Python project Report

Hand Written Digit Recognizer

DEPARTMENT: Bachelor in Aerospace Engineering (SMME)

NAME: Rana Muhammad Abdullah

CMS ID: 468092

Note: All the screenshots of the code are in chronological order & present the entire project.

TABLE OF CONTENTS

INTRODUCTION

What is Machine Learning?

Machine learning is a subfield of artificial intelligence (AI) that focuses on developing algorithms and statistical models to enable computers to improve their performance on specific tasks through experience. At its core, machine learning involves feeding large amounts of data to a computer and using various algorithms to find patterns within that data. These patterns allow the machine to make predictions or decisions without the need for explicit programming, this project follows just the same principles.

Python plays a crucial role in this field due to its simplicity, versatility, and extensive ecosystem of libraries and frameworks. Its popularity in machine learning is largely due to the ease with which complex tasks can be handled using Python's clear syntax and powerful tools. Some key Python libraries used in machine learning include NumPy, for handling large arrays and matrices, and Pandas, which provides data structures and data analysis tools essential for data manipulation.

Additionally, TensorFlow and Keras are essential for developing and training deep learning models. TensorFlow offers a comprehensive ecosystem for building machine learning applications, while Keras serves as an easy-to-use API that runs on top of TensorFlow. For data visualization, Matplotlib and Seaborn are widely used to create static, animated, and interactive visualizations, helping in understanding the data and evaluating the performance of machine learning models. By leveraging these Python tools, users can streamline the process of data preprocessing, model building, and evaluation, making Python an indispensable tool in the realm of machine learning.

What is Handwritten Digit Recognition?

Handwritten Digit Recognition is a technological process that enables computers to accurately identify and interpret handwritten numbers from images or scanned documents, all of this with

the help of machine learning. This task is particularly challenging due to the wide variability in human handwriting, where digits can vary significantly in style, size, and shape. The process involves several critical steps: acquiring the image, preprocessing it to enhance quality, segmenting the image into individual digits, extracting unique features, classifying the digits using advanced machine learning algorithms like Convolutional Neural Networks (CNNs), and finally, post-processing the results for accuracy. These steps ensure that each digit is recognized independently and accurately. Handwritten Digit Recognition has important applications in fields such as postal sorting, bank check processing, and digitizing handwritten data, making it a vital technology for automating and streamlining many everyday tasks.

Convolutional Neural Networks (CNNs)

A Convolutional Neural Network (CNN) is a specialized type of neural network primarily used for tasks involving image processing, computer vision, and pattern recognition. These networks automatically and adaptively learn spatial hierarchies of features from input images.

A CNN typically begins with an **input layer** that takes in the image data. This is followed by **convolutional layers**, which apply filters to the input image to create feature maps. Each filter detects different features, such as edges or textures. These convolutional layers are usually followed by an **activation function** like ReLU, which introduces non-linearity by turning negative values to zero.

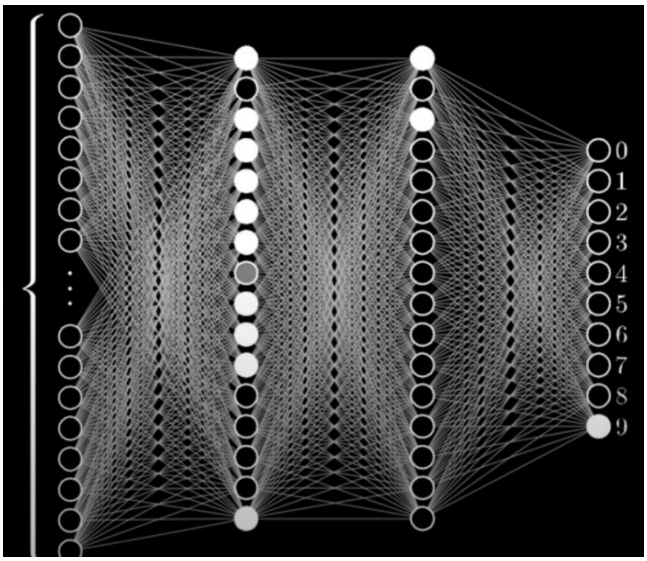


Figure 1: Illustration of CNN's layers

Next, a **pooling layer** reduces the spatial dimensions of the feature maps while retaining the depth, often using MaxPooling to take the maximum value in each pooling window. After the convolutional and pooling layers, the **flatten layer** converts the multi-dimensional feature maps into a one-dimensional vector, essential for transitioning to fully connected (dense) layers.

In a CNN, **dense layers** combine the features learned by the convolutional layers to make predictions or classifications. The final dense layer is often followed by a **softmax activation function** to produce a probability distribution over the possible classes.

Key Benefits:

- **Automatic Feature Learning**: CNNs can automatically detect important features in the input data.
- **Translation Invariance**: Pooling layers help CNNs recognize patterns regardless of their position in the image.
- **Reduced Parameters**: Weight sharing through convolutional layers makes CNNs more efficient and less prone to overfitting.

In summary, CNNs are powerful tools for image recognition and classification tasks, enabling the automatic and efficient extraction of complex features from input data. They are a cornerstone of modern computer vision applications, proving highly effective in various tasks, from digit classification to object detection and face recognition.

Use of External Frameworks and API's

TensorFlow

TensorFlow is an open-source library developed by Google for numerical computation and machine learning. It provides a robust framework for building and deploying machine learning models. The name TensorFlow comes from the way it handles data as multi-dimensional arrays, known as tensors, which flow through a series of operations as they are processed by the network.

TensorFlow is designed to be flexible and scalable, capable of running on various platforms, including CPUs, GPUs, and distributed systems. It supports a wide range of machine learning and deep learning algorithms, making it suitable for a variety of tasks such as image and speech recognition, natural language processing, and more. With TensorFlow, you can create complex

neural network architectures by stacking layers of computation, and it offers both low-level APIs for detailed customization and high-level APIs for quick prototyping.

Keras

Keras is a high-level neural networks API that runs on top of TensorFlow (as well as other frameworks like Theano and CNTK). It was developed with a focus on enabling fast experimentation and ease of use. Keras simplifies the process of building deep learning models by providing a user-friendly interface that abstracts much of the complexity involved in model creation and training.

With Keras, you can easily define neural networks by stacking layers and specifying their parameters. It supports all the commonly used layers such as Dense, Convolutional, and Recurrent layers, and provides various activation functions, loss functions, and optimizers. Keras is particularly appreciated for its simplicity and readability, allowing you to implement sophisticated models with minimal code. For instance, a deep learning model can be built in just a few lines of code, making it accessible to both newcomers and experienced practitioners.

Why integrated with Python

Python is the preferred language for TensorFlow and Keras due to its simplicity and extensive support for scientific computing. Combining TensorFlow's computational power with Keras's intuitive interface allows for efficient development, training, and deployment of machine learning models. This integration leverages Python's rich ecosystem of libraries, ensuring a seamless workflow from data preprocessing to model evaluation.

PROJECT EXPLANATION

Importing the libraries and loading dataset

The dataset used in this study is the MNIST (Modified National Institute of Standards and Technology) dataset, a well-known benchmark dataset for handwritten digit classification tasks. It consists of 60,000 training images and 10,000 testing images, with each image representing a handwritten digit ranging from 0 to 9.

Data Loading

The following code snippet demonstrates the process of importing necessary libraries, loading the MNIST dataset, and printing the shapes of the training and testing data:

```
[13]: import keras
      from keras.datasets import mnist
      from keras.models import Sequential
      from keras.layers import Dense, Dropout, Flatten
      from keras.layers import Conv2D, MaxPooling2D
      from keras import backend as K

      # the data, split between train and test sets
      (x_train, y_train), (x_test, y_test) = mnist.load_data()

      print(x_train.shape, y_train.shape)

      (60000, 28, 28) (60000,)
```

Dataset Structure

- **Training Data (**x_train**,** y_train**)**: This dataset is used to train the machine learning model. It comprises 60,000 images of handwritten digits (x_train) and corresponding labels (y_train).
- **Testing Data (**x_test**,** y_test**)**: This dataset evaluates the performance of the trained model. It contains 10,000 images (x_test) and their labels (y_test).

Data Dimensions

- **Input Images (**x_train **and** x_test**)**: Each image in the dataset is a 28x28 pixel grayscale image, represented as a 2D array of pixel intensities.
- **Labels (**y_train **and** y_test**)**: Each label is an integer value between 0 and 9, corresponding to the digit shown in the image.

The printed values of x_train and y_train are depicted in the output shown in the code snippet above.

Data Preprocessing

Reshaping Data:

The following lines of code reshape the training and testing data to include a channel dimension:

```
•[15]: x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
       x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
       input_shape = (28, 28, 1)
```

This reshaping ensures that each image is represented as a 28x28x1 array, which is necessary for the Convolutional Neural Network (CNN) layers to process the data correctly. The additional dimension (1) corresponds to the grayscale channel.

Defining classes and converting to binary class matrices:

```
num_classes = 10
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

This defines the number of output classes, which, in the case of the MNIST dataset, corresponds to the digits 0 through 9. One-hot encoding transforms the integer labels into binary matrices, making them suitable for use with categorical cross-entropy loss in the neural network. Each

label is represented by a binary vector where only the index corresponding to the label is 1, and all other indices are 0.

Data type conversion and data normalizing:

```
x_train = x_train.astype('float32') # helps when dealing with sigmoid or weighted sum
x_test = x_test.astype('float32')
#normalizing data, converting into [0,1] range
x_train /= 255
x_test /= 255
```

Converting the data type to float32 ensures compatibility with the neural network operations, which often require floating-point precision for computations, especially when dealing with activation functions like sigmoid or operations like weighted sum. Normalizing the pixel values to the range [0, 1] improves the training process by ensuring that the input data is on a consistent scale. This normalization is achieved by dividing each pixel value by 255, the maximum pixel value for an 8-bit image.

Output data shape:

Finally, the following lines print the shapes of the training data:

```
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

These print statements confirm the dimensions of the reshaped and normalized data, providing a quick verification step. The output indicates the number of samples and the shape of each sample, ensuring the data is correctly formatted for training the neural network.

Model Architecture

Importing necessary librearies, defining batch size and epochs for the model:

```
•[16]: from keras.models import Sequential
       from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense, BatchNormalization

       batch_size = 128
       epochs = 3

       model = Sequential()
```

This defines a linear stack of layers, where each layer has exactly one input and one output tensor.

First Block: Convolution, Batch Normalization, Activation, Pooling, Dropout:

```
# First block: Convolution, Batch Normalization, Activation, Pooling
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())#Stabilizes mean and variance, for speedy running
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())# Also to standardize so that the affects of weights of 1 layer do not relay to the coming layer
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.3))# this reduces chances of overfitting
```

- **Conv2D Layers**: Two convolutional layers with 64 filters each and a 3x3 kernel size to detect features from the input images.
- **Batch Normalization**: Applied to stabilize and accelerate training by normalizing activations.
- **MaxPooling2D**: Reduces the spatial dimensions by taking the maximum value in each pooling window, making the feature maps smaller and more manageable.
- **Dropout**: Randomly drops 30% of the neurons to prevent overfitting.

Second Block: Increased Complexity:

```
# Second block
model.add(Conv2D(128, (3, 3), activation='relu'))#128 to detect complex features
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.4))# greater drop out since the model has grown complex
```

- **Conv2D Layers**: Two convolutional layers with 128 filters each for detecting more complex features.
- **Batch Normalization**: Continues to stabilize and normalize activations.
- **MaxPooling2D**: Further reduces spatial dimensions.

- **Dropout**: Increases to 40% to manage the complexity and again to prevent overfitting.

Third Block: Further Feature Extraction:

```
# Third block
model.add(Conv2D(256, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))
```

- **Conv2D Layer**: A single convolutional layer with 256 filters to capture even more detailed features.
- **BatchNormalization**: Keeps the mean and variance stable.
- **MaxPooling2D**: Reduces the dimensions further.
- **Dropout**: Increased to 50% due to the model's increased complexity.

Flatten and Dense Layers: Transition to Fully Connected Layers

- **Flatten**: Converts the 3D feature maps into a 1D vector to be fed into fully connected layers.
- **Dense Layers**: Two fully connected layers with 512 and 256 neurons respectively, each followed by a dropout layer to prevent overfitting.
- **Output Layer**: A Dense layer with num_classes neurons and softmax activation, producing a probability distribution over the possible classes.

Model compilation:

```
# Compile the model
model.compile(loss='categorical_crossentropy',
              optimizer='adam',  # Using Adam for better performance
              metrics=['accuracy'])
```

- **Loss Function**: categorical_crossentropy is used for multi-class classification.
- **Optimizer**: adam, chosen for its efficiency and adaptive learning rate capabilities.
- **Metrics**: accuracy to evaluate the model's performance during training and testing.

This model architecture is designed to efficiently extract features from input images, stabilize training with batch normalization, prevent overfitting with dropout, and ultimately classify the images into their respective categories.

Train the Model

```
[17]: hist = model.fit(x_train, y_train,batch_size=batch_size,epochs=epochs,verbose=1,validation_data=(x_test, y_test))
      print("The model has successfully trained")

      model.save('mnist.h5')
      print("Saving the model as mnist.h5")
```

- **fit() Method**: This method trains the model on the training data (x_train, y_train).
- **batch_size**: Specifies the number of samples per gradient update. Here, it is set to 128.
- **epochs**: Defines the number of iterations over the entire training dataset. In this case, the model is trained for 3 epochs.
- **Verbose=1**: This parameter controls the verbosity of the output during training. Setting it to 1 provides progress updates.
- **validation_data**: This argument specifies the validation data (x_test, y_test). The model evaluates its performance on this data at the end of each epoch.

The model is trained using the training data, and its performance is validated on the testing data after each epoch. The training progress and metrics are displayed due to verbose=1.

- **save() Method**: This method saves the entire model, including the architecture, weights, and optimizer state, to a file.
- **'mnist.h5'**: The filename under which the model is saved. The .h5 extension indicates the HDF5 format, which is commonly used for saving Keras models.

By saving the model, you ensure that the trained model can be loaded and used later without retraining.

Evaluate the model

```
[17]: hist = model.fit(x_train, y_train,batch_size=batch_size,epochs=epochs,verbose=1,validation_data=(x_test, y_test))
      print("The model has successfully trained")

      model.save('mnist.h5')
      print("Saving the model as mnist.h5")

      Epoch 1/3
      469/469 ━━━━━━━━━━━━━━━━━━━━ 205s 402ms/step - accuracy: 0.6575 - loss: 1.1327 - val_accuracy: 0.4883 - val_loss: 2.7328
      Epoch 2/3
      469/469 ━━━━━━━━━━━━━━━━━━━━ 185s 395ms/step - accuracy: 0.9687 - loss: 0.1114 - val_accuracy: 0.9896 - val_loss: 0.0393
      Epoch 3/3
      469/469 ━━━━━━━━━━━━━━━━━━━━ 187s 400ms/step - accuracy: 0.9789 - loss: 0.0799 - val_accuracy: 0.9908 - val_loss: 0.0335
```

- **evaluate() Method**: This method evaluates the performance of the trained model on the testing data (x_test, y_test).
- **verbose=0**: This parameter suppresses the output during evaluation. Setting it to 0 ensures no output is printed during the evaluation process.
- **score**: The method returns a list containing the test loss and test accuracy.

The model's performance is assessed on the test data, providing two key metrics:

- **Test loss**: Indicates how well the model performs in terms of the chosen loss function (categorical_crossentropy).
- **Test accuracy**: Reflects the proportion of correctly classified instances in the test dataset.

Remarkably, even after running only 3 epochs, the model achieved an accuracy of more than 99% on the test data. This high accuracy indicates the model's excellent performance in classifying handwritten digits from the MNIST dataset, demonstrating the effectiveness of the chosen architecture and training process.

Create GUI to predict digits

```
from keras.models import load_model
from tkinter import *
import tkinter as tk
import win32gui
from PIL import ImageGrab, Image
import numpy as np

model = load_model('mnist.h5')

def predict_digit(img):
    #resize image to 28x28 pixels
    img = img.resize((28,28))
    #convert rgb to grayscale
    img = img.convert('L')
    img = np.array(img)
    #reshaping to support our model input and normalizing
    img = img.reshape(1,28,28,1)
    img = img/255.0
    #predicting the class
    res = model.predict([img])[0]
    return np.argmax(res), max(res)

class App(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)

        self.x = self.y = 0
```

```
        # Creating elements
        self.canvas = tk.Canvas(self, width=300, height=300, bg = "white", cursor="cross")
        self.label = tk.Label(self, text="Thinking..", font=("Helvetica", 48))
        self.classify_btn = tk.Button(self, text = "Recognise", command =         self.classify_handwriting)
        self.button_clear = tk.Button(self, text = "Clear", command = self.clear_all)

        # Grid structure
        self.canvas.grid(row=0, column=0, pady=2, sticky=W, )
        self.label.grid(row=0, column=1,pady=2, padx=2)
        self.classify_btn.grid(row=1, column=1, pady=2, padx=2)
        self.button_clear.grid(row=1, column=0, pady=2)

        #self.canvas.bind("<Motion>", self.start_pos)
        self.canvas.bind("<B1-Motion>", self.draw_lines)

    def clear_all(self):
        self.canvas.delete("all")

    def classify_handwriting(self):
        HWND = self.canvas.winfo_id() # get the handle of the canvas
        rect = win32gui.GetWindowRect(HWND) # get the coordinate of the canvas
        im = ImageGrab.grab(rect)

        digit, acc = predict_digit(im)
        self.label.configure(text= str(digit)+', '+ str(int(acc*100))+'%')

    def draw_lines(self, event):
        self.x = event.x
        self.y = event.y
        r=8
        self.canvas.create_oval(self.x-r, self.y-r, self.x + r, self.y + r, fill='black')

app = App()
mainloop()
```

This code creates a graphical user interface (GUI) for handwriting digit recognition using a trained Convolutional Neural Network (CNN) model. The primary components and their functions are as follows:

Imports and Model Loading**:**

- The necessary libraries (Keras, Tkinter, PIL, Numpy, and win32gui) are imported.
- The trained MNIST model (mnist.h5) is loaded using Keras.

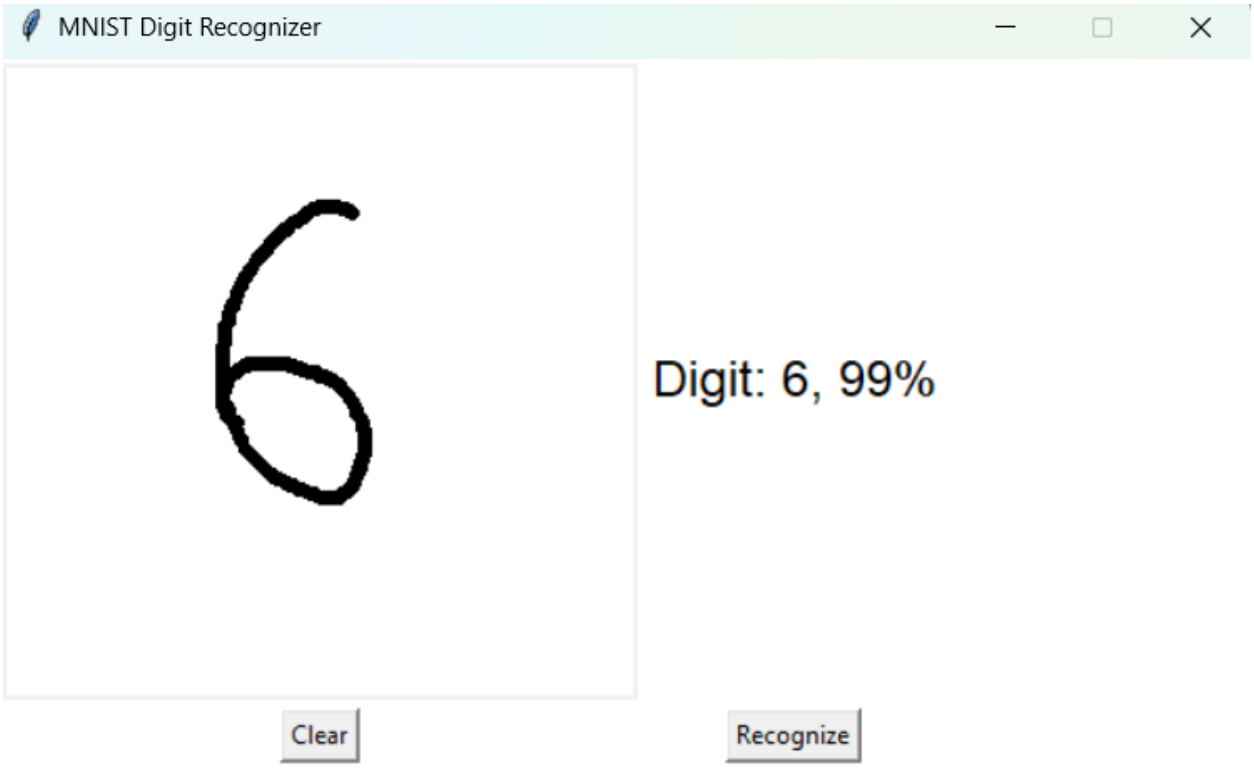
Prediction Function:

predict_digit(img): This function resizes the input image to 28x28 pixels, converts it to grayscale, normalizes it, and then predicts the digit using the loaded model. It returns the predicted digit and its confidence.

- **App Class**:
- Inherits from tk.Tk to create the main application window.
- **Canvas**: A drawable area where users can draw digits using the mouse.
- **Label**: Displays the predicted digit and confidence.
- **Buttons**:
    - "Recognise" button triggers the digit classification.
    - "Clear" button clears the canvas.
- **Draw Function**: Captures the mouse movement to draw lines on the canvas.
- **Classify Handwriting**:
- Captures the current drawing from the canvas, converts it to an image, and uses the predict_digit function to classify the drawn digit. The result is displayed on the label.

This GUI provides an interactive way for users to draw digits and have them recognized by the trained CNN model. It combines image processing with a user-friendly interface for digit classification tasks.

GUI's Output



Problems faced in building the GUI

During the development of the MNIST Digit Recognition project, several challenges emerged, requiring a combination of problem-solving and iterative refinement. Initially, issues with the graphical user interface (GUI) functionality were apparent, such as incorrect alignment of elements and difficulty ensuring a seamless user experience for drawing digits. Debugging these aspects required testing and modifying the canvas's input behavior, particularly the brush size and drawing sensitivity, to allow for natural and accurate user interaction.

Another significant hurdle was related to the preprocessing of input data. The raw images captured from the canvas often contained excessive whitespace or poorly centered digits, leading to inconsistent predictions. This necessitated implementing a cropping and centering mechanism to align the input more closely with the MNIST dataset's format, which was critical to improving model accuracy. Additionally, misclassification of certain digits, such as reading 9 as 6, highlighted the importance of refining preprocessing steps like resizing, grayscale conversion, and pixel value normalization.

On the model prediction side, the need for a confidence threshold became evident. Without a clearly defined threshold, the system occasionally provided uncertain or inaccurate predictions, confusing the user. Adjusting the confidence threshold and improving the feedback loop addressed this issue, ensuring predictions were reliable. These challenges, while initially frustrating, proved invaluable in building a more robust and user-friendly application.

CONCLUSION

A Convolutional Neural Network (CNN) is a specialized type of neural network primarily used for tasks involving image processing, computer vision, and pattern recognition. These networks automatically and adaptively learn spatial hierarchies of features from input images.

A CNN typically begins with an **input layer** that takes in the image data. This is followed by **convolutional layers**, which apply filters to the input image to create feature maps. Each filter detects different features, such as edges or textures. These convolutional layers are usually followed by an **activation function** like ReLU, which introduces non-linearity by turning negative values to zero.

Next, a **pooling layer** reduces the spatial dimensions of the feature maps while retaining the depth, often using MaxPooling to take the maximum value in each pooling window. After the convolutional and pooling layers, the **flatten layer** converts the multi-dimensional feature maps into a one-dimensional vector, essential for transitioning to fully connected (dense) layers.

In a CNN, **dense layers** combine the features learned by the convolutional layers to make predictions or classifications. The final dense layer is often followed by a **softmax activation function** to produce a probability distribution over the possible classes.

Key Benefits:

- **Automatic Feature Learning**: CNNs can automatically detect important features in the input data.

- **Translation Invariance**: Pooling layers help CNNs recognize patterns regardless of their position in the image.
- **Reduced Parameters**: Weight sharing through convolutional layers makes CNNs more efficient and less prone to overfitting.

In summary, CNNs are powerful tools for image recognition and classification tasks, enabling the automatic and efficient extraction of complex features from input data. They are a cornerstone of modern computer vision applications, proving highly effective in various tasks, from digit classification to object detection and face recognition.