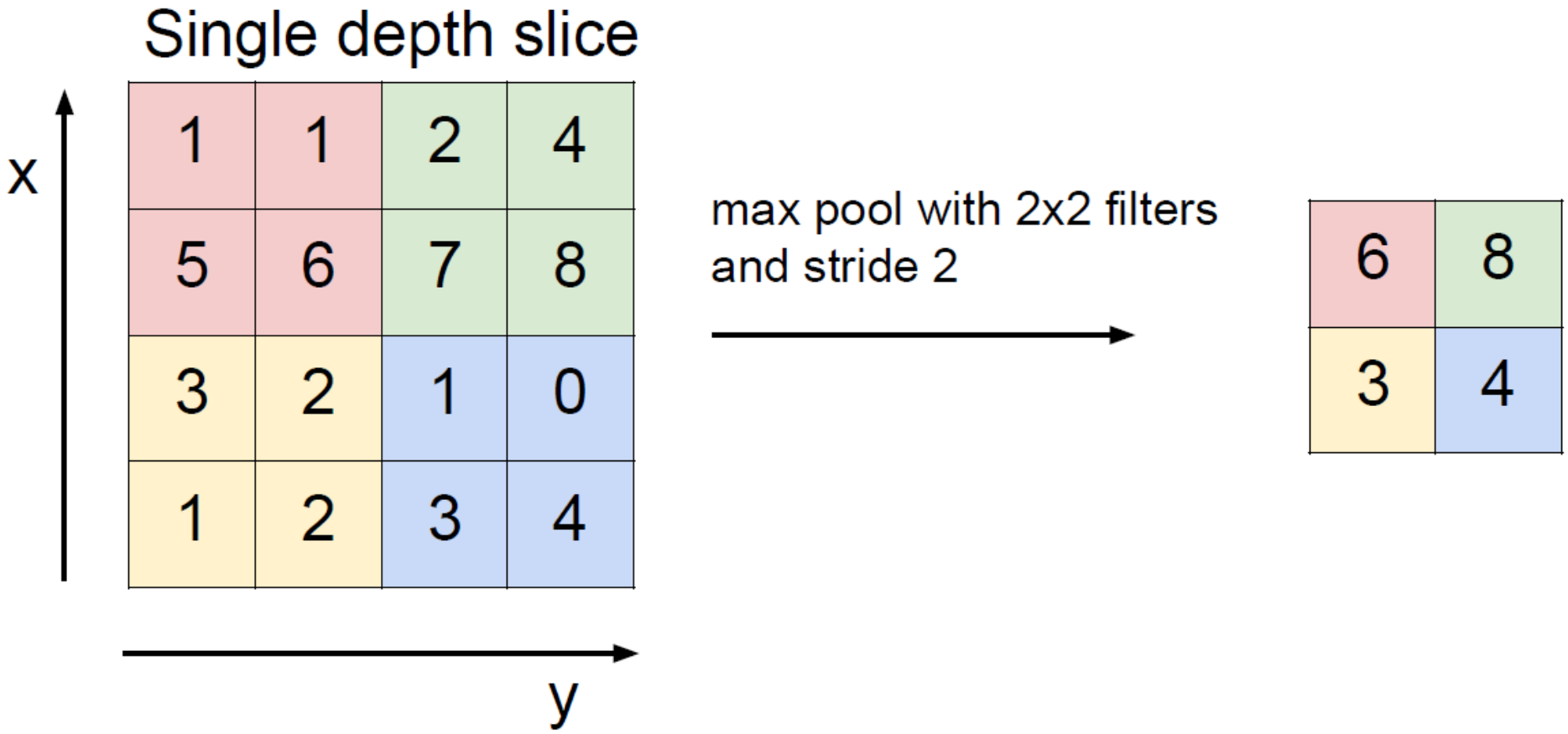
- The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height

Pooling Layer

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



Pooling Layer



Pooling Layer: summary

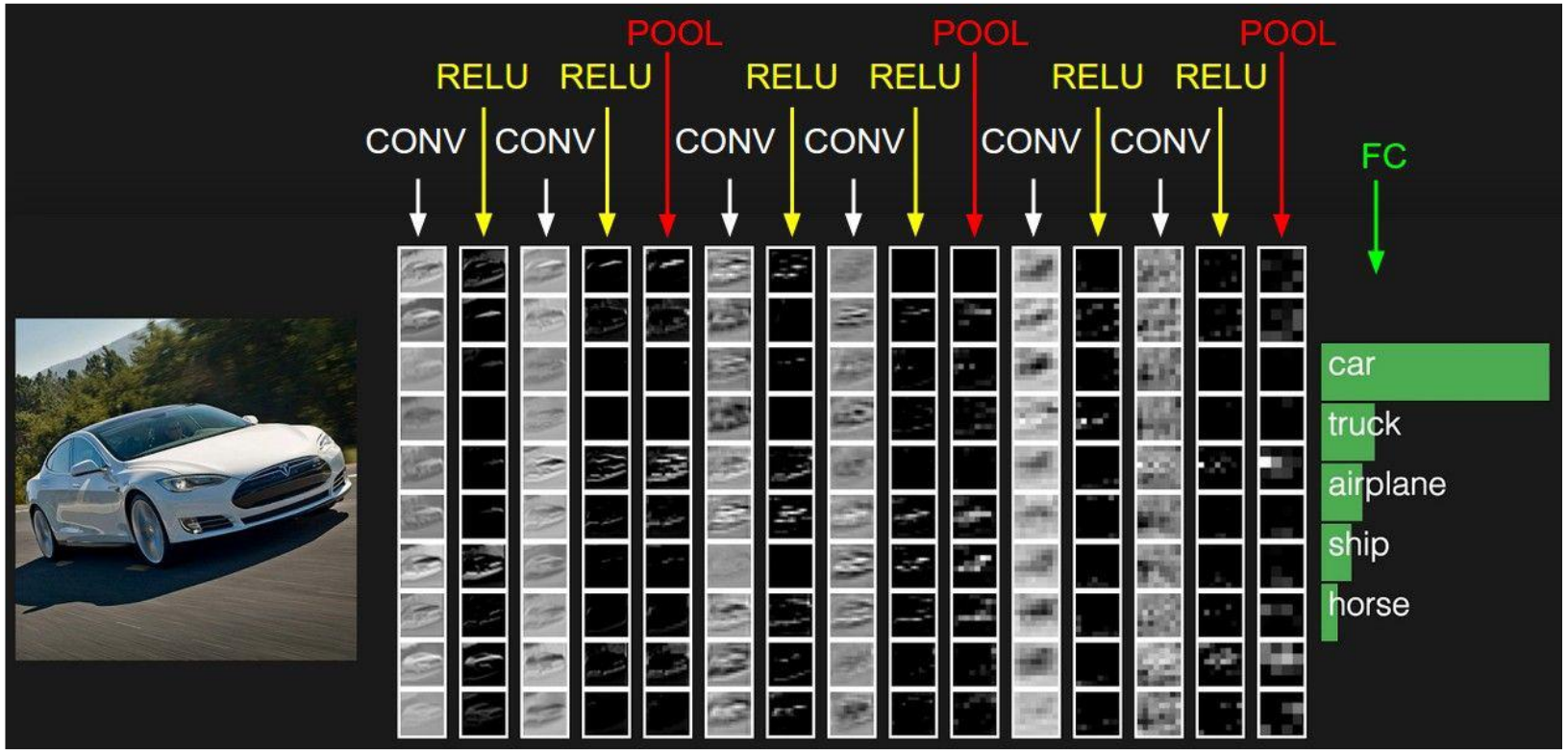
- Let's assume input is W_1 x H_1 x C
- Conv layer needs 2 hyperparameters:
  - The spatial extent F
  - The stride S

- This will produce an output of W_2 x H_2 x C where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$

- Number of parameters: 0

Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks.



```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10

(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()
train_images, test_images = train_images / 255.0, test_images / 255.0  # Normalize pixel values to [0, 1]

# Define the ConvNet model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

# print the model summary
model.summary()

# Compile the model with sparse_categorical_crossentropy
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(train_images, train_labels, epochs=10, validation_data=(test_images, test_labels))

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(test_images, test_labels)
print(f"Test Accuracy: {test_accuracy * 100}")
```