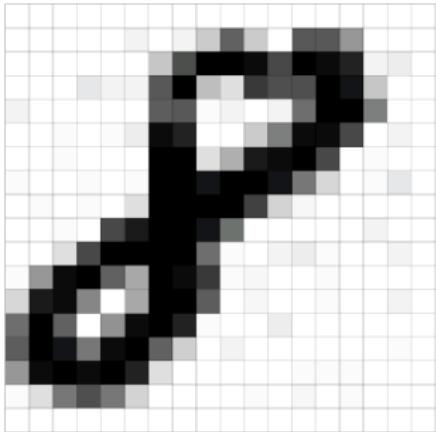


Convolutional Neural Networks

Shusen Wang

Convolution for Images

An Image is A Matrix/Tensor of Pixel Values



The MNIST Dataset

- Each grayscale image, \mathbf{x}_j , is a 28×28 matrix.
- (Previously, we reshape it to a 784-dim vector.)

Convolution

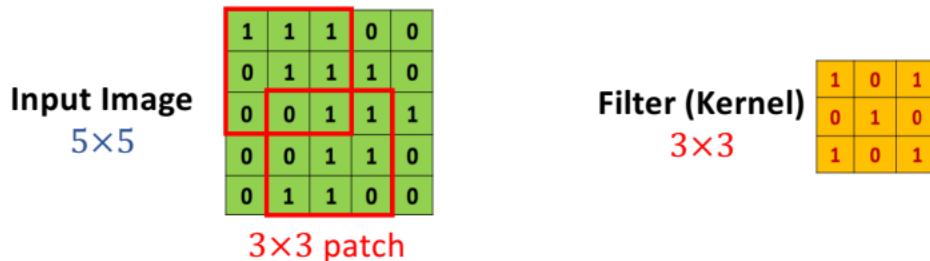
Input Image
 5×5

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Filter (Kernel)
 3×3

1	0	1
0	1	0
1	0	1

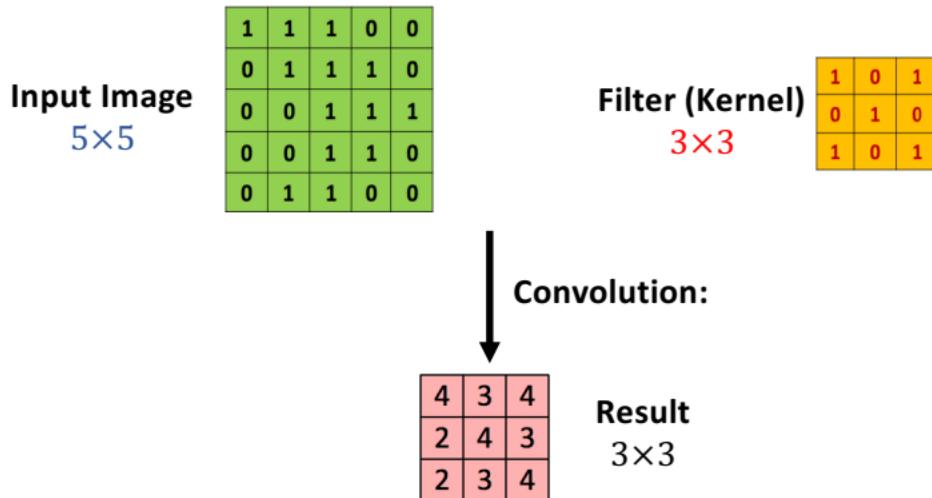
Convolution



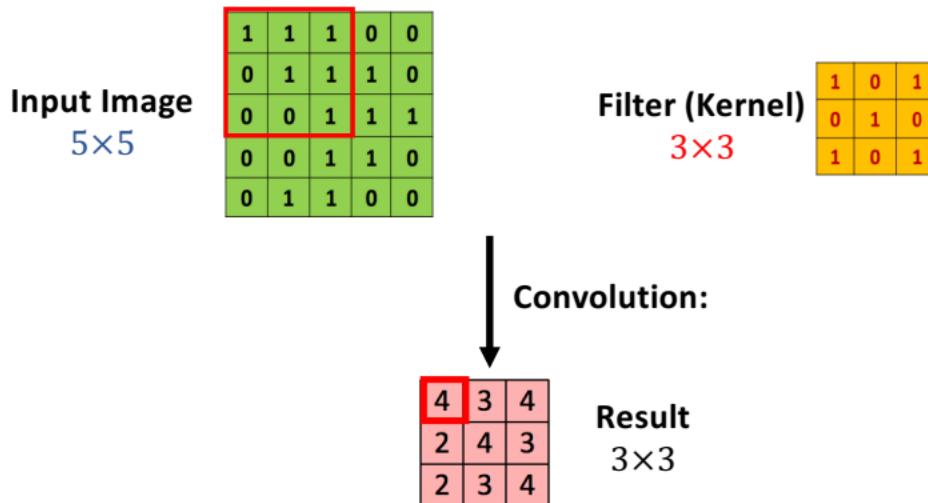
Question: how many 3×3 patches does a 5×5 image have?

Answer: $(5 - 3 + 1) \times (5 - 3 + 1) = 9.$

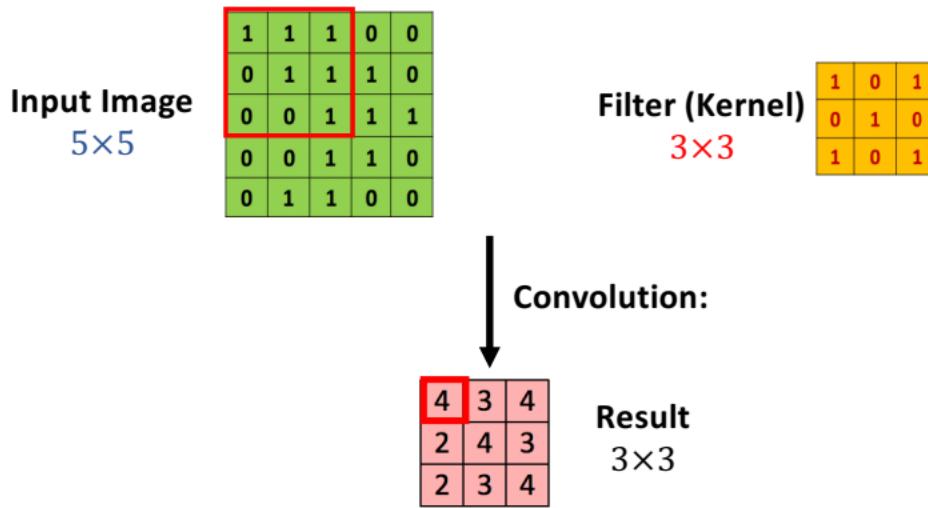
Convolution



Convolution



Convolution



The value **4** is the inner product of the patch

1	1	1
0	1	1
0	0	1

and the filter

1	0	1
0	1	0
1	0	1

Convolution

Input Image
 5×5

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Filter (Kernel)
 3×3

1	0	1
0	1	0
1	0	1

Convolution:

4	3	4
2	4	3
2	3	4

Result
 3×3

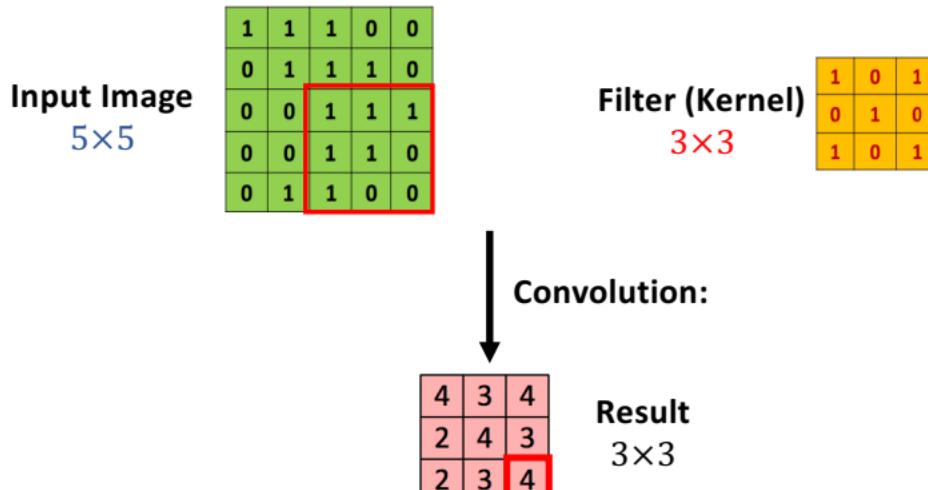
The value **3** is the inner product of the patch

1	1	0
1	1	1
0	1	1

and the filter

1	0	1
0	1	0
1	0	1

Convolution



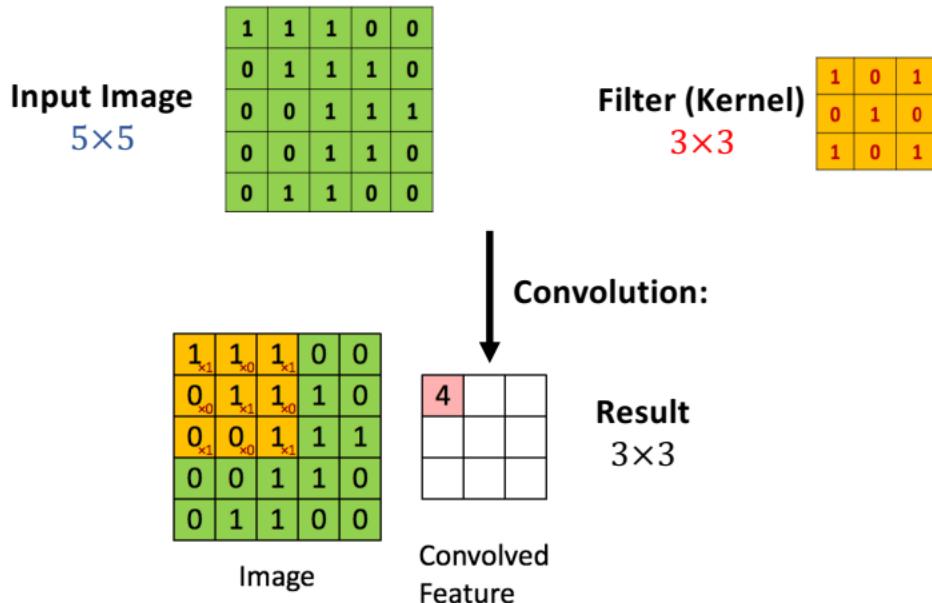
The value **4** is the inner product of the patch

1	1	1
1	1	0
1	0	0

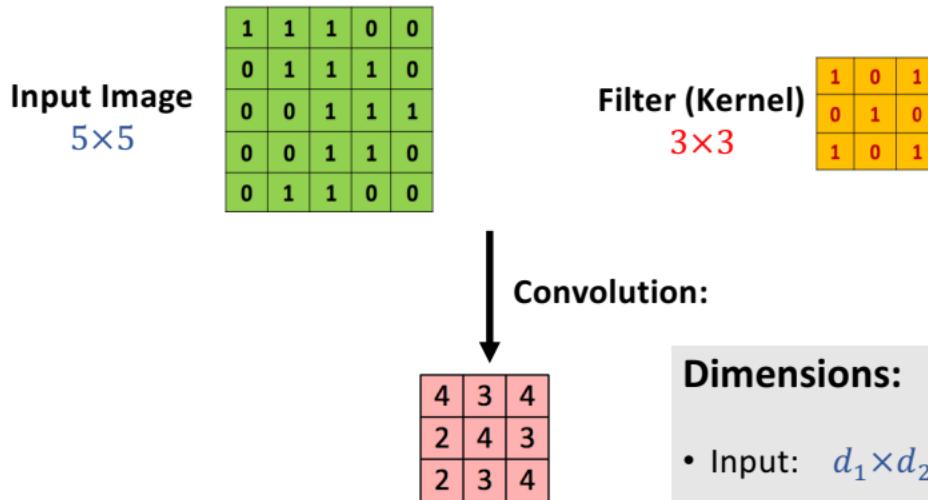
and the filter

1	0	1
0	1	0
1	0	1

Convolution



Convolution



Dimensions:

- Input: $d_1 \times d_2$
- Filter: $k_1 \times k_2$
- Output: $(d_1 - k_1 + 1) \times (d_2 - k_2 + 1)$

Convolution

Input Image



Filters

Identity

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Outputs



Convolution

Input Image



Filters

Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	A black and white output image showing the edges of the deer's head. It highlights the outline and some internal features like the eye and antlers.
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	A black and white output image showing the edges of the deer's head. It highlights the outline and some internal features like the eye and antlers.
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	A black and white output image showing the edges of the deer's head. It highlights the outline and some internal features like the eye and antlers.

Convolution

Input Image



Filters

Sharpen

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Outputs



- The inner product of $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ and $\begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & \textcolor{red}{0.1} & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix}$ is **0.1**.
- The inner product of $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ and $\begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & \textcolor{red}{0.2} & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix}$ is **0.6**.

Convolution

Input Image



	Filters	Outputs (or feature map)
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	A small image showing the input camel image processed with a 3x3 box blur filter. The result is a slightly smoothed version of the original image.
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	A small image showing the input camel image processed with a 3x3 Gaussian blur filter. The result is a smoother version of the original image compared to the box blur.

Convolution



Convolution: Zero Padding

Input Image
 5×5

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Filter (Kernel)
 3×3

1	0	1
0	1	0
1	0	1

Convolution:

Image

1 _{x1}	1 _{x0}	1 _{x1}	0	0
0 _{x0}	1 _{x1}	1 _{x0}	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0	0	1	1	0
0	1	1	0	0

Convolved Feature

Dimensions:

- Input: $d_1 \times d_2$
- Filter: 3×3
- Output: $(d_1 - 2) \times (d_2 - 2)$

Convolution: Zero Padding

1	1	1	0	0
---	---	---	---	---

Problem: the output is smaller than the input!

- If the input is 28×28 and the filter is 3×3 , you can have at most 13 convolutional layers. (The output of the 13th layer is 2×2 .)

Convolution:

1 _{x1}	1 _{x0}	1 _{x1}	0	0
0 _{x0}	1 _{x1}	1 _{x0}	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0	0	1	1	0

Image

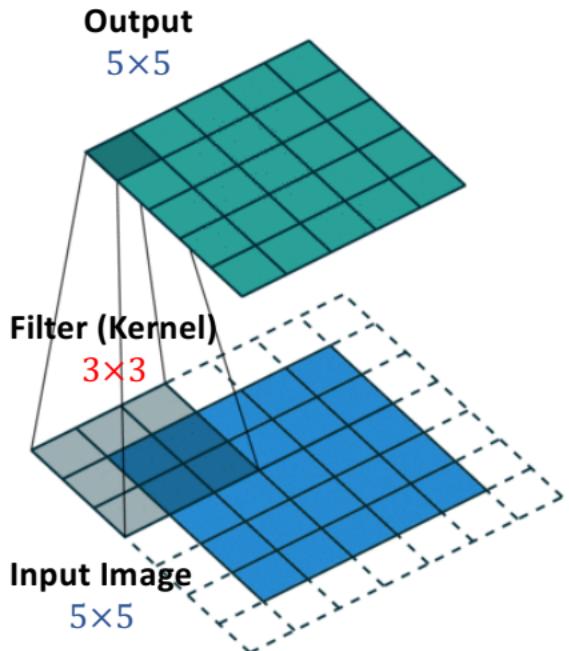
4		

Convolved Feature

Dimensions:

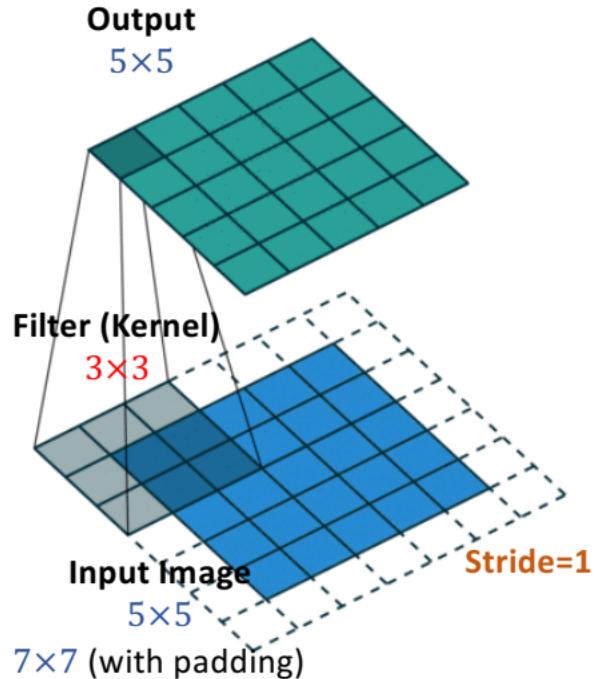
- Input: $d_1 \times d_2$
- Filter: 3×3
- Output: $(d_1 - 2) \times (d_2 - 2)$

Convolution: Zero Padding



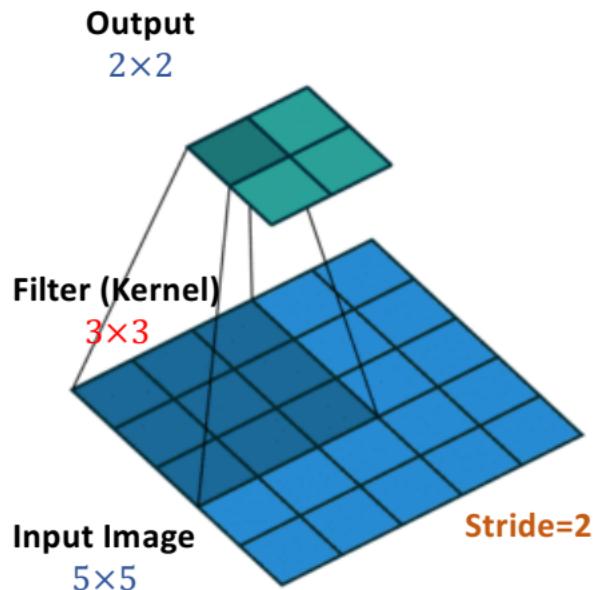
- Add a “boarder” of all-zeros.
- Increase the input’s size:
 - From $d_1 \times d_2$ to $(d_1 + 2) \times (d_2 + 2)$.
 - If the filter is 3×3 , the output is $d_1 \times d_2$.

Convolution: Stride



- In the previous examples, the **stride** is **1**.
 - The filter moves **1** step each time.

Convolution: Stride

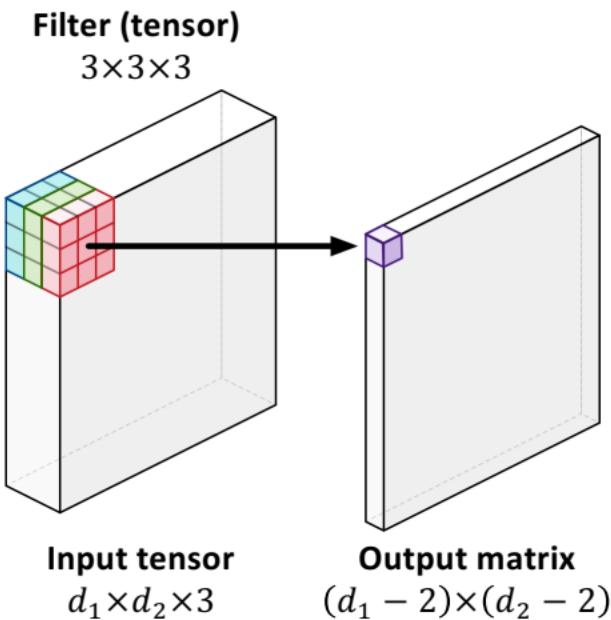


- In the previous examples, the **stride** is **1**.
 - The filter moves **1** step each time.
- The **stride** can be **2** or even larger.

Dimensions:

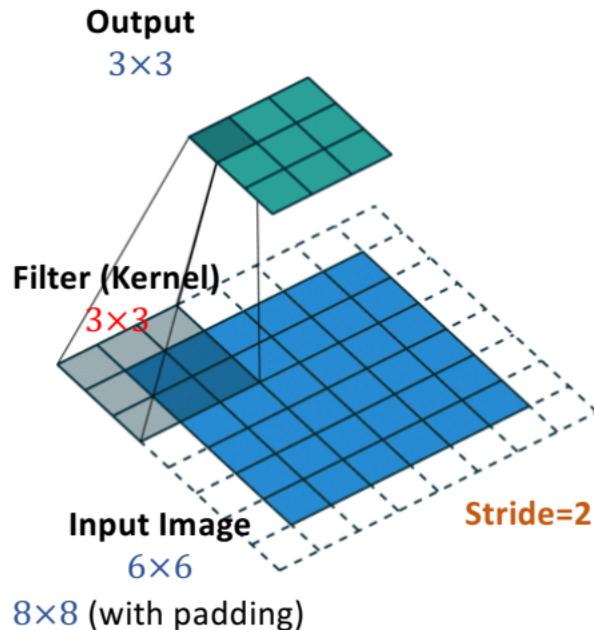
- Input: $d_1 \times d_2$
- Filter: $k_1 \times k_2$
- Stride: s
- Output: $\left(\left\lfloor \frac{d_1 - k_1}{s} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{d_2 - k_2}{s} \right\rfloor + 1\right)$

Convolution for Tensors



- The input is the order-3 tensor of size $d_1 \times d_2 \times 3$, e.g., a color image.
- The filter is a order-3 tensor of size $3 \times 3 \times 3$.
- The output is a matrix (order-2 tensor) of size $(d_1 - 2) \times (d_2 - 2)$.

Convolution: Key Concepts

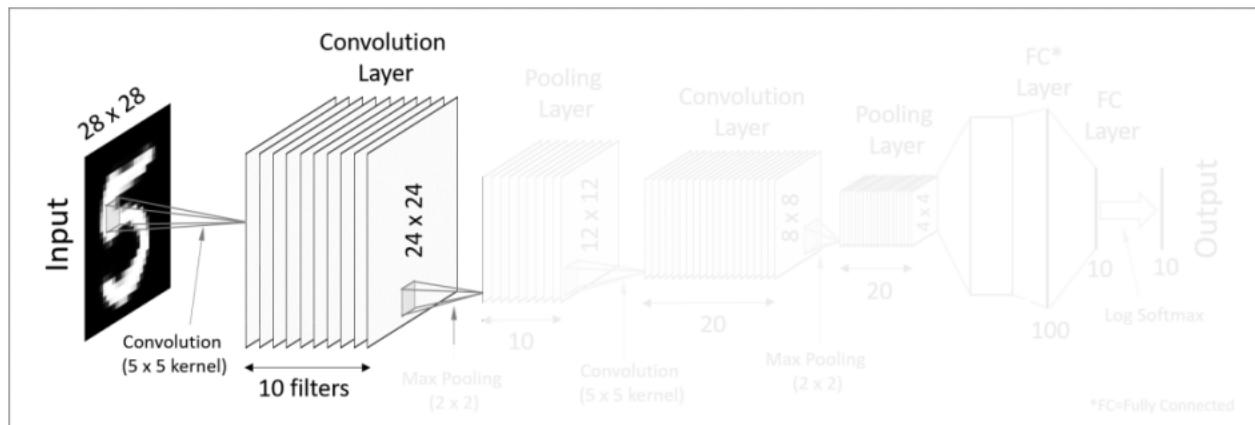


- Input, filter (kernel), and output (feature map).
- Convolution for matrix (order-2 tensor).
- Convolution for order-3 tensor.
- Zero padding.
- Stride.

Convolutional Neural Networks (CNNs)

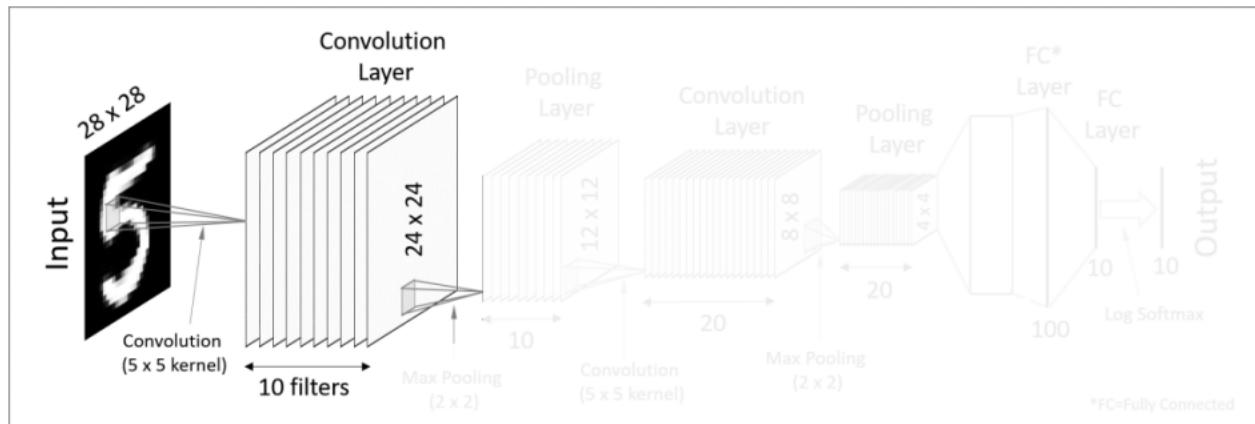
CNN: Convolutional Layer

- Use multiple filters (kernels) for the feature mapping. E.g.,
 - using ten 5×5 filters, (merely $10 \times 5 \times 5 = 250$ parameters!)
 - the results are ten 24×24 matrices.



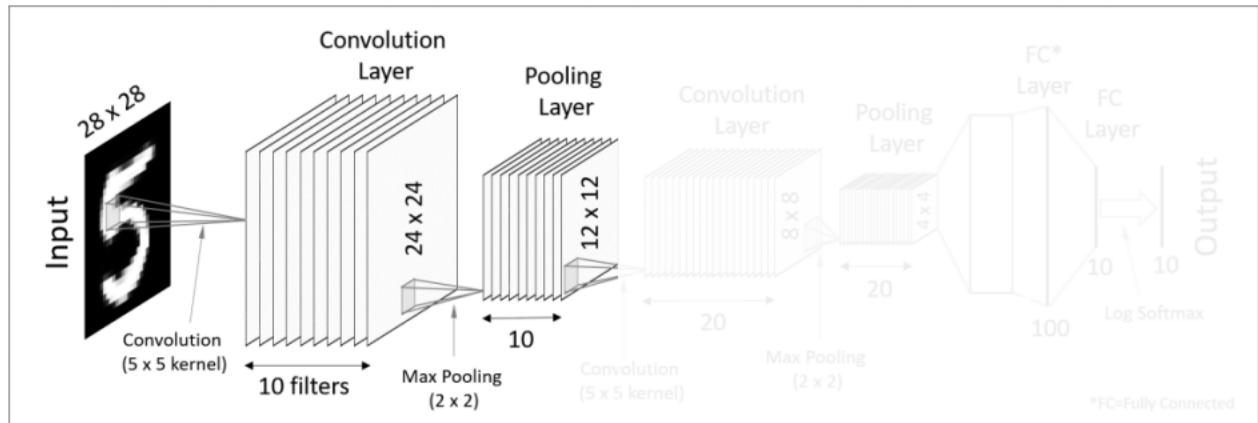
CNN: Convolutional Layer

- Use multiple filters (kernels) for the feature mapping.
- Apply an activation function (e.g., ReLU).
- A Convolutional layer = convolution + activation.



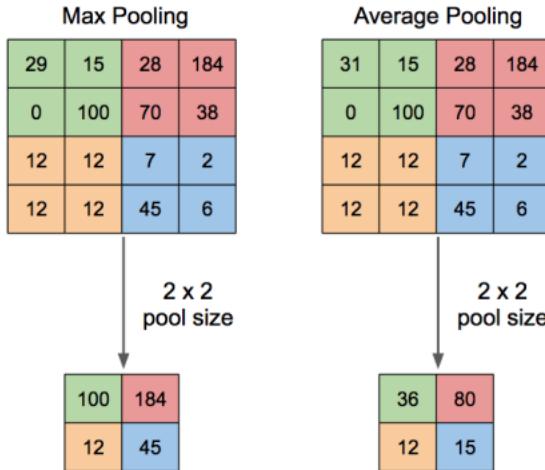
CNN: Pooling Layer

- Pooling reduces the dimensionality of each feature map.
- E.g., Max Pooling, Average Pooling, etc.



CNN: Pooling Layer

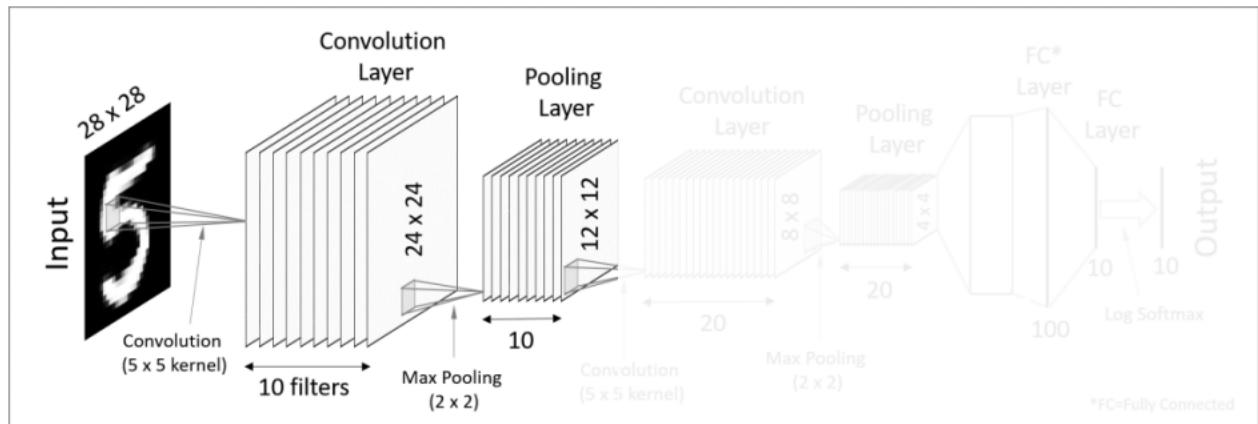
- Pooling reduces the dimensionality of each feature map.
- E.g., Max Pooling, Average Pooling, etc.



- Users specifies the pool size, e.g., 2×2 or 3×3 .
- The pooling can have overlap.

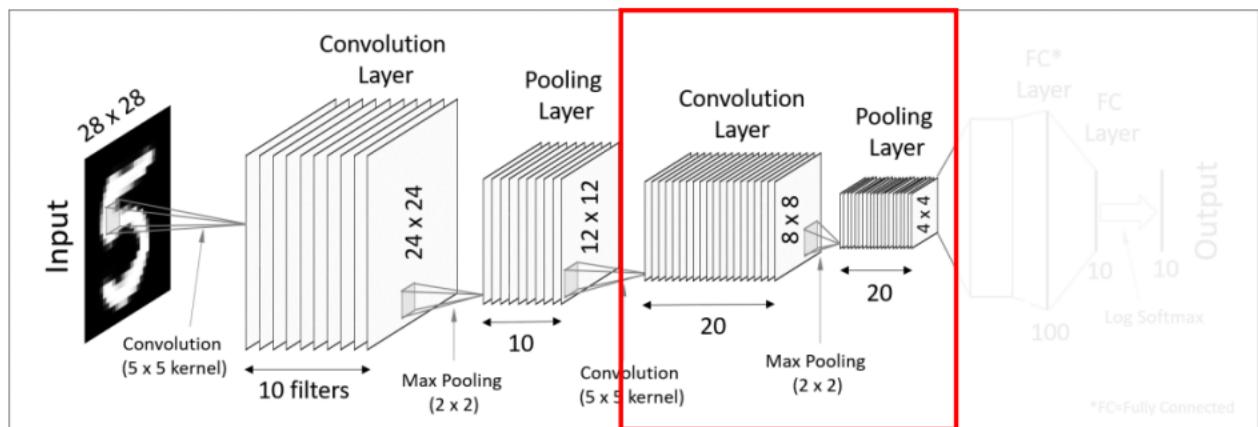
CNN: Pooling Layer

- Pooling reduces the dimensionality of each feature map.
- E.g., Max Pooling, Average Pooling, etc.

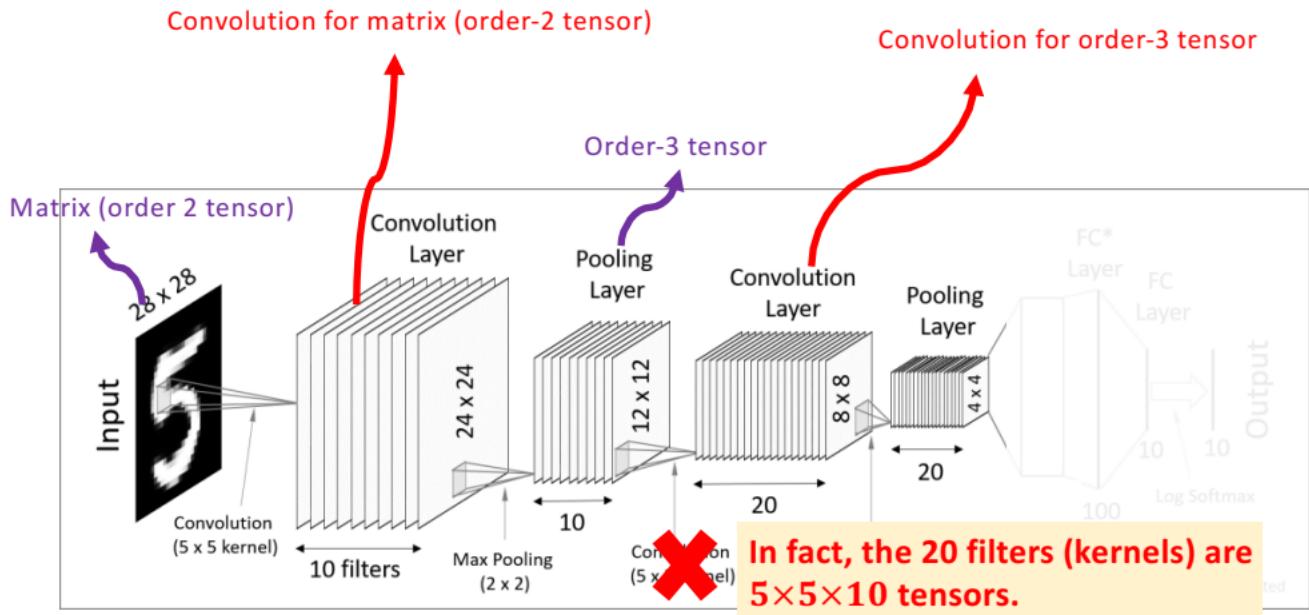


CNN

- Do the Convolution + Activation + Pooling again.

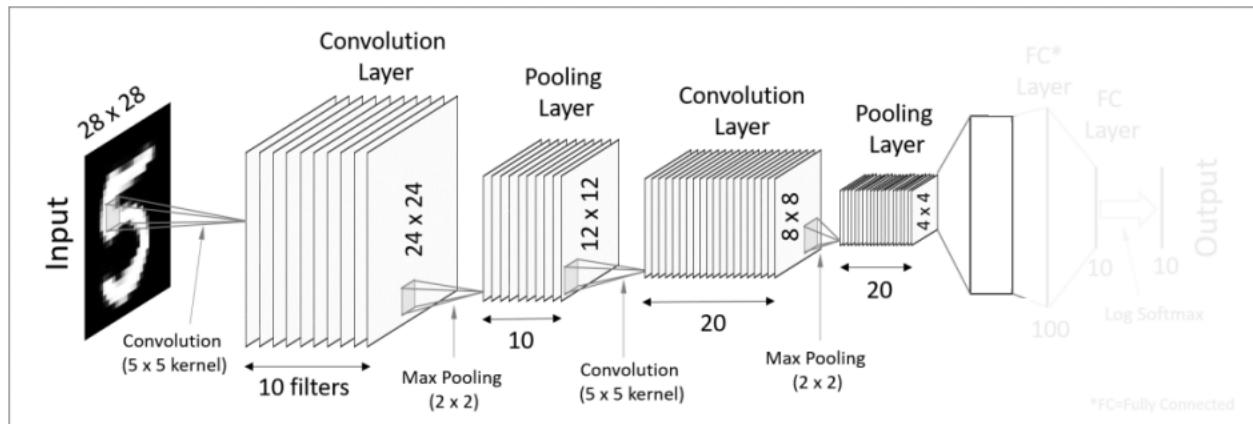


CNN



CNN: Flatten (Vectorization)

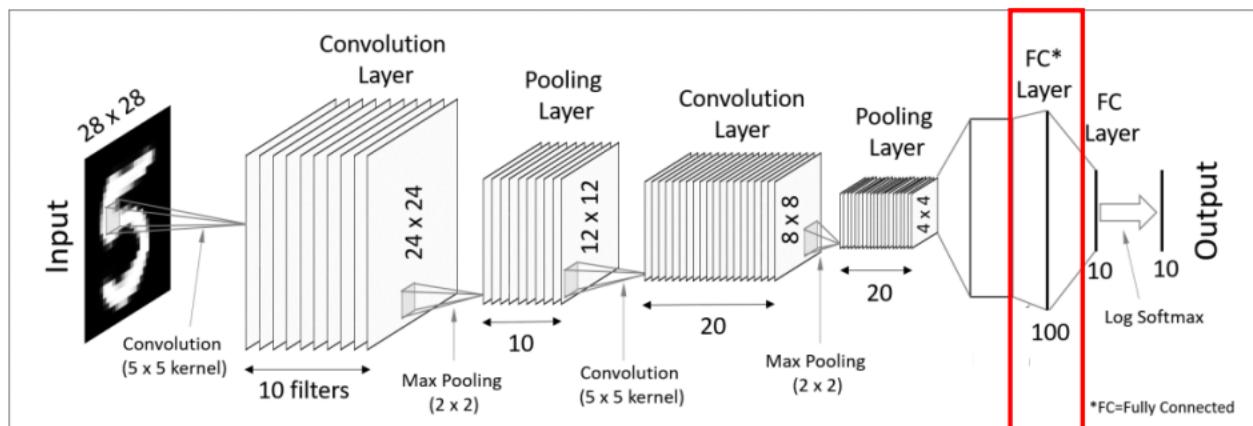
- Flatten the $4 \times 4 \times 20$ tensor.
- Get a 320-dim vector.



CNN: Fully-Connected Layers

- Add one or more fully-connected layers

This one is optional.



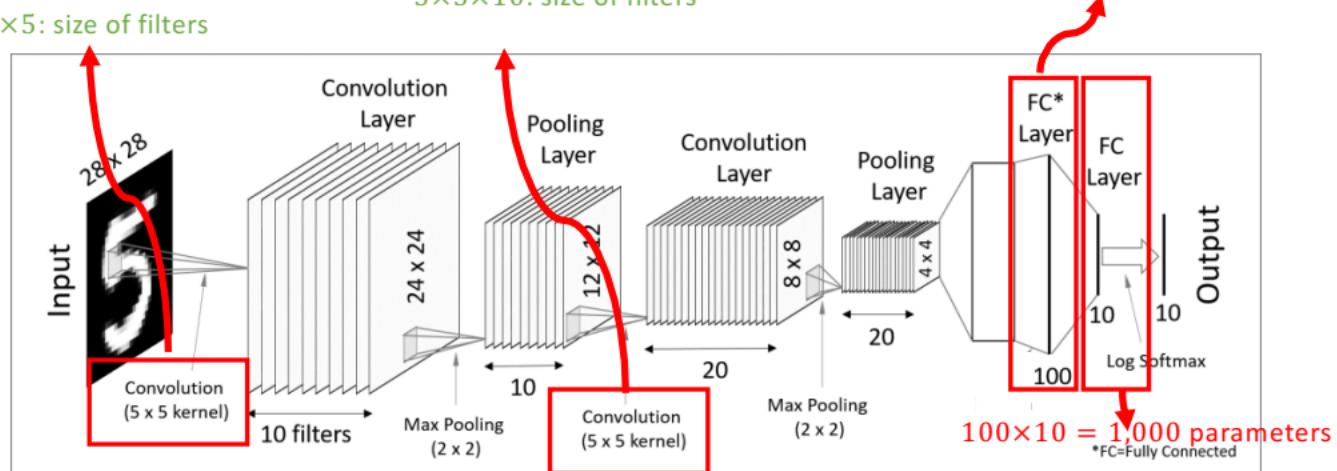
CNN: How Many Trainable Parameters?

$10 \times 5 \times 5 = 250$ parameters
• 10: number of filters
• 5×5 : size of filters

$$20 \times 5 \times 5 \times 10 = 5000 \text{ parameters}$$

- 20: number of filters
- $5 \times 5 \times 10$: size of filters

$$320 \times 100 = 32,000 \text{ parameters}$$



CNN: How Many Trainable Parameters?

Answer: > 38,250 parameters (not counting the intercections).

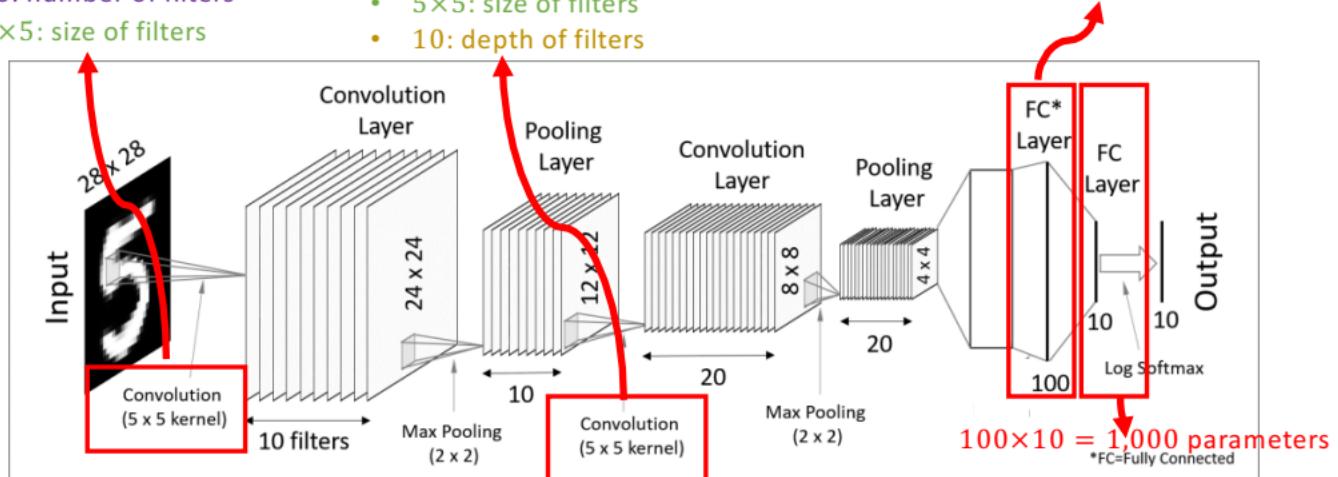
$$10 \times 5 \times 5 = 250 \text{ parameters}$$

- 10: number of filters
- 5×5: size of filters

$$20 \times 5 \times 5 \times 10 = 5000 \text{ parameters}$$

- 20: number of filters
- 5×5: size of filters
- 10: depth of filters

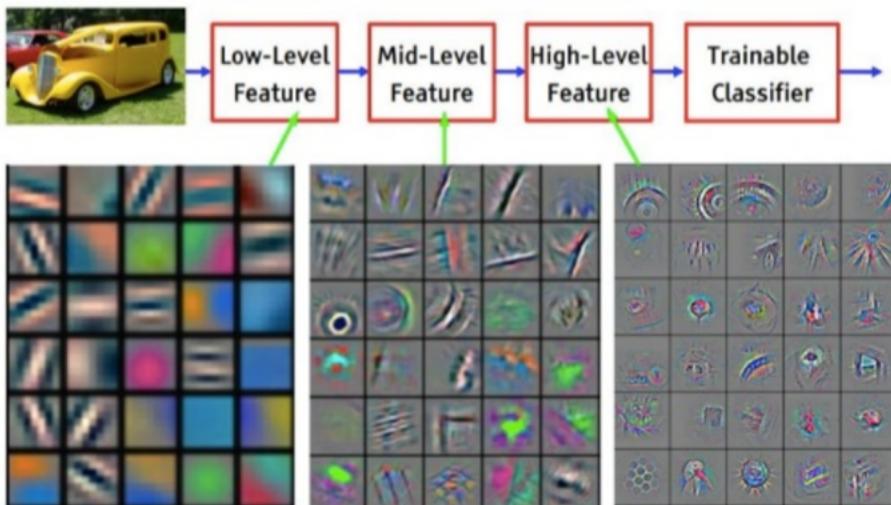
$$320 \times 100 = 32,000 \text{ parameters}$$



CNN: Re-Cap

Question: Why convolutional layers?

Answer: detect features like edge, shape, pattern, etc.



CNN: Re-Cap

Question: Why convolutional layers?

Answer: detect features like edge, shape, pattern, etc.

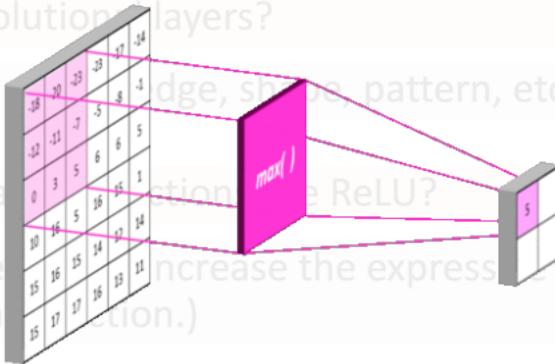
Question: Why activation functions like ReLU?

Answer: Add **nonlinearity** to increase the expressive power. (Note that convolution is a **linear** function.)

CNN: Re-Cap

Question: Why convolutional layers?

Answer: detect features like edge, shape, pattern, etc.



Question: Why activation function like ReLU?

Answer: Add nonlinearity to increase the expressiveness power. (Note that convolution is a linear operation.)

Question: Why pooling?

Answer: (1) Reduce the dimension of features;

CNN: Re-Cap

Example:

- A $d_1 \times d_2$ patch of the output of a convolution (5×5 filter) sees a $(d_1 + 4) \times (d_2 + 4)$ patch of the input.
- Then, after a 3×3 pooling, a $d_1 \times d_2$ patch sees a $(3d_1 + 12) \times (3d_2 + 12)$ patch of the input.

Answer: Add nonlinearity to increase the expressive power. (Note that convolution is a linear function.)

Question: Why pooling?

Answer: (1) Reduce the dimension of features; (2) Each element in the high-level feature “sees” a big patch of the input image.

Hyper-Parameters of CNN

Hyper-Parameters of CNN

- Number of layers (conv, pooling, and fully-connected layers).
- Convolutional layers:
 - Number of filters, size of filters, stride, zero-padding.
- Pooling layers:
 - Type of pooling: max, average, etc.
 - Size of pooling, stride, etc.
- Fully-connected layers:
 - Width of the layers.
- Activation functions.

Hyper-Parameters of Training

- Choice of the loss function.

Cross-entropy for multi-class classification

Logistic for binary classification

L1 or L2 for regression (the labels are continuous)

- Choose an optimization algorithm (and tune its parameters).

SGD

SGD with momentum

AdaGrad

RMSprop

Implement a CNN Using Keras

1. Load and Process the MNIST Dataset

Load data

```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

print('Shape of x_train: ' + str(x_train.shape))
print('Shape of x_test: ' + str(x_test.shape))
print('Shape of y_train: ' + str(y_train.shape))
print('Shape of y_test: ' + str(y_test.shape))
```

```
Shape of x_train: (60000, 28, 28)
Shape of x_test: (10000, 28, 28)
Shape of y_train: (60000,)
Shape of y_test: (10000,)
```

1. Load and Process the MNIST Dataset

Reshape: convert the $60000 \times 28 \times 28$ dataset to a $60000 \times 28 \times 28 \times 1$ tensor.

```
x_train_vec = x_train.reshape((60000, 28, 28, 1)) / 255.0
x_test_vec = x_test.reshape((10000, 28, 28, 1)) / 255.0

print('Shape of x_train_vec is ' + str(x_train_vec.shape))

Shape of x_train_vec is (60000, 28, 28, 1)
```

1. Load and Process the MNIST Dataset

Vectorization: convert the **labels** (scalars in $\{0, 1, \dots, 9\}$) to 10-dim vectors (by one-hot encoding).

```
def to_one_hot(labels, dimension=10):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

y_train_vec = to_one_hot(y_train)
y_test_vec = to_one_hot(y_test)
print('Shape of y_train_vec is ' + str(y_train_vec.shape))

Shape of y_train_vec is (60000, 10)
```

1. Load and Process the MNIST Dataset

Partition the **training** set to **training** and **validation** sets.

```
# Partition to training and validation sets

rand_indices = np.random.permutation(60000)
train_indices = rand_indices[0:50000]
valid_indices = rand_indices[50000:60000]

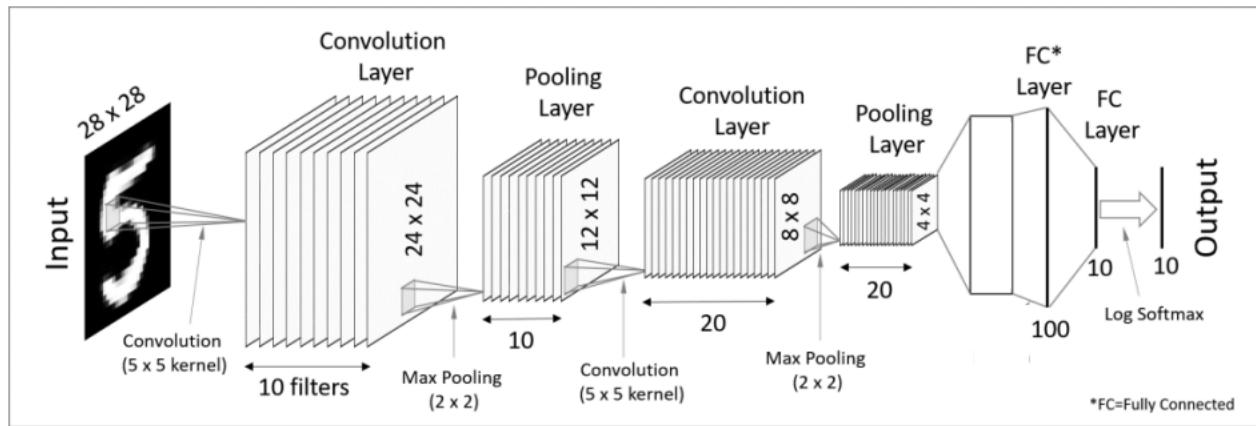
x_valid_vec = x_train_vec[valid_indices, :, :, :]
y_valid_vec = y_train_vec[valid_indices, :]

x_train_vec = x_train_vec[train_indices, :, :, :]
y_train_vec = y_train_vec[train_indices, :]

print('Shape of x_valid_vec: ' + str(x_valid_vec.shape))
print('Shape of y_valid_vec: ' + str(y_valid_vec.shape))
print('Shape of x_train_vec: ' + str(x_train_vec.shape))
print('Shape of y_train_vec: ' + str(y_train_vec.shape))

Shape of x_valid_vec: (10000, 28, 28, 1)
Shape of y_valid_vec: (10000, 10)
Shape of x_train_vec: (50000, 28, 28, 1)
Shape of y_train_vec: (50000, 10)
```

2. Build the CNN



2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
```

Number of filters Size of each filter

Default
• Stride: 1
• Zero-padding: no padding

2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
```

pool_size

Default
• Stride: pool_size

2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(20, (5, 5), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(20, (5, 5), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
```

2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(20, (5, 5), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

2. Build the CNN

```
# print the summary of the model.  
model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_1 (Conv2D)	(None, 24, 24, 10)	260
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 10)	0
conv2d_2 (Conv2D)	(None, 8, 8, 20)	5020
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 20)	0
flatten_1 (Flatten)	(None, 320)	0
dense_1 (Dense)	(None, 100)	32100
dense_2 (Dense)	(None, 10)	1010
<hr/>		
Total params: 38,390		
Trainable params: 38,390		
Non-trainable params: 0		

3. Train the CNN

Specify: optimization algorithm, learning rate (LR), loss function, and metric.

```
from keras import optimizers  
model.compile(optimizers.RMSprop(lr=0.0001),  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

3. Train the CNN

Specify: batch size and number of epochs.

```
history = model.fit(x_train_vec, y_train_vec,  
                      batch_size=128, epochs=50,  
                      validation_data=(x_valid_vec, y_valid_vec))
```

```
Train on 50000 samples, validate on 10000 samples  
Epoch 1/50  
50000/50000 [=====] - 12s 239us/step - loss: 1.2455 - acc: 0.7201 - val_loss: 0.4805 - val_acc: 0.8707  
Epoch 2/50  
50000/50000 [=====] - 12s 235us/step - loss: 0.3563 - acc: 0.8995 - val_loss: 0.2851 - val_acc: 0.9172  
Epoch 3/50  
50000/50000 [=====] - 12s 244us/step - loss: 0.2474 - acc: 0.9271 - val_loss: 0.2230 - val_acc: 0.9365
```

●
●
●

```
Epoch 49/50  
50000/50000 [=====] - 12s 240us/step - loss: 0.0244 - acc: 0.9924 - val_loss: 0.0439 - val_acc: 0.9875  
Epoch 50/50  
50000/50000 [=====] - 12s 239us/step - loss: 0.0238 - acc: 0.9927 - val_loss: 0.0446 - val_acc: 0.9881
```

4. Examine the Results

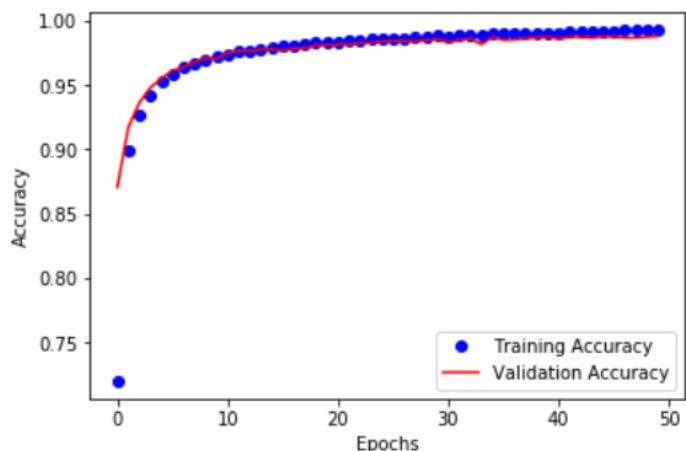
Plot the *accuracy* against *epochs* (1 epoch = 1 pass over the data).

```
import matplotlib.pyplot as plt
%matplotlib inline

epochs = range(50) # 50 is the number of epochs
train_acc = history.history['acc']
valid_acc = history.history['val_acc']
plt.plot(epochs, train_acc, 'bo', label='Training Accuracy')
plt.plot(epochs, valid_acc, 'r', label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

4. Examine the Results

Plot the *accuracy* against *epochs* (1 epoch = 1 pass over the data).



Why using the validation set?

- Tune the parameters, e.g., learning rate, regularization.
- Check whether over-fit or not.

4. Examine the Results

Evaluate the model on the ***test*** set. (So far, the model has not seen the ***test*** set.)

```
loss_and_acc = model.evaluate(x_test_vec, y_test_vec)
print('loss = ' + str(loss_and_acc[0]))
print('accuracy = ' + str(loss_and_acc[1]))

10000/10000 [=====] - 1s 127us/step
loss = 0.03570800104729715
accuracy = 0.9883
```

4. Examine the Results

- Training accuracy: 99.3%
- Validation accuracy: 98.8%
- Test accuracy: 98.8%