A Gentle Guide to Convolutional Neural Network

1. Overview of Neural Network

A neural network is **computational model** that is inspired by the structure and functioning of the human brain, designed to recognize pattern and make prediction by learning from data without being explicitly programmed to do so. At a basic level, a neural network consists of many connected nodes called **neurons**, which is responsible for the computation within the network.

One of the earliest and simplest neural network models is the **Perceptron**, invented by Frank Rosenblatt in 1958. The perceptron laid the foundation for modern neural network It consists of a single **neuron**, multiple **inputs** (representing the features of the dataset), and a single **output**. The perceptron learns by processing input features through the neuron, where each input is multiplied by an associate weight (weighted inputs) to determined its importance. These weighted inputs are then summed, and a bias term is added to influence the final result (as shown in the equation below).

$$\text{Linear equation: } z = \sum_{i=1}^{n}(Xi.wi) + b$$

- **n** refers to the number of input features
- $\mathbf{X_i}$ refers to the i-th features
- $\mathbf{w_i}$ refers to the i-th weights
- **b** refers to the bias

The result of the linear combination is passed through a non-linear activation function, such as the sigmoid (logistic function), which generates the perceptron's output. The **sigmoid** function produces a value between 0 and 1, making it ideal for binary classification tasks.

$$\text{Sigmoid (logistic) function: } \hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$$

- $\hat{y}$ refers to the predicted output
- **σ** refers to the sigmoid function
- **Z** refers to the linear combination

Furthermore, the perceptron learns to adjust its weights and biases through a process called **backpropagation**. To determine how much the weights and biases need to be updated, the perceptron first calculates the loss—a measure of the difference between the predicted output and the true output. Then, the **gradient** of the loss function is computed with respect to the perceptron's parameters, allowing for **iterative updates**. Over time, this process minimizes the loss, indicating the perceptron's improving ability to learn and make accurate predictions.

Building on the concept of the perceptron, more advanced and sophisticated neural network architectures have been developed, including **Convolutional Neural Networks** (for image-related tasks), **Recurrent Neural Networks** (for short sequential data-related tasks), and **Transformers** (for long sequential data-related tasks). These architectures were designed to overcome the limitations of the simple perceptron and enable the performance of more complex tasks, such as image classification and natural language processing.

2. The challenge of working with images

Regular neural networks face several challenges when applied to image tasks due to how computers perceive images. Images are represented as **multi-dimensional** arrays of numbers, with each number corresponding to the intensity of a pixel in the image (see Fig. 1).

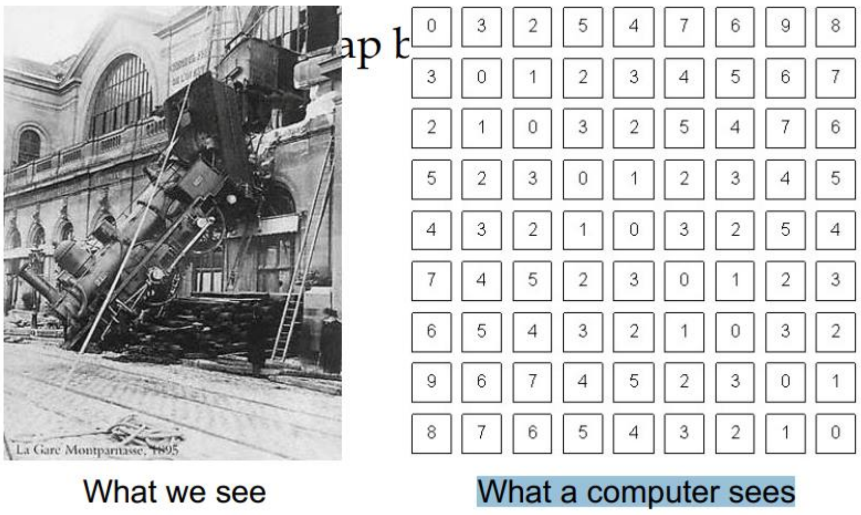


Figure 1: Comparison between how we and computer perceive images

If an image is the size of **256x256**, then there are **65,536 pixels** in total in an image. A typical image has 3 channels: red, green and blue (see Fig. 2), each containing 65,536 pixels.

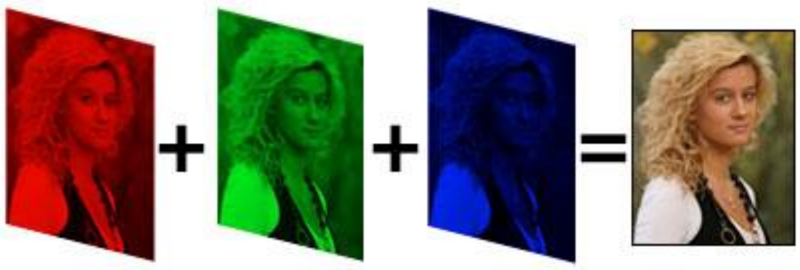


Figure 2: Colored images with 3 channels

Treating each pixel as an independent input in a regular neural network results in an enormous number of parameters (weights and biases), leading to high computational costs and inefficient training. Moreover, when training images on a regular neural network, the images must be **flattened** into one-dimensional vectors (see Fig. 3), causing the loss of **spatial relationships** between pixels. To overcome these limitations, a specialized type of neural network known as the **Convolutional Neural Network (CNN)** was introduced.

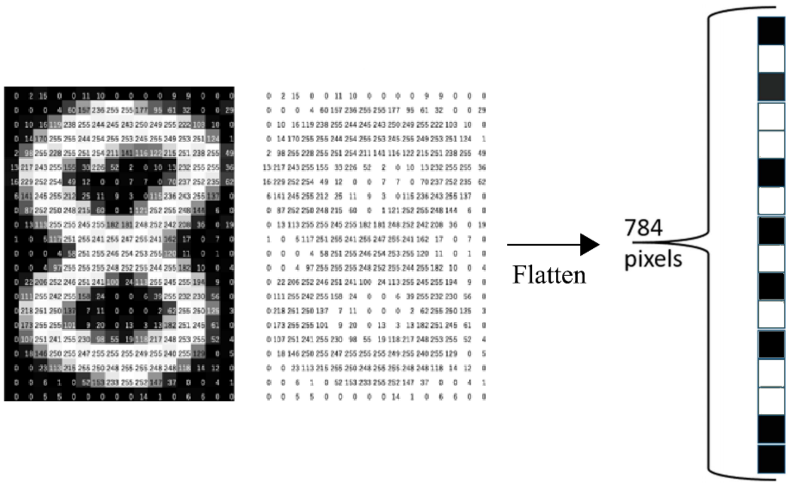


Figure 3: The flattening process of an image with the size of 28x28

3. Convolutional neural network

Convolutional Neural Networks (CNNs) are specialized neural networks designed to process and learn from visual data, such as images. Unlike traditional neural networks, which process flattened pixel data, CNNs are specifically structured to **preserve** and **capture spatial relationships** between pixels, making them highly effective for image classification and other image-related tasks.

Before the invention of CNNs, traditional neural networks relied on manually crafted **feature detectors**, such as **edge detectors** or **Histogram of Oriented Gradients (HOG)**, to extract meaningful features from images (see Fig. 4). These features were then fed into fully connected neural networks to make predictions. This approach reduced computational costs and captured meaningful features for classification. However, it was time-consuming to manually design feature detectors and limited in generalization, as different datasets often required unique feature detectors.

CNNs overcome these limitations by integrating two key components: **feature learning layers** and **classification layers** (see Fig. 5). These layers enable CNNs to automatically learn and extract **hierarchical features** directly from the data, eliminating the need for manually crafted feature detector.

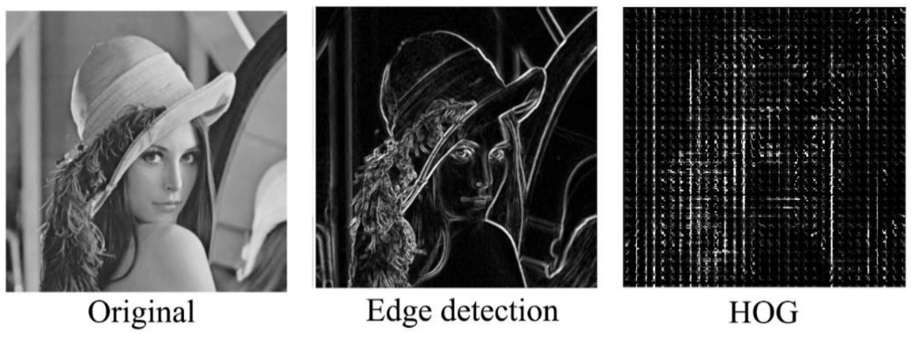


Figure 4: Original image and it detected feature using Edge detector and HOG algorithm.

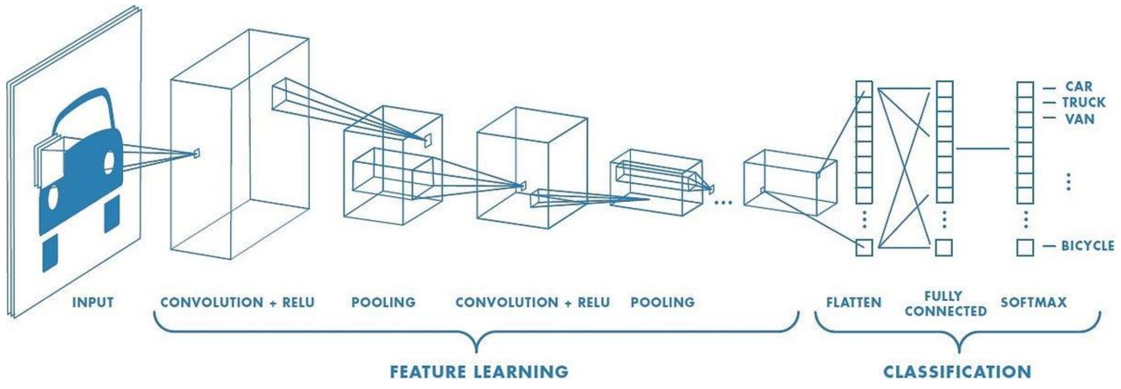


Figure 5: Convolutional neural network architecture

3.1 Feature learning layer

The feature learning layer consists of several sub-layers: **convolutional layers**, **pooling layers**, and activation functions. These components work together to **automatically extract** hierarchical features (ranging from simple to abstract features) from input image data. In the convolutional layers, **small filters** (or **kernels**) are used to slide across the input images, detecting patterns such as edges and textures, producing **feature maps**. These feature maps represent the important information contained within the image. Figure 6 visualizes the convolutional operation on a 3x3 input image with a 2x2 kernel, producing a 2x2 feature map.

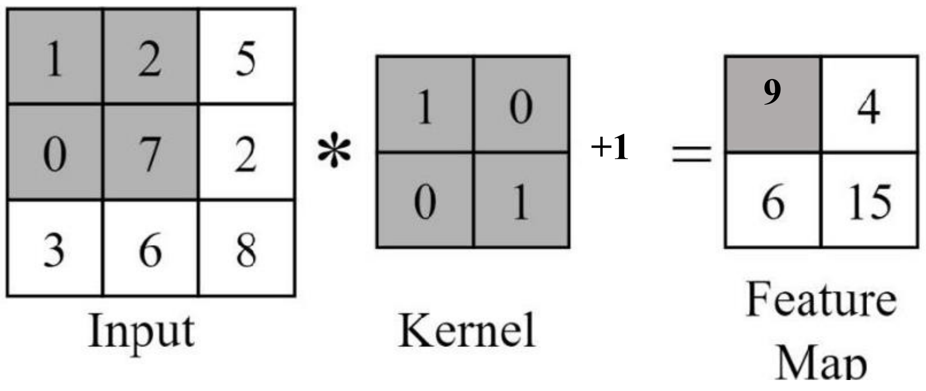


Figure 6: Convoluting a 3x3 image with a 2x2 kernel to get a 2x2 feature map

The convolution operation can be mathematically expressed as follows:

$$Z_1=(1x1+0x2+0x0+1x7)+1=\mathbf{9}$$

- $\mathbf{Z_1}$ refers to the first pixel in the feature map.
- **1, 0, 0, and 1** are the values of the **kernel weights**.
- **1** is the bias term added to the result of the convolution.
- **9** is the output, which is the result of the **dot product** between the input patch and the kernel, plus the **bias**.

The above equation can be expressed as a linear equation as followed:

$$Z_i = \sum_{i=1}^{n}(Xi.wi) + b$$

- $\mathbf{Z_i}$ refers to the i-th pixels calculated after taking the dot products between the input and the kernel plus a bias.
- $\mathbf{X_i}$ refers to the i-th pixels in an image
- $\mathbf{w_i}$ refers to the i-th weights of the kernel
- **b** refers to the bias associated with each kernel
- **n** refers to the number of pixels in the image

However, it is common that an activation function is applied to $\mathbf{Z_i}$ before it gets mapped to the feature map (Figure 7 illustrates the convolutional process with an activation). The most popular activation function used in this process is **Rectified Linear Unit (ReLU)** due to its simplicity and computational efficiency. ReLU can be mathematically expressed as follows:

$$\text{ReLU: } a_i(z_i) = \max(0, z_i)$$

For any input of Zi:

- If $Z_i > 0$, $a_i(Z_i) = Z_i$
- If $Z_i < 0$, $a_i(Z_i) = 0$

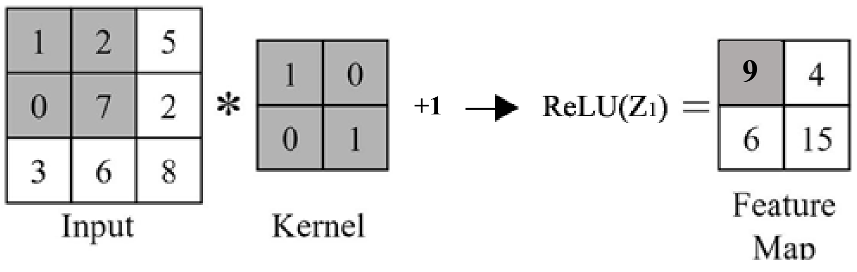


Figure 7: Convoluting a 3x3 image with a 2x2 kernel and an activation function to get a 2x2 feature map

However, in a typical convolutional neural network, multiple filters are applied, resulting in several feature maps at each convolutional layer. These feature maps then serve as the input for subsequent convolutional layers, making the mathematics of convolution with feature maps slightly different from that of the initial input image.

Suppose there are 2 feature maps created after the first convolution and pooling layer. We want to create 2 filters (kernels) to slide over these 2 feature maps. However, 4 filters are created instead. This happens because each filter in the convolutional layer is applied to every feature map from the previous layer, and the results are summed together to produce a new feature map. Each filter can convolve over a feature map only once, so 4 filters are needed to convolve over 2 feature maps, ensuring that each feature map is processed by 2 filters. (see Fig. 8)

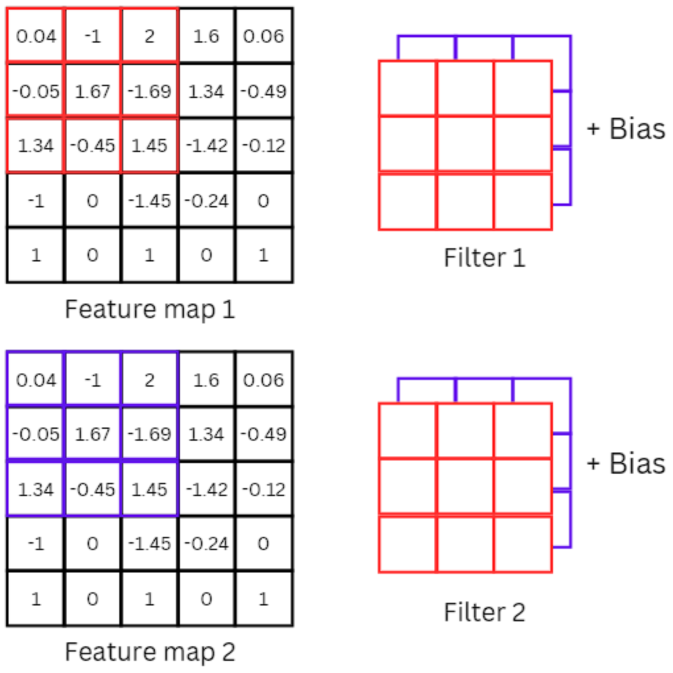


Figure 8: 4 filters are created with 2 filters convolving over each feature map

The weights of the filter and the corresponding patch of the feature map are multiplied element-wise and then summed (an operation referred to as the dot product between the filter and the feature map). Applying this process to 2 filters on 2 different feature maps produces 2 values, which are then summed together, followed by the addition of a bias term. Lastly, an activation function is applied before mapping the result onto a new feature map (see Fig. 9).

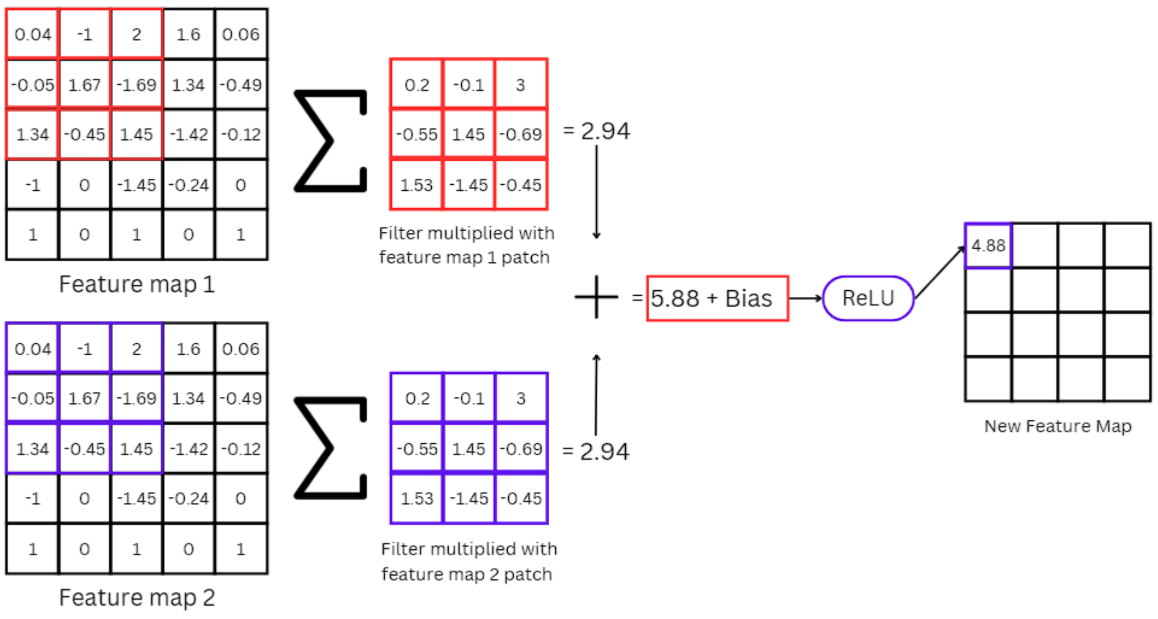


Figure 9: Convolution process over multiple feature maps

Suppose there are M input feature maps, each with n pixels (or values), and a filter with M corresponding kernels (one for each input feature map). The output at a specific pixel Zi in the resulting feature map is calculated as:

$$Z_i = \sum_{m=1}^{M} \sum_{j=1}^{n} (X_{mj} . w_{mj}) + b$$

- $\mathbf{Z_i}$ refers to the i-th pixel in the output feature map.
- $\boldsymbol{X_{mj}}$ refers to the j-th pixel in the m-th input feature map.
- $\boldsymbol{w_{mj}}$ refers to the j-th weight in the kernel corresponding to the m-th input feature map.
- **b** refers to the bias associated with the filter.
- **M** refers to the number of input feature maps.
- **N** refers to the number of pixels in each feature map.

Similarly, the **pooling** layer is responsible for **reducing** the size of the feature map. This reduction helps decrease the computational power required to train the network. The pooling layer scales down the feature map size while retaining only the important features in the image. There are two common types of pooling layers: **max-pooling** and **average-pooling**.

Max-pooling works by sliding an $\boldsymbol{n \times n}$ **filter** across the feature map, where $\boldsymbol{n}$ refers to the **height** and **width** of the filter, and mapping the highest value within the filter to a new matrix called the **pooled feature map**. Figure 10 demonstrates the max-pooling process on a 4x4 feature map with a 2x2 max-pooling filter, producing a 2x2 pooled feature map. This process reduces the size of the feature map while preserving only the most important features.

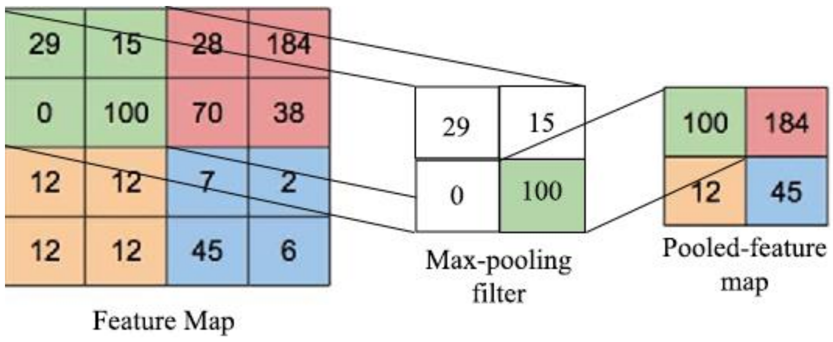


Figure 10: Max-pooling a 4x4 feature map with a 2x2 max-pool filter to get a 2x2 pooled-feature map

In contrast to max-pooling, **average-pooling** calculates the average value within the sliding window and maps it to a new matrix called the pooled feature map. While max-pooling focuses on the most prominent feature in each region, average-pooling **summarizes** the information by **averaging** all values, helping to retain subtle features that max-pooling might overlook. Figure 11 demonstrates the average-pooling process on a 4x4 feature map using a 2x2 average-pooling filter, resulting in a 2x2 pooled feature map.

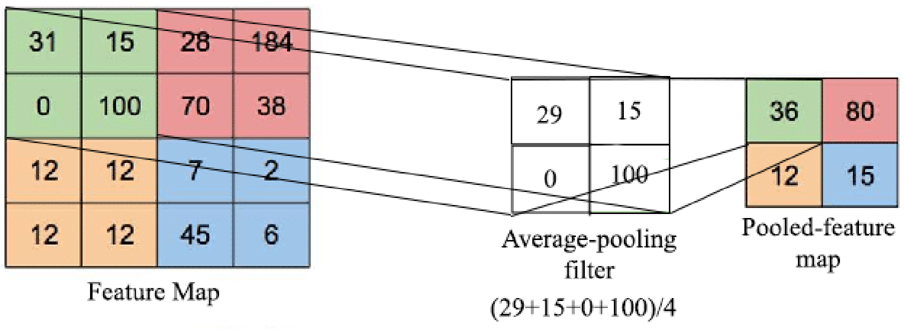


Figure 11: Average-pooling a 4x4 feature map with a 2x2 max-pool filter to get a 2x2 pooled-feature map

3.2 Classification layer

The second component, the classification layer, consists of one or more fully connected layers that take the processed features from the feature learning layer and classify them into **distinct categories**. These layers act as decision-makers, transforming the extracted features into a one-dimensional vector and using traditional neural network techniques to make predictions. The classification layer is composed of **neurons** that perform a standard **linear combination** of the input features, followed by an **activation function** to produce the final output.

$$Z_i = \sum_j \left(w_{ij} \,.\, x_j\right) + b_i$$

Where:

- $\mathbf{w_{ij}}$ refers to the weights between the input $\mathbf{x_j}$ and the neuron $\mathbf{y_i}$
- $\mathbf{x_j}$ refers to the input values (flattened features from the previous layer)
- $\mathbf{b_i}$ refers to the bias term of i-th neuron
- $\mathbf{Z_i}$ refers to the output of the i-th neuron

After the linear combination, an activation function (e.g., ReLU) is applied to introduce non-linearity.

$$\text{ReLU: } a_i(z_i) = \max(0, z_i)$$

For any input of Zi:

- If $Z_i > 0$, $a_i(Z_i) = Z_i$
- If $Z_i < 0$, $a_i(Z_i) = 0$

Lastly, the output layers contain either **Softmax** (multi-class classification) or **Sigmoid** (binary classification) activation function to produce the final output of the model.

$$\text{Sigmoid (logistic) function: } \hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$$

- $\hat{y}$ refers to the predicted output which range from 0 to 1
- **σ** refers to the sigmoid function
- **Z** refers to the linear combination

$$\text{Softmax function: } \hat{y} = \sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

- z_i refers to the output from previous layers
- $\hat{y}$ refers to the output of softmax which is a probability distribution across i-th class

To illustrate, the feature learning layer extracts meaningful information from images of animals, such as their ears, eyes, jaws, and noses. The classification layer then takes these features and determines whether the animal is a cat, dog, or cow based on the extracted characteristics.

By combining automatic feature extraction with classification, CNNs eliminate the need for manual feature detection, making them both powerful and versatile for a wide range of tasks, from image classification to object detection and beyond. This versatility has led to the development of various specialized CNN architectures designed to address specific challenges in computer vision tasks.

4. LeNet-5

To demonstrate the potential of deep learning in real-world applications, the **LeNet-5** architecture, developed by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, was designed to recognize **handwritten** and **machine-printed** characters. LeNet-5 consists of **6 hidden layers**: 3 convolutional layers, 2 subsampling (average-pooling) layers, and 1 fully connected layer. Figure 12 illustrates the LeNet-5 architecture for digit recognition.

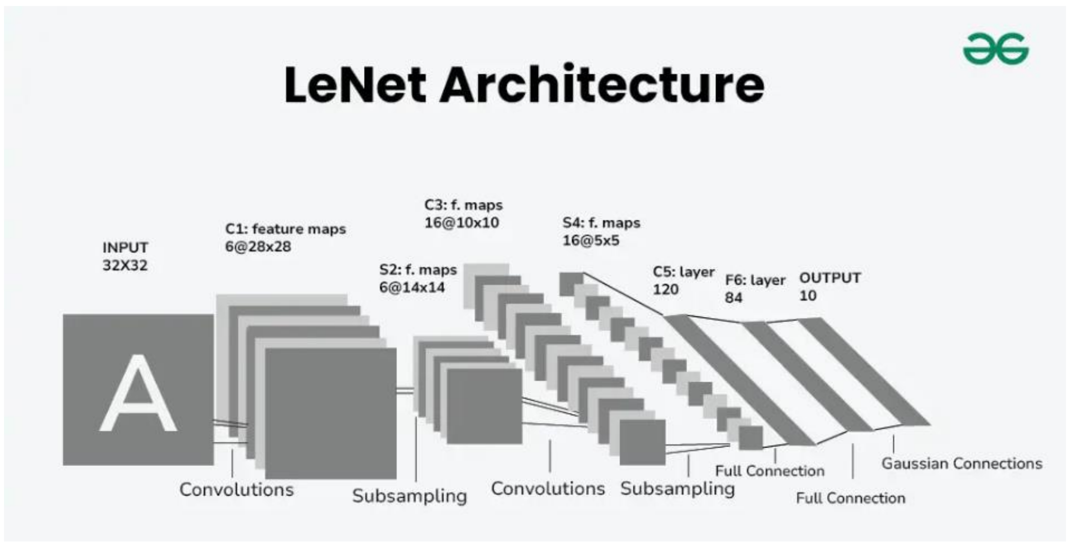


Figure 12: LeNet-5 for Digit Recognition

5. AlexNet

AlexNet, developed by Geoffrey Hinton and his student Alex Krizhevsky, won the 2012 ImageNet competition. It consists of **10 layers**: 5 convolutional layers, 3 max-pooling layers, and 2 fully connected layers. What sets AlexNet apart from its competitors is its use of the **ReLU (Rectified Linear Unit)** activation function, which accelerates training. Additionally, AlexNet introduced **dropout** to prevent overfitting by randomly setting some neurons to zero during training, forcing the network to avoid relying too heavily on specific neurons. Figure 13 illustrates the architecture of AlexNet.

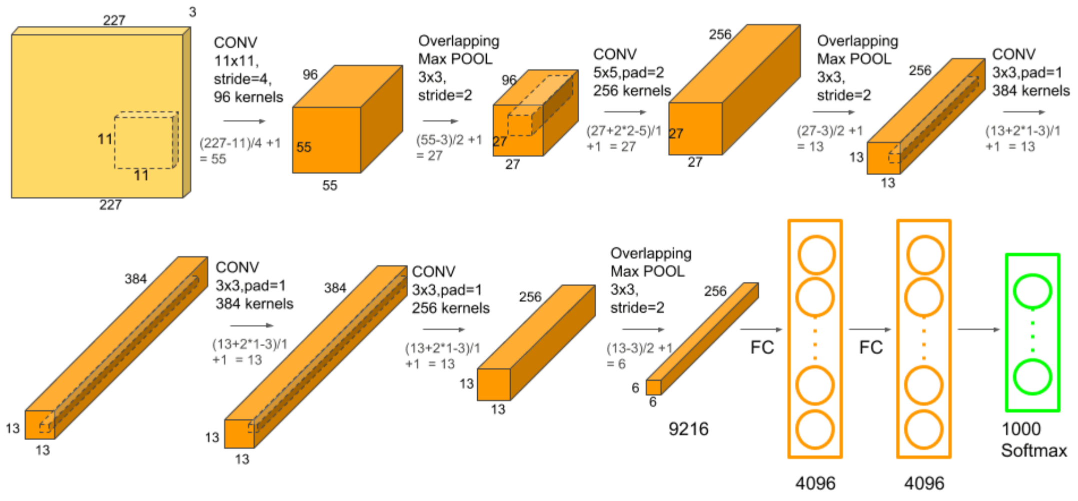

227
3
227
11
11
CONV 11x11, stride=4, 96 kernels
(227-11)/4 +1 = 55
96
55
55
Overlapping Max POOL 3x3, stride=2
(55-3)/2 +1 = 27
96
27
27
CONV 5x5,pad=2 256 kernels
(27+2*2-5)/1 +1 = 27
256
27
27
Overlapping Max POOL 3x3, stride=2
(27-3)/2 +1 = 13
256
13
13
CONV 3x3,pad=1 384 kernels
(13+2*1-3)/1 +1 = 13
384
13
13
CONV 3x3,pad=1 384 kernels
(13+2*1-3)/1 +1 = 13
384
13
13
CONV 3x3,pad=1 256 kernels
(13+2*1-3)/1 +1 = 13
256
13
13
Overlapping Max POOL 3x3, stride=2
(13-3)/2 +1 = 6
256
6
6
9216
FC
4096
FC
4096
1000 Softmax

Figure 13: AlexNet architecture for ImageNet 2012 competition

6. Convolutional Autoencoder (CAE)

A **convolutional autoencoder (CAE)** is a type of convolutional neural network designed for **unsupervised learning tasks** such as image compression, noise reduction, or feature extraction. It consists of two main parts (see Fig. 14):

1. **Encoder**: Uses convolutional layers to compress the input image into a lower-dimensional representation (latent space), preserving important features while discarding redundant information.
2. **Decoder**: Reconstructs the input image from the compressed latent representation using transposed convolutional layers.

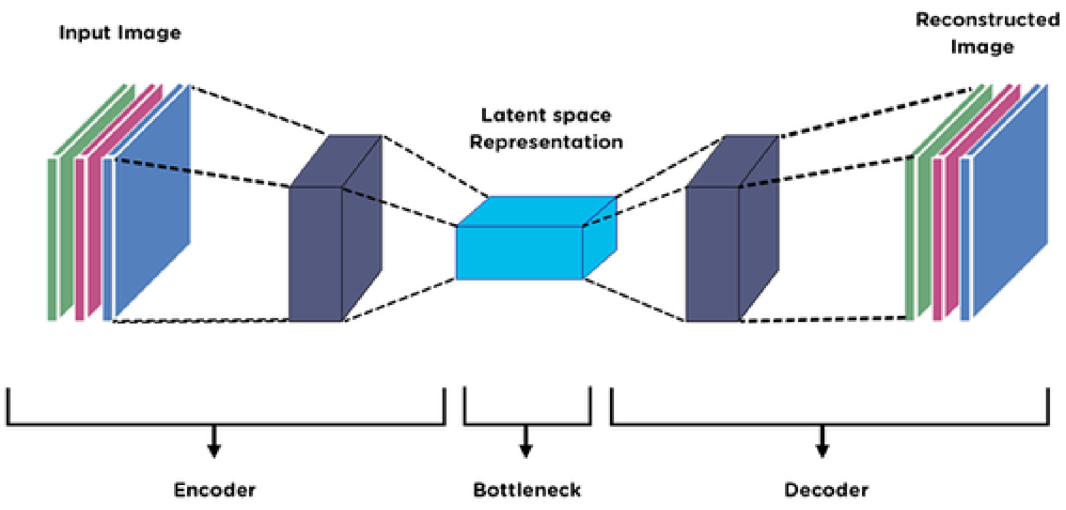


Figure 14: Architecture of convolutional autoencoder

As shown in Figure 11, the **encoder** compresses the image into a smaller dimension, capturing only the essential information. This compressed representation, often referred to as the **latent space** or **bottleneck**, serves as a condensed version of the input image, retaining critical features while discarding less relevant details. This **dimensionality reduction** allows the model to focus on the most informative patterns, making it ideal for tasks such as **image compression** or **denoising**. The **decoder** then **reconstructs** the original image from the **latent representation** by progressively expanding the compressed features back to their original size. This process is evaluated using a **reconstruction loss** function, such as **mean squared error (MSE)**, which measures the difference between the original image pixels and their reconstructed counterparts. MSE can be mathematically expressed as:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- **N** refers to the number of pixels in the image
- $\boldsymbol{y_i}$ refers to the i-th true pixel value (e.g., 0-255)
- $\boldsymbol{\hat{y}_i}$ refers to the i-th reconstructed pixel value

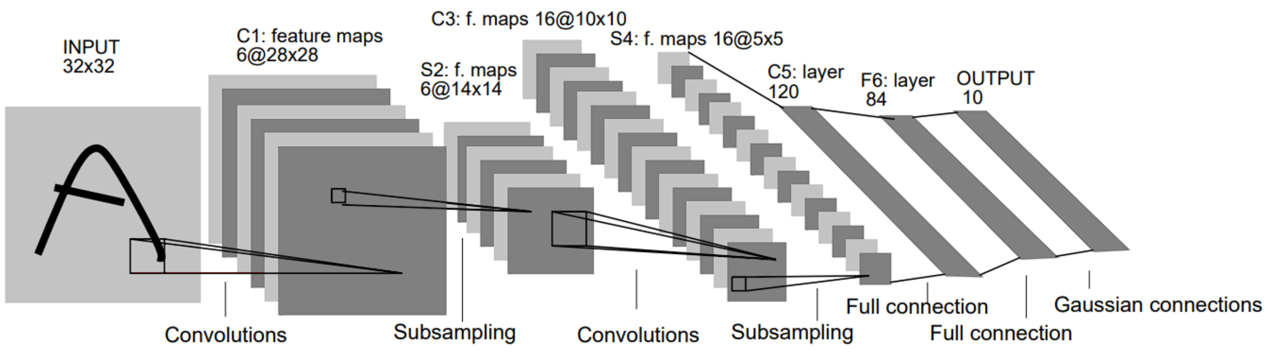
7. Code implementation of basic CNN architecture

This section will go over the syntax and implementation of CNN using Tensorflow. Furthermore, each parameter will be explained and visualized in detail.

7.1 LeNet-5

To implement a simple CNN model, we will be following the LeNet-5 architecture taken directly from a famous research paper titled **“Gradient-Based Learning Applied to Document Recognition”** written by Yann LeCun, and others. For the sake of simplicity, we will be **modifying** some layers of the architecture to fit with modern technique (e.g., switching subsampling layers with average-pooling).

To start, we will break down the architecture layer-by-layer and rebuild it using Tensorflow.



7.2 Layer-by-layer breakdown of Modified LeNet-5

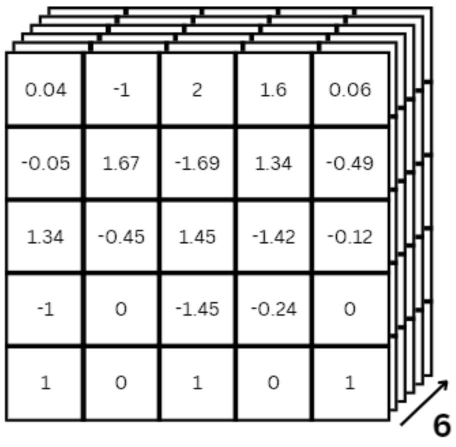
1. Input layer:
    - Input Size: 32×32 (grayscale image)
2. Convolutional layer 1 (C1)
    - Filters: 6 filters of size 5x5
    - Padding: None
    - Strides: 1
    - Activation: Tanh
    - Output: 6 features map of size 28x28
3. Average-pooling layer 1 (S2)
    - Filters size: 2x2
    - Strides: 2
    - Output: 6 pooled feature map of size 14x14
4. Convolutional layer 2 (C3)
    - Filters: 16 filters of size 5x5
    - Padding: None
    - Strides: 1
    - Activation: Tanh
    - Output: 16 features map of size 10x10
5. Average-pooling layer 2 (S4)
    - Filters size: 2x2
    - Strides: 2
    - Output: 16 pooled feature map of size 5x5
6. Convolutional layer 3 (C5)
    - Filters: 120 filters of size 5x5
    - Padding: None
    - Strides: 1
    - Activation: Tanh
    - Output: 120 features map of size 1x1
7. Fully connected layer (F6)
    - Neurons: 84
    - Activation: Tanh
8. Output layer
    - Neurons: 10

- Activation: Softmax

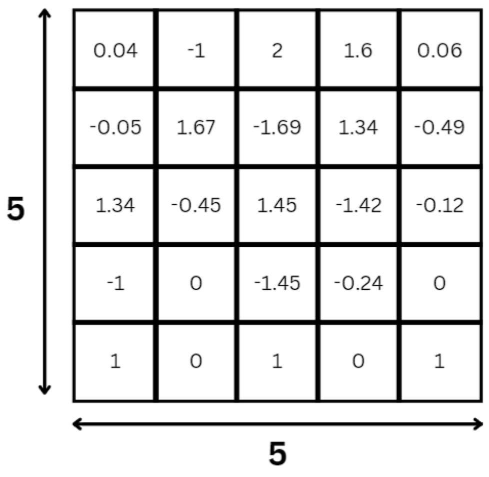
a. Convolutional layer 1 (C1)

```
Conv2D(filters=6, kernel_size=(5, 5), strides=(1, 1), activation='tanh',
input_shape=(input_shape), padding="valid", name="C1")
```

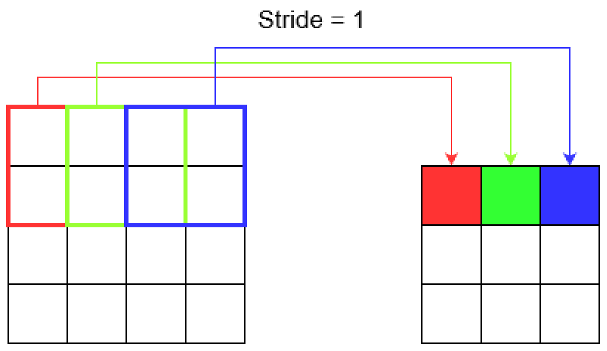
Filters: Small matrices or window that slides across input images extracting the image's features. The code creates 6 of these filters that will slide across the image, creating 6 feature maps.



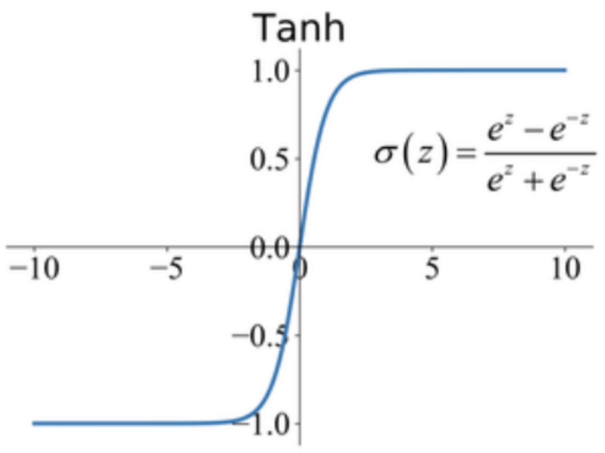
Kernel size: The size of the filters. In the case of this code, the filters will have the size of 5x5.



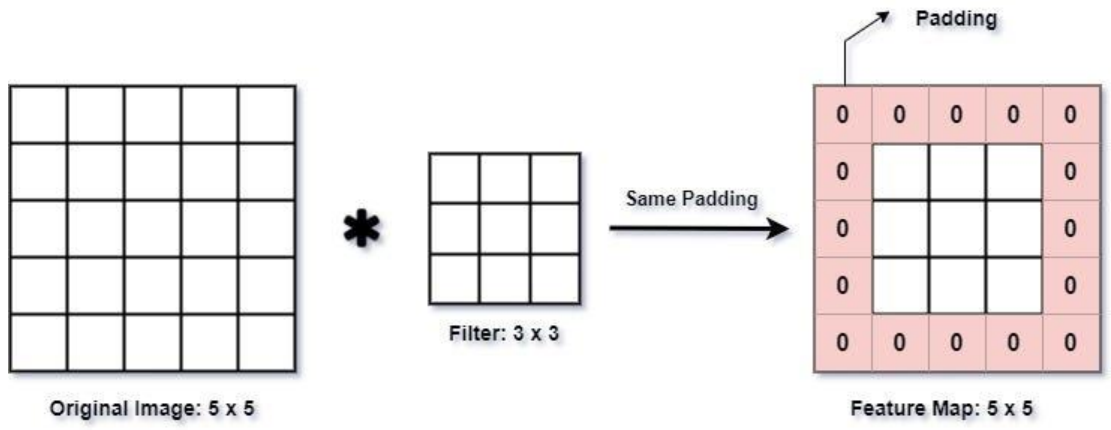
Strides: The steps at which the filters move. Strides of (1, 1) indicating the filters move 1 pixel at a time to the right, and after reaching the end of the image, it will move 1 pixel down starting from the beginning.



Tanh activation: Tanh produces an output that centered around zero (-1 to 1). This is useful in a case where you want to ensure that the output of a neuron can have both positive and negative values.



Padding: Adding extra pixels (zero) around the border of the feature map. This is done to ensure that after the filter convolved over the input image, the feature map will have the exact same dimension of the input image. In the case of our code (padding="valid"), we do not add any adding, therefore the feature map is slightly smaller than the input image. However, if we want to add padding, we would specify padding="same".

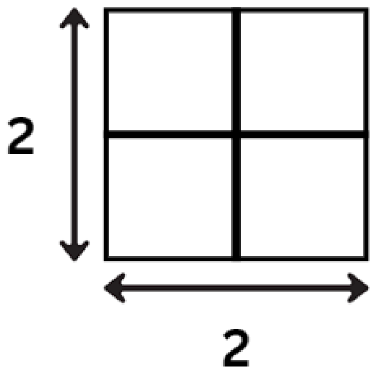


Input shape: The dimension (shape) of the input image (e.g., a grayscale image of size 32x32 would have a shape of (32, 32, 1), where 1 indicates a single channel)

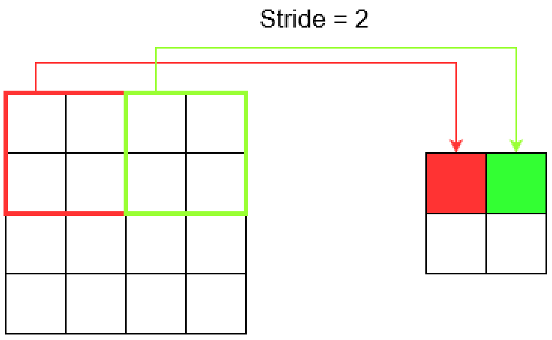
b. Average-pooling layer 1 (S2)

```
AveragePooling2D(pool_size=(2, 2), strides=(2, 2), name="S2")
```

Pool size: The size of pooling filter. In the case of our code, the pooling filter is the size of 2x2.



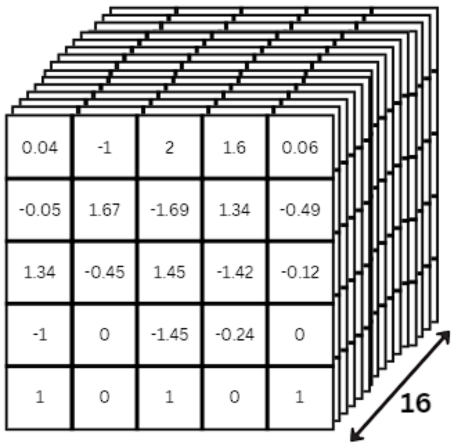
Strides: The pooling filter in the case of our code (Strides-(2, 2)) will slides across the feature map 2 pixels at a time, meaning there will be no overlapping. Additionally, once it reaches the end of the image, it will slide down 2 pixels.



c. Convolutional layers 3 (C3)

```
Conv2D(filters=16, kernel_size=(5, 5), strides=(1, 1), activation='tanh',
padding="valid", name="C3")
```

Filters: 16 convolutional filters are created to convolved across S2 pooled feature map.



0.04	-1	2	1.6	0.06
-0.05	1.67	-1.69	1.34	-0.49
1.34	-0.45	1.45	-1.42	-0.12
-1	0	-1.45	-0.24	0
1	0	1	0	1

d. Average-pooling layer 2 (S4)

```
AveragePooling2D(pool_size=(2, 2), strides=(2, 2), name="S4")
```

The pooling filter slides across all 16 feature maps producing 16 pooled feature maps

e. Convolution layers 4 (C5)

```
Conv2D(filters=120, kernel_size=(5, 5), strides=(1, 1), activation='tanh',
padding="valid", name="C5")
```

Filters: 120 filters are created to convolved over the 16 feature map, producing 120 new feature maps.

f. Flattening

```
layers.Flatten(name="Flatten")
```

These layers effectively compress the filters into a one-dimensional vector, which is essential because the neurons in the fully connected layers can only process one-dimensional input.

g. Fully connected layer (F6)

```
Dense(units=84, activation='tanh', name="F6")
```

Units: Number of neurons in a single layer. In the case of our code, 84 neurons is created with a tanh activation function.

h. Output layer

```
Dense(units=10, activation='softmax', name="Output")
```

Units: The number of neurons in the output layers is determined by the number of classes needed to be predicted. In the case of our code, the model is trained to predict whether a written digit is in the range of 0 to 9.

With the code above, you have successfully built a simple LeNet-5 model using TensorFlow. The code below provides the complete process for building the model, including selecting an optimizer and choosing an appropriate loss function.

```
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import SGD
def create_LeNet5(input_shape):
  model = models.Sequential([

      # C1 and inputs
      layers.Conv2D(filters=6, kernel_size=(5, 5), strides=(1, 1), activation='tanh',
input_shape=(input_shape), padding="valid", name="C1"),

      # S2 Average pooling
      layers.AveragePooling2D(pool_size=(2, 2), strides=(2, 2), name="S2"),

      # C3
      layers.Conv2D(filters=16, kernel_size=(5, 5), strides=(1, 1), activation='tanh',
padding="valid", name="C3"),

      #S4
      layers.AveragePooling2D(pool_size=(2, 2), strides=(2, 2), name="S4"),

      #C5 (kernel size is smaller than original LeNet-5 in this layer because our images are smaller
than what LeNet-5 was trained on)
      layers.Conv2D(filters=120, kernel_size=(5, 5), strides=(1, 1), activation='tanh',
padding="valid", name="C5"),

      #F6
      layers.Flatten(name="Flatten"), # Convert all features map into 1D vector for fully connected
layer to process

      layers.Dense(units=84, activation='tanh', name="F6"),
      # layers.Dropout(0.5, name="Dropout"), # Disable 50% of the neuron to avoid overfitting,

      # Output Layer
      layers.Dense(units=10, activation='softmax', name="Output") # Units=10 because there are 10
classes (0-9)

  ], name="modifiedLeNet5")
  return model

LeNet5 = create_LeNet5(input_shape=(32, 32, 1)) # (32, 32, 1) is the shape of our images

# Compile the model
SGD = SGD(learning_rate=0.01)  # Specify the learning rate
# sparse categorical crossentropy is a common loss function for multi-class classification
LeNet5.compile(optimizer=SGD, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Reference and material

Google colab:

You can run the code directly in here: Implementing modified LeNet-5 on handwritten digits recognition

Google drive:

You can download the source code and the dataset from: LeNet-5 code and MNIST dataset

Articles:

A Gentle Introduction To The Math Behind Neural Networks

A Comprehensive Guide to Convolutional Neural Network

LeNet-5 Architecture

Convolutional Neural Networks: A Comprehensive Guide

Understanding the Math Behind Deep Neural Networks

AlexNet Architecture: A Complete Guide

Convolutional Autoencoder

Implementing an Autoencoder Using Tensorflow

YouTube videos:

Convolutional Neural Network Explained (CNN visualized)

Convolutional Neural Network from Scratch | In Depth

CNN: Convolutional Neural Networks Explained - Computerphile

Why do Convolutional Neural Networks work so well?

Autoencoders Explained Easily

Research paper:

Yan Lecun., et al. (1998). Gradient-Based Learning Applied to Document Recognition