Deep Learning
BS-Elective

Instructor : Dr Muhammad Ismail Mangrio

Slides credit : Dr M. Asif Khan

ismail@iba-suk.edu.pk

Unit 02 CNN Week 1

Contents

Image Processing VS Computer Vision

Image processing involves manipulating and enhancing images to improve their quality or to extract useful information. This can include techniques for filtering, noise reduction, image enhancement, and transformation.

The primary goal is to prepare images for further analysis or to improve their visual appearance.

Techniques: Filtering, image restoration, compression, color adjustment, edge detection

Computer vision is a broader field that focuses on enabling computers to interpret and understand visual information from the world. It encompasses the techniques and algorithms that allow machines to analyze and make decisions based on images or video.

The goal is to extract meaningful information from images or sequences of images, enabling tasks such as object detection, recognition, tracking, and scene understanding.

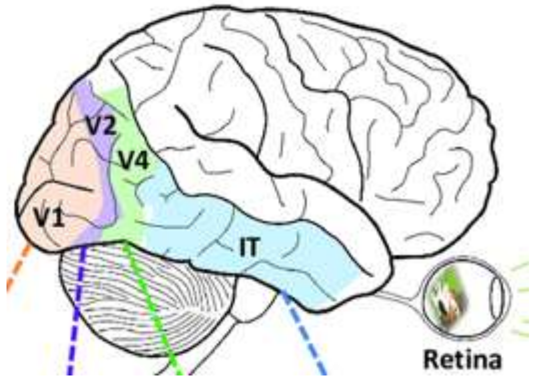
Techniques: Image segmentation, Object detection and recognition (e.g., using deep learning),Motion analysis, 3D reconstruction, Feature extraction

Emergence of CNN (Convolutional Neural Network)

- Mankind is an awesome natural machine and can look at multiple images every second and process them without realizing how the processing is done.
- But same is not with machines.
- The first step in image processing is to understand, how to represent an image so that the machine can read it?
- Every image is a cumulative arrangement of dots (a pixel) arranged in a special order.
- If you change the order or color of a pixel, the image will change as well.

Emergence of CNN (Convolutional Neural Network)

- CNNs emerged from the study of the brain's **visual cortex**, and they have been used in **image recognition** since the 1980s.
- In last few years, thanks to increase in **computational power**, the amount of available **training data**, and the tricks for training deep nets.
- CNNs have managed to achieve **superhuman** performance on some **complex visual tasks**.
- They power image search services, self-driving cars, automatic video classification systems, and more.



The Architecture of the Visual Cortex

- Many neurons in the visual cortex have a **small local receptive field**.
- Meaning **they react only to visual stimuli** located in a limited region of the visual field (see Figure 13-1, in which the local receptive fields of five neurons are represented by dashed circles).
- The **receptive fields** of different neurons **may overlap**, and together they **tile the whole visual field**.

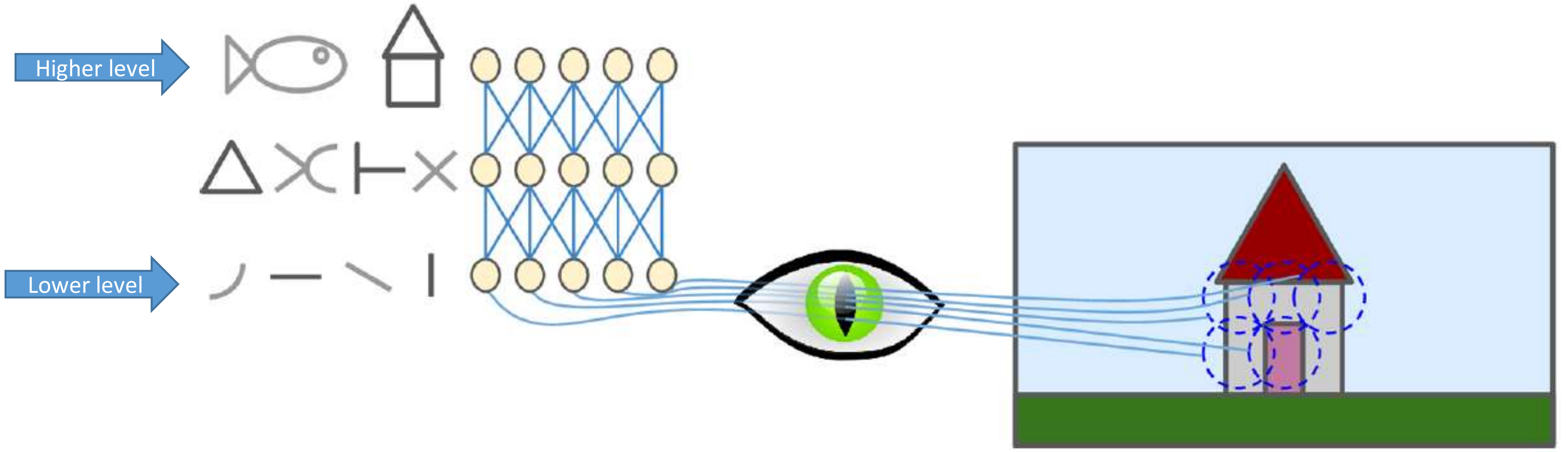


Figure 13-1. Local receptive fields in the visual cortex

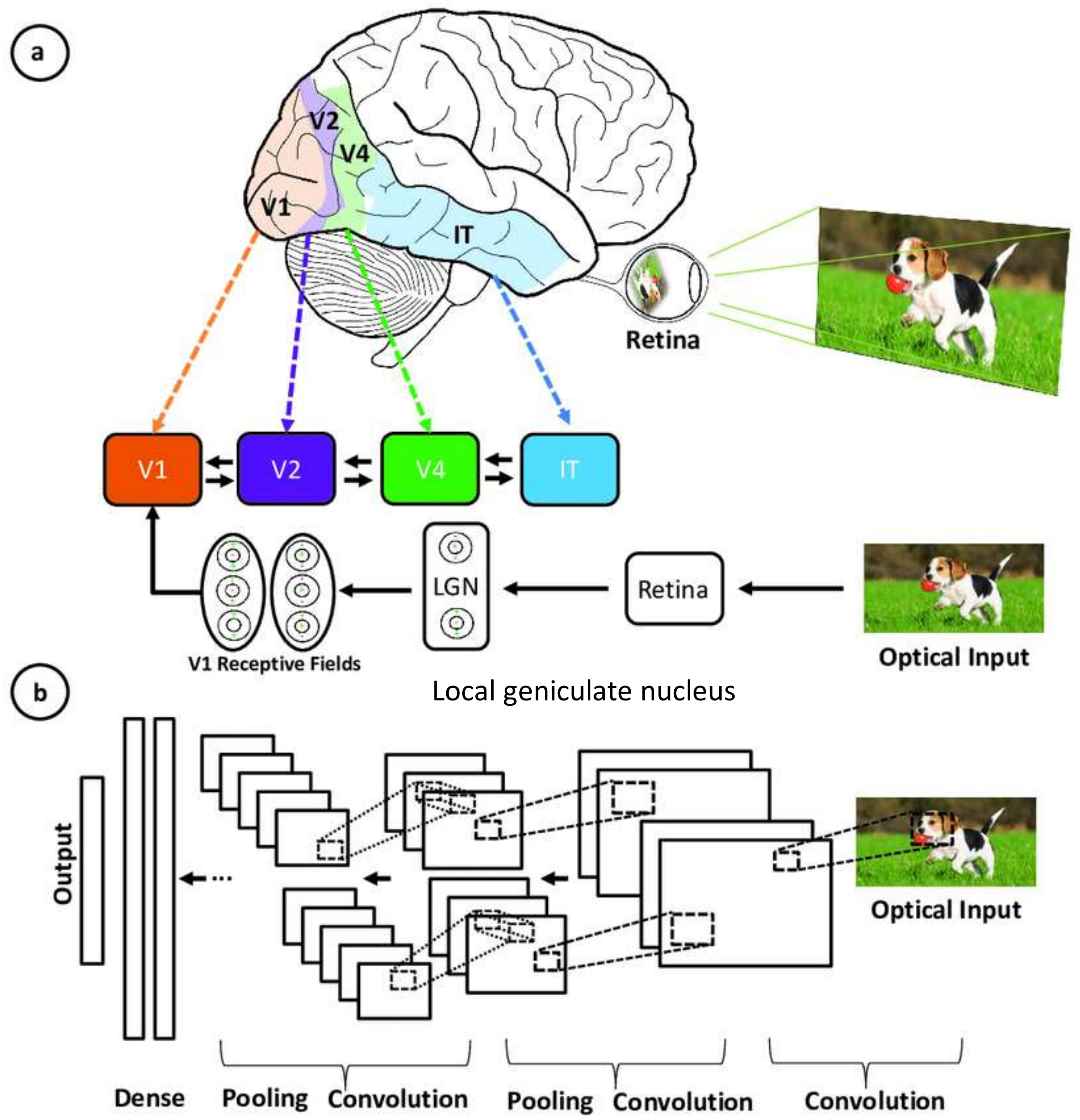
*The receptive field: region of the input image that a particular neuron in a convolutional layer is "looking at" or taking into account when making its predictions or feature extractions.

The Architecture of the Visual Cortex

- Some neurons **react only to** images of **horizontal lines**, while others react only to **lines with different orientations** (two neurons may have the **same receptive field** but react to **different line orientations**).
- Some neurons have **larger receptive fields**, and they react to more **complex patterns** that are **combinations of the lower-level patterns**.
- These observations led to the idea that the **higher-level neurons** are **based on** the **outputs** of neighboring **lower-level neurons** (in Figure 13-1, notice that each neuron is connected only to a few neurons from the previous layer).
- This powerful architecture is **able to detect all sorts of complex patterns** in any area of the visual field.

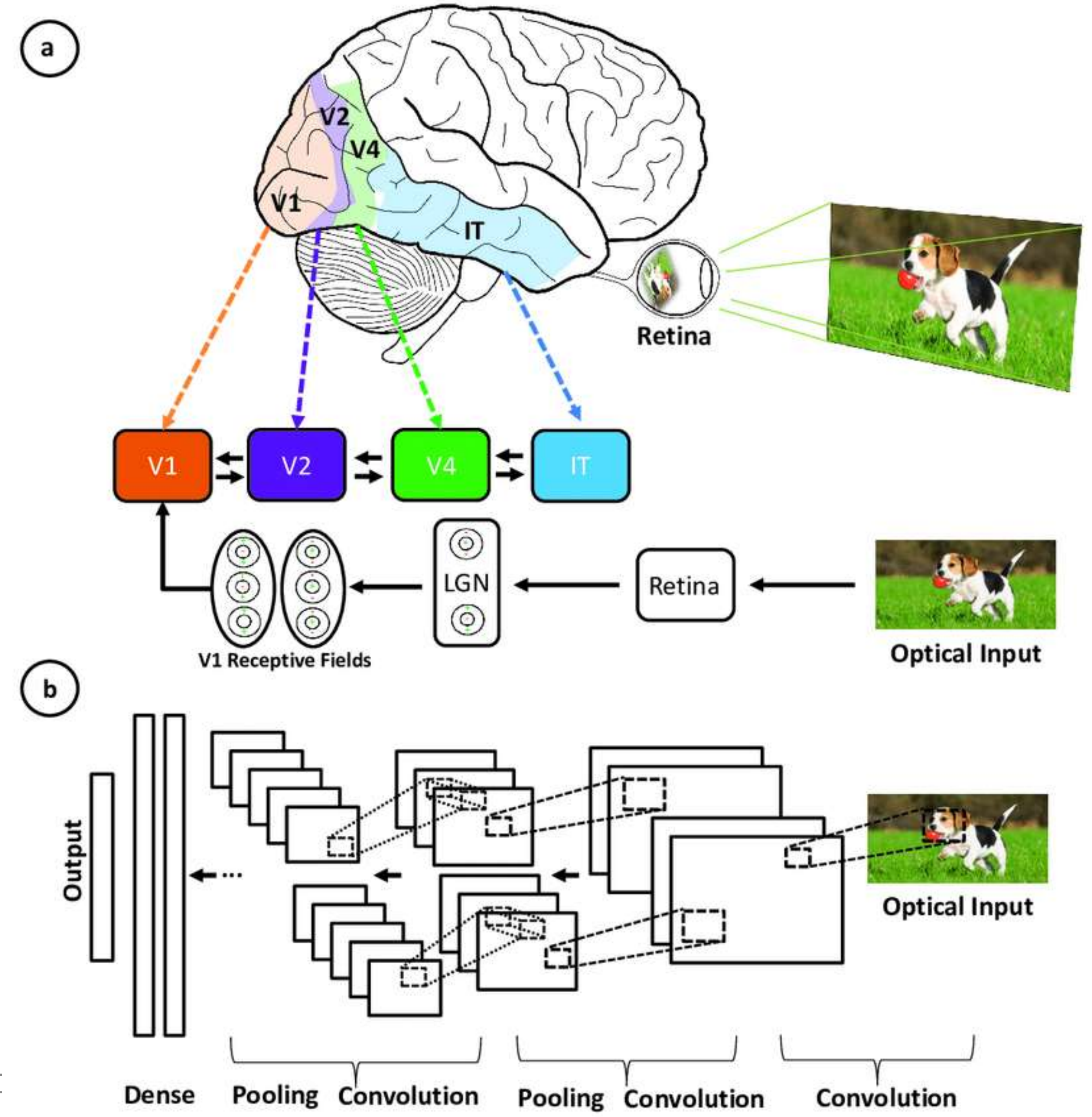
The Architecture of the Visual Cortex

- **Convolutional** neural networks were inspired by the **layered architecture** of the **human visual cortex**, and below are some key similarities and differences:
- V1 - Primary Visual Cortex:
  - **Role in Brain**: V1 is the first stage of cortical processing in the visual pathway. It primarily processes basic visual information such as **edges, orientations, and simple textures**.
  - **Analogy in CNNs**: The first layers of a CNN are similar to V1 in that they **detect simple features like edges and corners in an image**. These layers have **small receptive fields** and **extract low-level features**.



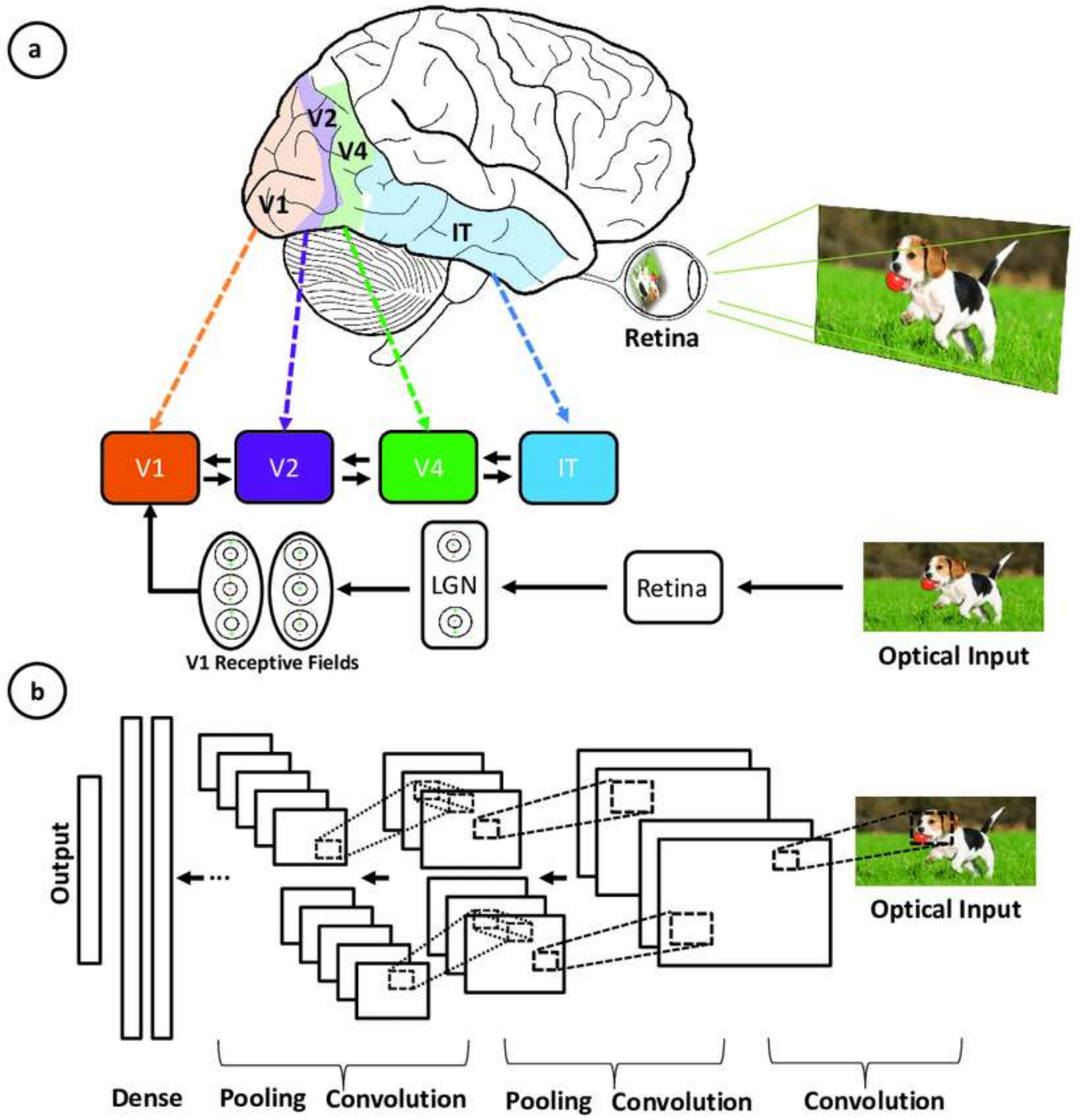
The Architecture of the Visual Cortex

- V2 - Secondary Visual Cortex:
  - **Role in Brain: V2 receives input from V1** and processes more **complex patterns**, such as **combinations of edges**, textures, **and some depth information**.
  - **Analogy in CNNs:** The **subsequent layers** in a CNN (after the first few layers) correspond to V2, where the network starts to **combine simple features into** more **complex patterns**, like **shapes** and textures.
- V4 - Fourth Visual Area:
  - **Role in Brain:** V4 is heavily involved in **processing color** information, as well as more complex shapes and patterns. It integrates information from earlier visual areas.
  - **Analogy in CNNs**: The deeper layers in a CNN, which deal with more abstract representations of the input image, are similar to V4. These layers might be responsible for **identifying specific objects or colors**.



The Architecture of the Visual Cortex

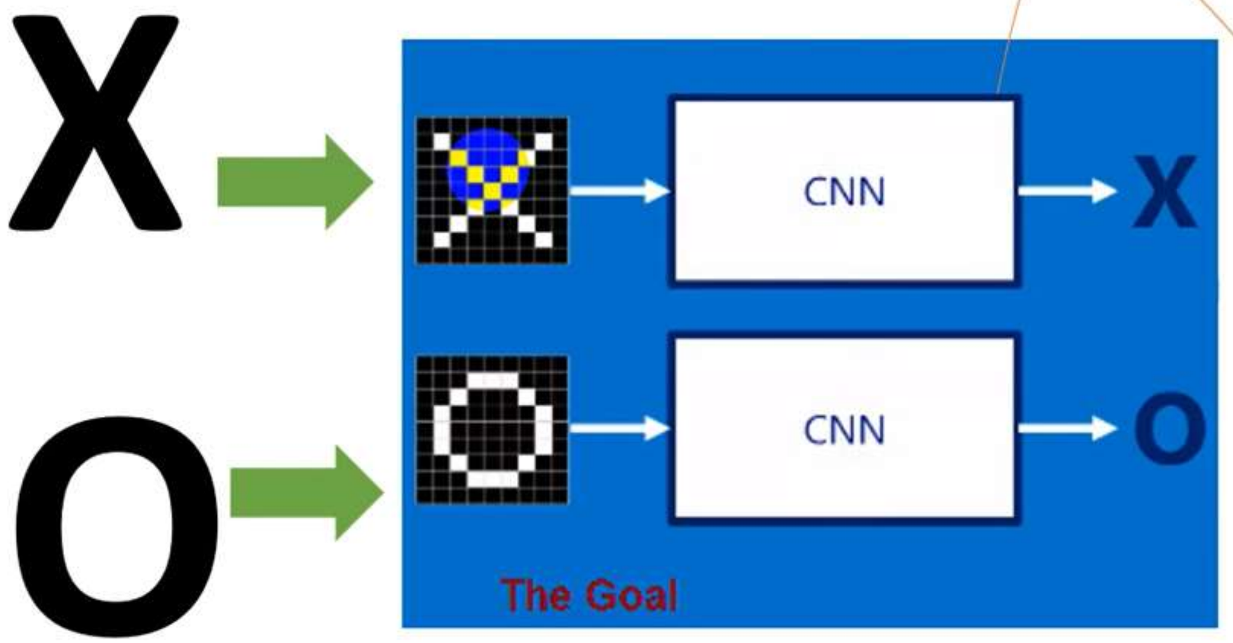
- IT – Inferotemporal Cortex:
  - **Role in Brain:** High-Level Processing: The IT cortex is involved in processing complex visual stimuli, particularly related to **object recognition**, **face recognition**, and categorization. It integrates information from earlier stages of visual processing to form a **complete understanding of what is being seen**.
  - **Analogy in CNNs:** The IT cortex can be compared to the **final layers** of a CNN, where the network has already **processed and combined various lower-level features** (like edges, textures, and shapes) **into more complex** and abstract representations. In these deep layers, the CNN is able to recognize specific objects, just as the IT cortex does.



CNN Learning

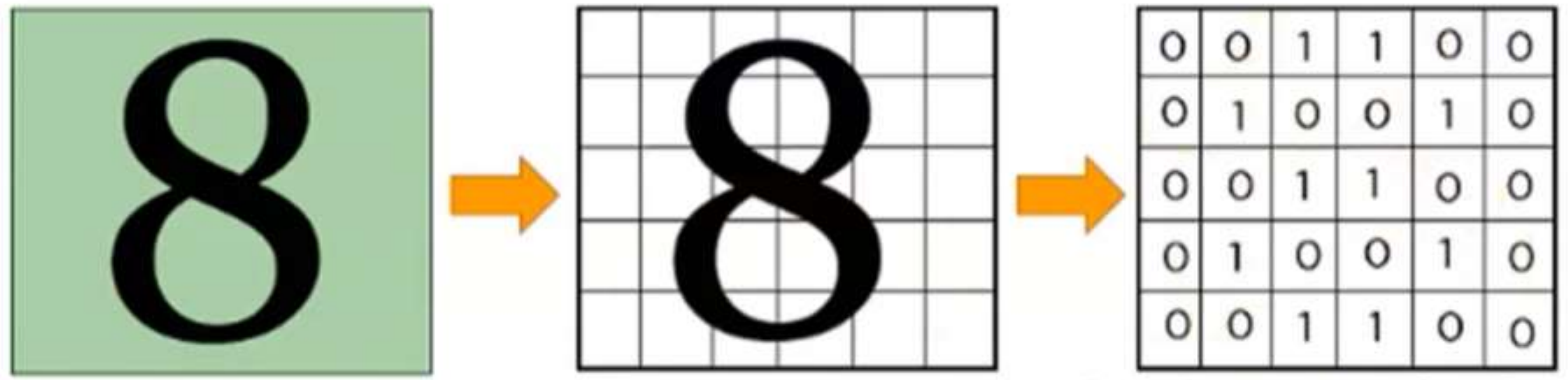
So, what about the Computer? CNN?

Learning...



Here CNN work as like black box, so what is inside the black box!

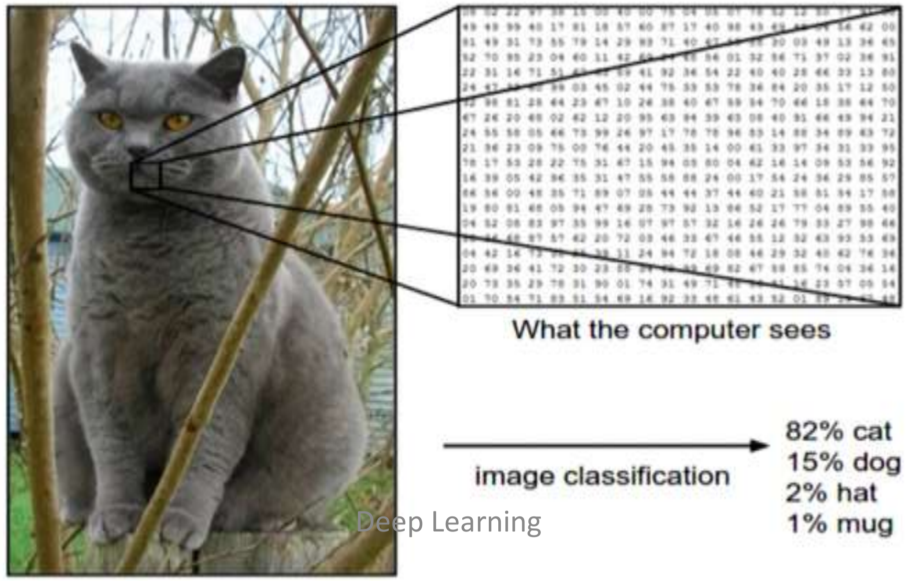
CNN Learning



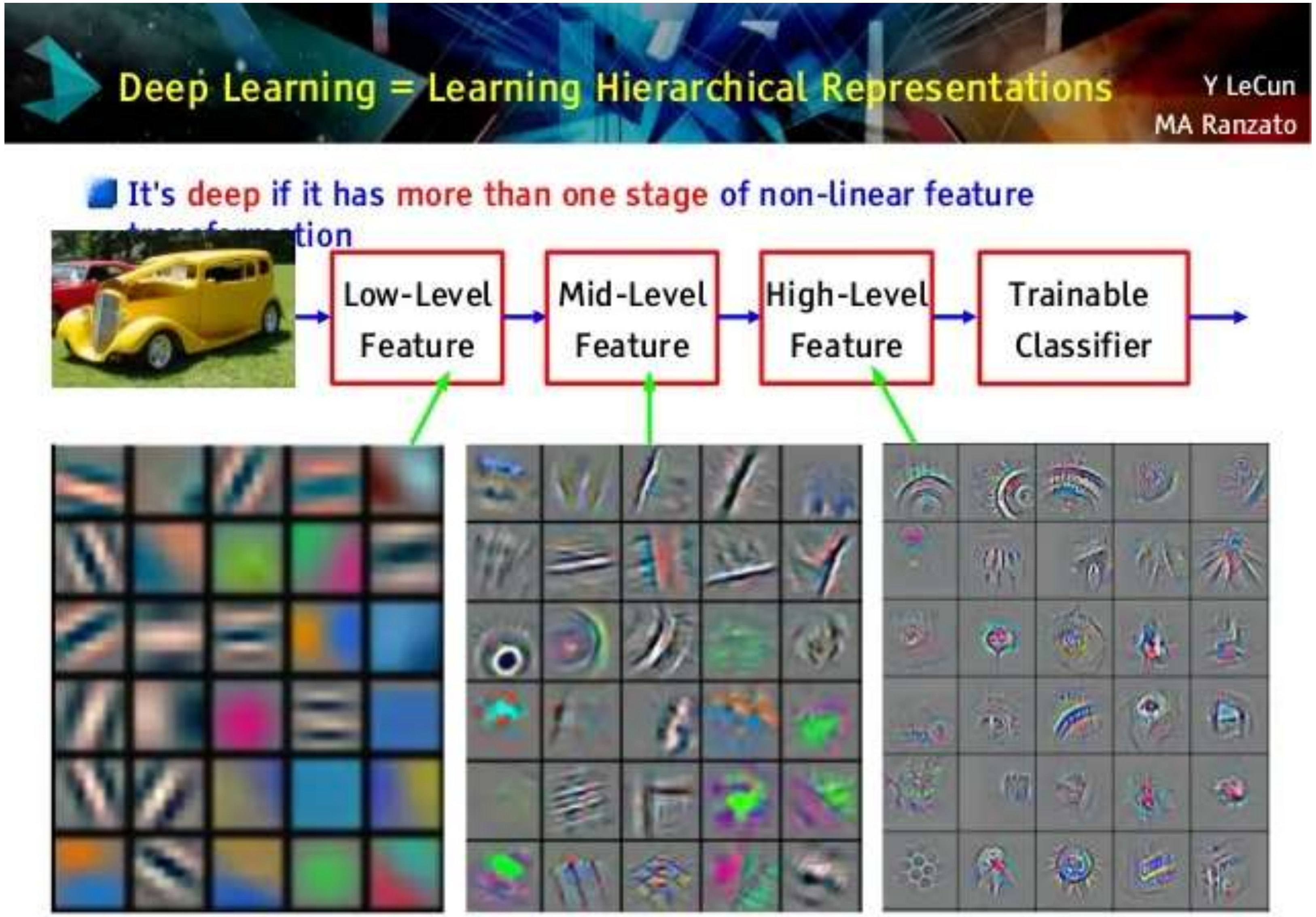
Real Image of the digit 8

Represented in the form of an array

Digit 8 represented in the form of pixels of 0's and 1's



Visual features in an image



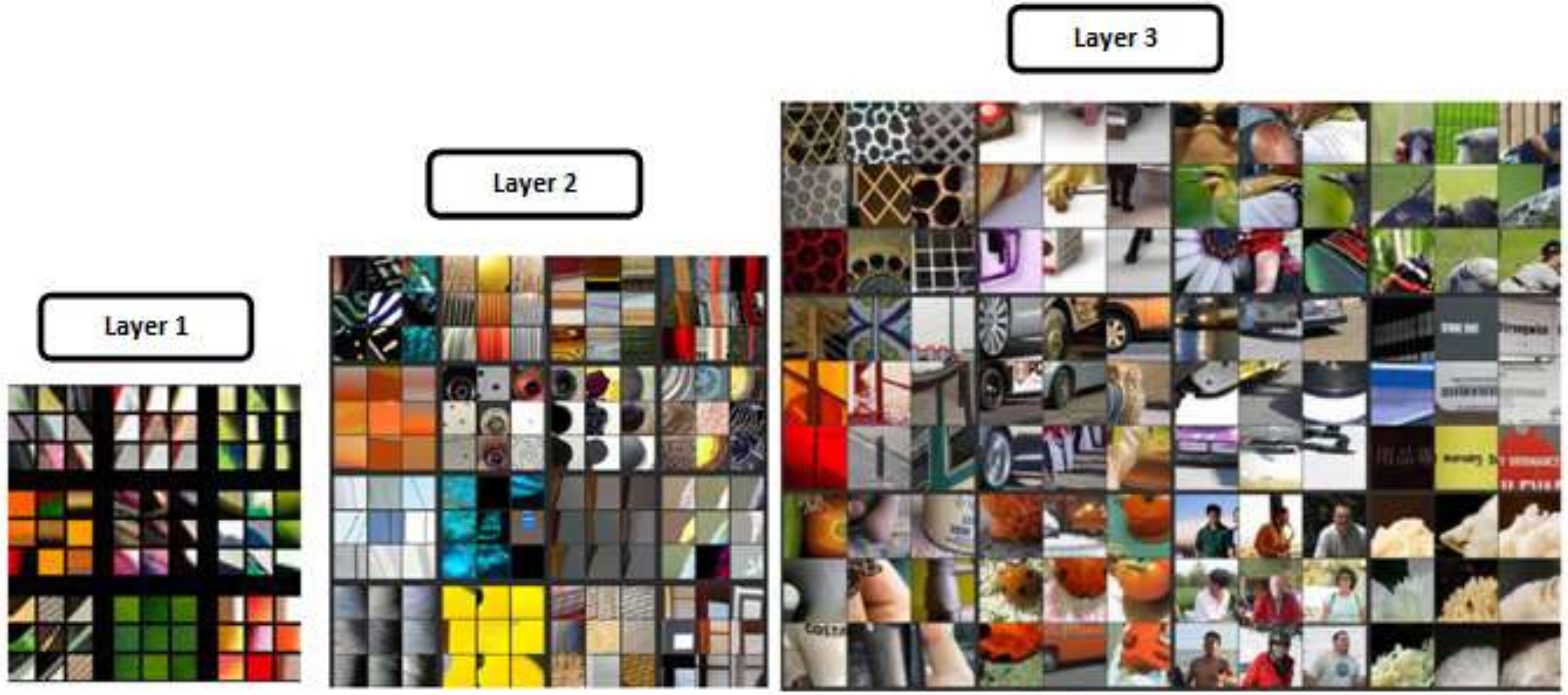
Deep Learning = Learning Hierarchical Representations

Y LeCun
MA Ranzato

- It's deep if it has more than one stage of non-linear feature transformation

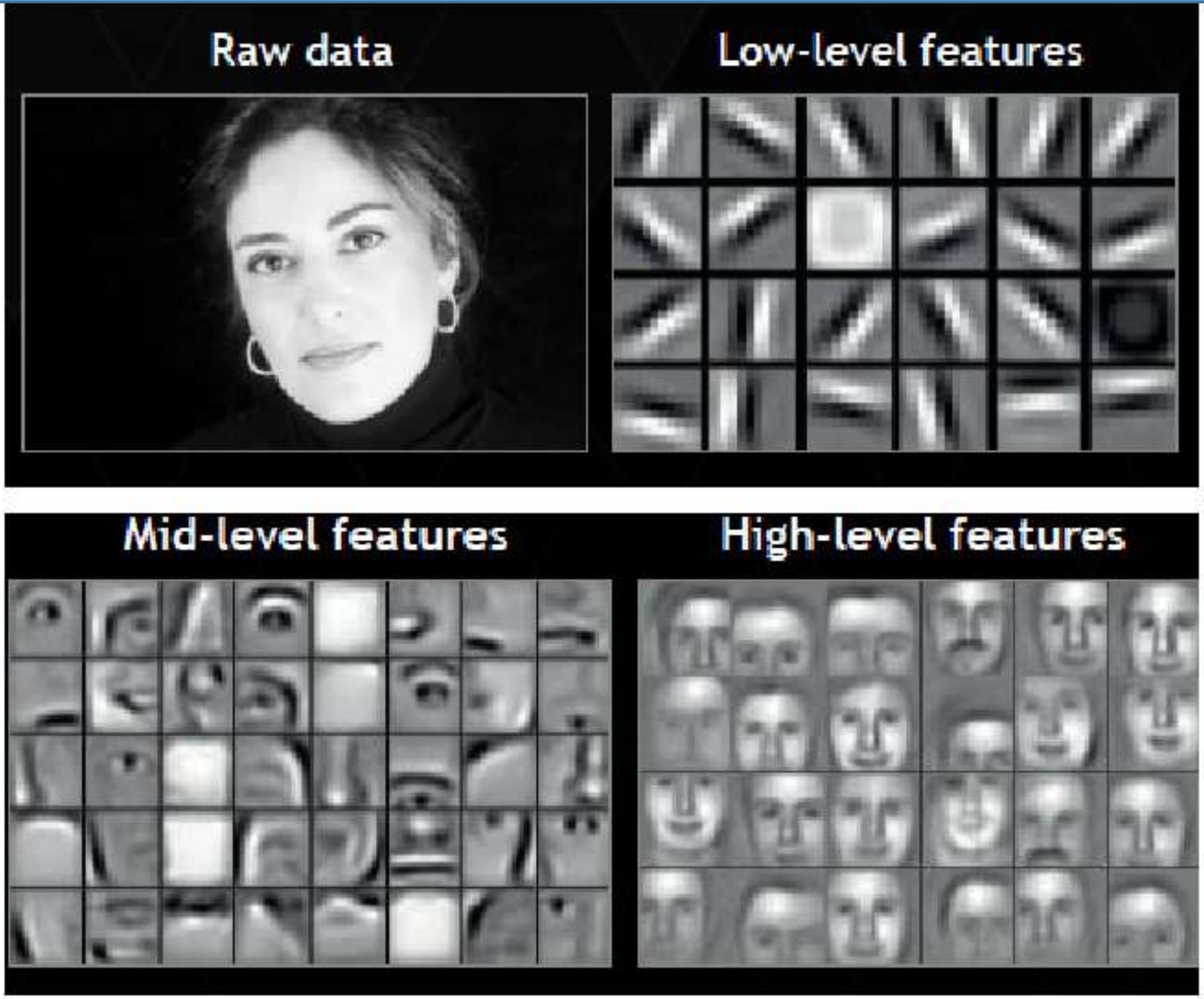
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Visual features in an image

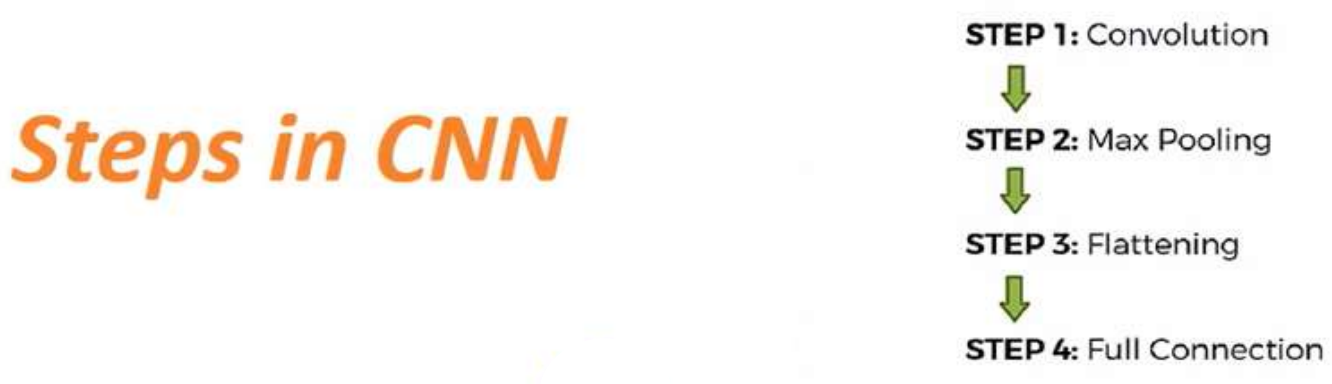


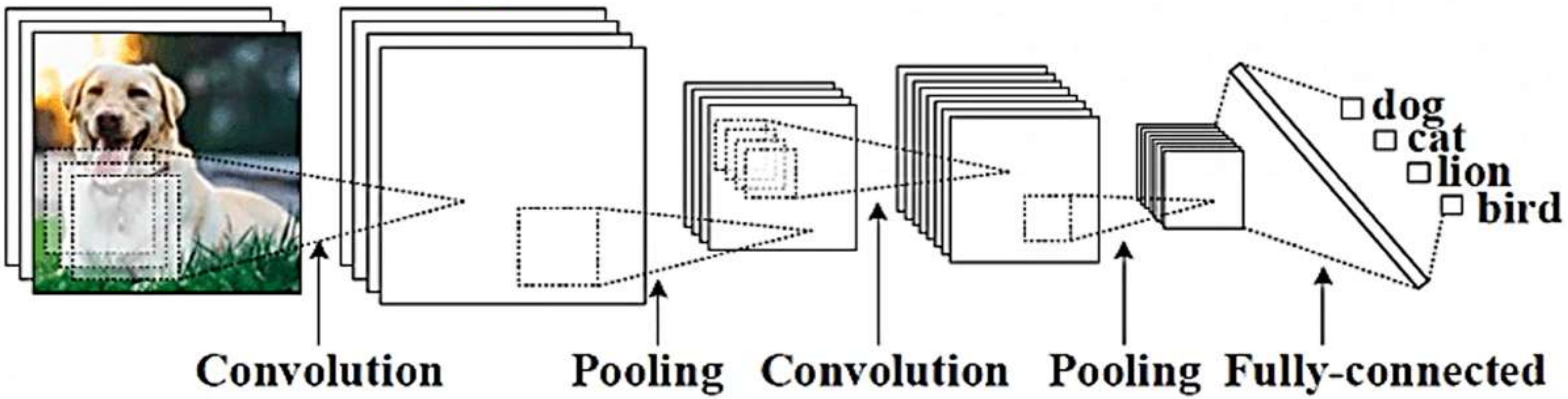
Feature Visualization of Convolutional net trained on ImageNet (from Matthew D. Zeiler and Rob Fergus)

Visual features in an image

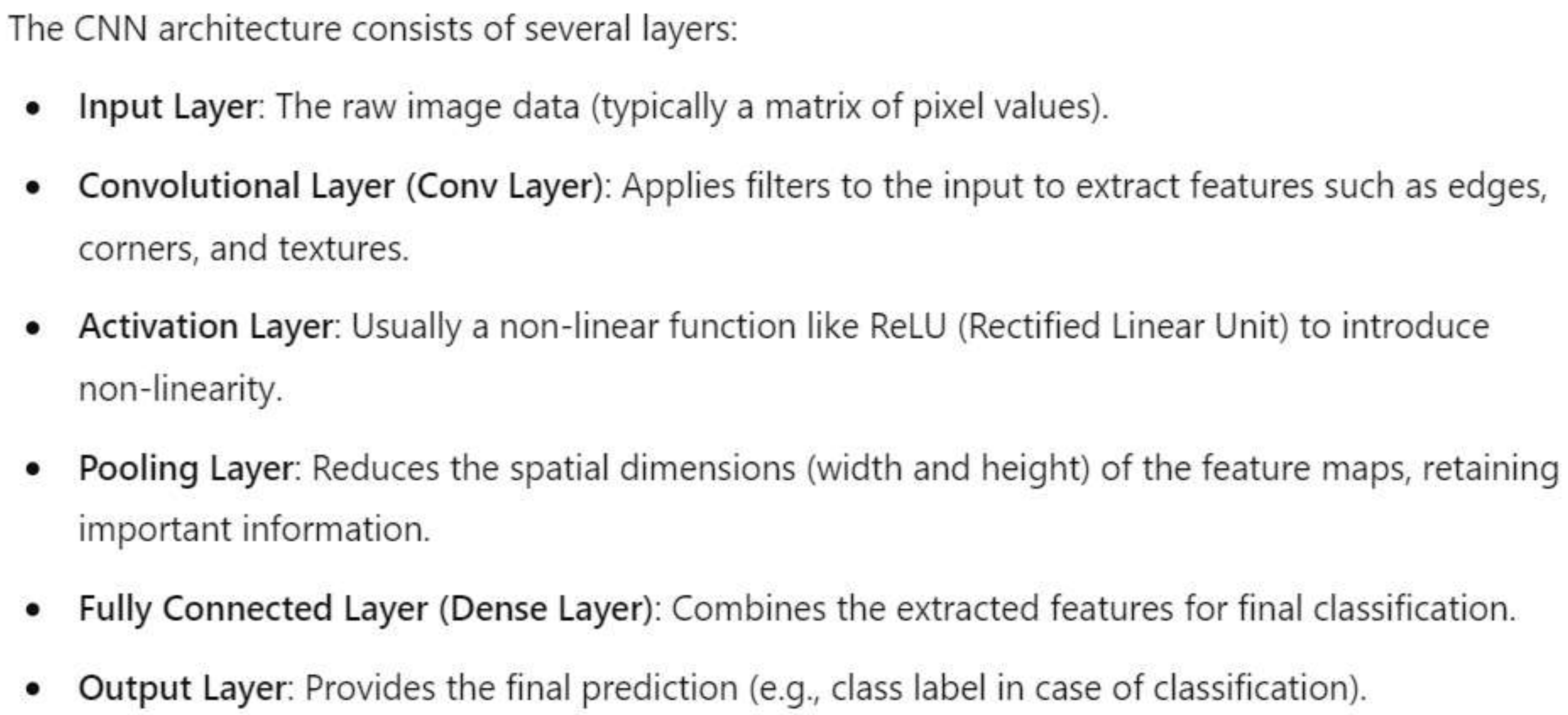


Basic Components of CNN





CNN architecture



The CNN architecture consists of several layers:

- **Input Layer**: The raw image data (typically a matrix of pixel values).
- **Convolutional Layer (Conv Layer)**: Applies filters to the input to extract features such as edges, corners, and textures.
- **Activation Layer**: Usually a non-linear function like ReLU (Rectified Linear Unit) to introduce non-linearity.
- **Pooling Layer**: Reduces the spatial dimensions (width and height) of the feature maps, retaining important information.
- **Fully Connected Layer (Dense Layer)**: Combines the extracted features for final classification.
- **Output Layer**: Provides the final prediction (e.g., class label in case of classification).

Convolutional Layer

- The most important building block of a CNN is the **convolutional layer**.
- The **neurons in the first convolutional layer** are not **connected to** every single pixel in the input image (like they were simple ANN), but only to **pixels in their receptive fields** (see Figure 13-2).
- Each neuron in the **second convolutional layer** is **connected** only to neurons located **within a small rectangle in the first layer**.
- This **architecture allows** the network to **concentrate on low-level** features in the first hidden layer, then **assemble them into higher-level** features in the next hidden layer, and so on.
- This **hierarchical structure** is common in real-world images, which is one of the reasons **why CNNs work so well for image recognition**.

Note: Until now, all multilayer neural networks we looked at had layers composed of a long line of neurons, and we **had to flatten input images to 1D before** feeding them to the neural network. **Now each layer is represented in 2D**, which makes it easier to match neurons with their corresponding inputs.

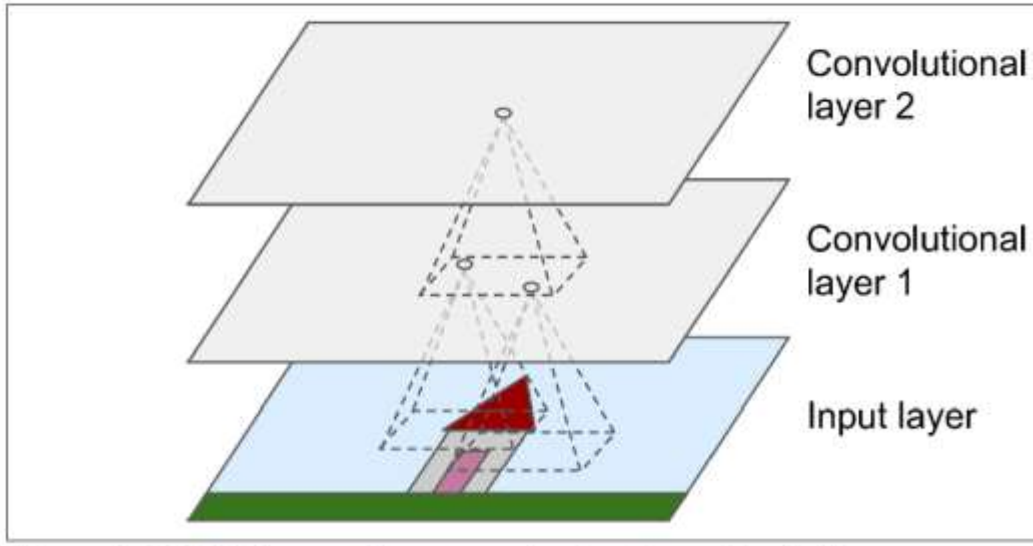


Figure 13-2. CNN layers with rectangular local receptive fields

Convolutional Layer

- Convolution demo link:
- https://hannibunny.github.io/mlbook/neuralnetworks/convolutionDemos.html
- https://medium.com/@er_95882/convolution-image-filters-cnns-and-examples-in-python-pytorch-bd3f3ac5df9c
- In digital image processing in particular, **convolution** is a mathematical method for **combining two images** to **produce a third image**.
- Typically, one of **the two combined images is not an image itself**, but **a matrix** whose size and values determine the **nature of the effect of the convolution process**; this matrix is called a **filter or kernel**.

Filters

- A **neuron's weights** can be **represented** as a **small image the size of the receptive field**.
- The **receptive field** refers to the **specific area of the input space** (e.g., a portion of an image) that a particular neuron or unit is "responsible" for or sensitive to.
- For example, Figure 13-5 shows two possible sets of **weights, called filters** (or **convolution kernels**).
- The first one is represented as a **black square with a vertical white line** in the middle (it is a 7 × 7 matrix full of 0s except for the central column, which is full of 1s);
- Neurons using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will get multiplied by 0, except for the ones located in the central vertical line).
- The **second filter** is **a black square with a horizontal white line in the middle**.
- Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

Filters

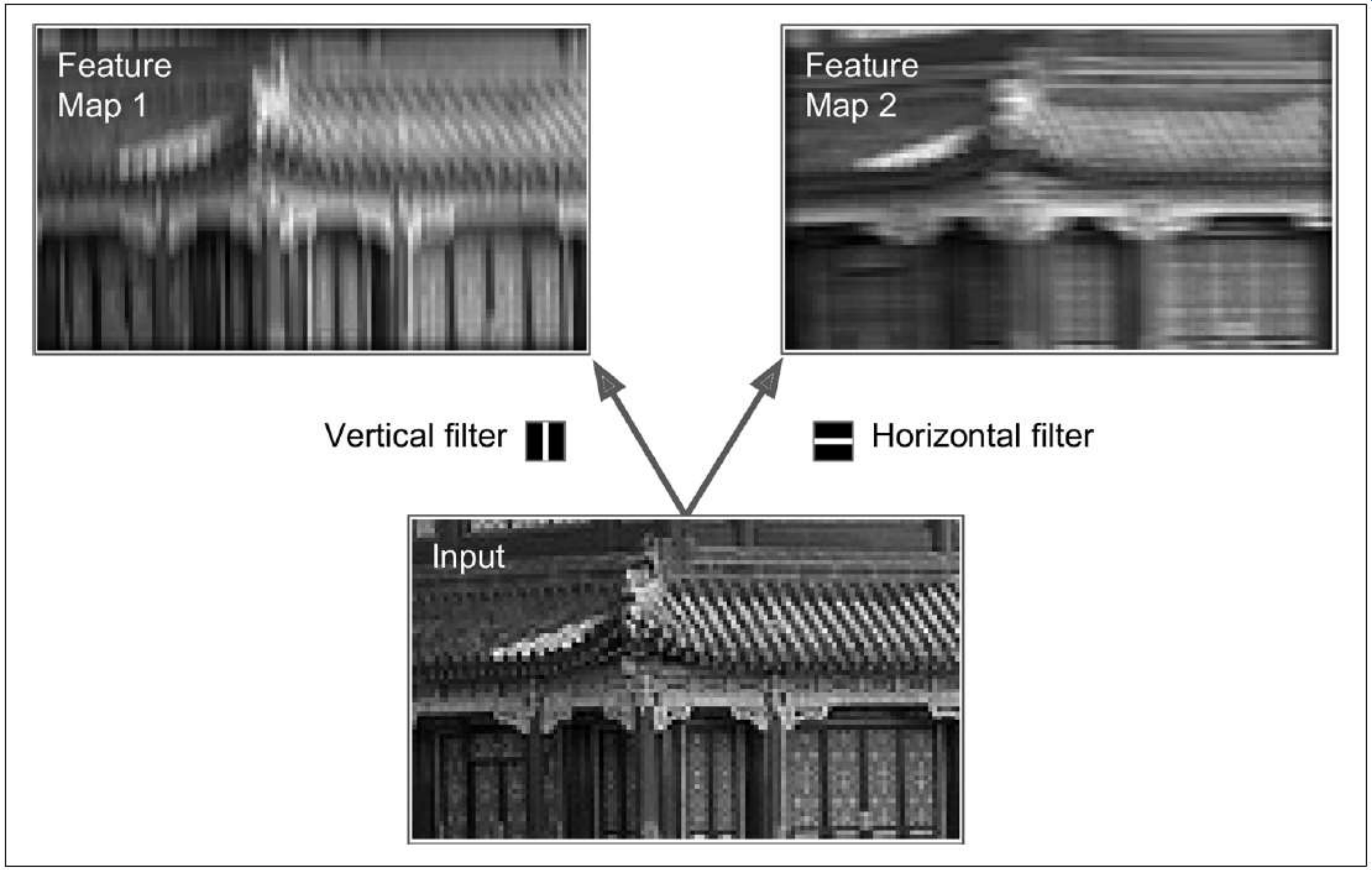


Figure 13-5. Applying two different filters to get two feature maps

Filters

- Now if all neurons in a layer use the **same vertical line filter** (and the same bias term), and you feed the network the input image shown in Figure 13-5 (bottom image), the layer will **output the top-left image**. Notice that the **vertical white lines get enhanced** while the **rest gets blurred**.
- Similarly, the **upper-right image** is what you get if all neurons **use the horizontal line filter**; notice that the **horizontal white lines get enhanced** while the **rest is blurred out**.
- Thus, a layer full of neurons using the same **filter gives you a feature map**, which **highlights the areas** in an image that **are most similar to the filter**.
- **During training, a CNN finds the most useful filters** for its task, and it **learns to combine them** into more **complex patterns** (e.g., a cross is an area in an image where both the vertical filter and the horizontal filter are active).

Filters

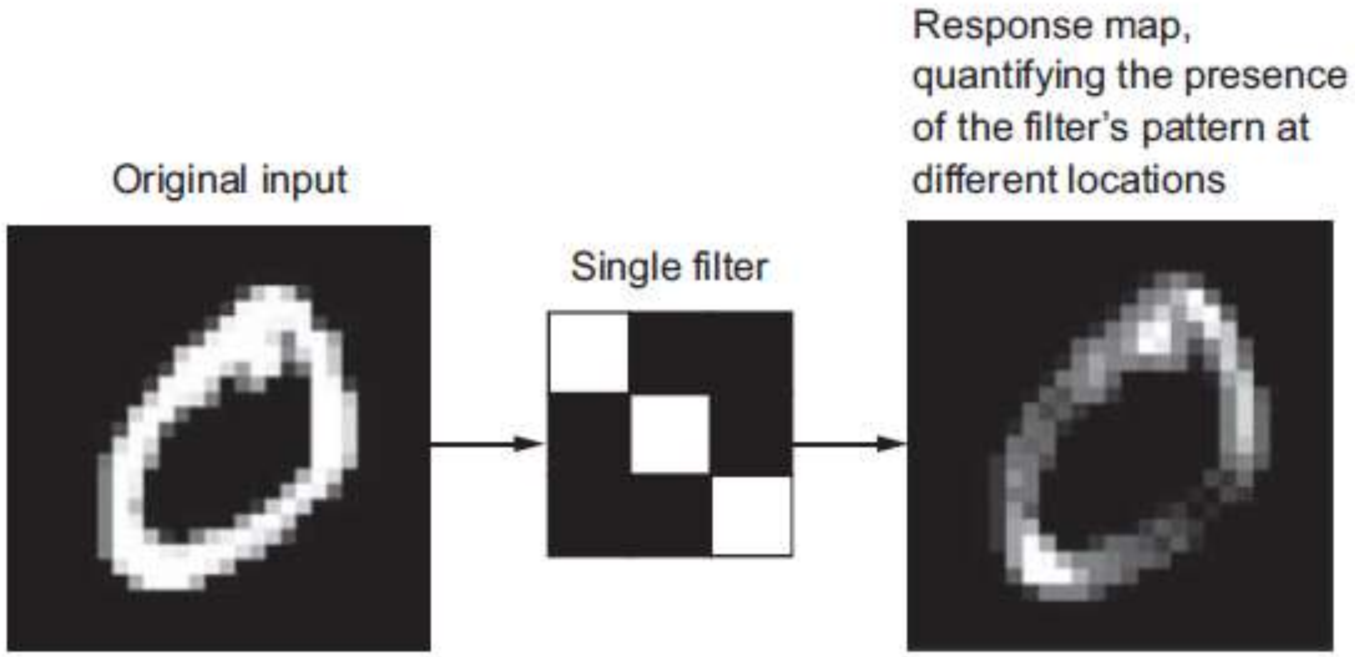

Original input
Single filter
Response map, quantifying the presence of the filter's pattern at different locations

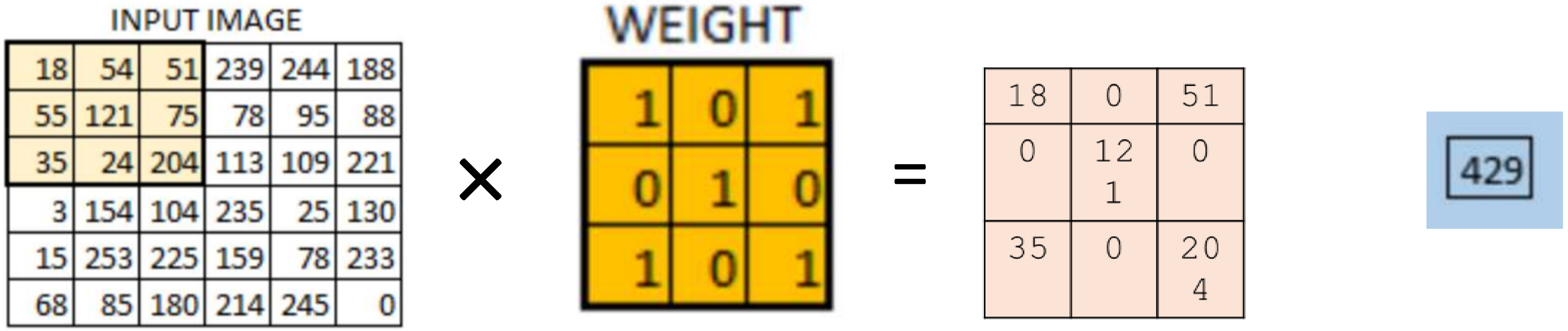
Figure 5.3 The concept of a *response map*: a 2D map of the presence of a pattern at different locations in an input

Filters

- Pixel values are used again when the weight matrix moves along the image.
- This basically enables parameter sharing in a convolutional neural network.
- The weight matrix behaves like a filter in an image, extracting useful information from the original image matrix.
- A weight combination might be extracting edges, while another one might a particular color, while another one might just blur the unwanted noise.
- The weights are learnt such that the loss function is minimized and extract features from the original image which help the network in correct prediction.
- When we use multiple convolutional layers, the initial layer extract more generic features, and as network gets deeper the features get complex

Filters

The value 429, is obtained by the adding the values obtained by element wise multiplication of the weight matrix and the highlighted 3 x 3 part of the input image.



INPUT IMAGE

18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

×

WEIGHT

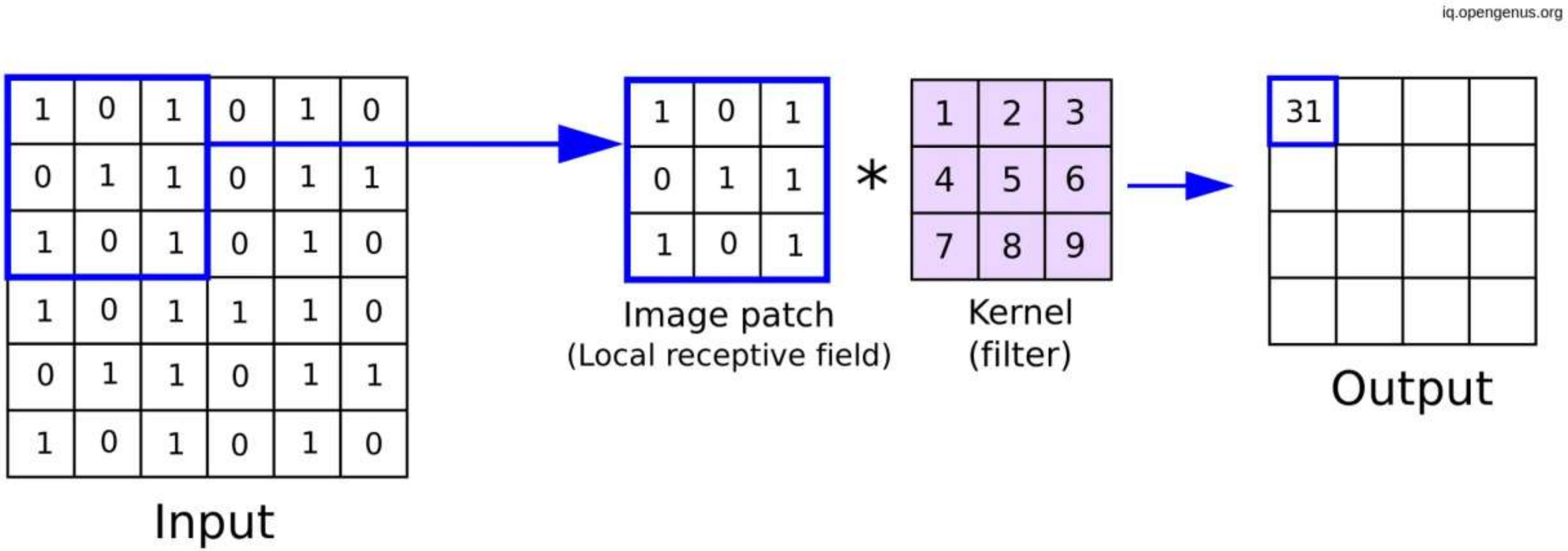
1	0	1
0	1	0
1	0	1

=

18	0	51
0	121	0
35	0	204

429

Filters

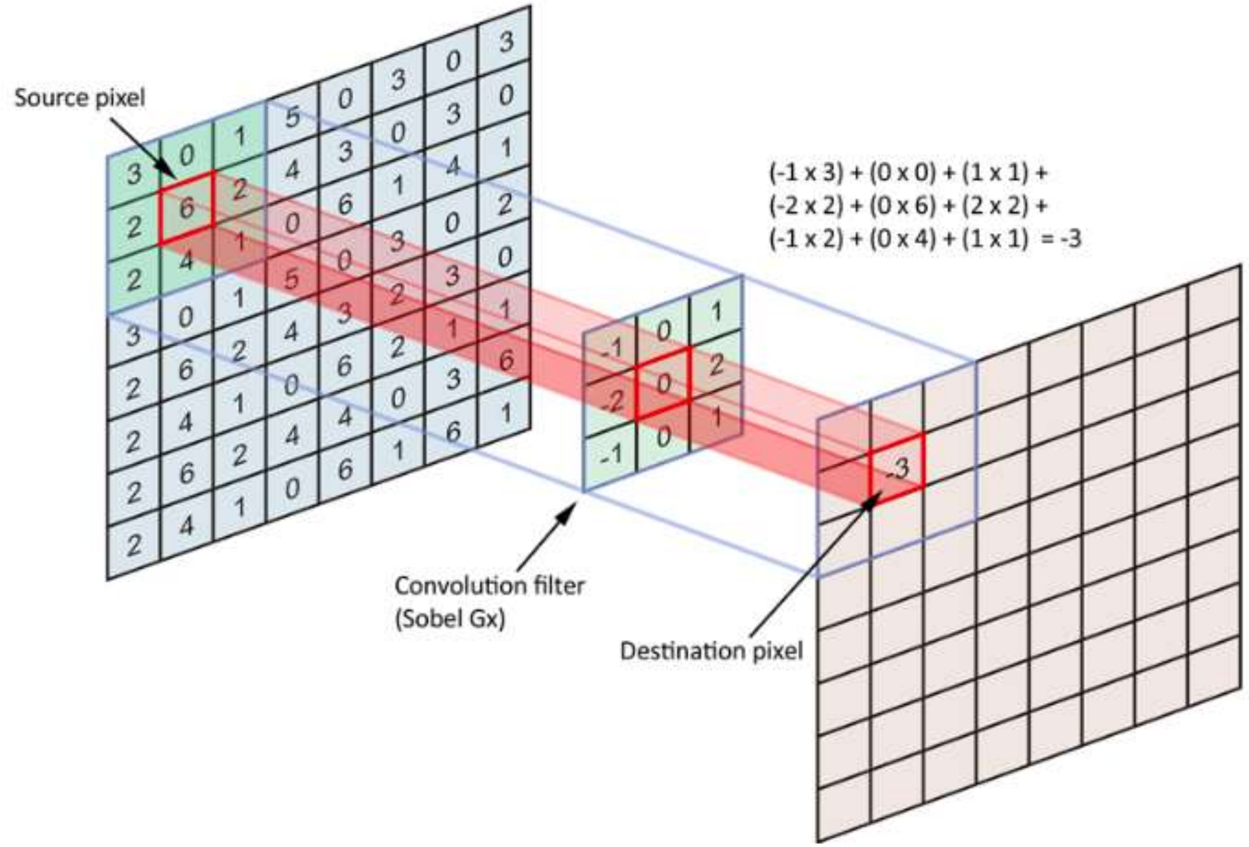


iq.opengenus.org
1 0 1 0 1 0
0 1 1 0 1 1
1 0 1 0 1 0
1 0 1 1 1 0
0 1 1 0 1 1
1 0 1 0 1 0
Input
1 0 1
0 1 1
1 0 1
Image patch
(Local receptive field)
*
1 2 3
4 5 6
7 8 9
Kernel
(filter)
31
Output

Filters

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

4		


Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

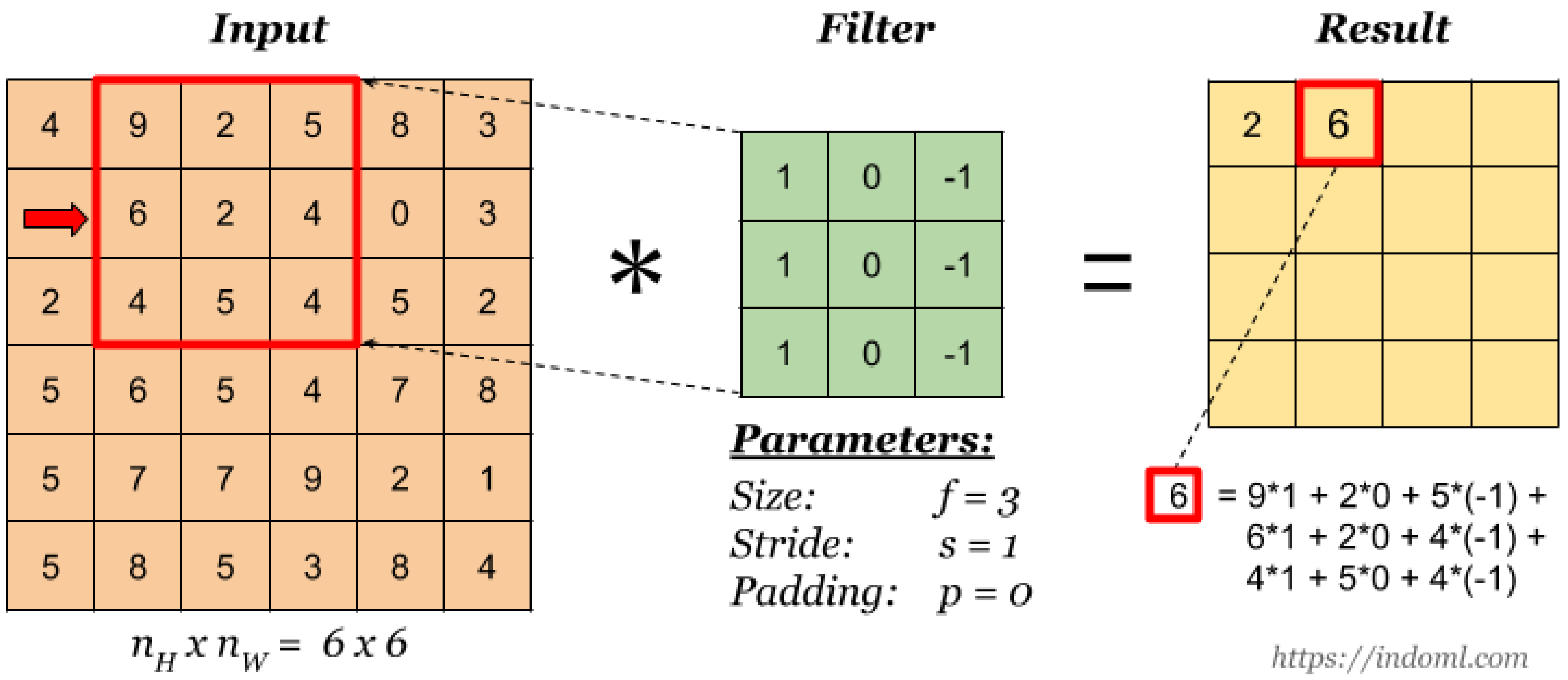
Animation source: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Filters - Homework

- Visit the following link, explore various filters and apply it on any basic image in python

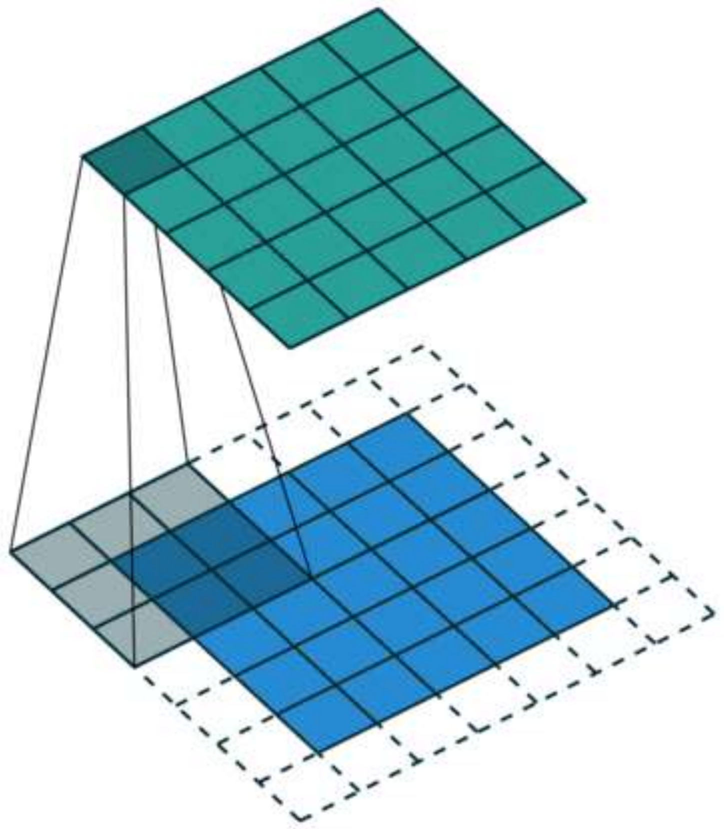
https://setosa.io/ev/image-kernels/

Filters (stride, filter size and padding)



Filters (stride, filter size and padding)

- **Padding** refers to the process of **adding extra pixels** (usually **zeros**) **around the borders** of an image before it is passed through a convolutional layer in a CNN.
- It **controls the Output Size**: Helps maintain the spatial dimensions of the output feature map relative to the input, **avoiding size reduction**.
- Padding **Ensures that edge features** are adequately **captured** during convolution operations.
- Types of Padding:
  - **Valid Padding**: No padding is applied, potentially reducing the output size.
  - **Same Padding**: Padding is added so that the output size is the same as the input size.
  - Example: Show a small grid (e.g., 5x5) with padding applied around the edges, resulting in a larger grid (e.g., 6x6) before convolution.



Filters (stride, filter size and **padding**)

Zero Padding: For a layer to have the same height and width as the previous layer, add zeros around the inputs.

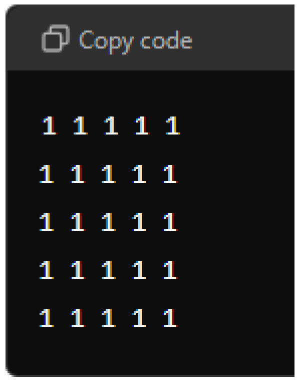
- **SAME** padding is a type of padding used in convolutional neural networks (CNNs) to ensure that the output feature map has the same spatial dimensions (height and width) as the input.
- This is achieved by **adding zeros around the border** of the input image.

Example: SAME Padding with a 3x3 Filter
Case 1: 1x1 Stride
Input Image: 5x5 (example)
Filter Size: 3x3
Stride: 1x1
Padding: SAME

$$P = \frac{K - 1}{2}$$

P = padding size
K = kernel size

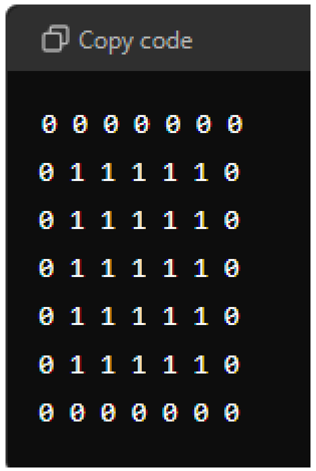
Input Image (5x5):



Copy code

```
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
```

Padded Image (7x7):



Copy code

```
0 0 0 0 0 0 0
0 1 1 1 1 1 0
0 1 1 1 1 1 0
0 1 1 1 1 1 0
0 1 1 1 1 1 0
0 1 1 1 1 1 0
0 0 0 0 0 0 0
```

$$\text{Feature Map Size} = \frac{N + 2P - K}{S} + 1$$

P = Padding size
K = Kernel size
S = Stride

Filters (stride, filter size and **padding**)

VALID

- convolutional layer does *not* use zero padding
- may ignore some rows and columns at the bottom and right of the input image depending on the stride

SAME

- convolutional layer uses zero padding if necessary.
- the number of output neurons is equal to the number of input neurons divided by the stride, rounded up
  - ceil (13 / 5) = 3
- zeros are added as evenly as possible around the inputs.

For simplicity, only the horizontal dimension is shown, but same logic applies to the vertical dimension too

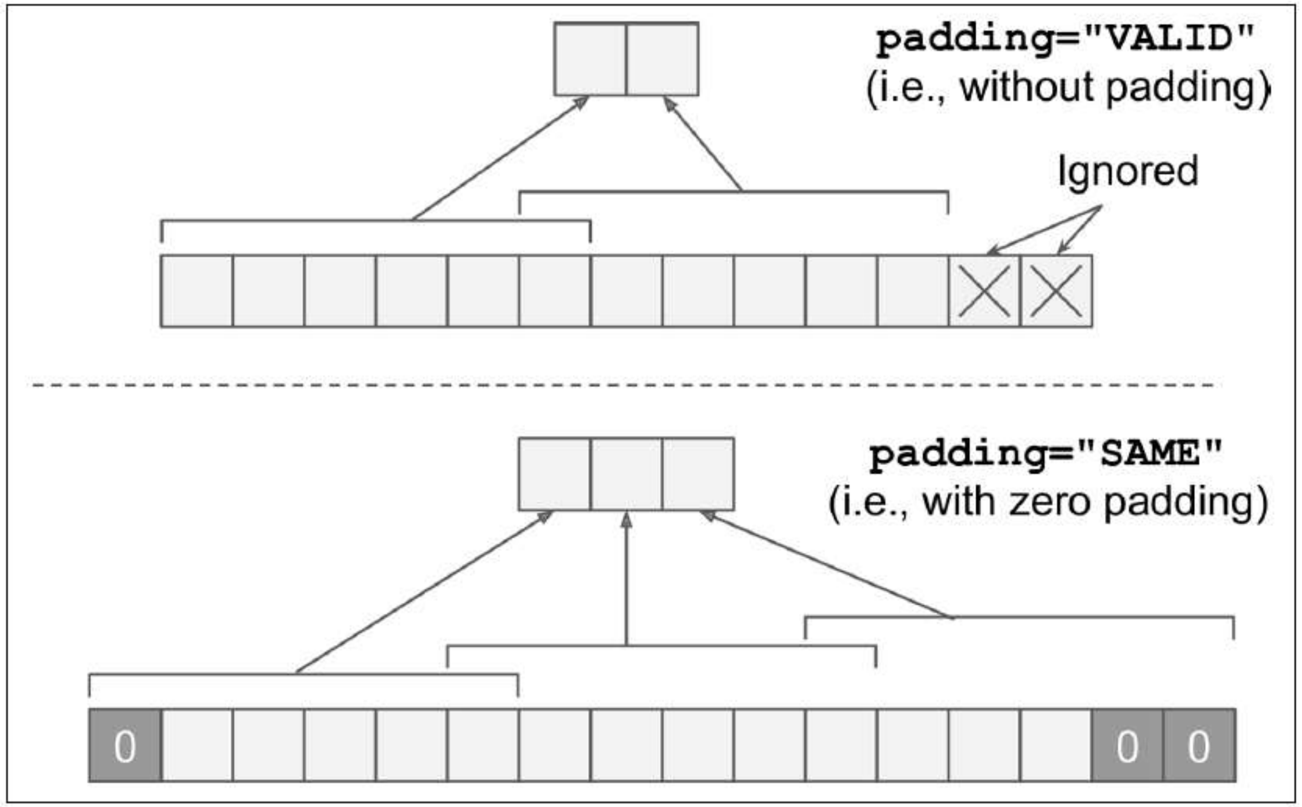
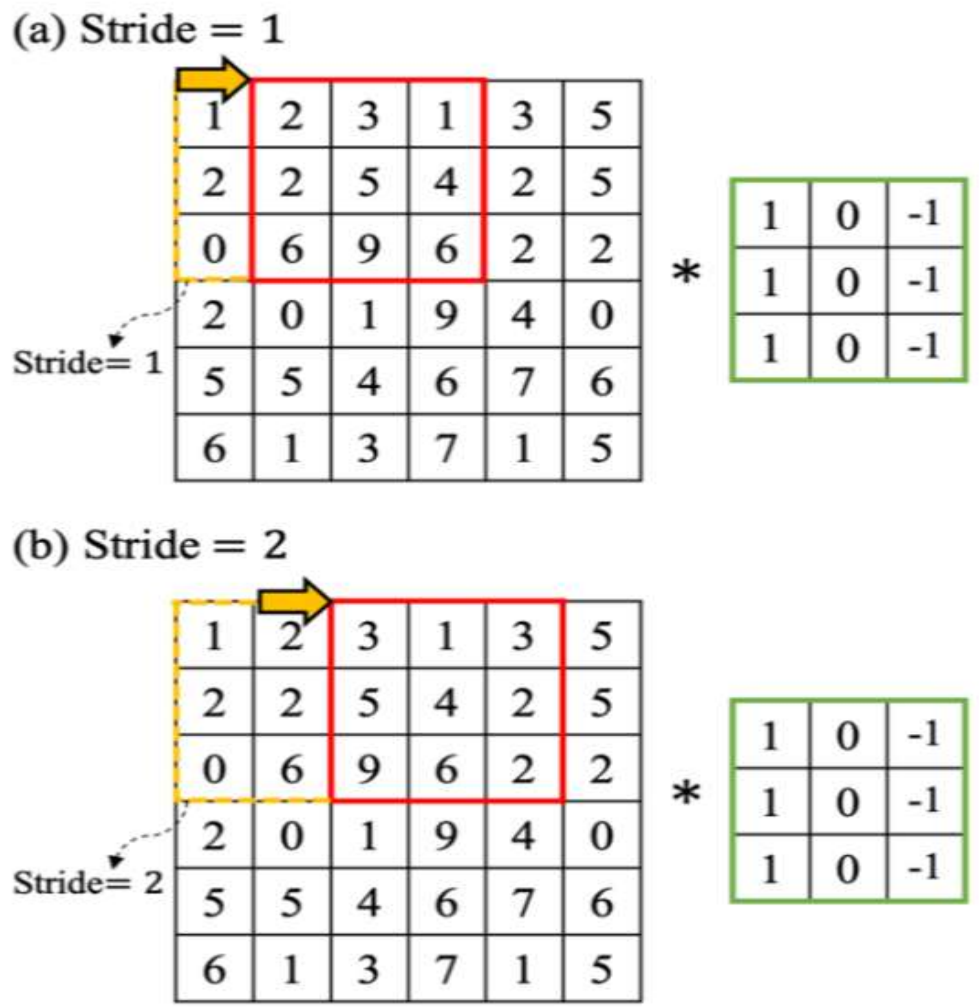


Figure 13-7. Padding options—input width: 13, filter width: 6, stride: 5

Filters (**stride**, filter size and padding)

Stride: The distance between two consecutive receptive fields.

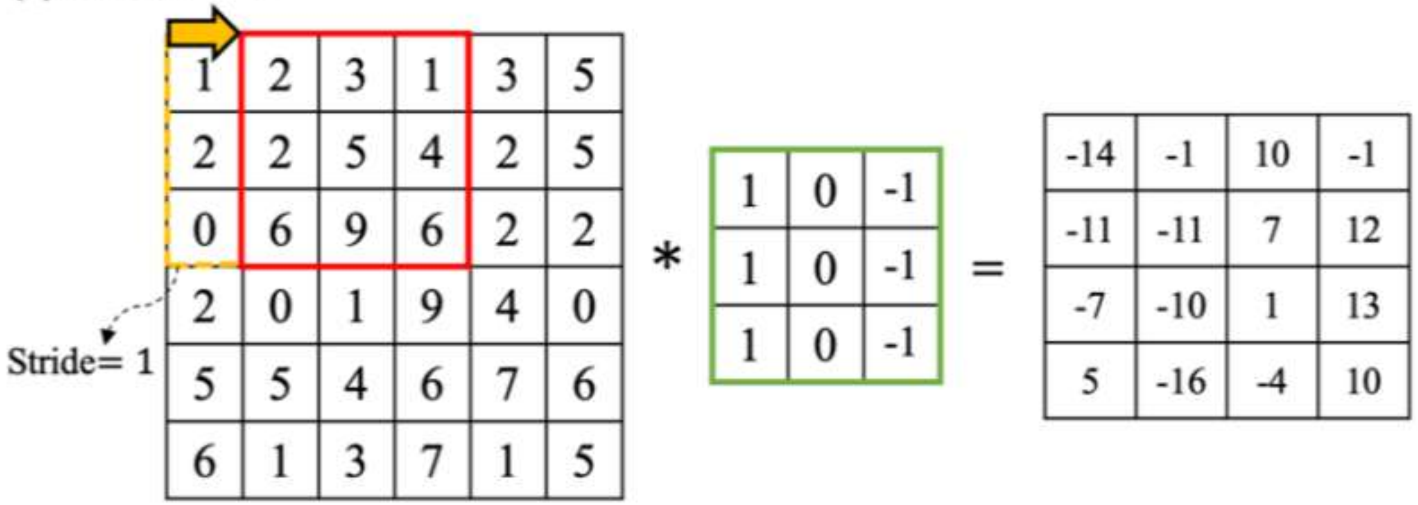


????

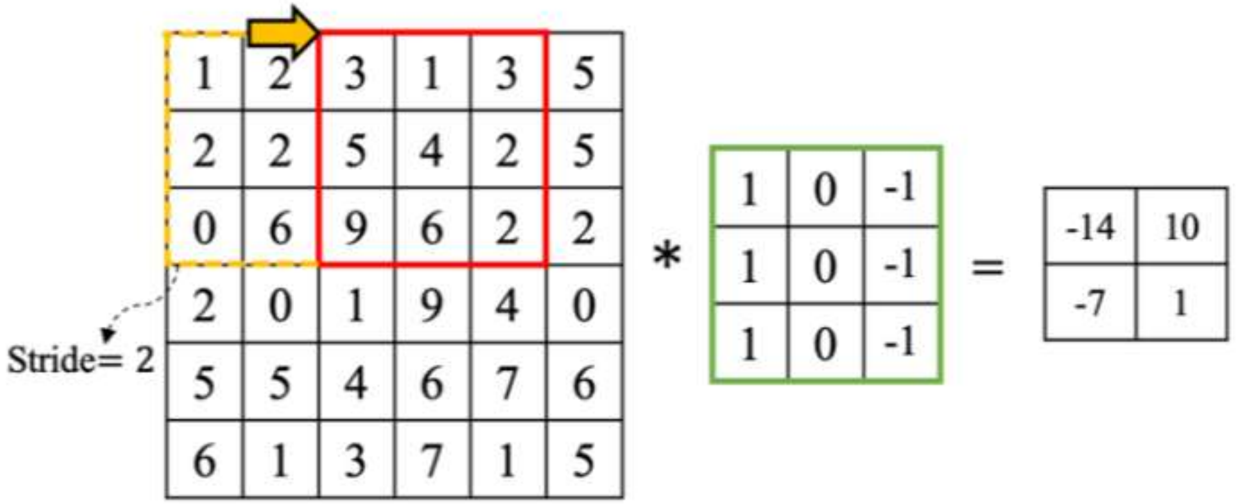
????

Filters (**stride**, filter size and padding)

(a) Stride = 1



(b) Stride = 2



Stacking Multiple Feature Maps

- Convolutional layer is represented as a **thin 2D layer**.
- However, in reality it is composed of **several feature maps of equal sizes**, so it is more **accurately represented in 3D** (see Figure 13-6).
- A **feature map** is the **output of a convolutional layer after applying a filter** (or **kernel**) to the input data.
- Within **one feature map**, all neurons **share the same parameters** (weights and bias term), but different feature maps may have different parameters.
- A **neuron's receptive field** is the same as described earlier, but it extends **across all the previous layers' feature maps**.
- In short, **a convolutional layer simultaneously applies multiple filters** to its inputs, making it capable of **detecting multiple features** anywhere in its inputs.

Fact: The fact that all neurons in a feature map share the same parameters **dramatically reduces the number of parameters** in the model, but most importantly it means that once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location.

Stacking Multiple Feature Maps

- Moreover, input images are also composed of **multiple sublayers**: one per color channel.
- There are typically three: red, green, and blue (**RGB**).
- **Grayscale** images have just **one channel**, but some images may have much more—for example, satellite images that capture extra light frequencies (such as infrared).

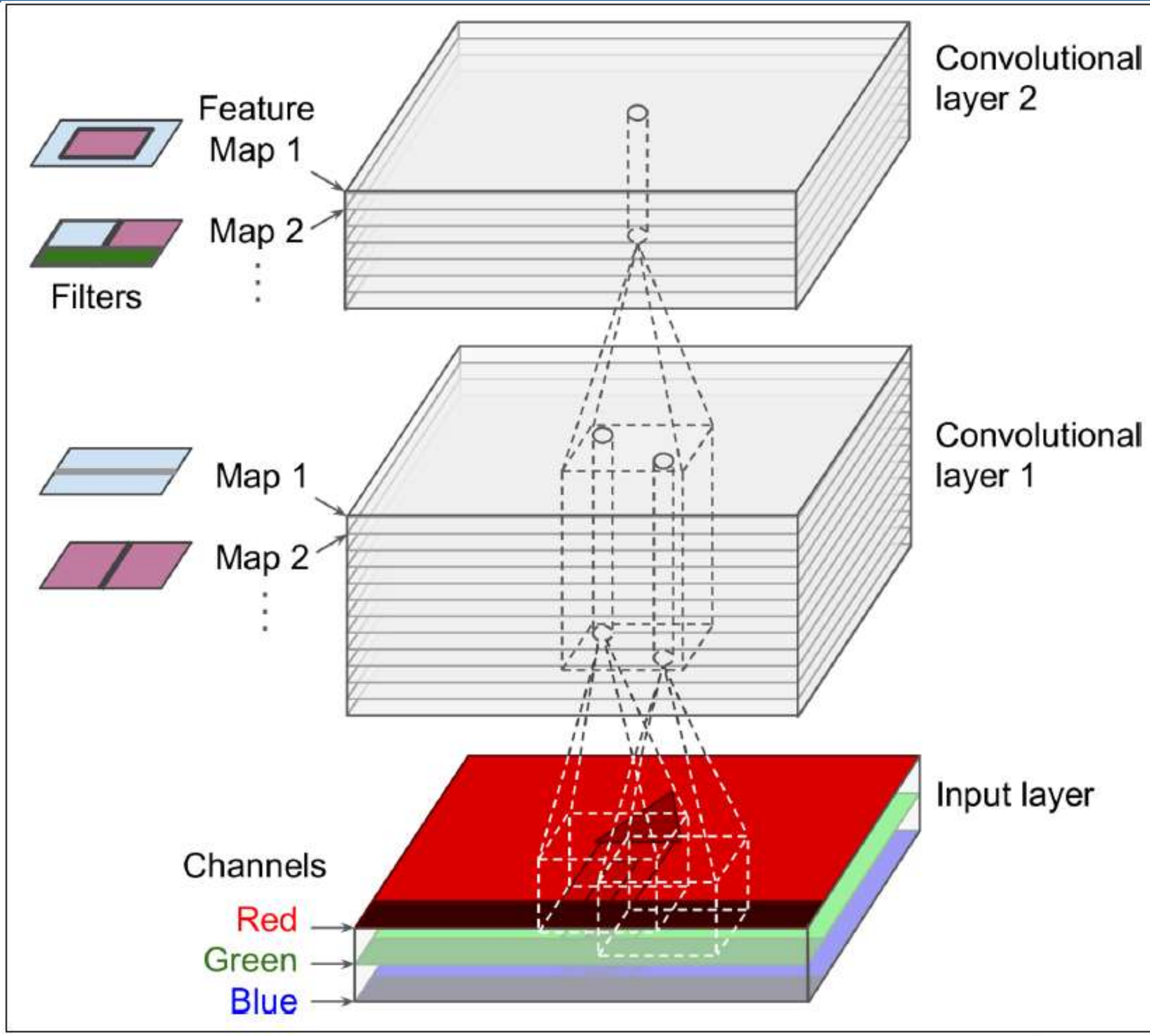


Figure 13-6. Convolution layers with multiple feature maps, and images with three channels

Pooling Layer

- The pooling layers goal is to subsample/down sample (i.e., shrink) the **input image** to **reduce the computational load**, the **memory usage**, and the **number of parameters** (thereby **limiting** the risk of **overfitting**).
- Reducing the input image size also makes the neural network tolerate a little bit of **image shift** (**location invariance**).
- Just **like in convolutional layers**, **each neuron** in a pooling layer is **connected to** the outputs of a **limited number of neurons in the previous layer**, located within a small rectangular receptive field.
- **You must define** its size, the stride, and the padding type, just like before.
- However, a **pooling neuron has no weights**; all it does is aggregate the inputs using an aggregation function such as the max or mean.

Pooling Layer

- Figure 13-8 shows a **max pooling layer**, which is the most common type of pooling layer.
- In this example, we use a 2 × 2 pooling kernel, a stride of 2, and no padding.
- Note that only the max input value in each kernel makes it to the next layer.
- The other inputs are dropped.

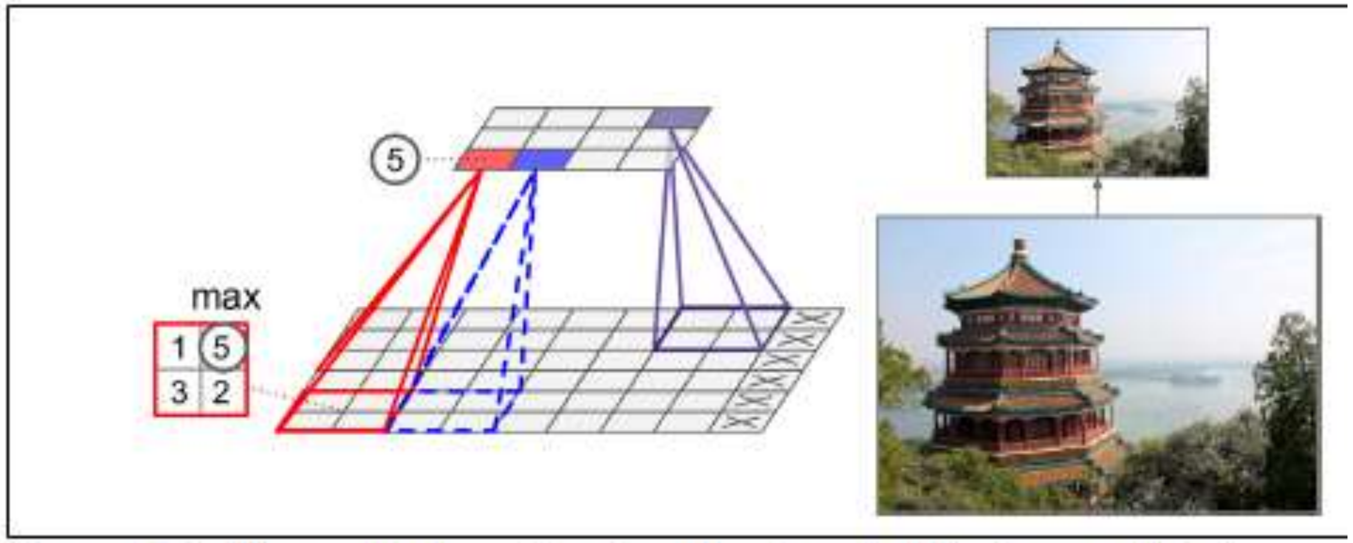
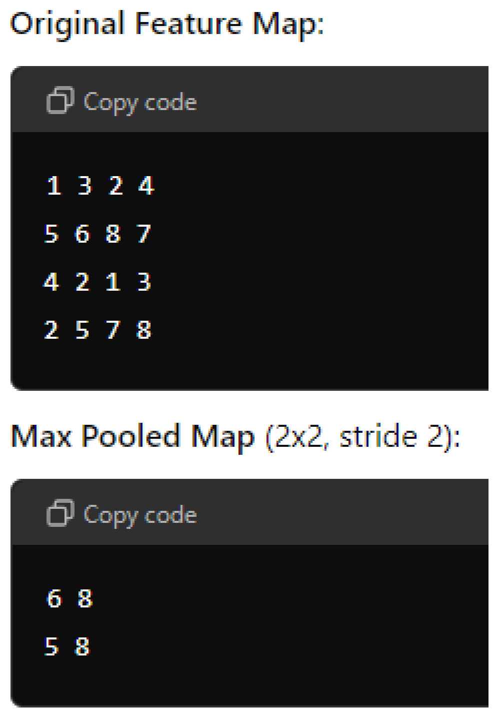


Figure 13-8. Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

Pooling Layer

- Common **types** of pooling layer are:
- **Max Pooling**:
  - Takes the maximum value in each patch of the feature map.
- **Average Pooling**:
  - Takes the average value of each patch.
- Max pooling example:
  - Input image 4 x 4
  - Max kernel 2 x 2
  - Output 2 x 2

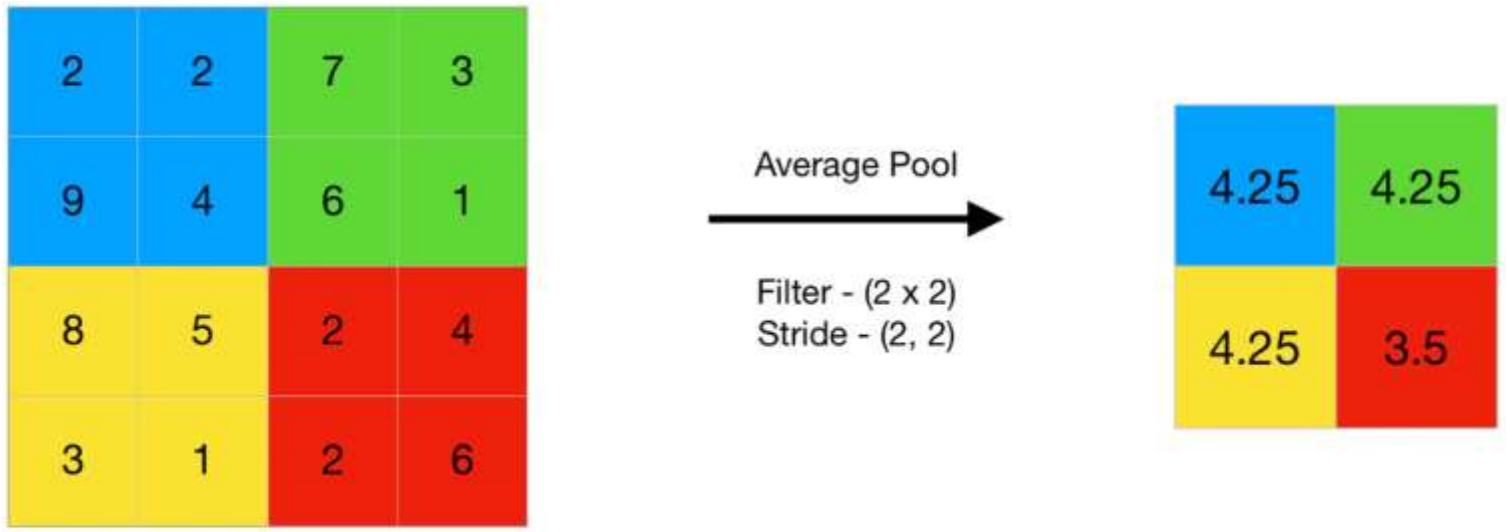

Original Feature Map:

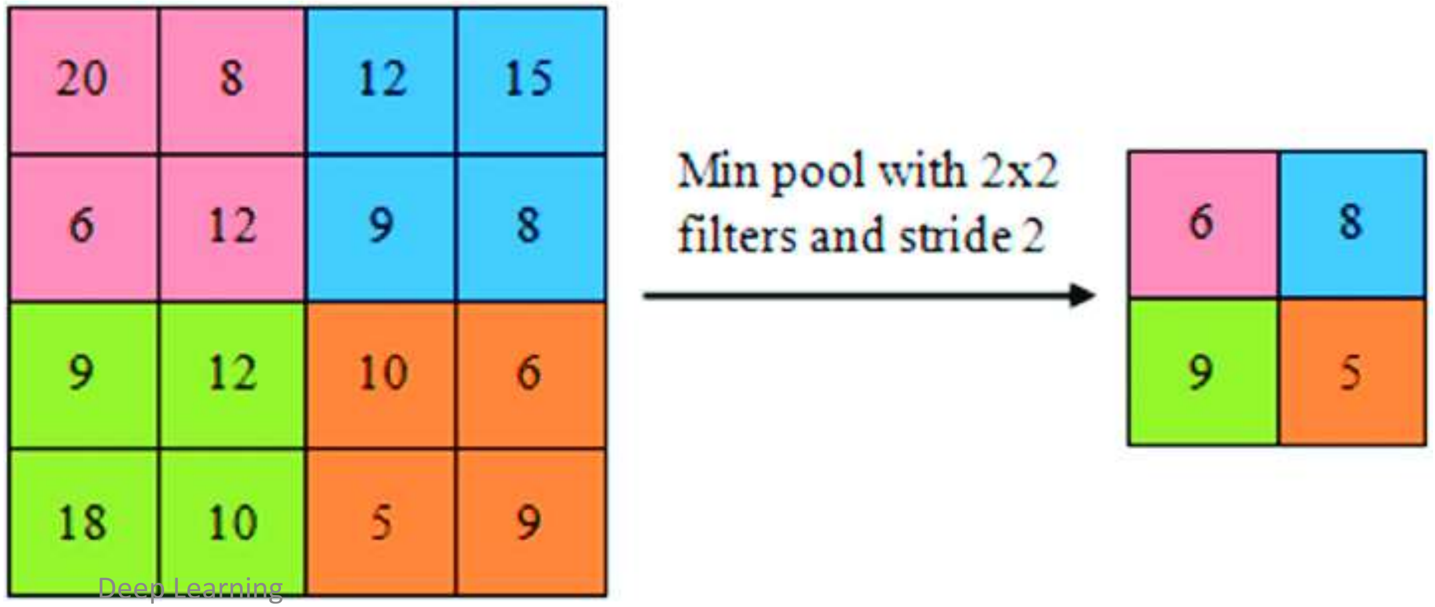
```
1 3 2 4
5 6 8 7
4 2 1 3
2 5 7 8
```

Max Pooled Map (2x2, stride 2):

```
6 8
5 8
```

Max/Min Pooling





Weights and Bias in CNN

- A filter (kernel) slides over the input, performing **element-wise multiplication** between the filter and the input region it covers (receptive field).
- If you are using a kernel (filter) at any convolutional layer, e.g., with the size of **3 x 3**. This kernel mean **9 weights on its receptive field** in an image or input.
- The result is a weighted sum for each location:

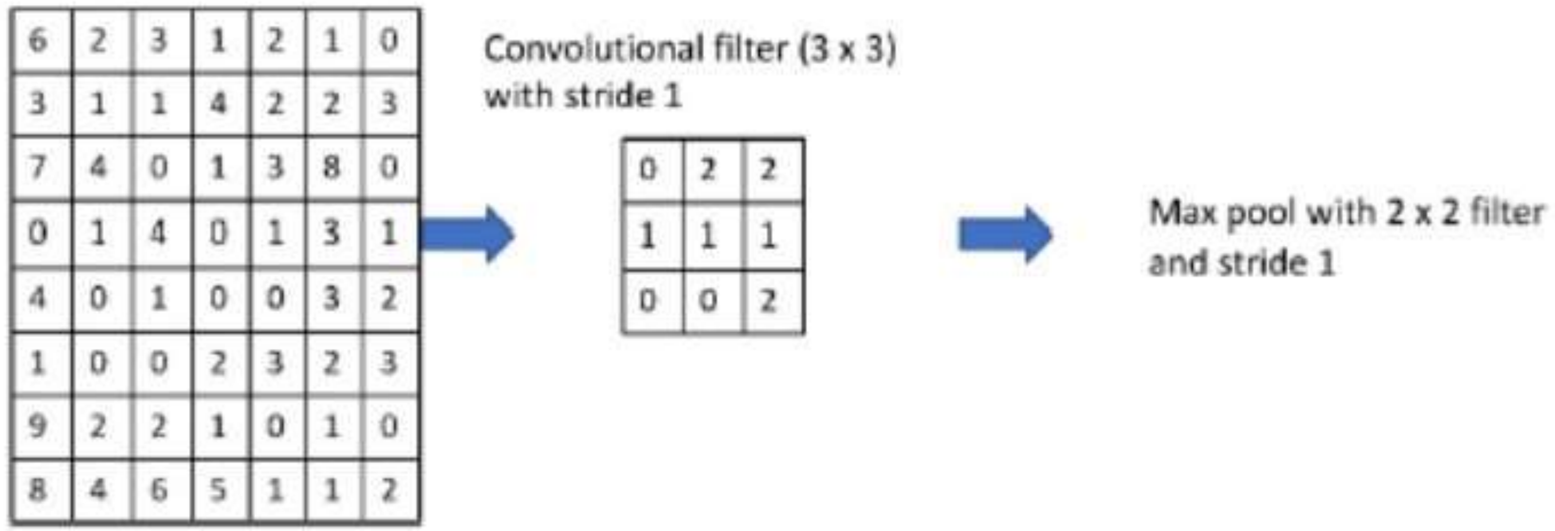
$$z(x, y) = \sum_{i=1}^{h} \sum_{j=1}^{w} W_{ij} \cdot X(x+i, y+j)$$

  - W_{ij} = Filter weights
  - X(x+i,y+j) = Input pixel values
  - Bias is a scalar value added to the weighted sum at each location in the feature map.

$$z(x, y) = \left(\sum W \cdot X\right) + b$$

Practice Example

Given a convolutional neural network (CNN) with convolutional layer and a pooling layer as shown in Figure, the convolutional layer is a filter of size 3*3 with a stride of 1, and the pooling layer is max pooling filter of size 2*2 with a stride of 1.


Convolutional filter (3 x 3) with stride 1
Max pool with 2 x 2 filter and stride 1

6	2	3	1	2	1	0
3	1	1	4	2	2	3
7	4	0	1	3	8	0
0	1	4	0	1	3	1
4	0	1	0	0	3	2
1	0	0	2	3	2	3
9	2	2	1	0	1	0
8	4	6	5	1	1	2

0	2	2
1	1	1
0	0	2

Given an input image of size 8*7 as shown above, give the output after the convolutional layer and output after the pooling layer.

CNN Architecture

- Typical CNN architectures **stack a few convolutional layers** (each one generally followed by a ReLU layer),
- then a **pooling layer**, then **another few convolutional layers** (+ReLU),
- then another pooling layer, **and so on**.
- The image **gets smaller and smaller** as it progresses through the network,
- but it also typically **gets deeper and deeper** (i.e., with more feature maps) thanks to the convolutional layers (see Figure 13-9).

CNN Architecture

- At the **top** of the stack, a regular **feedforward neural network** is added, composed of a **few fully connected layers** (+ReLUs),
- and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

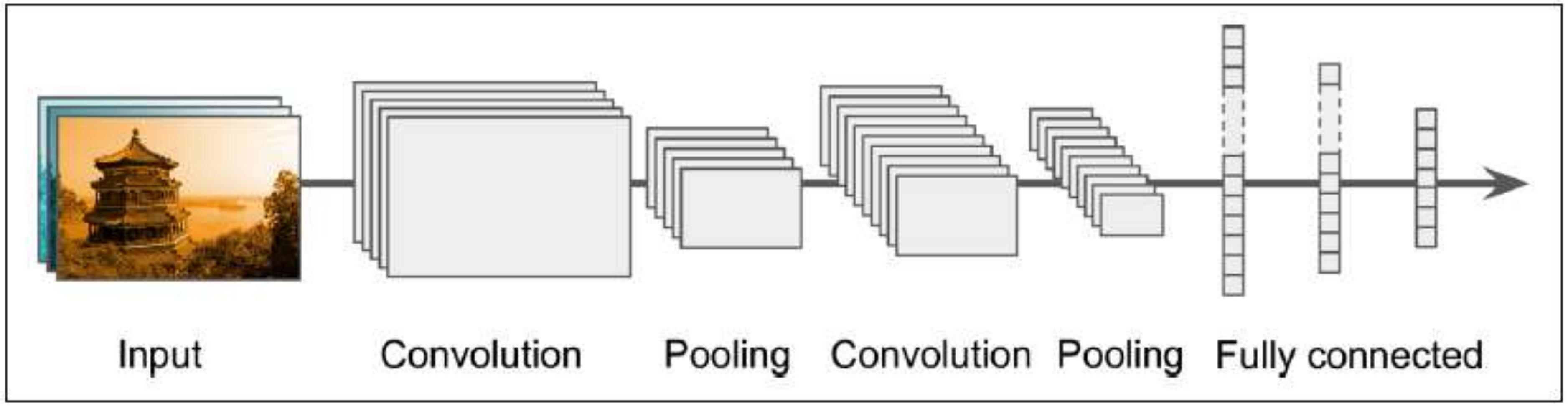


Figure 13-9. Typical CNN architecture

CNN Architecture

- A common **mistake** is to use **convolution kernels that are too large**.
- You can often get the same effect as a 9 × 9 kernel by stacking two 3× 3 kernels on top of each other, for a lot less compute.

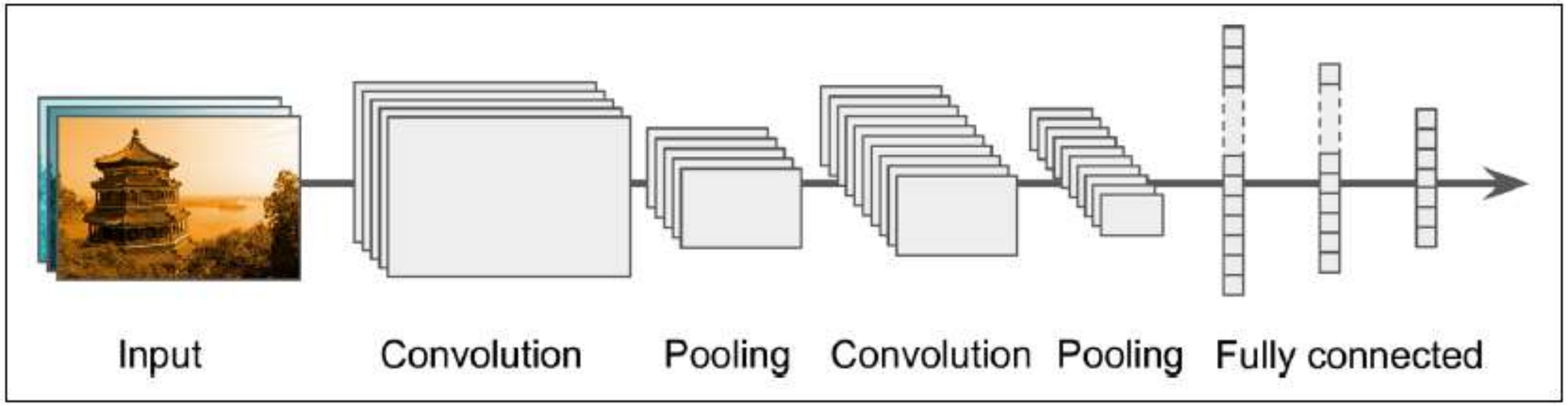


Figure 13-9. Typical CNN architecture

CNN example

```
from keras import layers
from keras import models
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

```
________________________________________________________________
Layer (type)                     Output Shape          Param #
================================================================
conv2d_1 (Conv2D)                (None, 26, 26, 32)    320
________________________________________________________________
maxpooling2d_1 (MaxPooling2D)    (None, 13, 13, 32)    0
________________________________________________________________
conv2d_2 (Conv2D)                (None, 11, 11, 64)    18496
________________________________________________________________
maxpooling2d_2 (MaxPooling2D)    (None, 5, 5, 64)      0
________________________________________________________________
conv2d_3 (Conv2D)                (None, 3, 3, 64)      36928
================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

CNN example (break down of layers)

- **conv2d_1 (Conv2D)**
  - **Input shape**: Not provided directly, but based on the output shape, we can infer the input was likely 28×28 (commonly for MNIST-like datasets) and grayscale (1 channel).
  - **Filter size**: 3x3
  - **Number of filters**: 32
  - **Input channels**: 1 (since grayscale)
  - **Output shape**: 26×26×32 (height, width, number of filters). The output is 26x26 as it doesn't have padding.
  - **Parameters calculation:**
  - 3 x 3 is filter size that will be applied by current layer
  - 32 different filters of size 3 x 3 will be convolved with input feature maps
  - 32 biases will be added by each of 32 filters to their weighted sum
  - Weights = (3 × 3 × 1) × 32 = 288
  - Total parameters = 288 + 32 = 320

CNN example (break down of layers)

- **maxpooling2d_1 (MaxPooling2D)**
  - **Input shape**: 26×26×32
  - **Pool size**: 2×2 pool with stride 2.
  - **Output shape**: 13×13×32 (13x13 is the dimension of feature map, 32 is no of maps or depth)
  - **Parameters**: **0** (MaxPooling doesn’t have learnable parameters).

CNN example (break down of layers)

- **conv2d_2 (Conv2D)**
  - **Input shape**: 13×13×32
  - **Filter size**: 3x3
  - **Number of filters**: 64
  - **Input channels**: 32 (from the previous layer)
  - **Output shape**: 11×11×64
  - **Parameters calculation**:
  - 3 x 3 is filter size that will be applied by current layer
  - 64 different filters of size 3 x 3 will be convolved with input feature maps
  - 32 input feature maps were produced by previous layer
  - 64 biases will be added by each of 64 filters to their weighted sum
  - Weights = (3 × 3 × **32**) × 64 = 18432
  - Total parameters = 18432 + 64 = 18496

CNN example (break down of layers)

- **maxpooling2d_2 (MaxPooling2D)**
  - **Input shape**: 11×11×64
  - **Pool size**: 2×2 pool with stride 2.
  - **Output shape**: 5×5×64
  - **Parameters**: **0** (MaxPooling doesn’t have learnable parameters).

CNN example (break down of layers)

- **conv2d_3 (Conv2D)**
  - **Input shape**: 5×5×64
  - **Filter size**: 3x3
  - **Number of filters**: 64
  - **Input channels**: 64 (from the previous layer)
  - **Output shape**: 3×3×64
  - **Parameters calculation**:
  - 3 x 3 is filter size that will be applied by current layer
  - 64 different filters of size 3 x 3 will be convolved with input feature maps
  - 64 input feature maps were produced by previous layer
  - 64 biases will be added by each of 64 filters to their weighted sum
  - Weights = (3 × 3 × 64) × 64 = 36864
  - Total parameters = 36864 + 64 = 36928

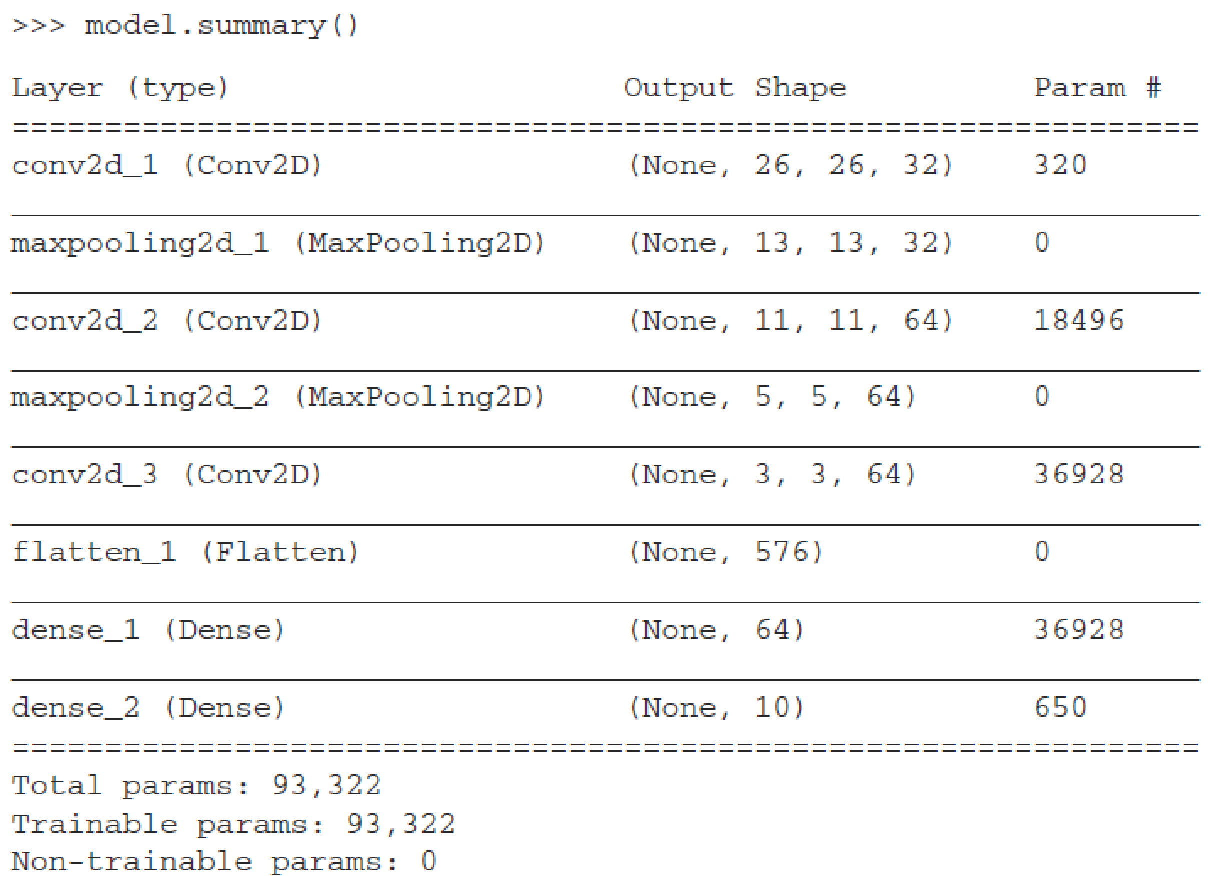
CNN example (break down of layers)

- The next step is to feed the last output tensor (of shape (3, 3, 64)) into a **densely connected classifier network**.
- These **classifiers process** vectors, which are **1D**, whereas the current **output is a 3D tensor**.
- First we have to **flatten the 3D outputs to 1D**, and then add a few Dense layers on top.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

CNN example (break down of layers)

```
from keras import layers
from keras import models
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
 input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```



```
>>> model.summary()
Layer (type)                     Output Shape          Param #
================================================================
conv2d_1 (Conv2D)                (None, 26, 26, 32)    320
________________________________________________________________
maxpooling2d_1 (MaxPooling2D)    (None, 13, 13, 32)    0
________________________________________________________________
conv2d_2 (Conv2D)                (None, 11, 11, 64)    18496
________________________________________________________________
maxpooling2d_2 (MaxPooling2D)    (None, 5, 5, 64)      0
________________________________________________________________
conv2d_3 (Conv2D)                (None, 3, 3, 64)      36928
________________________________________________________________
flatten_1 (Flatten)              (None, 576)           0
________________________________________________________________
dense_1 (Dense)                  (None, 64)            36928
________________________________________________________________
dense_2 (Dense)                  (None, 10)            650
================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

Implementation of CNN using Fashion MNIST Dataset

Implement CNN model in realtime using Fashion MNIST dataset

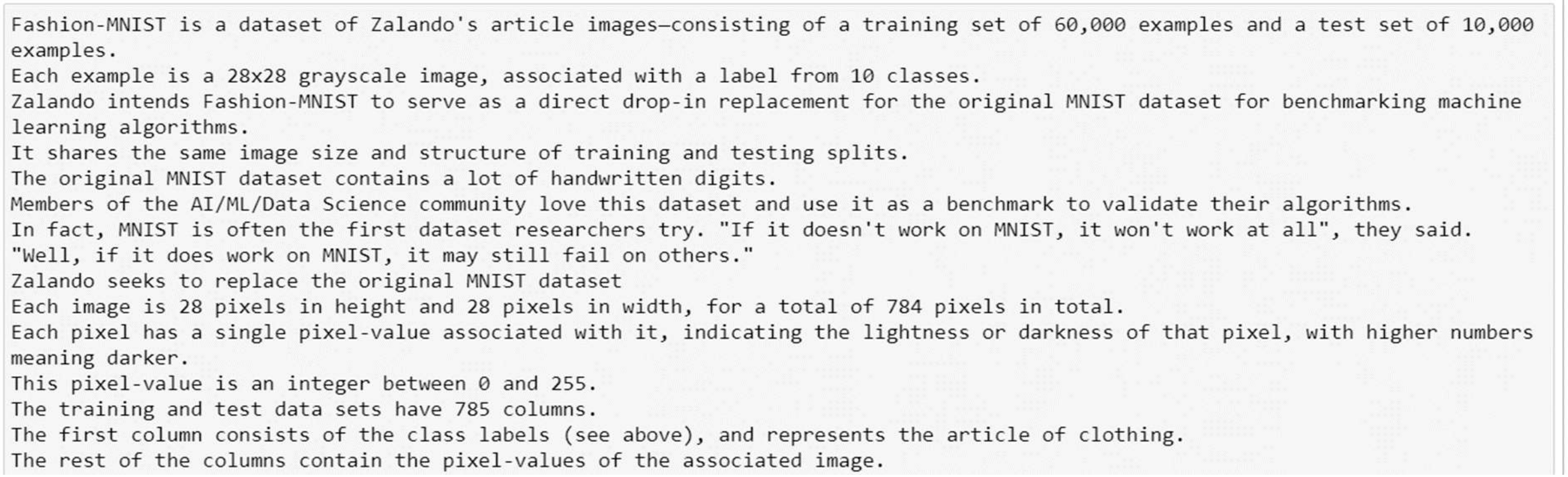
Implementation of CNN

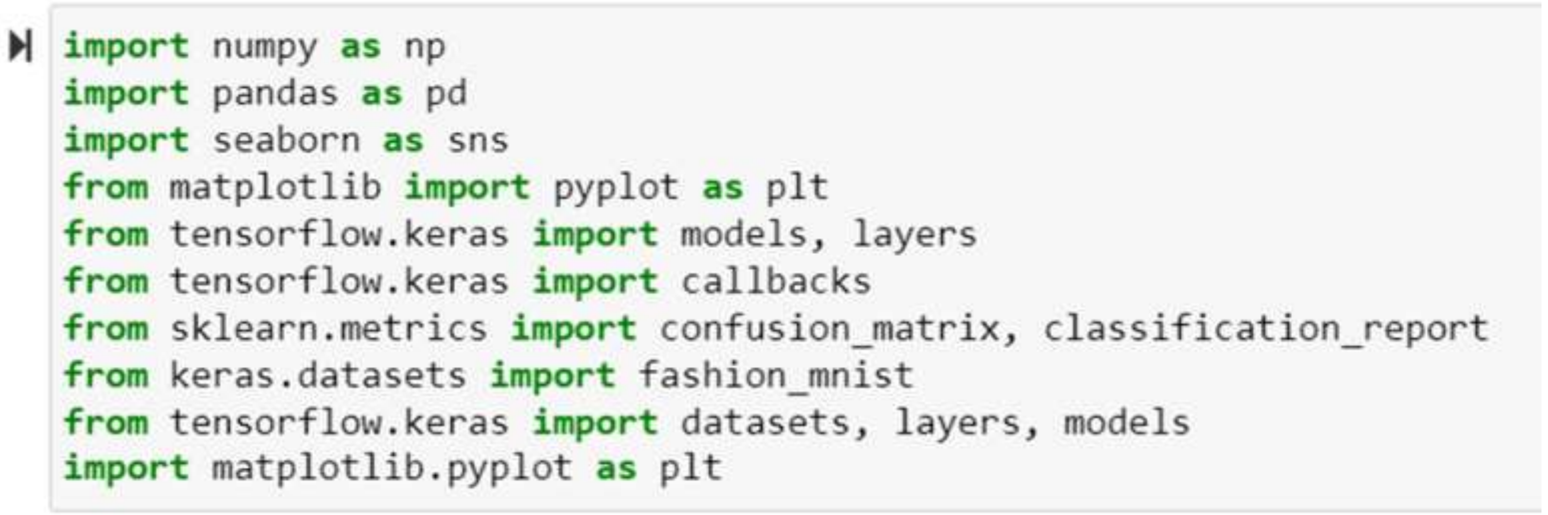
- Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.
- Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total.
- Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255.
- The training and test data sets have 785 columns.
- The first column consists of the class labels (see above) and represents the article of clothing.
- The rest of the columns contain the pixel-values of the associated image.

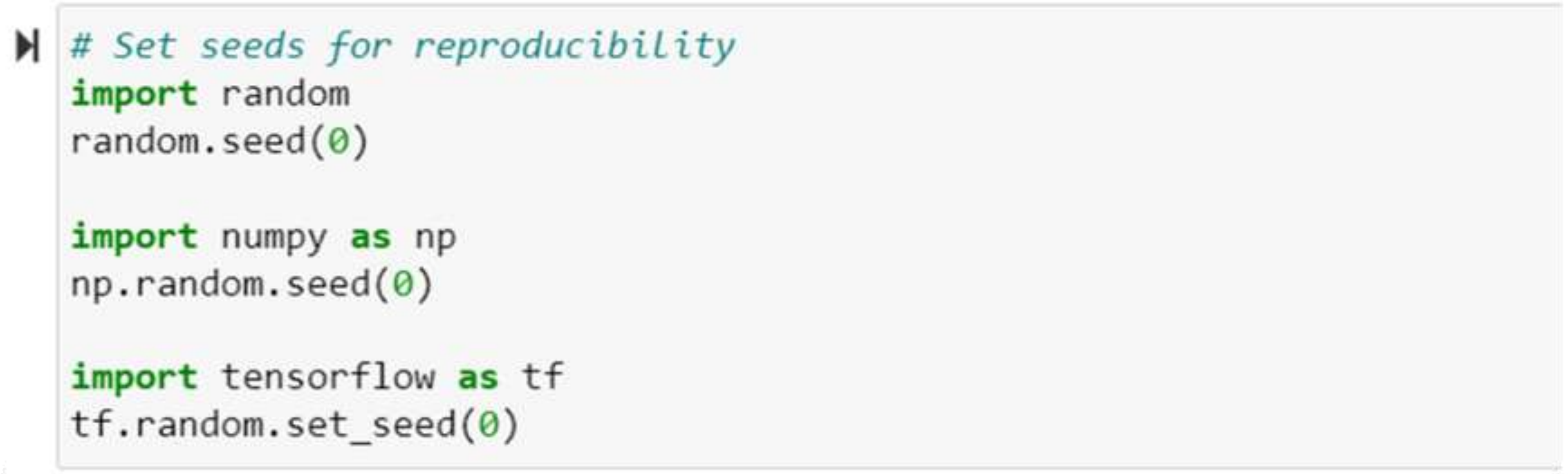
Each training and test example is assigned to one of the following labels:

- 0 T-shirt/top
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt
- 7 Sneaker
- 8 Bag
- 9 Ankle boot

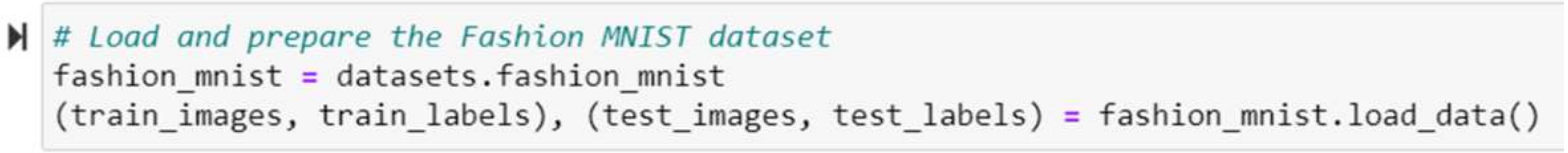
- Each row is a separate image
- Column 1 is the class label.
- Remaining columns are pixel numbers (784 total).
- Each value is the darkness of the pixel (1 to 255)


Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples.
Each example is a 28x28 grayscale image, associated with a label from 10 classes.
Zalando intends Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms.
It shares the same image size and structure of training and testing splits.
The original MNIST dataset contains a lot of handwritten digits.
Members of the AI/ML/Data Science community love this dataset and use it as a benchmark to validate their algorithms.
In fact, MNIST is often the first dataset researchers try. "If it doesn't work on MNIST, it won't work at all", they said.
"Well, if it does work on MNIST, it may still fail on others."
Zalando seeks to replace the original MNIST dataset
Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total.
Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker.
This pixel-value is an integer between 0 and 255.
The training and test data sets have 785 columns.
The first column consists of the class labels (see above), and represents the article of clothing.
The rest of the columns contain the pixel-values of the associated image.


```python
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from tensorflow.keras import models, layers
from tensorflow.keras import callbacks
from sklearn.metrics import confusion_matrix, classification_report
from keras.datasets import fashion_mnist
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

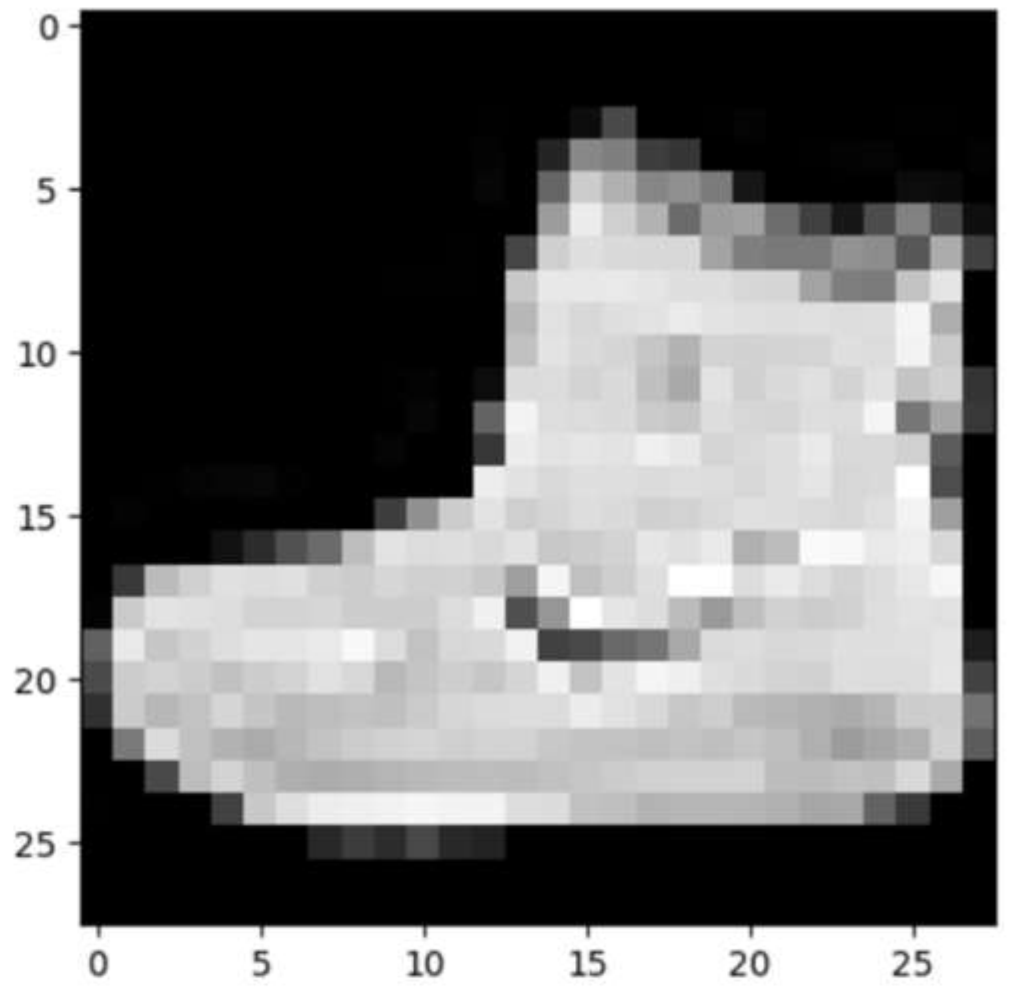


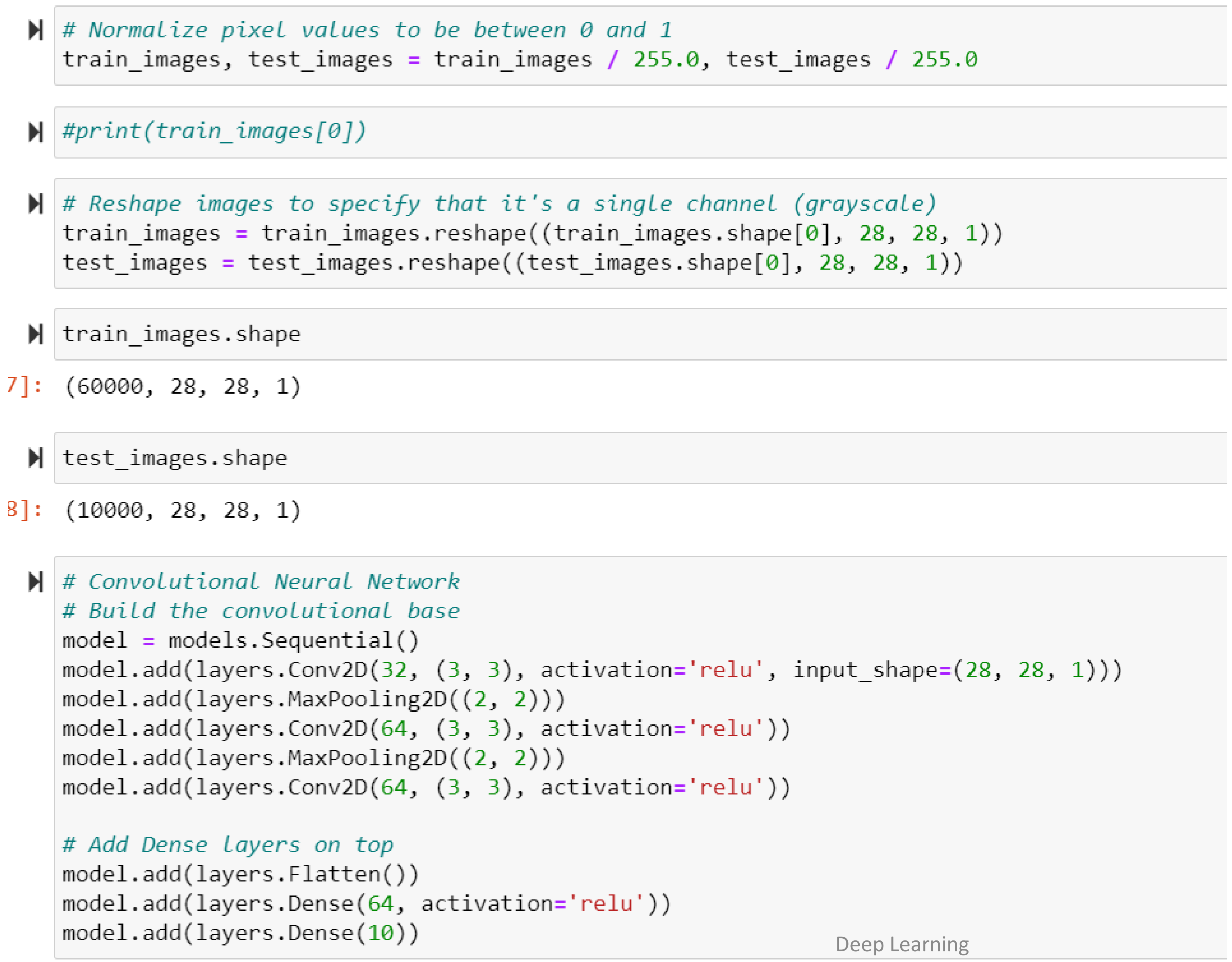
```python
# Set seeds for reproducibility
import random
random.seed(0)

import numpy as np
np.random.seed(0)

import tensorflow as tf
tf.random.set_seed(0)
```



```python
# Load and prepare the Fashion MNIST dataset
fashion_mnist = datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

```
# Display an image from the dataset
plt.imshow(train_images[0], cmap='gray')
plt.show()
print(train_labels[0])
# class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```





```
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

```
#print(train_images[0])
```

```
# Reshape images to specify that it's a single channel (grayscale)
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))
```

```
train_images.shape
```

```
7]: (60000, 28, 28, 1)
```

```
test_images.shape
```

```
8]: (10000, 28, 28, 1)
```

```
# Convolutional Neural Network
# Build the convolutional base
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Add Dense layers on top
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

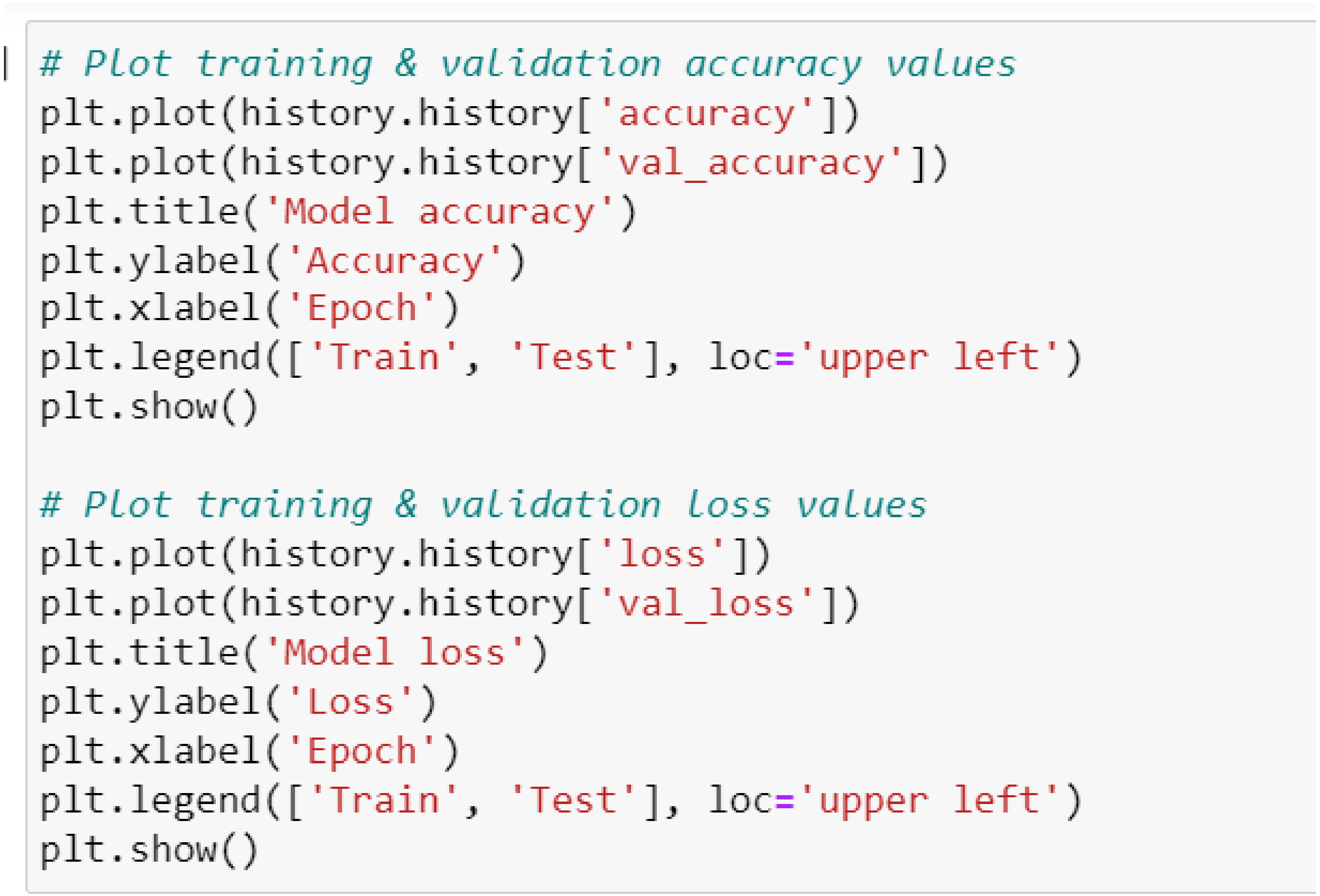
```
# Compile and train the model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
history = model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))
```

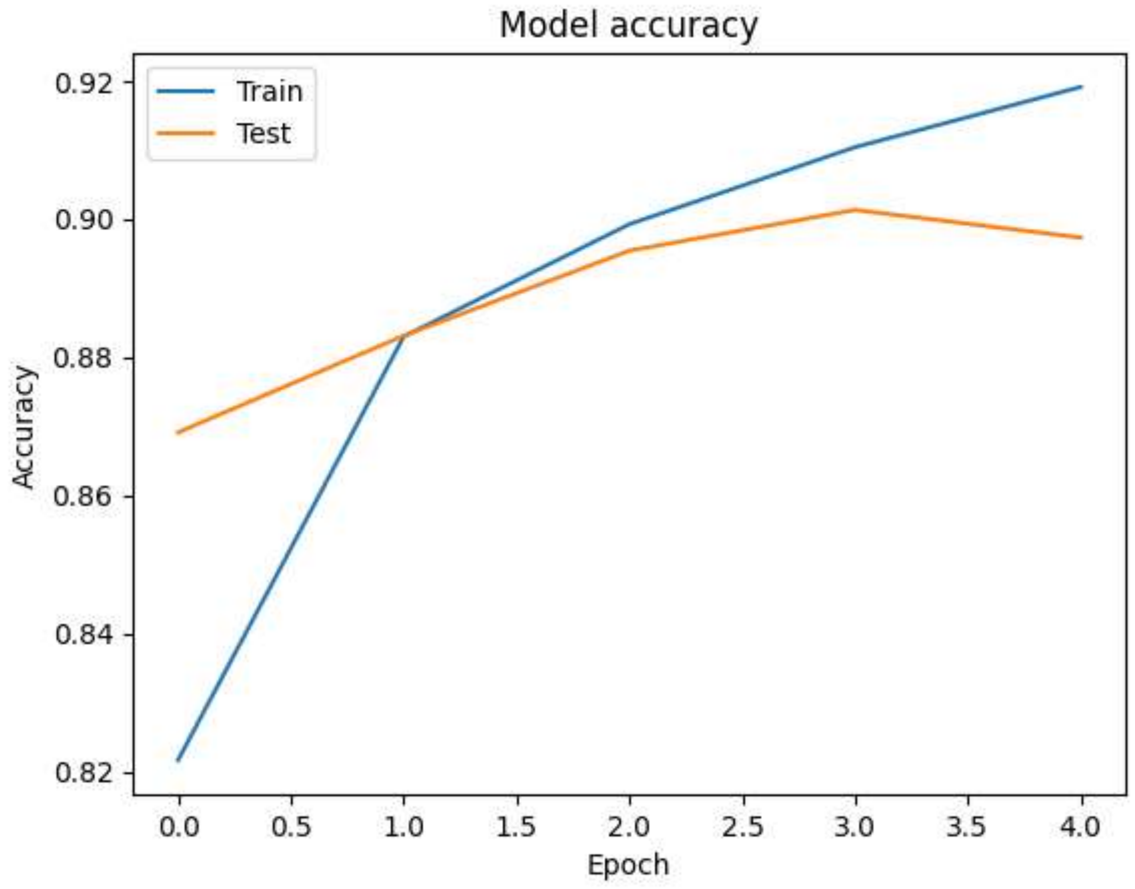
```
Epoch 1/5
1875/1875 [==============================] - 29s 15ms/step - loss: 0.4892 - accuracy: 0.8217 - val_loss: 0.3646 - val_accura
cy: 0.8691
Epoch 2/5
1875/1875 [==============================] - 28s 15ms/step - loss: 0.3171 - accuracy: 0.8830 - val_loss: 0.3171 - val_accura
cy: 0.8831
Epoch 3/5
1875/1875 [==============================] - 30s 16ms/step - loss: 0.2709 - accuracy: 0.8992 - val_loss: 0.2885 - val_accura
cy: 0.8954
Epoch 4/5
1875/1875 [==============================] - 29s 15ms/step - loss: 0.2410 - accuracy: 0.9104 - val_loss: 0.2655 - val_accura
cy: 0.9013
Epoch 5/5
1875/1875 [==============================] - 25s 13ms/step - loss: 0.2177 - accuracy: 0.9191 - val_loss: 0.2837 - val_accura
cy: 0.8973
```

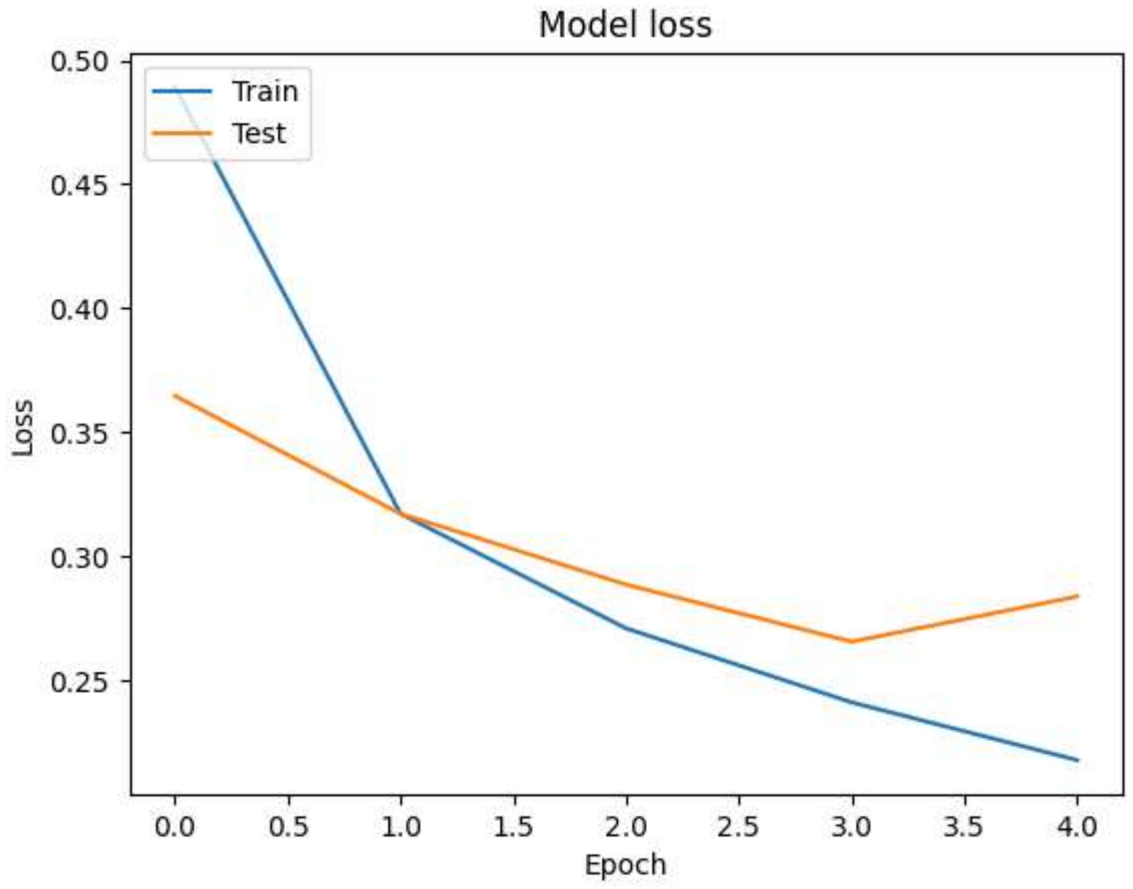
```
# Evaluate the model
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
print('\nTest accuracy:', test_acc)
```

```
313/313 - 2s - loss: 0.2837 - accuracy: 0.8973 - 2s/epoch - 7ms/step

Test accuracy: 0.8973000049591064
```



```python
# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```


Model accuracy
Accuracy
Epoch
Train
Test
0.82
0.84
0.86
0.88
0.90
0.92
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Model loss
Loss
Epoch
Train
Test
0.25
0.30
0.35
0.40
0.45
0.50
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

The convolutional layer vs Densely connected layer

- Dense layers learn **global patterns** in their input feature space (for example, an MNIST digit, patterns involving all pixels),
- whereas convolution layer learn local patterns (see figure 5.1): in case of images, patterns found in small 2D windows of inputs. E.g., these windows were all 3 × 3.
- key characteristic of convent:
  - The patterns they learn are translation invariant.
  - They can learn spatial hierarchies of patterns (see figure 5.2).

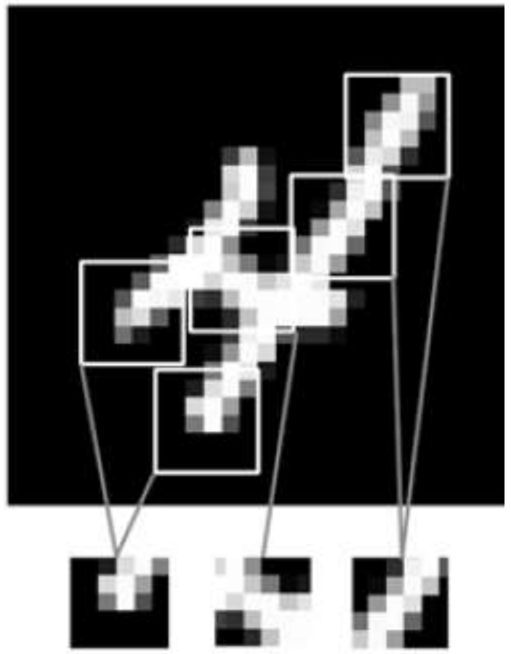


Figure 5.1 Images can be broken into local patterns such as edges, textures, and so on.

The convolutional layer vs Densely connected layer

- The **first convolution layer** will learn small **local patterns** such as edges.
- The **second convolution layer** will learn **larger patterns** made of the **features of the first layers**, and so on.
- This **allows convnets** to efficiently **learn** increasingly **complex and abstract** visual concepts (because the visual world is fundamentally spatially hierarchical).

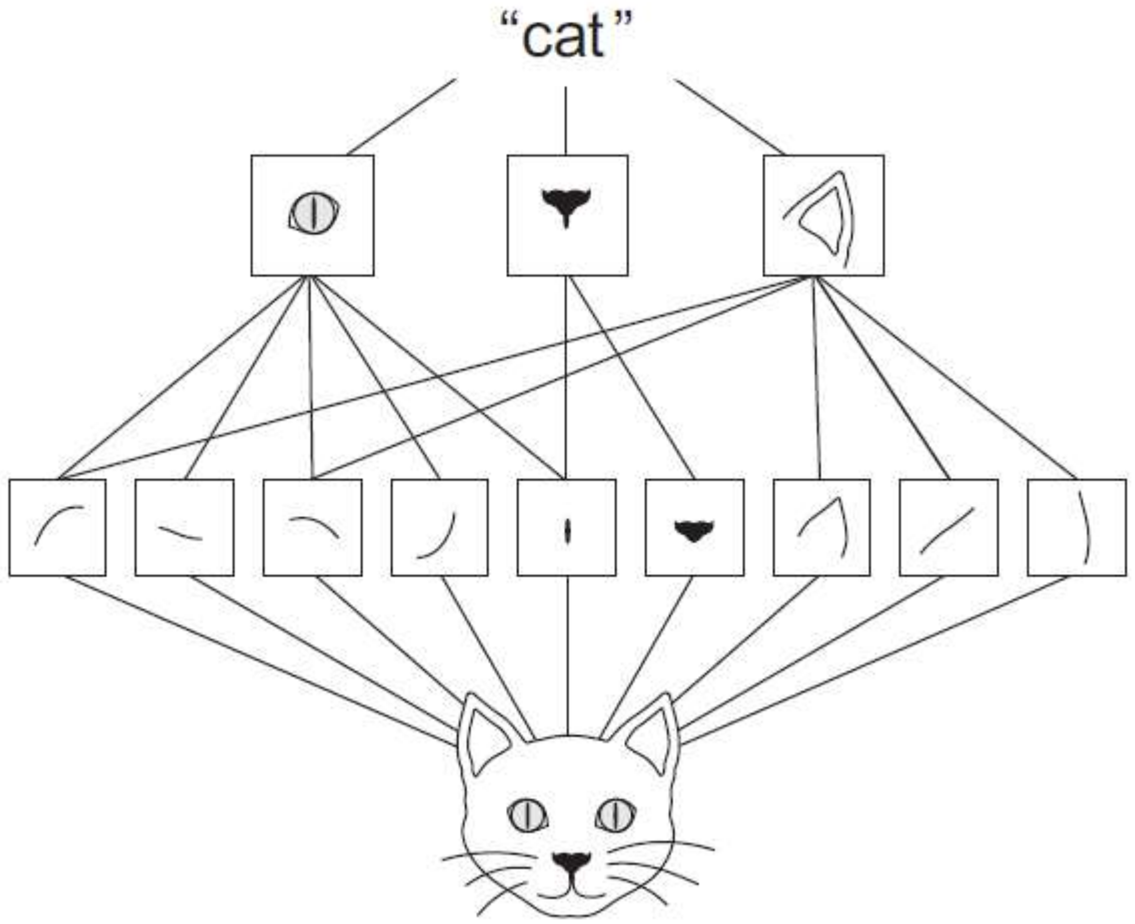


Figure 5.2 The visual world forms a spatial hierarchy of visual modules: hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as "cat."

ImageNet Competition

- In this competition the top-5 error rate for image classification fell from over 26% to barely over 3% in just five years.
- The top-five error rate is the number of test images for which the system's top 5 predictions did not include the correct answer.
- The images are large (256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds).
- We first look at the classical **LeNet-5** architecture (1998), then three of the winners the challenge: **AlexNet** (2012), **GoogLeNet** (2014), and **ResNet** (2015).

ImageNet Competition (LeNet-5)

- The **LeNet-5 architecture** is the most widely known CNN architecture.
- Created by Yann LeCun (1998), **widely used** for handwritten digit recognition (**MNIST**).
- MNIST images are 28 × 28, but **are zero-padded** to 32 × 32 pixels and **normalized** before being fed to the network.
- The **rest of the network does not use any padding**, which is why the si**ze keeps shrinking** as the image progresses through the network.

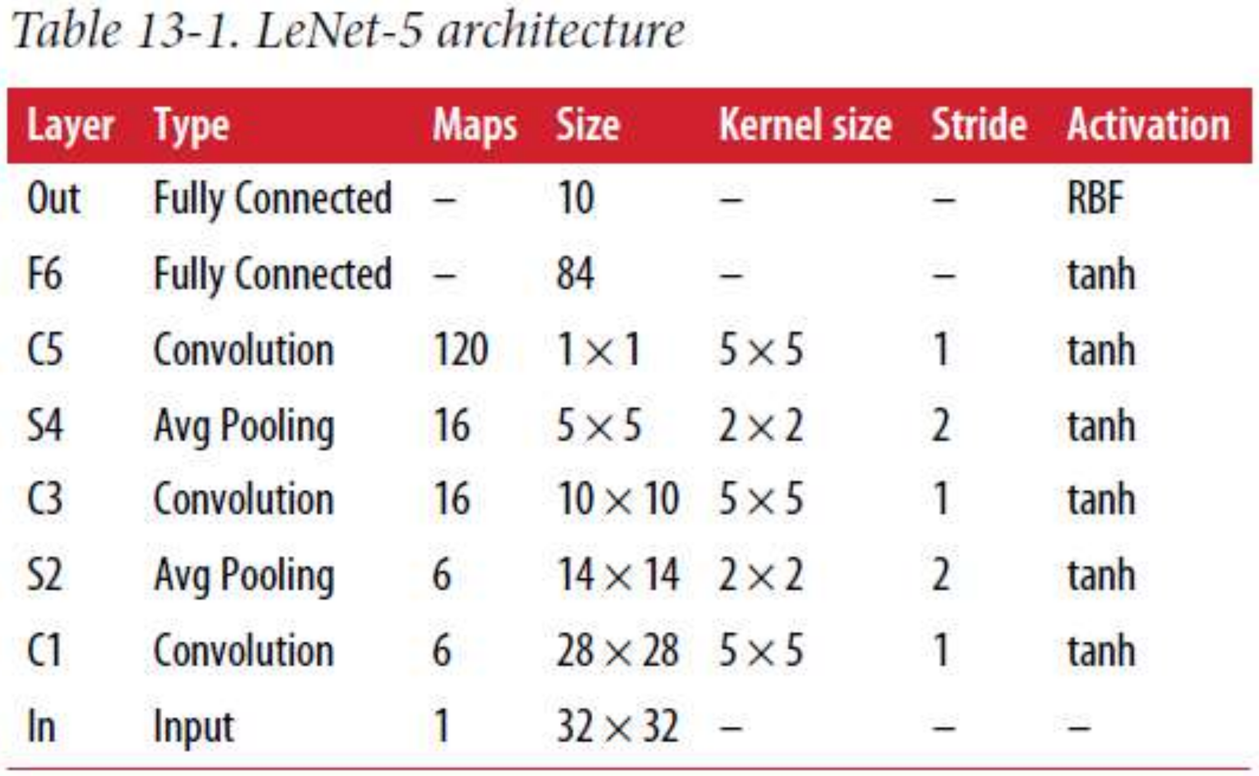


Table 13-1. LeNet-5 architecture

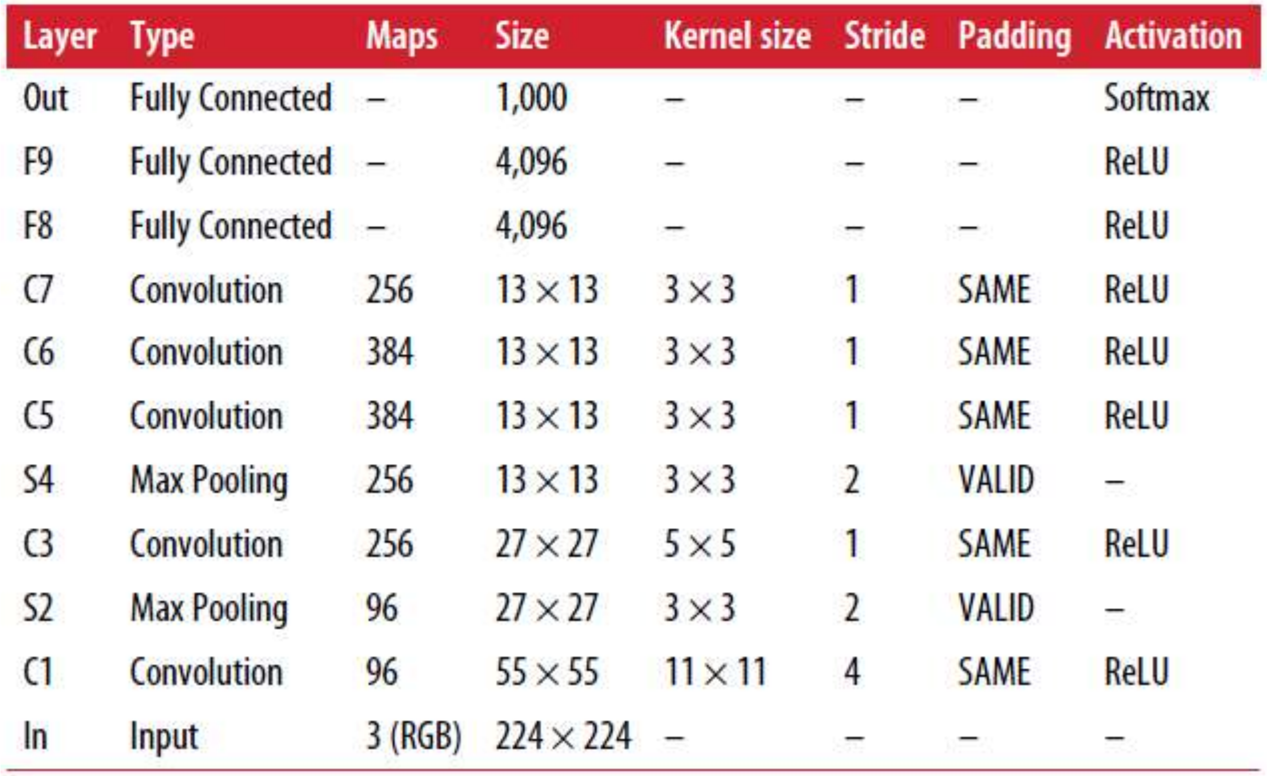
Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg Pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg Pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–

ImageNet Competition (AlexNet)

- Developed by **Alex** Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton.
- It is quite **similar to LeNet-5**, only much larger and **deeper**, and it was **the first to stack convolutional layers** directly on top of each other, instead of stacking a pooling layer on top of each convolutional layer.
- To **reduce overfitting**, the authors used **two regularization** techniques:
  - First they applied dropout (with a **50% dropout rate**) during training to the outputs of layers F8 and F9.
  - Secondly, **data augmentation** by randomly shifting training images by various offsets, flipping them horizontally, and changing the lighting conditions.

ImageNet Competition (AlexNet)

Table 13-2. AlexNet architecture



Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

Calculation of Output Size for first CNN layer

$$\text{Output size} = \left\lfloor \frac{\text{Input size} - \text{Kernel size} + 2 \times \text{Padding}}{\text{Stride}} + 1 \right\rfloor$$

1. **Input Size**: 224 (for both width and height)
2. **Kernel Size**: 11 (11x11)
3. **Padding**: 4 (SAME padding)
4. **Stride**: 4

$$\text{Padding} = \frac{F-1}{2}$$

F= filter size, in this case F=11

Now, substituting into the formula:

$$\text{Output size} = \left\lfloor \frac{224 - 11 + 2 \times 4}{4} + 1 \right\rfloor$$

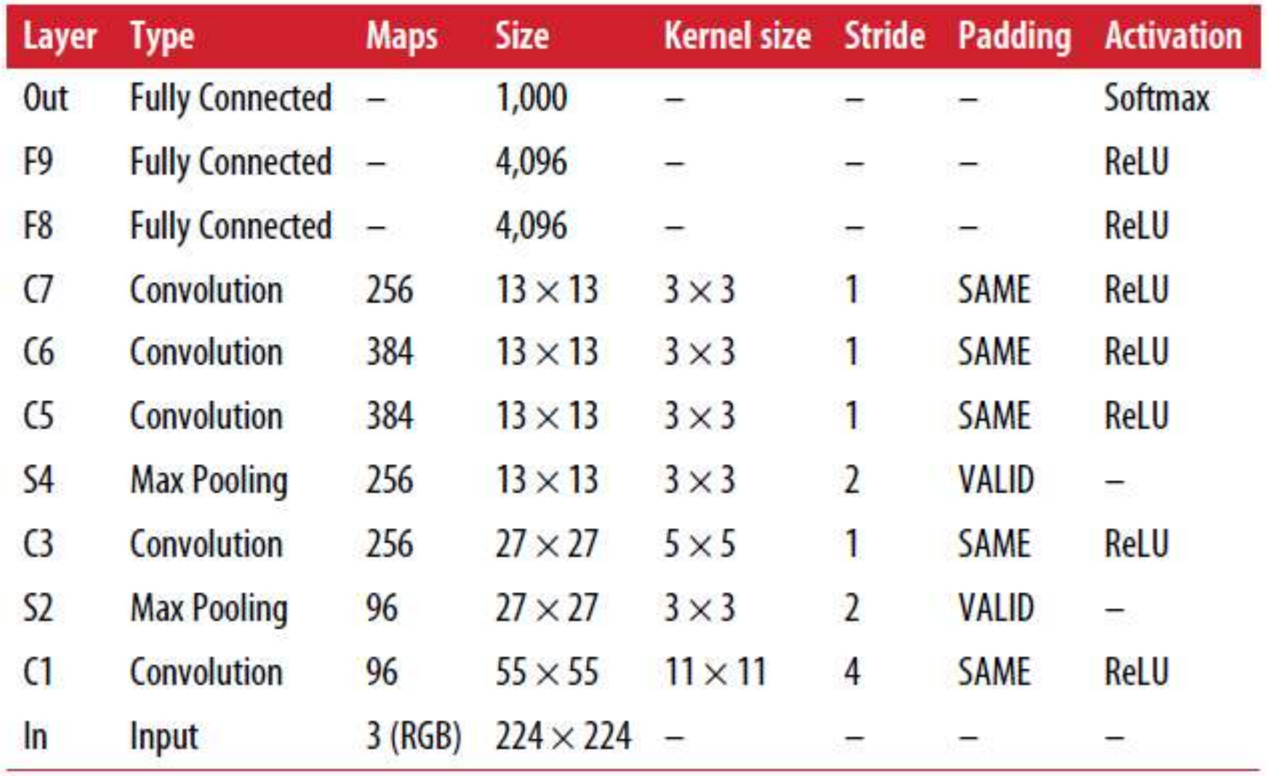
224−11+8=221

221/4=55.25

the output size from the first convolutional layer indeed would be 55x55, as specified.

ImageNet Competition (AlexNet)

Table 13-2. AlexNet architecture



Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

Calculation of Output Size for pooling layer

$$\text{Output Size} = \left(\frac{W-F}{S}\right) + 1$$

Where.

- W is the input size (height or width of the image or feature map before pooling).
- F is the size of the pooling filter (e.g., 2x2, 3x3).
- S is the stride, which determines the step size when sliding the pooling window across the input.
- The output size will be the height or width of output feature map after pooling.
- For a pooling operation with the following parameters
- Input size W=32 (e.g., 32x32 image),
- Filter size F=2×2 = 2,
- Stride S=2
- The output size will be:
  Output Size=((32−2)/2)+1=16 (h x w) feature map.

ImageNet Competition (GoogleNet)

- This was made possible by sub-networks called **inception modules**, which allow **GoogLeNet** to use parameters much more efficiently than previous architectures:
- GoogLeNet actually has **10 times fewer parameters than AlexNet** (roughly 6 million instead of 60 million).

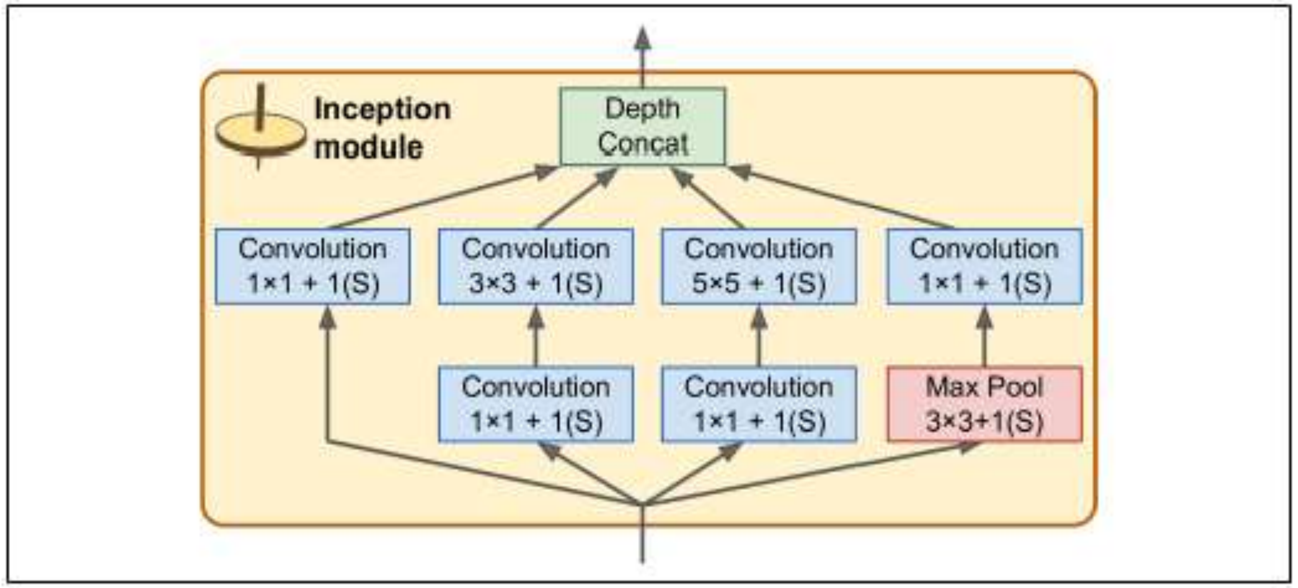


Figure 13-10. Inception module

ImageNet Competition (GoogleNet)

- The notation “3 × 3 + 2(S)” means that the layer uses a 3 × 3 kernel, stride 2, and **SAME padding**.
- The **input signal is first copied** and **fed to four different layers**.
- All convolutional layers use the **ReLU activation** function.
- Note that the **second set of convolutional** layers **uses different kernel si**zes (1 × 1, 3 × 3, and 5 × 5), allowing them to **capture patterns at different scales**.
- Also note that every single layer uses a stride of 1 and SAME padding (even the max pooling layer), so their outputs have same height and width as their inputs.
- This makes it possible to **concatenate all the outputs** along the depth dimension in the final depth concat layer (i.e., **stack the feature maps from all four top convolutional layers**).

ImageNet Competition (GoogleNet)

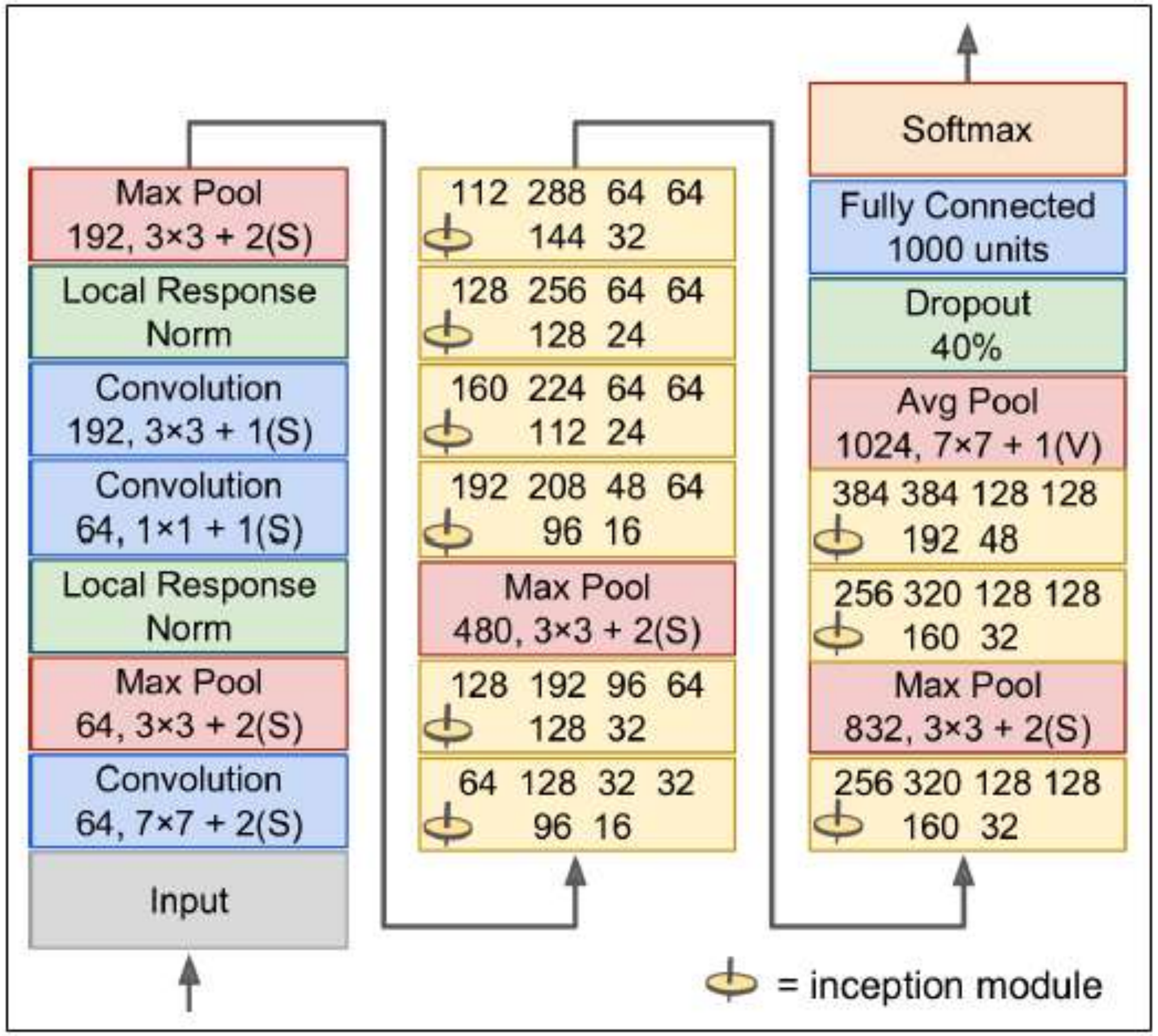


Figure 13-11. GoogLeNet architecture

The **Local Response Normalization (LRN)** layer in architectures like GoogleNet serves to enhance the generalization ability of the model by normalizing the responses of neurons across feature maps.

The six numbers on inception module represents the **number of filers** applied by each convolutional layer on its input.

ImageNet Competition (ResNet)

- **ResNet** (Residual Network) is a deep learning architecture that was introduced by Microsoft Research in 2015.
- It is designed to address the **vanishing gradient problem** and improve the training of very deep neural networks.
- ResNet won the **ILSVRC (ImageNet Large Scale Visual Recognition Challenge)** in 2015 and became one of the most popular architectures for image classification tasks.
- The main idea behind **ResNet** is the concept of **residual learning**, which involves the use of **skip connections** (or shortcut connections) that allow the input to bypass one or more layers and be added to the output of a layer.

ImageNet Competition (ResNet)

- For example, in a typical residual block, the equation looks like this:
  - x is the input to the block.
  - $F(x,\{Wi\})$ is the learned transformation applied to the input (e.g., convolution, batch normalization, activation).
  - y is the final output, which is the sum of the transformation and the original input.

$$y = \mathcal{F}(x, \{W_i\}) + x$$

ImageNet Competition (ResNet)

- The main idea behind **ResNet** is the concept of **residual learning**, which involves the use of **skip connections** (or shortcut connections) that allow the input to bypass one or more layers and be added to the output of a layer.
- This bypass mechanism helps solve several problems, including:
  - **Vanishing gradients**: By using residual connections, the gradient can "skip" certain layers, making it easier to train deep networks.
  - **Degradation problem**: As the network becomes deeper, traditional networks tend to reach a point where increasing the number of layers actually leads to worse performance.
  - This is called the degradation problem. Residual connections help mitigate this problem by providing a direct path for the gradient flow.

ImageNet Competition (ResNet)

- It used extremely deep CNN composed of 152 layers
- The key to being able to train such a deep network is to use skip connections (also called shortcut connections): the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack. Let's see why this is useful.
- When training a neural network, the goal is to make it model a target function h(x).
- If you add the input x to the output of the network (i.e., you add a skip connection), then the network will be forced to model f(x) = h(x) – x rather than h(x). This is called residual learning.

ImageNet Competition (ResNet)

- Solving vanishing gradient problem

$$y = \mathcal{F}(x) + x$$

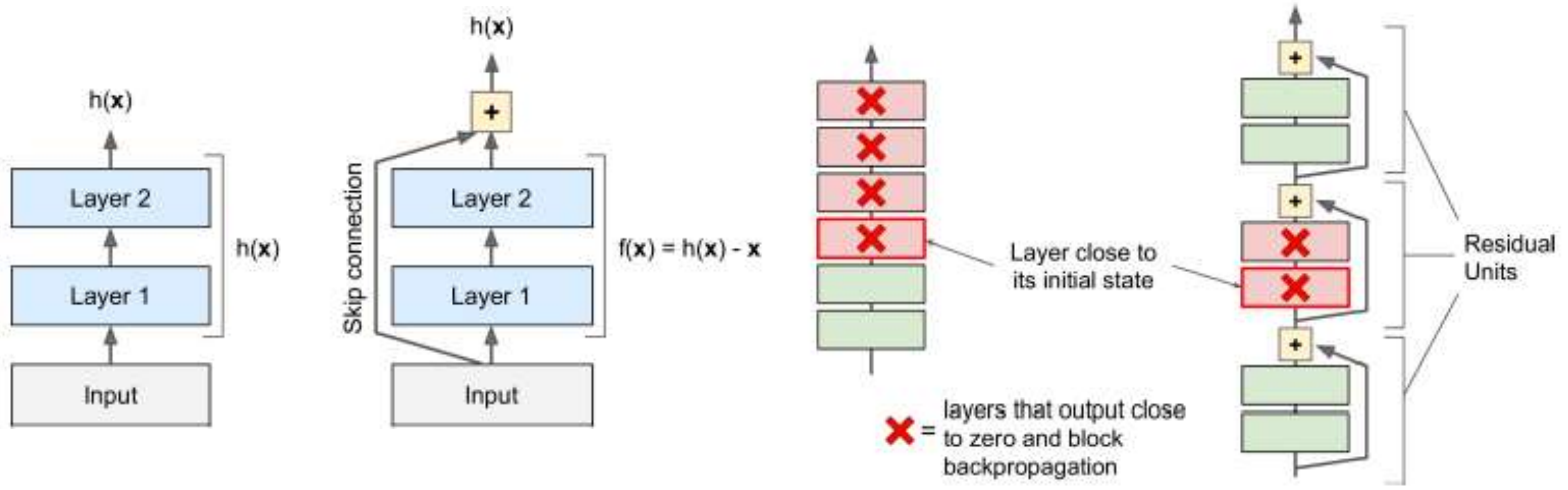
$$\frac{\partial y}{\partial x} = \frac{\partial}{\partial x}(\mathcal{F}(x) + x) = \frac{\partial \mathcal{F}(x)}{\partial x} + 1$$

- The important point here is the **"+1"** term.
- This term means that **even if the derivative of F(x) is very small**, the gradient flowing through the skip connection will be **non-zero** and will contribute to the gradient flow.
- This makes it much easier for the network to learn in deeper layers because it prevents the gradients from vanishing completely.

ImageNet Competition (ResNet)

- Transformation layers (convolution + activation) might discard or distort some details of the input data.
- For example, if a transformation isn't ideal (e.g., it introduces noise or loses fine details), the deeper layers might not have access to the original, potentially important features.
- By adding the original input to the output, skip connections allow the network to retain important information that might have been lost during the layer transformation.
- This ensures that useful features from earlier layers are available for the deeper layers to work with.

ImageNet Competition (ResNet)



ImageNet Competition (ResNet)

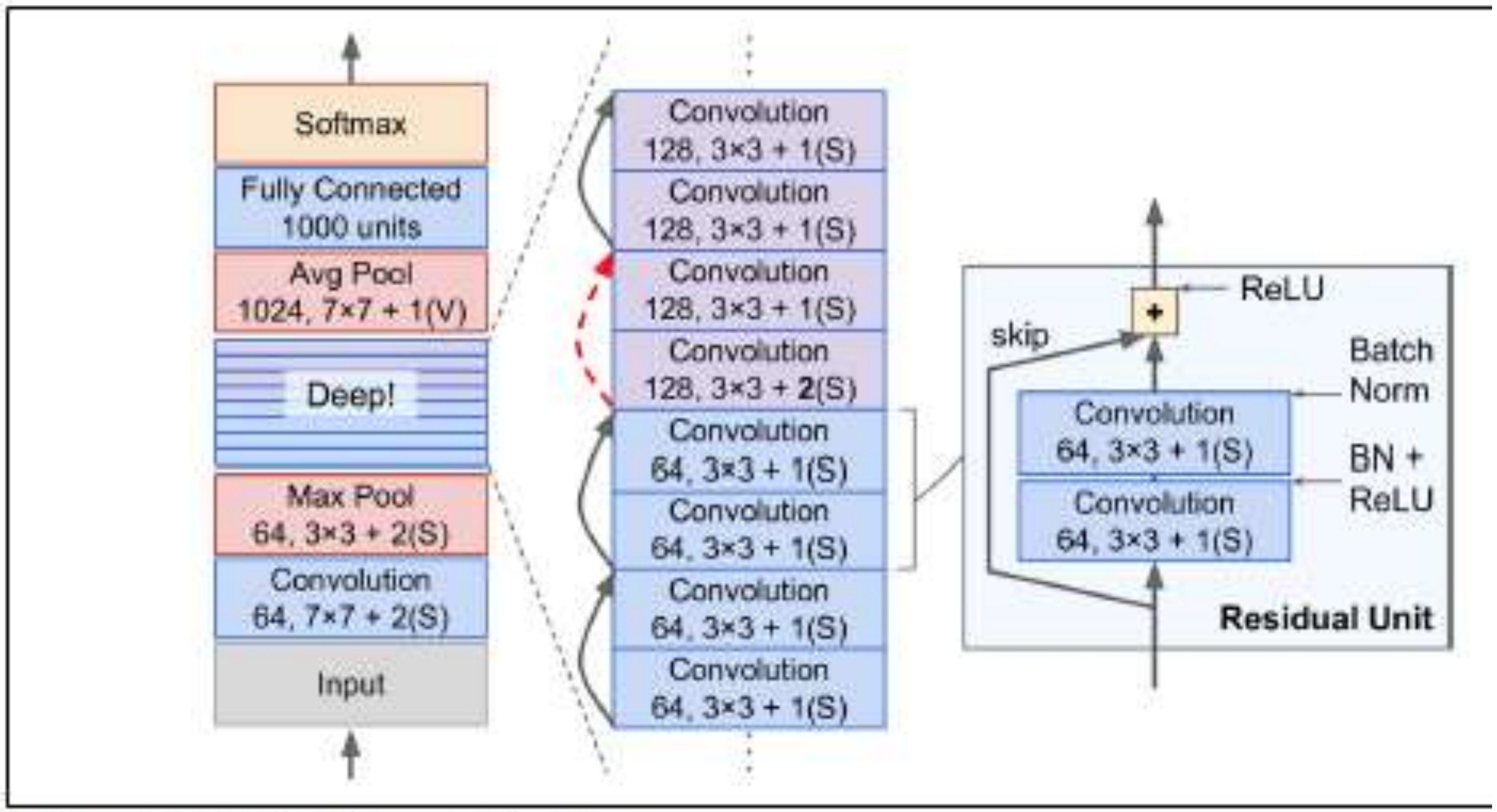

Softmax
Fully Connected 1000 units
Avg Pool 1024, 7×7 + 1(V)
Deep!
Max Pool 64, 3×3 + 2(S)
Convolution 64, 7×7 + 2(S)
Input
Convolution 128, 3×3 + 1(S)
Convolution 128, 3×3 + 1(S)
Convolution 128, 3×3 + 1(S)
Convolution 128, 3×3 + 2(S)
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
ReLU
skip
Batch Norm
Convolution 64, 3×3 + 1(S)
BN + ReLU
Convolution 64, 3×3 + 1(S)
Residual Unit

Figure 13-14. ResNet architecture

Summary

- CNN inspiration from human brain visual cortex was introduced
- Convolutional operations with filters was presented
- Pre-trained models are playing vital role in CNN evolution

Resources

- https://www.kaggle.com/code/pavansanagapati/a-simple-cnn-model-beginner-guide
- https://www.analyticsvidhya.com/blog/2022/03/basics-of-cnn-in-deep-learning/
- https://www.freecodecamp.org/news/convolutional-neural-network-tutorial-for-beginners/
- https://medium.com/@prathammodi001/convolutional-neural-networks-for-dummies-a-step-by-step-cnn-tutorial-e68f464d608f
- https://www.tensorflow.org/tutorials/images/cnn
- https://medium.com/@prathammodi001/convolutional-neural-networks-for-dummies-a-step-by-step-cnn-tutorial-e68f464d608f