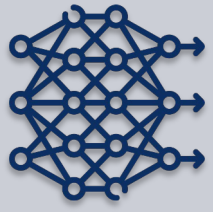
Convolutional Neural Networks

Zeham Management Technologies BootCamp by SDAIA

August 13th, 2024

SDAIA
الهيئة السعودية للبيانات والذكاء الاصطناعي
Saudi Data & AI Authority

Agenda



Introduction to Convolutional Neural Network

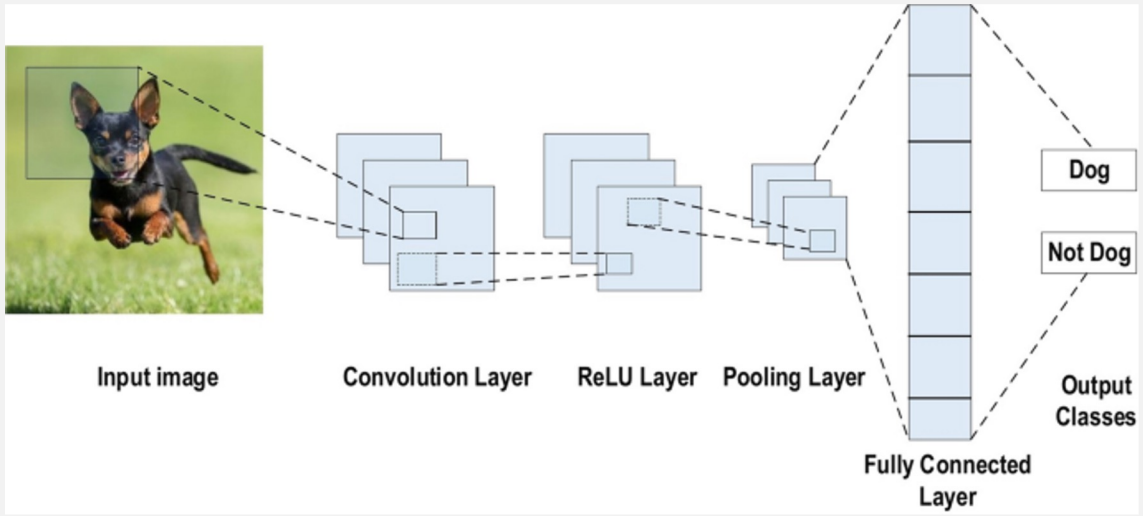
CNN Architecture

Image Dataset & Preprocessing

Introduction to Convolutional Neural Network

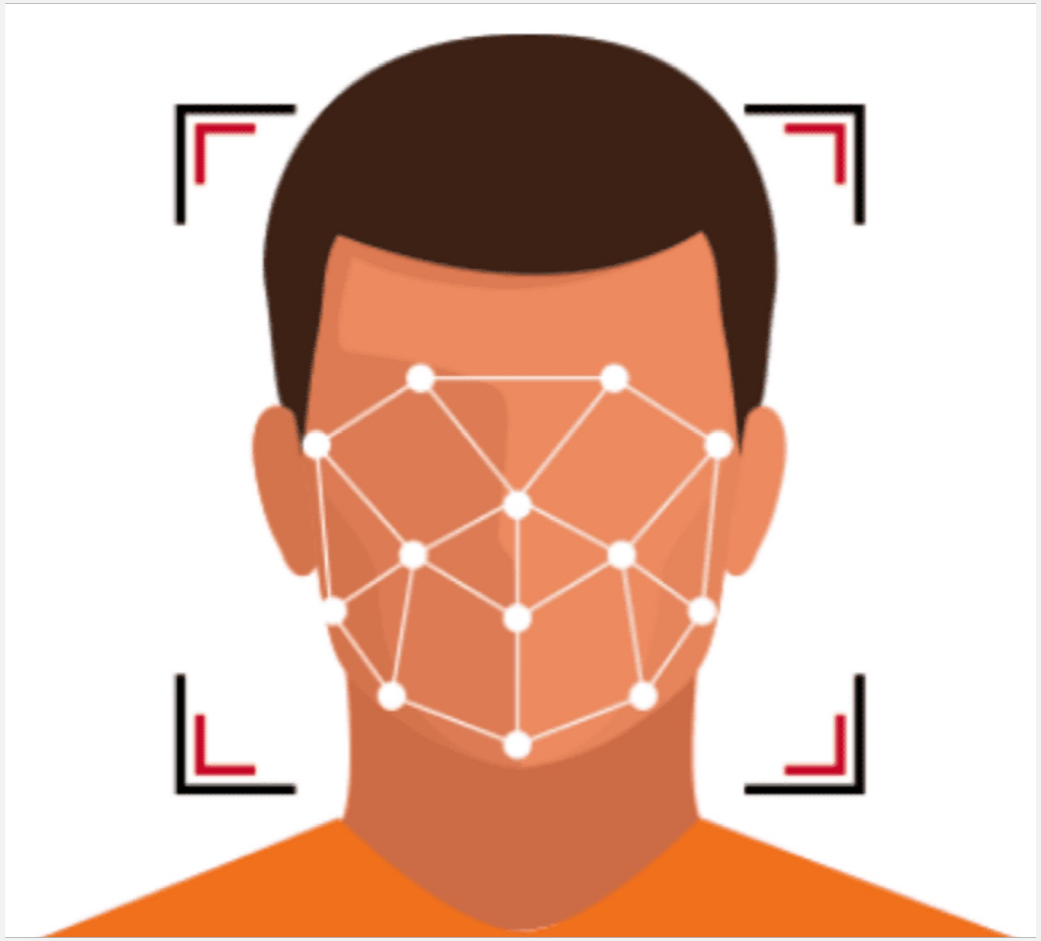
What is a Convolutional Neural Network (CNN)?

- A Convolutional Neural Network (CNN), also known as **ConvNet**, is a specialized type of deep learning algorithm mainly designed for tasks like object recognition, including image classification, detection, and segmentation.
- CNNs are employed in a variety of practical scenarios, such as autonomous vehicles, security camera systems, and others.



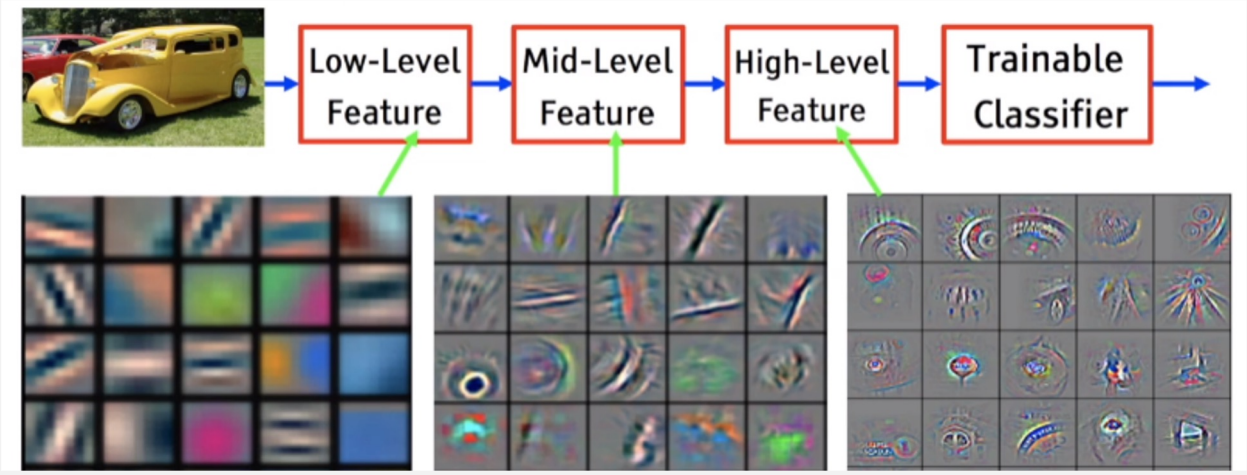
Applications of CNN

- **Face Recognition Applications -** Social Media, Identification, and Surveillance
- **Medical Image Computing -** Predictive Analytics, Healthcare Data Science
- **Image Classification -** Search Engines, Social Media, Recommender Systems



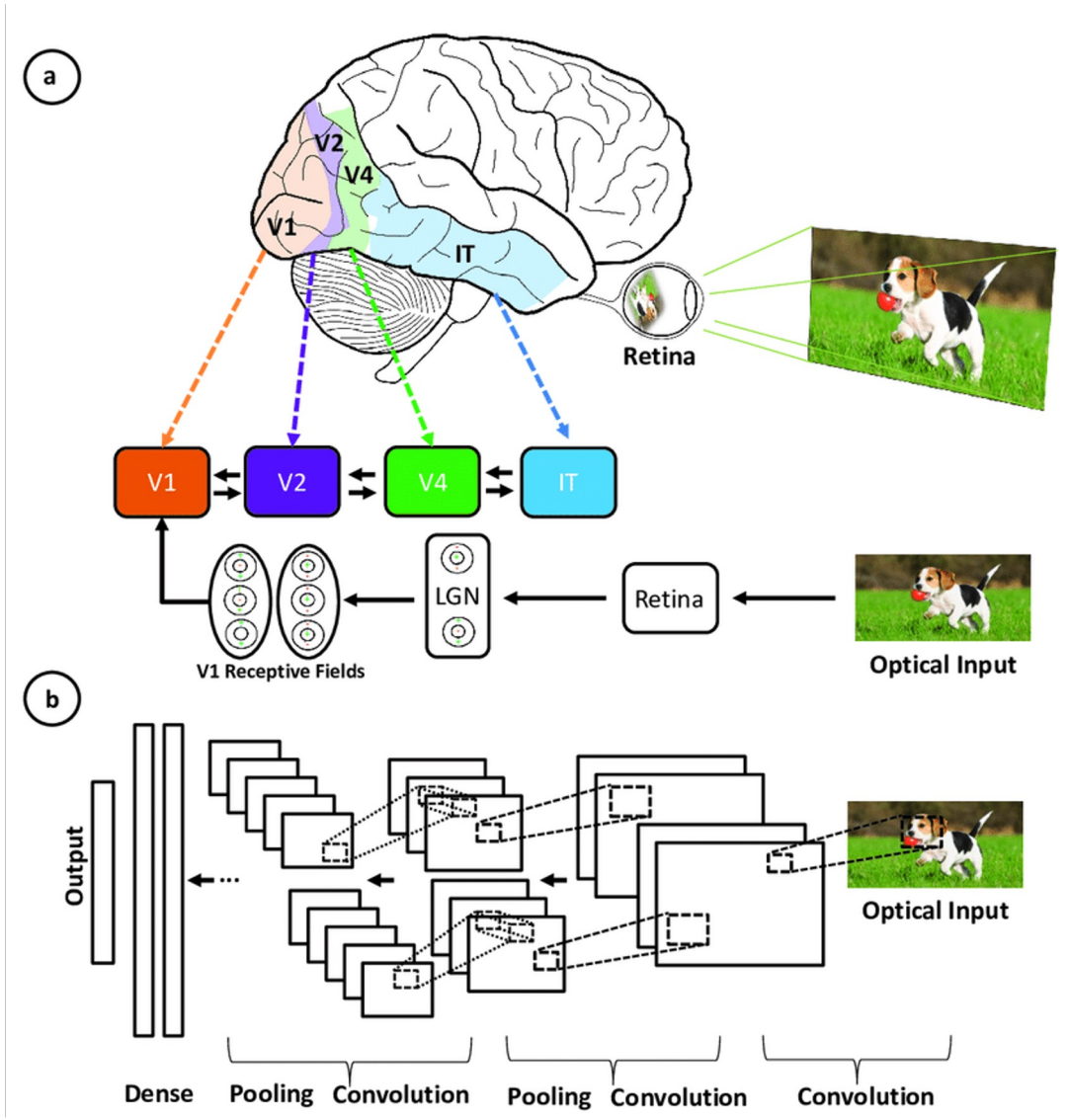
Importance of CNNs

- CNNs autonomously extract features at a large scale, bypassing the need for manual feature engineering and thereby enhancing efficiency.
- The convolutional layers enables the CNNs to identify and extract patterns and features from data irrespective of variations in position, orientation, scale, or translation.
- Beyond image classification tasks, CNNs can be applied to a range of other domains, such as natural language processing, time series analysis, and speech recognition.
- A variety of pre-trained CNN architectures, including VGG-16, ResNet50, Inceptionv3, and EfficientNet, have demonstrated top-tier performance.



Inspiration Behind CNN

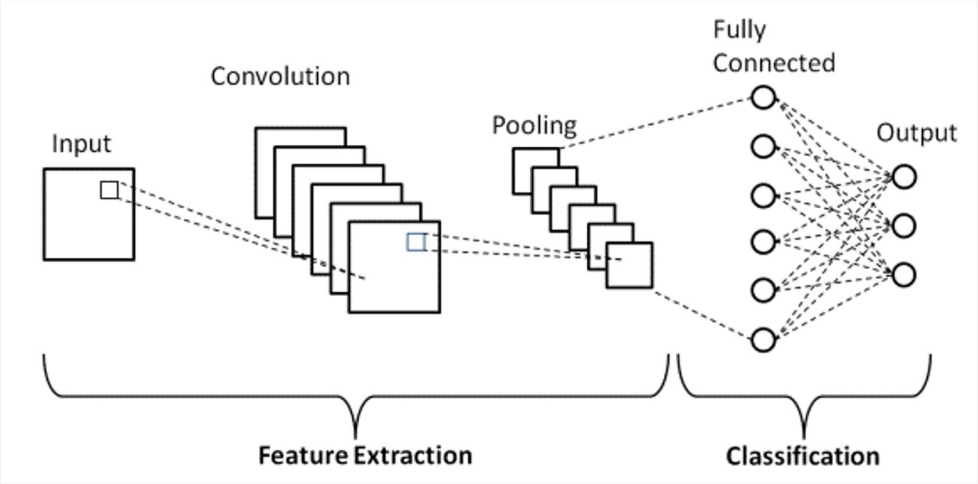
Convolutional neural networks were inspired by the layered architecture of the human visual cortex, and below are some key similarities and differences:



CNN Architecture

CNN Architecture

Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Pooling layer, and fully connected layers.

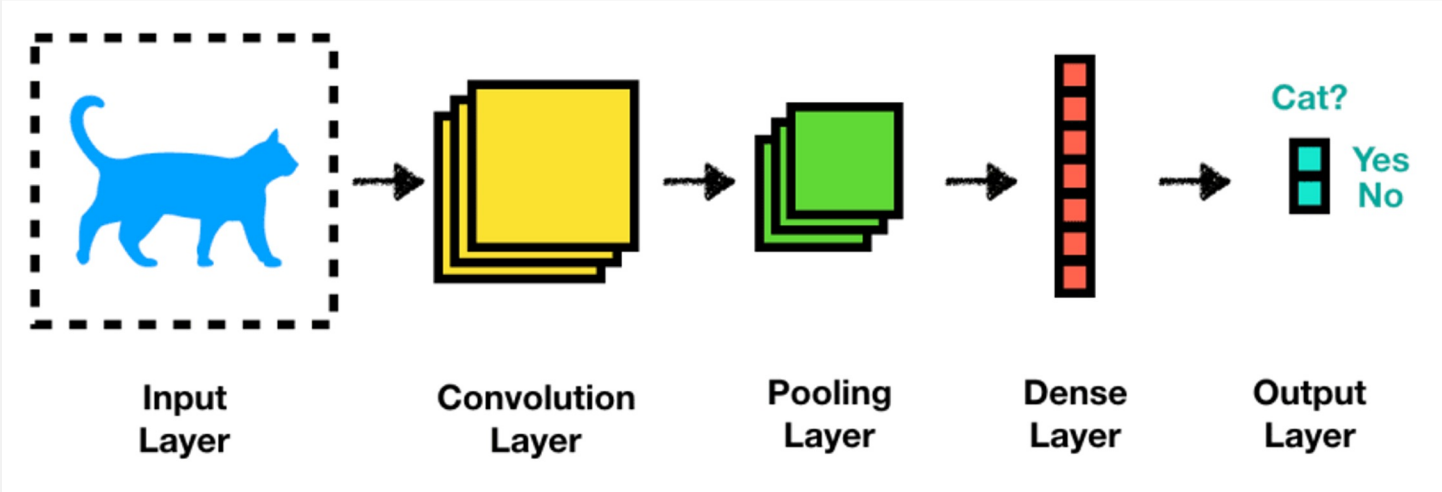


Simple CNN architecture

- The **Convolutional** layer applies filters to the input image to extract features, the **Pooling** layer downsamples the image to reduce computation, and the **fully connected** layer makes the final prediction.

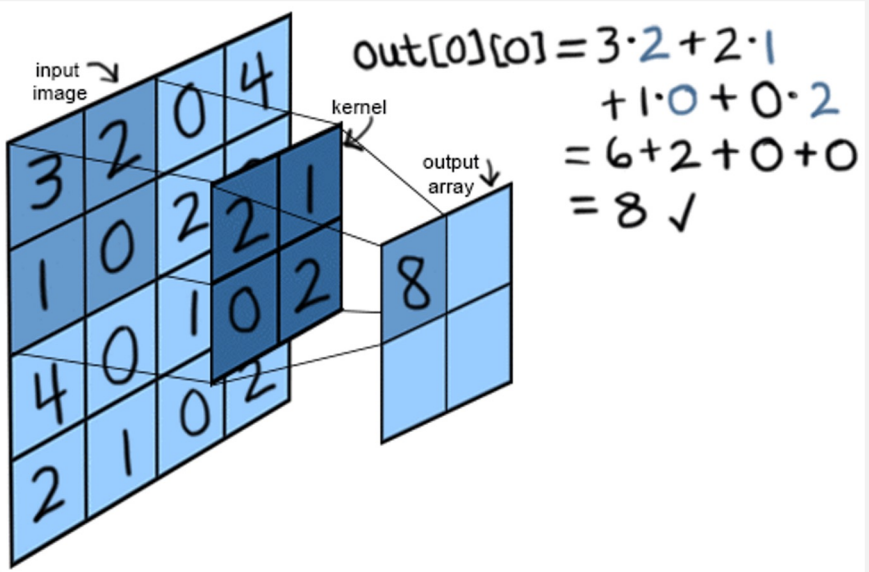
Layers of CNN

- Convolutional Layers
- Pooling layers
- Flattening layer
- Fully connected layers
- Output layer



Convolution Layers

- This is the first building block of a CNN, it applies a sliding window function to a matrix of pixels representing an image.
- The sliding function applied to the matrix is called **kernel** or filter.
- In the convolution layer, several filters of equal size are applied, and each filter is used to recognize a specific pattern from the image, such as the curves, the edges, the whole shape and more.
- The small grids *(called filters or kernels)* move over the image like a mini magnifying glass that looks for specific patterns in the photo, like lines, curves, or shapes.

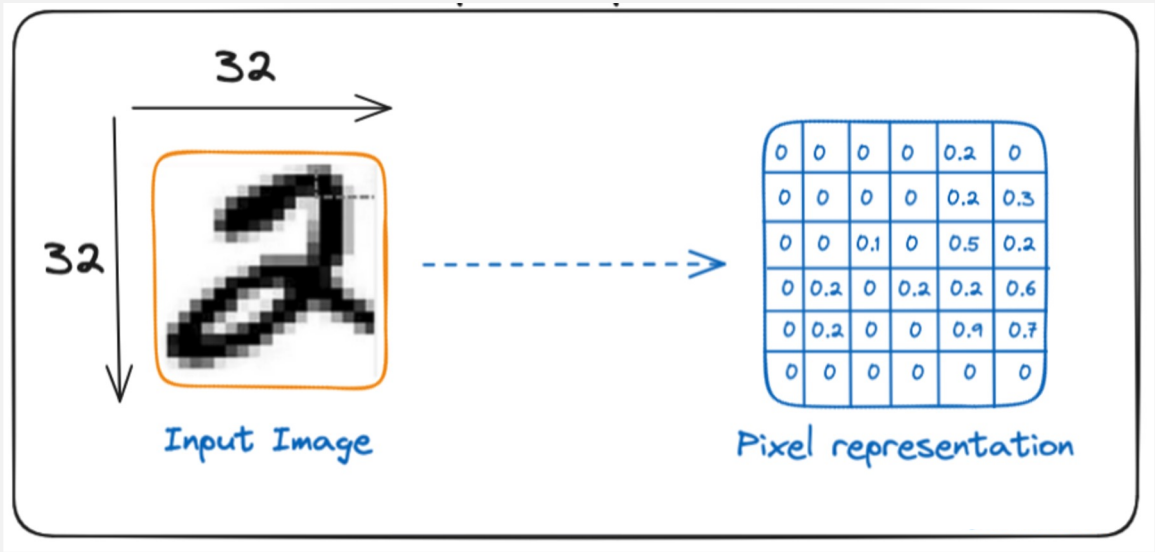


Convolution Layers (cont'd)

- Let's consider the kernel used for the convolution. It is a matrix with a dimension of 3x3.
- The weights of each element of the kernel is represented in the grid. Zero weights are represented in the black grids and ones in the white grid.

Do we have to manually find these weights?

- In real life, the weights of the kernels are determined during the training process of the neural network.



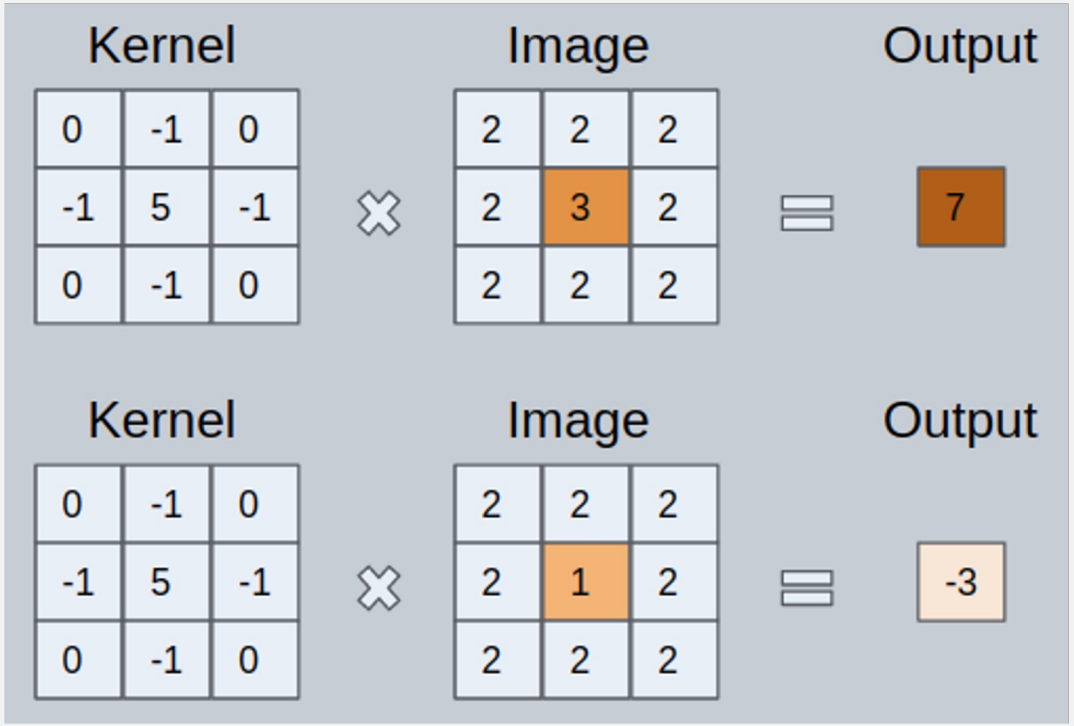
Convolution Layers (cont'd)

- Another name associated with the kernel in the literature is Feature Detector because the weights can be fine-tuned to detect specific features in the input image.

For example:

- **Averaging neighboring pixels kernel** can be used to blur the input image.
- **Subtracting neighboring kernel** is used to perform edge detection.

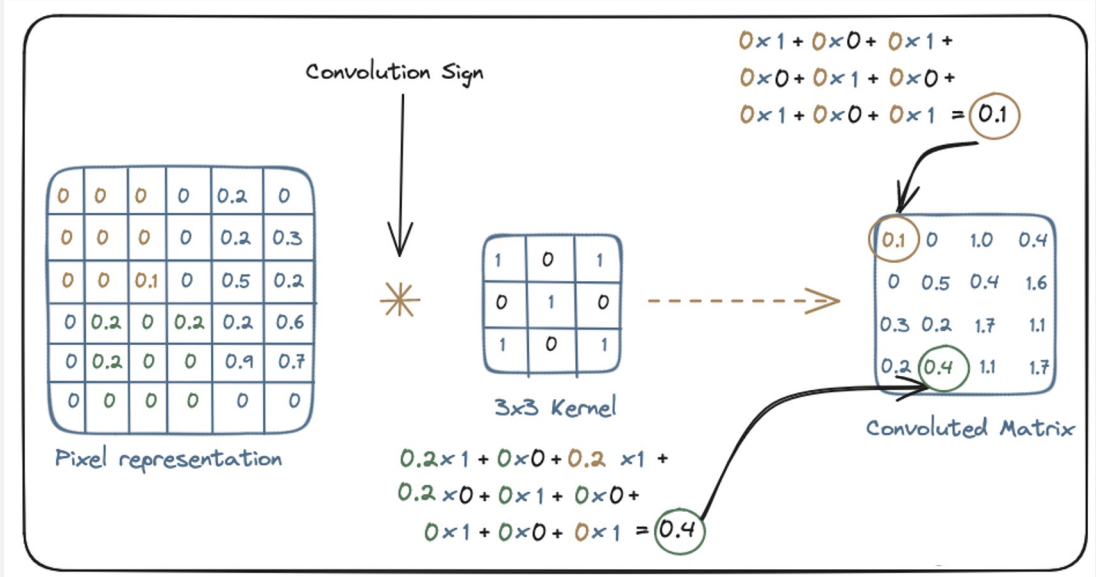
- The more convolution layers the network has, the better the layer is at detecting more abstract features.



Convolution Layers (cont'd)

Using these two matrices *(weights and pixels)*, we can perform the convolution operation by applying the dot product, and work as follows:

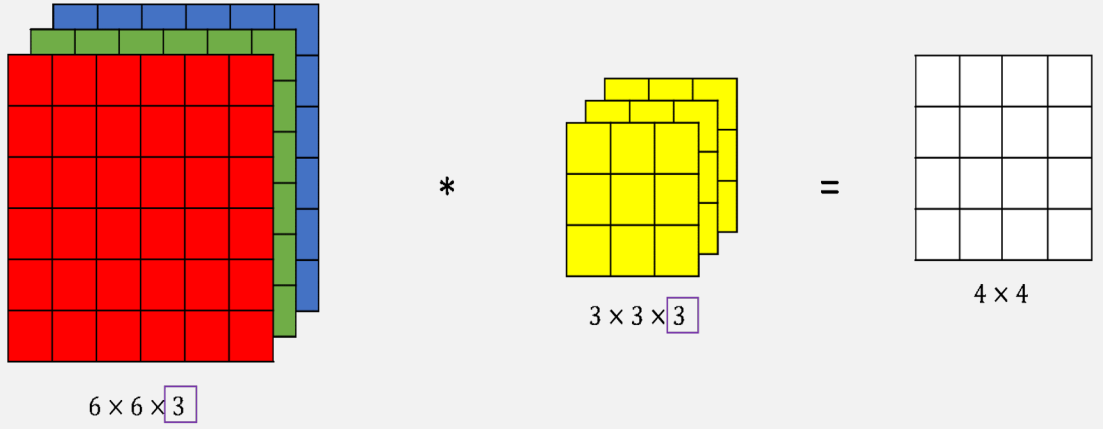
1. Apply the kernel matrix from the top-left corner to the right.
2. Perform element-wise multiplication.
3. Sum the values of the products.
4. The resulting value corresponds to the first value (top-left corner) in the convoluted matrix.
5. Move the kernel down with respect to the size of the sliding window.
6. Repeat steps 1 to 5 until the image matrix is fully covered.



Convolution Layer: Example

- Imagine you have an image. It can be represented as a cuboid having its length, width *(dimension of the image),* and height *(i.e the channel as images generally have red, green, and blue channels).*
- Now imagine taking a small patch of this image and running a mini magnifier, called a filter or *kernel* on it, with say, K outputs and representing them vertically.

- Now slide that magnifier across the whole image, as a result, we will get another image with different widths, heights, and depths.
- Instead of just R, G, and B channels now we have more channels but lesser width and height.
- If the patch size is the same as that of the image it will be a regular neural network. Because of this small patch, we have fewer weights.



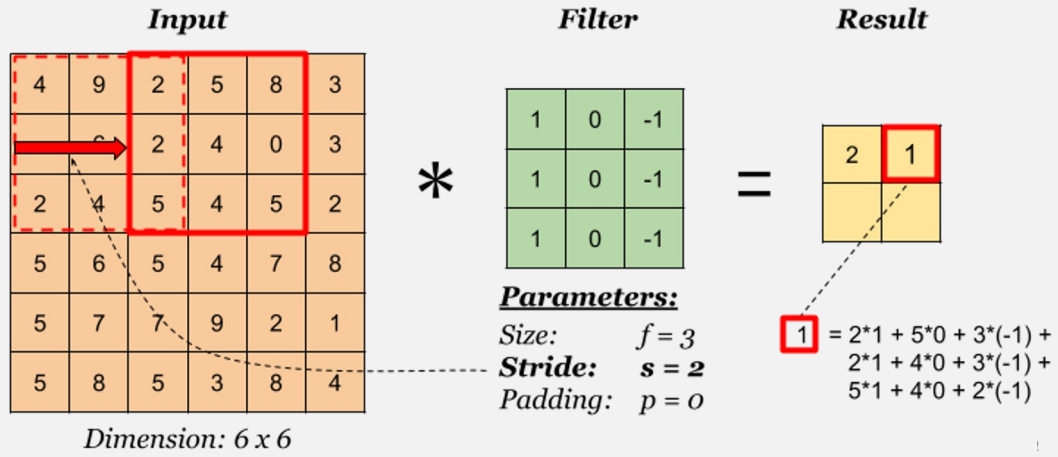
Convolution Layer Example *(cont'd)*

Now let's talk about a of mathematics in the convolution process.

- Convolution layers consist of a filters (or kernels) having small widths and heights and the same depth as that of input volume (3 if the input layer is image input).

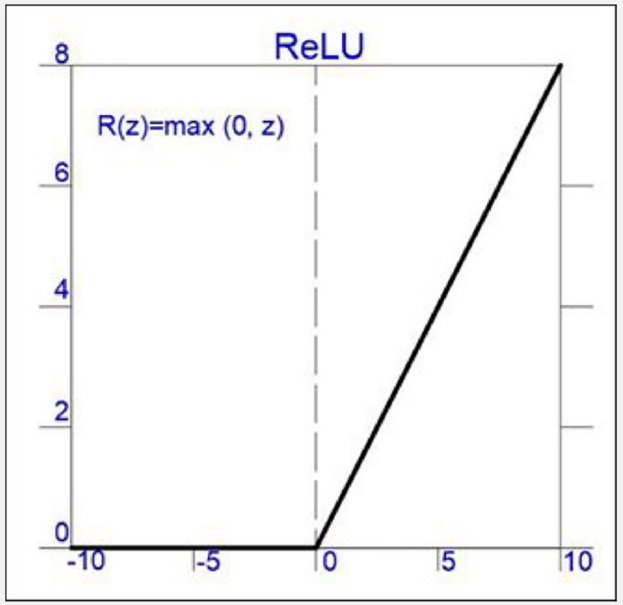
- A Convolution on an image with dimensions 34x34x3. The possible size of filters can be axax3, where 'a' can be anything like 3, 5, or 7 but smaller as compared to the image dimension.

- During the forward pass, each Kernel slide across the whole input volume step by step where each step is called stride (which can have a value of 2, 3, or even 4 for high-dimensional images) and compute the dot product between the kernel weights and patch from input volume.



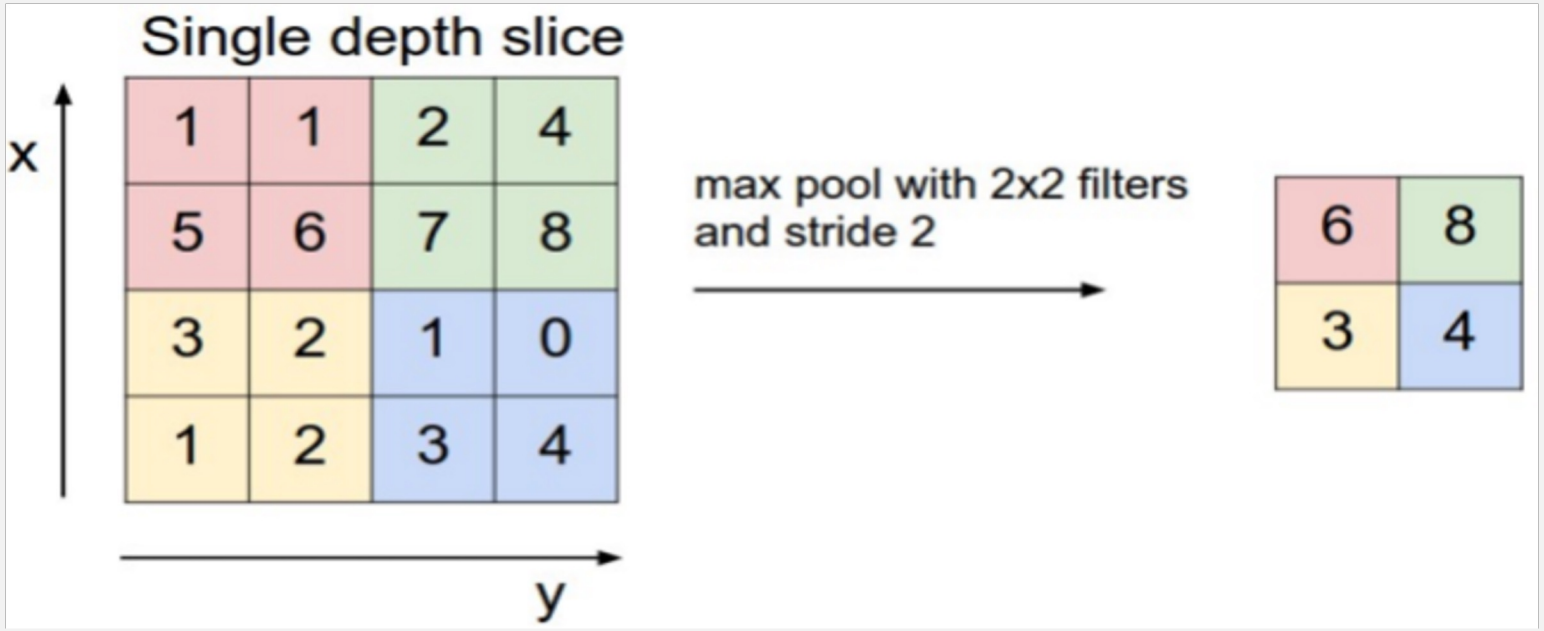
CNN Activation Function

- A ReLU activation function is applied after each convolution operation.
- This function helps the network learn non-linear relationships between the features in the image, hence making the network more robust for identifying different patterns.
- It also helps to mitigate the vanishing gradient problems.



Pooling Layer

- The goal of the pooling layer is to pull the most significant features from the convoluted matrix.
- This is done by applying some aggregation operations, which reduce the dimension of the feature map (convoluted matrix), reducing the memory used while training the network.
- Pooling is also relevant for mitigating overfitting.



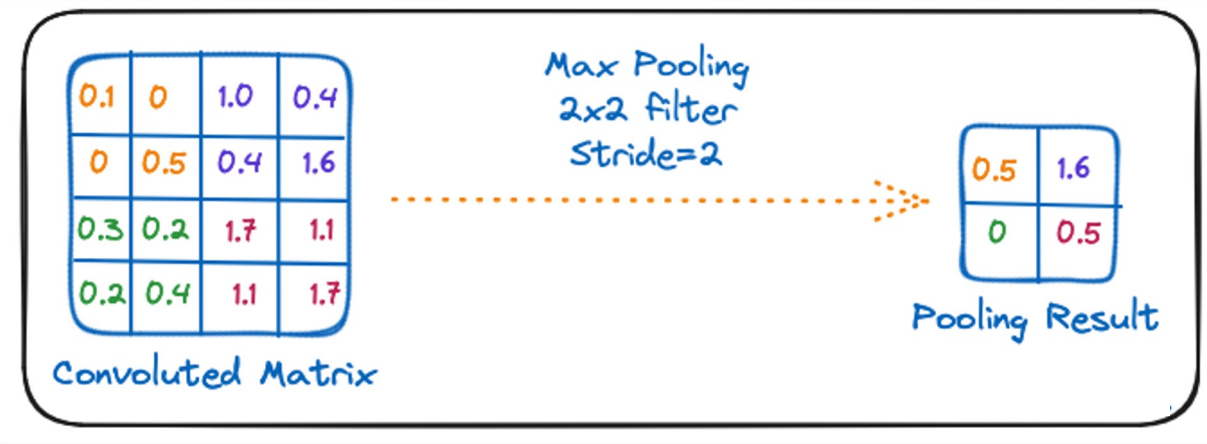
Pooling Layer *(cont'd)*

- The most common aggregation functions that can be applied are:

  A. **Max pooling:** which is the maximum value of the feature map.

  B. **Sum pooling:** corresponds to the sum of all the values of the feature map.

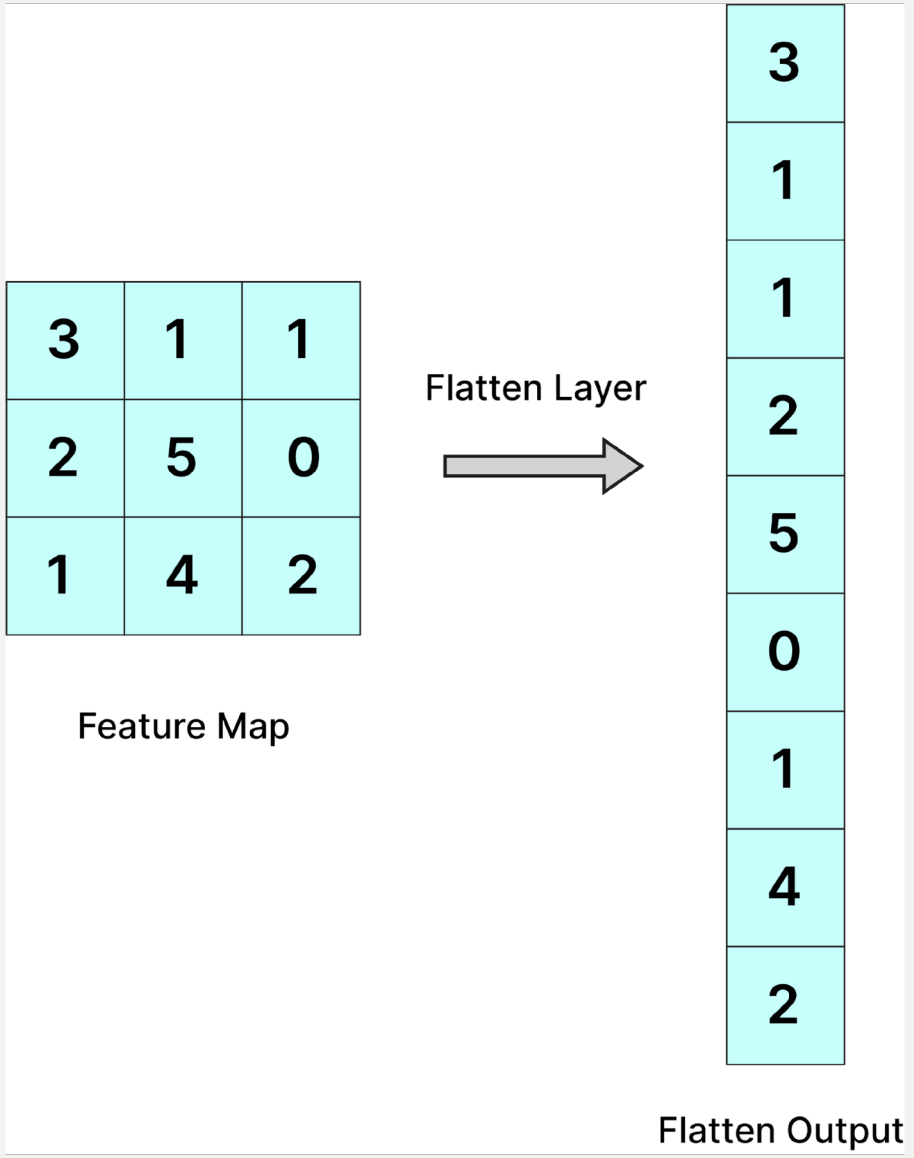
  C. **Average pooling:** is the average of all the values.



Flattening Layer

- The resulting feature maps are flattened into a one-dimensional vector after the convolution and pooling layers.

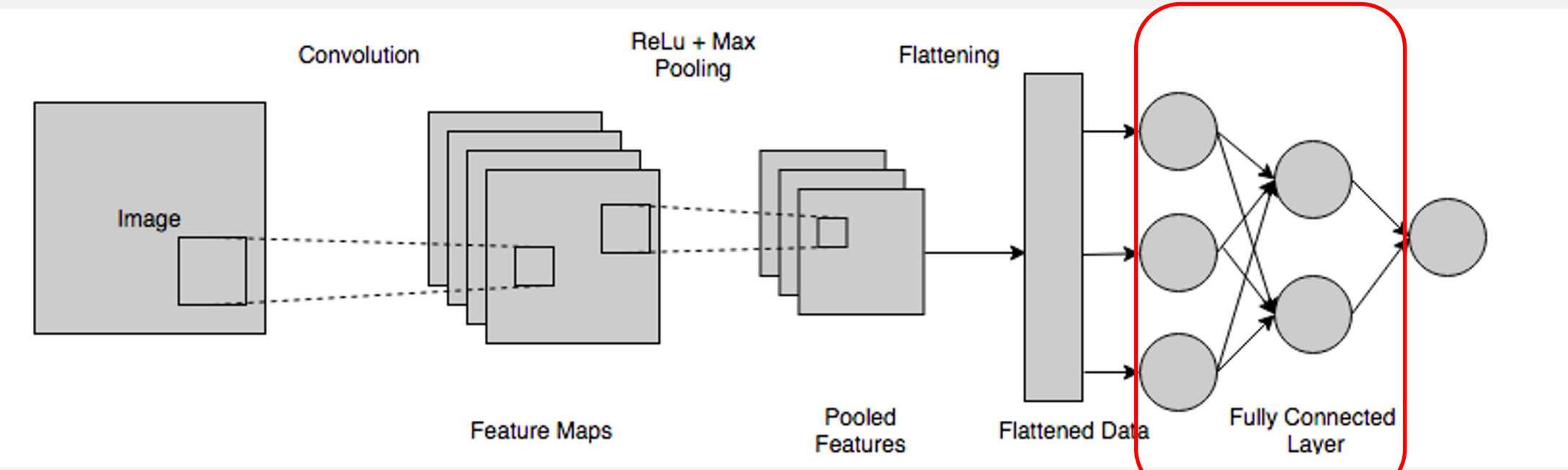
- Feature map after the Flattening layer can be passed into the Fully Connected Layer for categorization or regression.



Fully connected layers

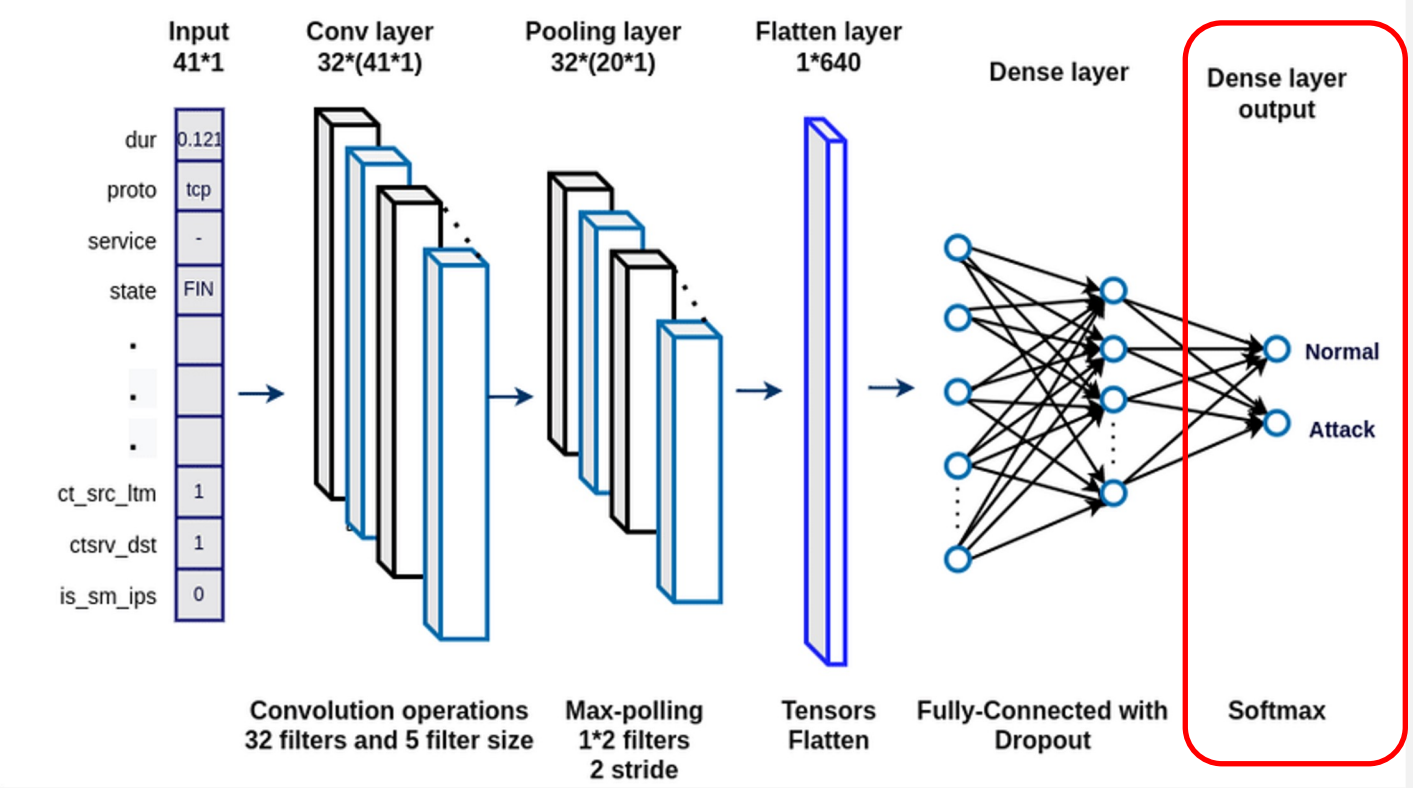
- These layers are in the last layer of the convolutional neural network, and their inputs correspond to the flattened one-dimensional matrix generated by the last pooling layer.

- It takes the input from the previous layer and computes the final classification or regression task.



Output Layer

- The output from the fully connected layers is then fed into a logistic function for classification tasks like sigmoid or SoftMax depending on whether the target column is binary, multi-class or multi-label.



A Classical CNN Architecture

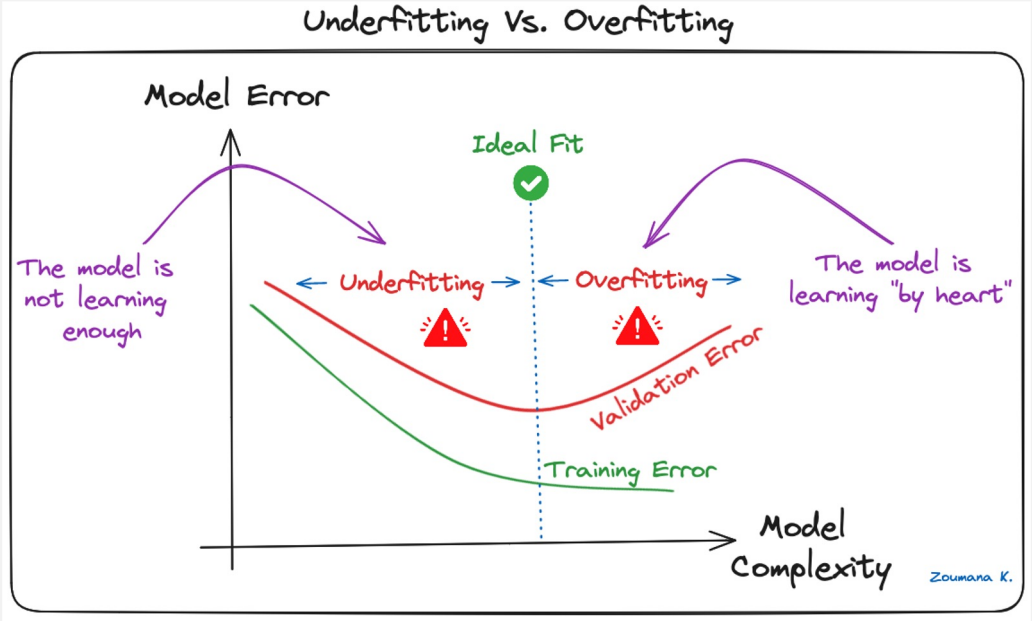
- The first hidden layer is a Convolutional layer call Convolution2D. It has 32 feature maps with size 5×5. This is also the input layer.
- Next is the pooling layer that takes the maximum value called MaxPooling2D. In this model, it is configured as a 2×2 pool size.
- The fifth layer is the flattened layer that converts the 2D matrix data into a vector called Flatten. It allows the output to be fully processed by a standard fully connected layer.
- Next, Fully connected layer Dense with 128 neurons.
- Finally, the output layer has 10 neurons for the 10 classes and a SoftMax activation function.

```
model=Sequential()
model.add(Conv2D(32,5,5,    padding='same',input_shape=(1,28,28),   activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2), padding='same'))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy',  optimizer='adam', metrics=['accuracy'])
```

Overfitting and Regularization in CNNs

- Overfitting is a common challenge in CNN Deep Learning modes. It happens when the model learns the training data too well **("learning by heart")**, including its noise and outliers.

- This can be observed when the performance on training data is too low compared to the performance on validation or testing data, and a graphical illustration is given below:



Overfitting and Regularization in CNNs *(cont'd)*

- Convolutional Neural Networks are particularly susceptible to overfitting due to their capacity for high complexity and their ability to learn detailed patterns in large-scale data.

- Some regularization techniques in CNNs are:

  Dropout

  Batch Normalization

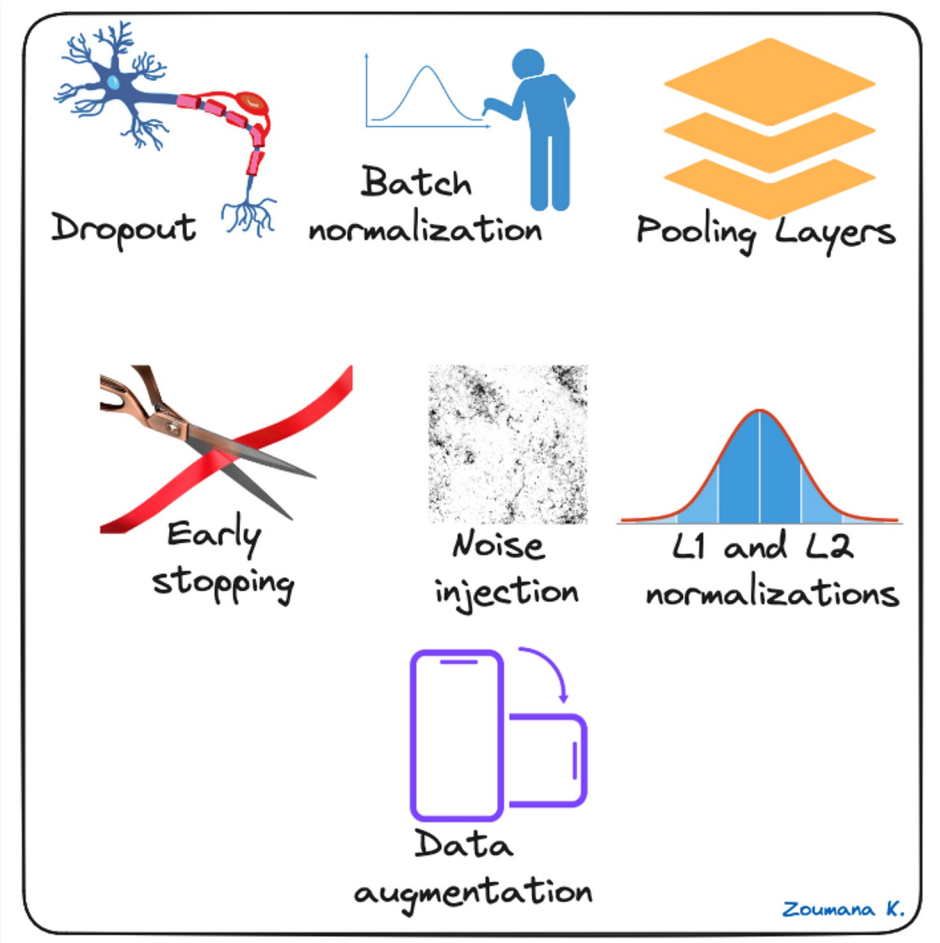
  Pooling Layers

  Early stopping

  Noise Injection

  L1 and L2 Regularization.

  Data Augmentation



Exercise

Notebook: Convolutional_Neural_Networks_Basic_Design.ipynb

Notebook: Convolutional_Neural_Networks.ipynb

Image Dataset & Preprocessing

What is an Image Dataset?

An image dataset includes digital images used in testing, training, and evaluating the performance of machine learning, commonly computer vision algorithms.

Image datasets help algorithms learn to identify and recognize information in images and perform related cognitive activities.

For example, AI algorithms can be trained to tag photographs, read car plates, and identify tumors in medical images.

ImageNet Datasets

The ImageNet project contains millions of images and thousands of objects for image classification.

It is widely used in the research community for benchmarking state-of-the-art models.

ImageNet Datasets include:

- ImageNet Synset
- Corel-1000 Dataset
- Caltech-256 Dataset
- Clatech-101 Dataset

(a) ImageNet Synset: One sample image from each category

(b) Corel-1000 Dataset: Sample images from each category

(c) Caltech-256 Dataset: One sample image from each category

(d) Caltech-101 Dataset: One sample image from each category

Top Image-Processing Python Libraries

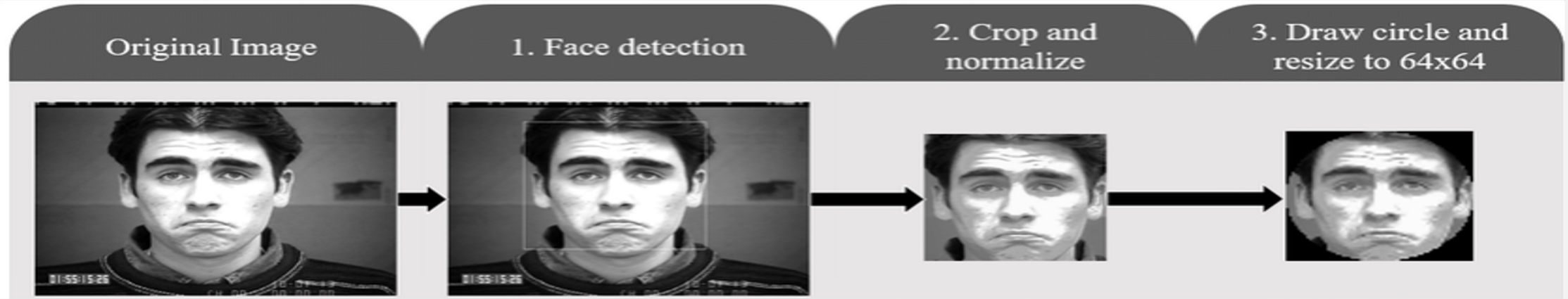
To process a large amount of data with efficiency and speed without compromising the results data scientists need to use image processing tools for machine learning and deep learning tasks.

The most useful image processing libraries in Python include:

- OpenCV
- Pillow/PIL
  (Python Imaging Library)
- Scikit-Image

Image Preprocessing

- To prepare image data for model input, some pre-processing is required. One example of this is for **Convolutional Neural Networks** (CNN), where the images need to be in arrays of the same size for fully connected layers.
- Pre-processing can reduce model training time and improve model inference speed as reducing their size can significantly shorten the Model training time.
- Geometric transformations of images, such as rotation, scaling, and translation, are considered pre-processing techniques.



Techniques for Image Preprocessing

Image pre-processing steps aim to improve the quality, consistency, and suitability of images for model training and inference.

Various techniques are employed for image pre-processing, including but not limited to:

- **Resizing**
- **Normalization**
- **Data Augmentation**
- **Image Filtering**
- **Greyscale**

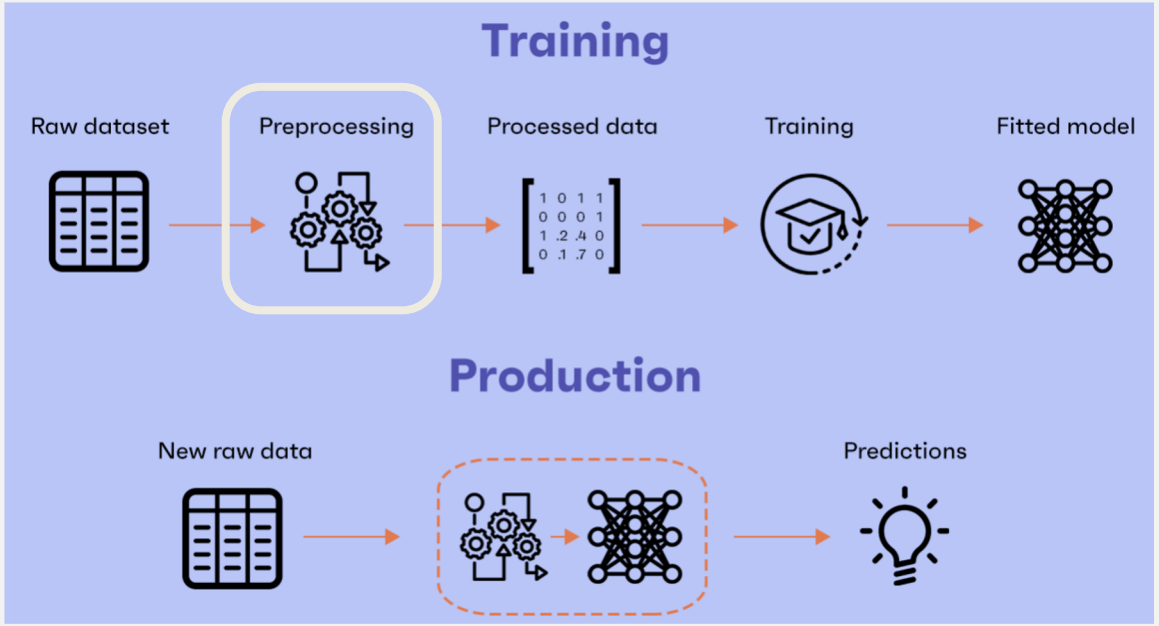


Image Preprocessing: Resizing and Scaling

- Most of the neural network models assume a square shape input image.
- This means that each image needs to be checked if it is a square or not and cropped appropriately.
- Images can be resized to a smaller or larger size and scaled to have a certain range of pixel values.

```
1  img = cv2.imread ('original.jpg')
2  half = cv2.resize(img, (0, 0), fx = 0.1, fy = 0.1)
3  bigger = cv2.resize(img, (1050, 1610))
```

Image Preprocessing: Normalization

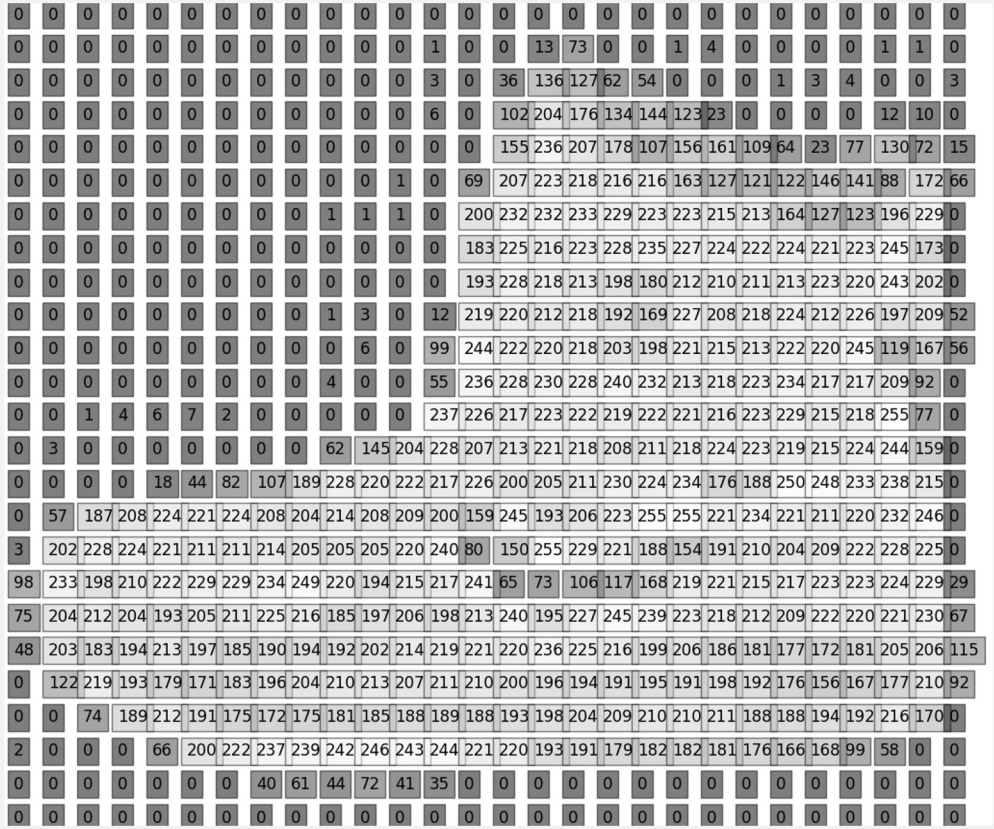
- **Normalization** is a technique in image processing that alters the range of pixel intensity values.
- **Normalization** convert pixel values to a range between 0 and 1.

Why we do Normalization?

- Normalization is used to improve the model's performance; The objective is to ensure uniformity in the dynamic range of a collection of data, signals, or images to prevent exhaustion or distraction.
- Normalization of input data for deep learning (DL) applications is an important step that impacts network convergence and final results.

Image Preprocessing: Normalization *(cont'd)*

- The MNIST Fashion image on the right show the pixel values ranges from 0 to 255, where 255 represents the **lightest-color** cell and 0 represents the **darkest-color** cell.
- Pixel's normalization will convert values to be from 0 to 1 or -1 to 1.
- We can normalize/scale the image by three methods:
  - Using Scikit learn's `MinMaxScaler`
  - Using Scikit learn's `StandardScaler`
  - Dividing both training and test set features by 255.

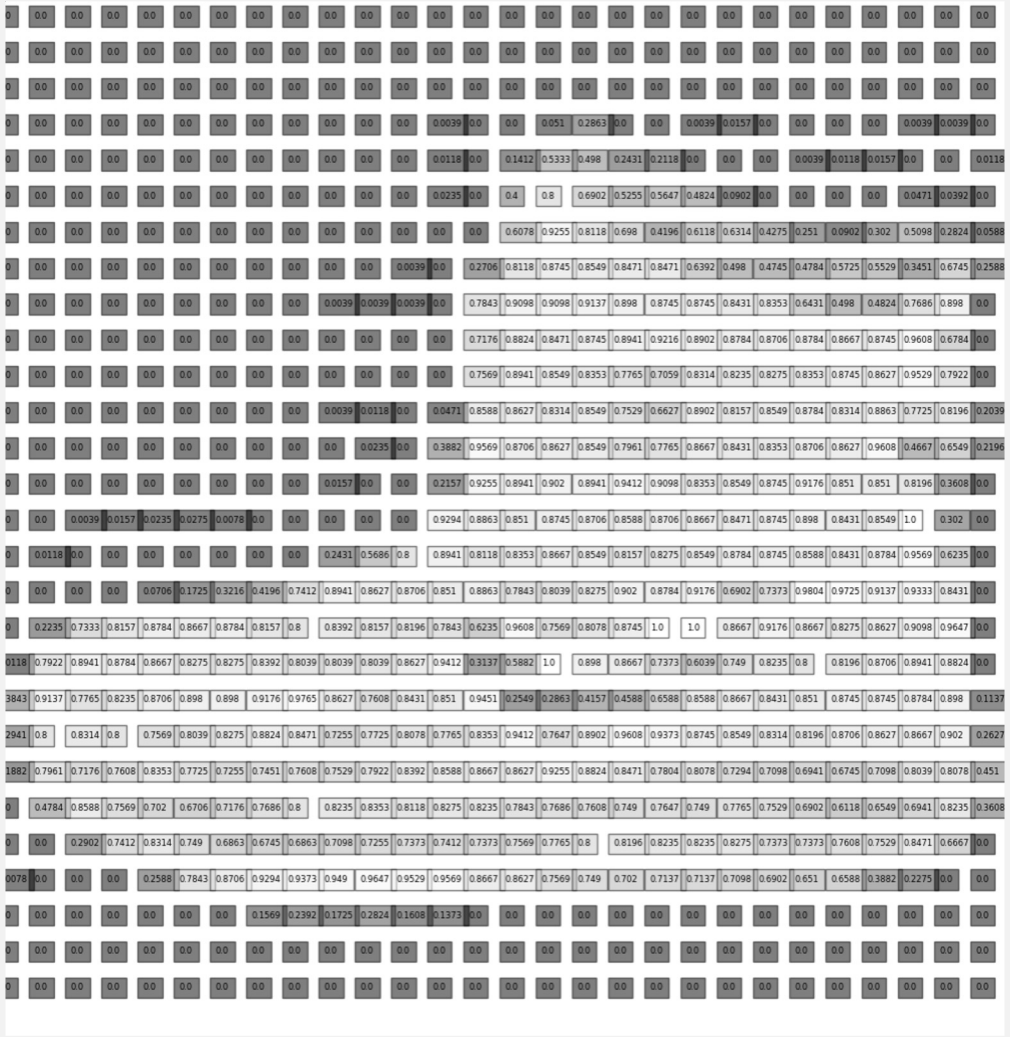


MNIST Fashion before Normalization

Image Preprocessing: Normalization *(cont'd)*

```python
Img = cv2.imread ('image.jpg')
normalized = img / 255.0

# Normalization could be done by subtracting the mean and scaling to unit variance:
mean, std = cv2.meanStdDev (img)
std_img = (img - mean) / std
```



MNIST Fashion after Normalization

Image Preprocessing: Data Augmentation

- **Data augmentation** is a technique used to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data.
- Common operations used for data augmentation for images:

    1. Rotation
    2. Shearing

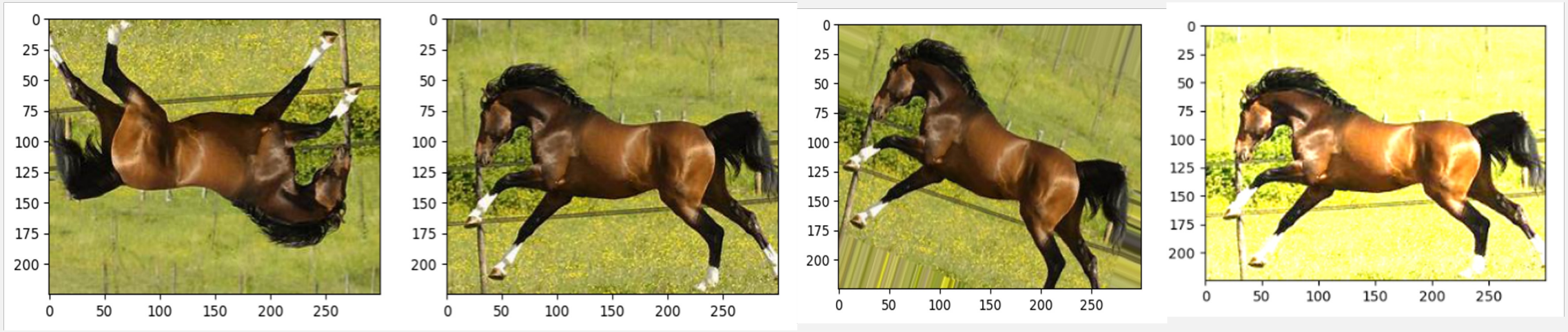
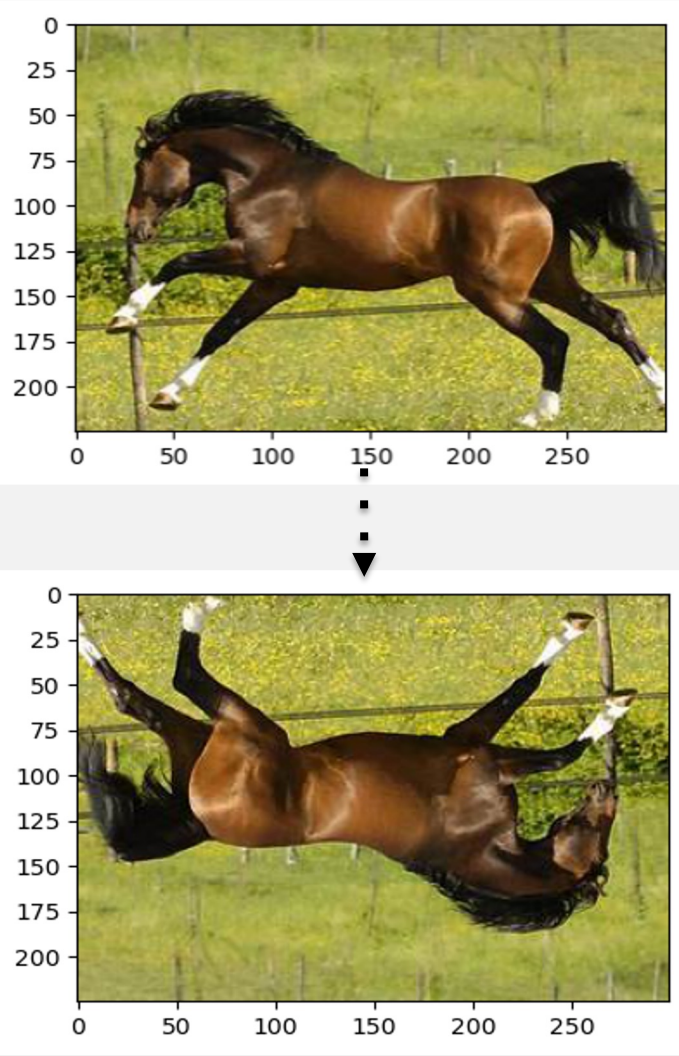


Image Preprocessing: Data Augmentation

For example, Flipping:

- This reverses the rows or columns of pixels in either vertical or horizontal cases, respectively.




```
samples = 10
# ImageDataGenerator for flipping
datagen = ImageDataGenerator(horizontal_flip=True,  vertical_flip=True)
# Create an iterator
it = datagen.flow(samples, batch_size=1)
```

Image Preprocessing: Greyscale

- **Grayscale** is converting images from colorful image to black and white image. It is normally used to reduce computation complexity in machine learning algorithms.
- Since most pictures don't need color to be recognized, it is wise to use grayscale, which reduces the number of pixels in an image, thus, reducing the computations required.

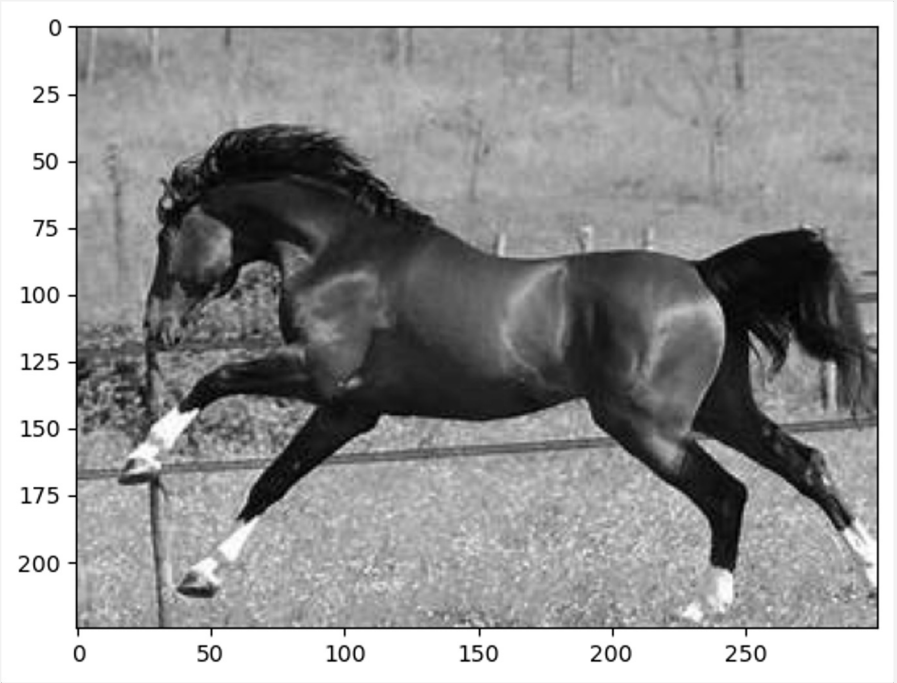
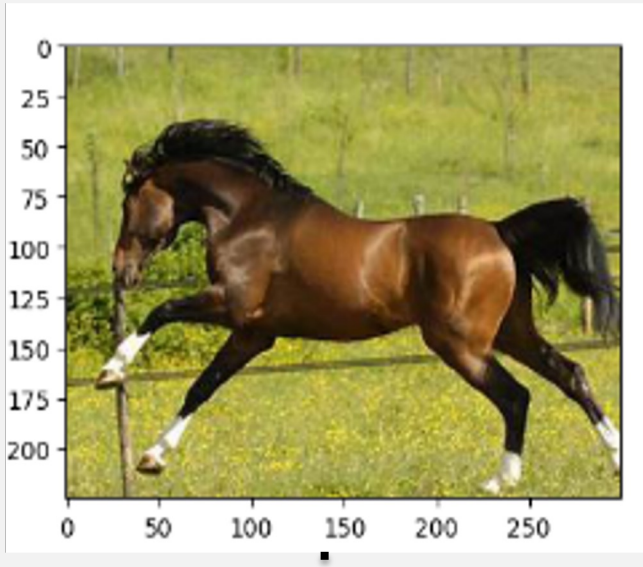


Image Preprocessing: Greyscale *(cont'd)*

```
1   #Convert RGB to Grayscale:
2   gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```



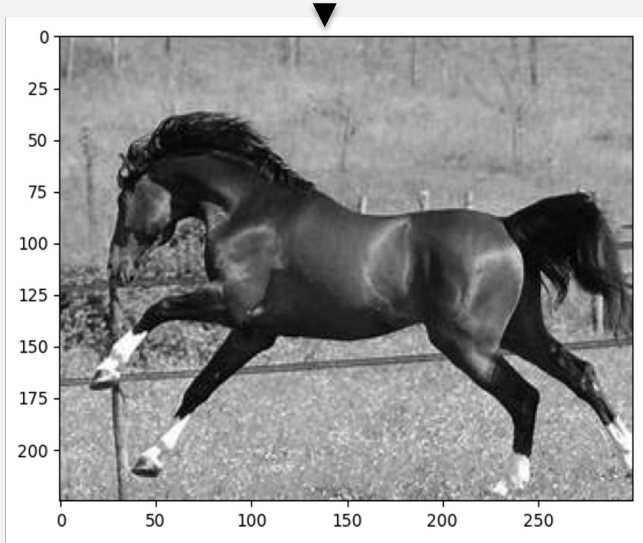


Image Preprocessing: Image Filtering

- Applying a digital filter involves taking the convolution of an image with a kernel *(a small matrix).*
- Convolution involves applying a filter or kernel to the input image by computing the element-wise multiplication between the kernel and the overlapping region of the image, and then summing up the results.
- Image processing operations implemented with filtering include smoothing, sharpening, and edge enhancement.

Image Preprocessing: Image Filtering *(cont'd)*

For example, Image Blurring:

- It removes high frequency content *(eg: noise, edges)* from the image. So, edges are blurred a little bit in this operation (*there are also blurring techniques which don't blur the edges)*.

```
1  img = cv2.imread('opencv-logo-white.png')
2  blur = cv2.blur(img,(5,5))
```

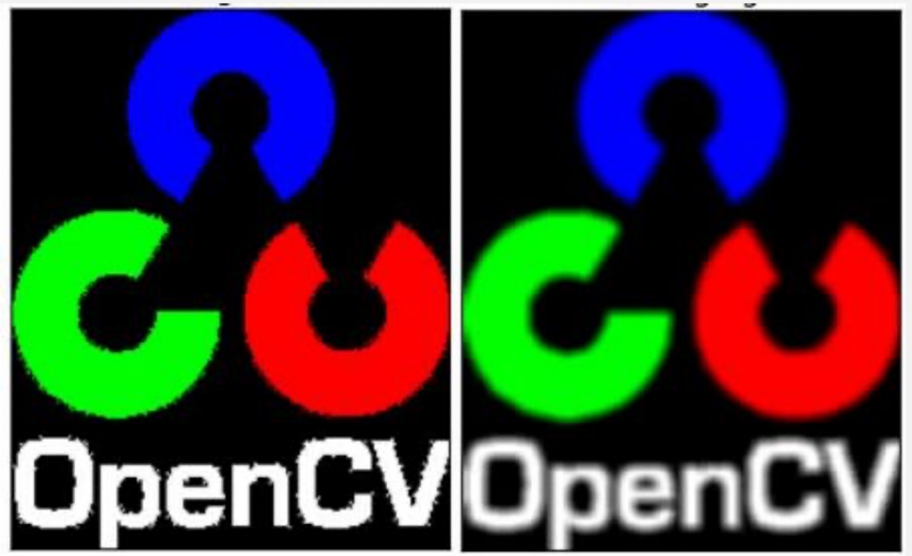


Image Connected Components

- Connected components refer to sets of pixels in an image that is connected to each other by some criterion, such as sharing the same color, intensity, or texture.
- Connected components are important in computer vision for several reasons:
    1. **Object detection and recognition**
    2. **Image segmentation**
    3. **Feature extraction**

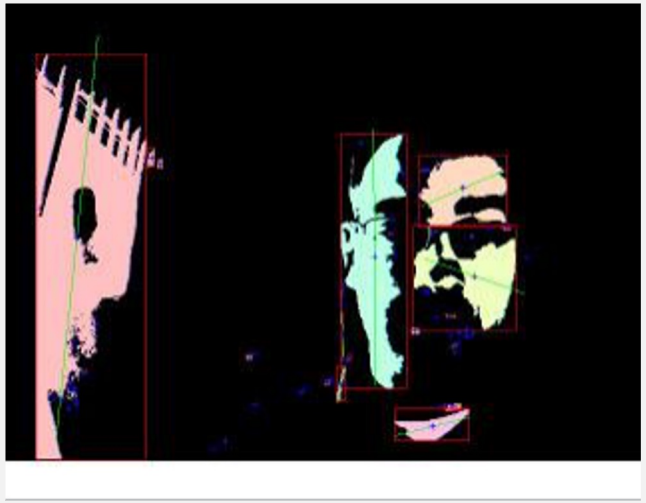
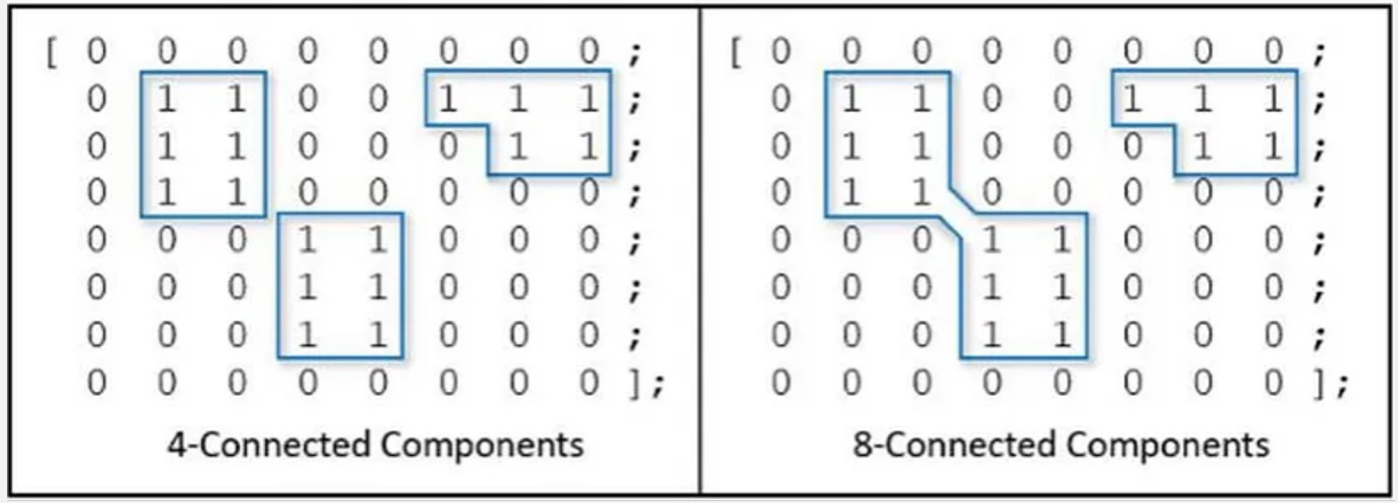


Image Connected Components *(cont'd)*

4 Connected Components: If two pixels' edges contact, they are connected. If two pixels are connected in either the horizontal or vertical direction and are both on, they are a single object.

8 Connected Components: If two pixels' edges or corners meet, they are connected. If two adjacent pixels are both on and connected in a horizontal, vertical, or diagonal direction, they are a single object.



Exercise

Notebook: ImageDataGenerator.ipynb

Notebook: DataAugmented_CNN.ipynb

Putting it all together

Now lets put that all together:

Notebook: CNN_Exercise.ipynb