

# **SWE-532: Deep Learning Lab Practical File**

*Submitted towards the partial fulfilment  
of the requirements of the award of the degree of*

## **Master of Technology In Software Engineering**

**Submitted To:-**

**Dr. Sonika Dahiya**  
Assistant Professor  
Department of Software Engineering

**Submitted By:-**

Aman Chauhan (24/SWE/08)  
II SEM, I YEAR



**Delhi Technological University**  
(FORMERLY Delhi College of Engineering)  
SHAHBAD DAULATPUR, BAWANA ROAD, DELHI – 110042

April, 2025

# INDEX

S.No.	Experiment Name	Page No.	Date	Signature
1	Implementation of ANN model on MNIST dataset without hidden layer.			
2	Implementation of ANN model on MNIST dataset with hidden layer.			
3	Implementation of CNN model on MNIST dataset			
4	Read about the CIFAR-10 dataset, implementing a simple CNN with 2 hidden layers. Show the accuracy using confusion matrix.			
5	Write a program to implement k-Nearest Neighbour algorithm to classify any dataset of your choice. Use cross validation and following metrics for performance evaluation: a. Accuracy b. Misclassification Rate c. Sensitivity d. Specificity e. Precision f. F-Score			
6	Write a python code for the following 5 activation functions: a) Step b. Sigmoid c. tanh d. ReLU (rectified linear unit) e. Leaky ReLU			
7	Implement a sliding window approach to classify regions of an image using a pre-trained CNN on CIFAR-10 dataset.			
8	Use Faster R-CNN with pre-trained weights for object detection on simple datasets like Pascal VOC.			
9	To perform fast and accurate object detection using the YOLOv5 model. Run inference on sample images and videos.			
10	To extract named entities from text using a pre-trained BERT model fine-tuned for NER.			

# Experiment - 1

**Aim:** Implementation of ANN model on MNIST dataset without hidden layer.

## **Introduction:**

### ***ANN Model***

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of the human brain. Just like our brains learn from experience, ANNs can be trained on data to recognize patterns, make predictions, and perform various tasks.

### **Neural Network Architecture:**

At the core of ANN lies its architecture, which comprises layers of interconnected nodes, also known as neurons. These layers are categorized into three types: input layer, hidden layers, and output layer. Information is processed through these layers via weighted connections, where each connection is assigned a weight representing its importance in the learning process.

### **Neuron Functionality:**

Individual neurons in the network perform a mathematical operation on the weighted sum of their inputs, followed by the application of an activation function. This non-linear activation function introduces complexity and allows the network to capture intricate patterns and relationships within the data.

### **Training Process:**

Training an ANN involves presenting it with a labeled dataset, allowing the network to learn the underlying patterns and relationships. The learning process is facilitated through an optimization algorithm, often stochastic gradient descent, which adjusts the weights to minimize the difference between the predicted and actual outputs. This iterative process, known as backpropagation, is the backbone of ANN learning.

## **Dataset Description:**

### **MNIST: The Handwritten Digit Dataset**

The MNIST database (Modified National Institute of Standards and Technology) is a widely used benchmark dataset for image classification and machine learning tasks.

#### **Key characteristics:**

**Content:** Contains handwritten digits (0-9) from various individuals.

#### **Size:**

Training set:

60,000 images

Testing set: 10,000

images

**Format:** Each image is a 28x28 pixel grayscale image, representing a single handwritten digit.

**Normalization:** Images are centred and size-normalized for consistency.

#### **Popularity:**

**Simplicity:** Easy to understand and work with, making it ideal for beginners in machine learning and image processing.

**Freely available:** Accessible for anyone to download and use for research or educational purposes.

**Baseline performance:** Commonly used as a baseline to compare the performance of new algorithms and models in image classification tasks.

## **Code:**

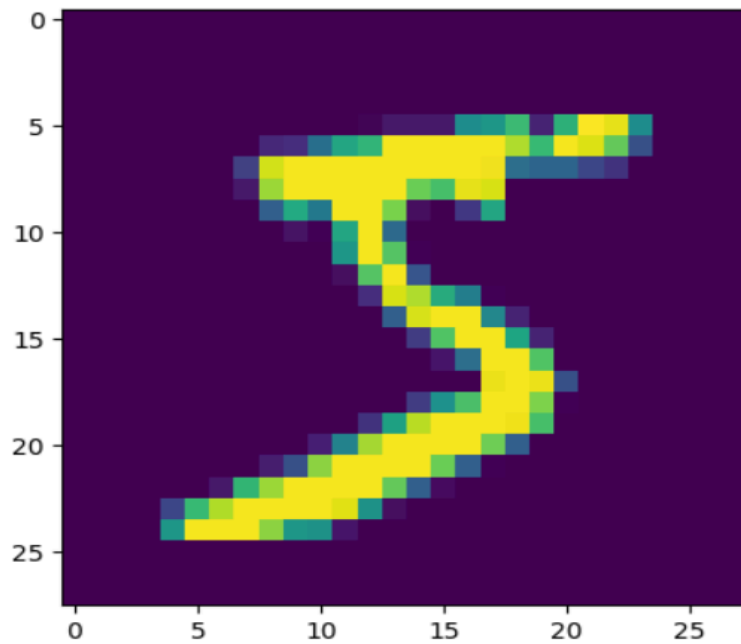
```
✓ [1] #importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
✓ [2] #loading MNIST dataset
6s from tensorflow.keras.datasets import mnist
(X_train,y_train) , (X_test,y_test)=mnist.load_data()
```

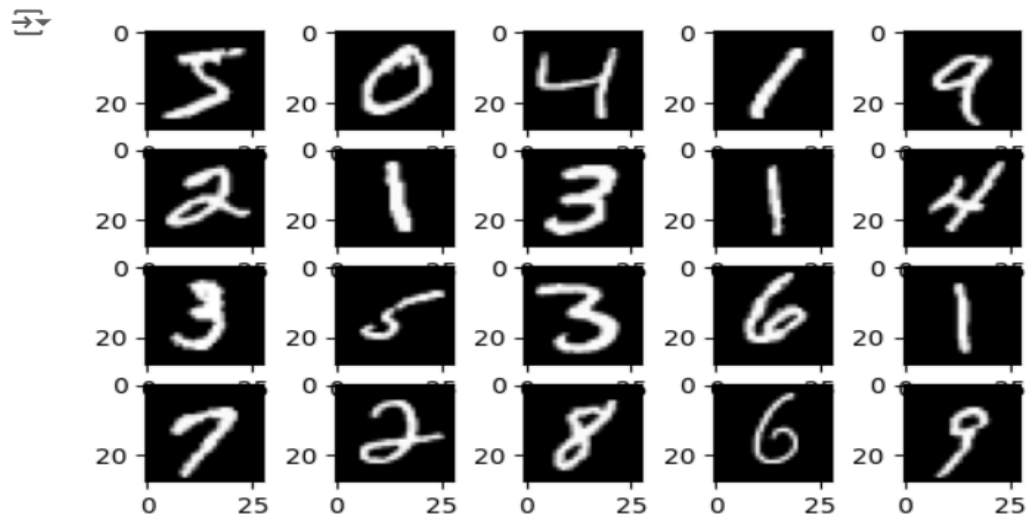
⇨ Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>  
11490434/11490434 [=====] - 0s 0us/step

```
✓ [3] #visualizing the image in train data
0s plt.imshow(X_train[0])
```

⇨ <matplotlib.image.AxesImage at 0x7bcde2c0b7c0>




```
[4] #visualizing the first 20 images in the dataset
for i in range(20):
    #subplot
    plt.subplot(5, 5, i+1)
    # plotting pixel data
    plt.imshow(X_train[i], cmap=plt.get_cmap('gray'))
```



✓  
0s [6] #Checking the shape of train and test data  
print(X\_train.shape)  
print(X\_test.shape)

⇒ (60000, 28, 28)  
(10000, 28, 28)

✓  
0s [7] # the image is in pixels which ranges from 0 to 255  
X\_train[0]

⇒ ndarray (28, 28) show data  


[8] #flattening the images  
X\_train\_flat=X\_train.reshape(len(X\_train),28\*28)  
X\_test\_flat=X\_test.reshape(len(X\_test),28\*28)  
  
#checking the shape after flattening  
print(X\_train\_flat.shape)  
print(X\_test\_flat.shape)

⇒ (60000, 784)  
(10000, 784)

✓  
0s [9] #checking the representation of image after flattening  
X\_train\_flat[0]

⇒ array([ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 126, 136, 175, 26, 166, 255, 247, 127, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 30, 36, 94, 154, 170, 253,  
 253 253 253 253 225 172 253 212 195 61 0 0 0])

```
[10] #we will normalize pixel values in such a way that it ranges from 0-1
      #normalizing the pixel values
      X_train_flat=X_train_flat/255
      X_test_flat=X_test_flat/255

      #print this code to check the pixel values after normalization
      X_train_flat[0]
```

[illegible]

- Building a simple ANN model without hidden layer

```
[11] #importing necessary libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
[12] #Step 1 : Defining the model
model=Sequential()
model.add(Dense(10,input_shape=(784,),activation='softmax'))
```

```
[13] #Step 2: Compiling the model
      model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
```



✓ [14] #Step 3: Fitting the model  
model.fit(X\_train\_flat,y\_train,epochs=10)

⇒ Epoch 1/10  
1875/1875 [=====] - 5s 2ms/step - loss: 0.4682 - accuracy: 0.8784  
Epoch 2/10  
1875/1875 [=====] - 3s 2ms/step - loss: 0.3038 - accuracy: 0.9151  
Epoch 3/10  
1875/1875 [=====] - 3s 2ms/step - loss: 0.2836 - accuracy: 0.9204  
Epoch 4/10  
1875/1875 [=====] - 3s 2ms/step - loss: 0.2729 - accuracy: 0.9235  
Epoch 5/10  
1875/1875 [=====] - 5s 3ms/step - loss: 0.2670 - accuracy: 0.9263  
Epoch 6/10  
1875/1875 [=====] - 3s 2ms/step - loss: 0.2616 - accuracy: 0.9277  
Epoch 7/10  
1875/1875 [=====] - 3s 2ms/step - loss: 0.2582 - accuracy: 0.9293  
Epoch 8/10  
1875/1875 [=====] - 4s 2ms/step - loss: 0.2557 - accuracy: 0.9292  
Epoch 9/10  
1875/1875 [=====] - 4s 2ms/step - loss: 0.2530 - accuracy: 0.9295  
Epoch 10/10  
1875/1875 [=====] - 3s 2ms/step - loss: 0.2507 - accuracy: 0.9309  
<keras.src.callbacks.History at 0x7bcd97c2a40>

✓ [15] #Step 4: Evaluating the model  
model.evaluate(X\_test\_flat,y\_test)

⇒ 313/313 [=====] - 1s 2ms/step - loss: 0.2678 - accuracy: 0.9273  
[0.267774373292923, 0.927299976348877]


✓  
3s [20] #Step 5 :Making predictions  
y\_predict = model.predict(X\_test\_flat)  
y\_predict[3] #printing the 3rd index

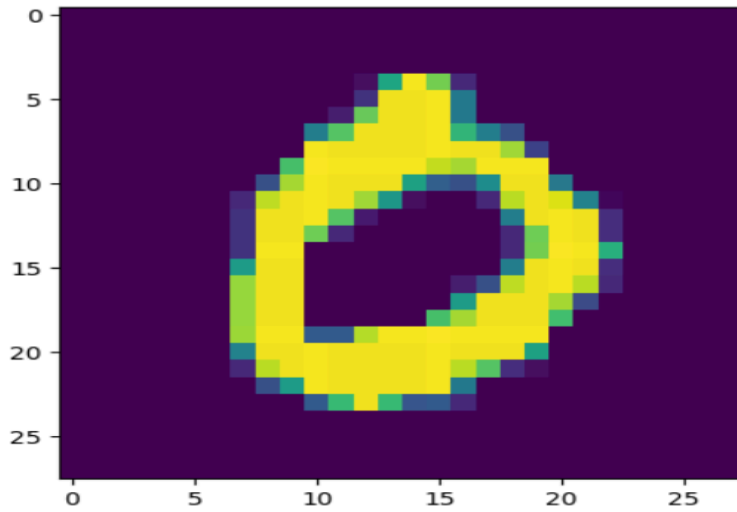
⇒ 313/313 [=====] - 1s 3ms/step  
array([9.9990350e-01, 4.2141594e-12, 2.3381870e-05, 1.2660194e-06,  
1.6458474e-08, 2.0254107e-05, 3.8249091e-05, 1.9950796e-06,  
5.5995993e-06, 5.6966992e-06], dtype=float32)

✓  
0s [21] # Here we get the index of the maximum value in the above-encoded vector.  
np.argmax(y\_predict[3])

⇒ 0

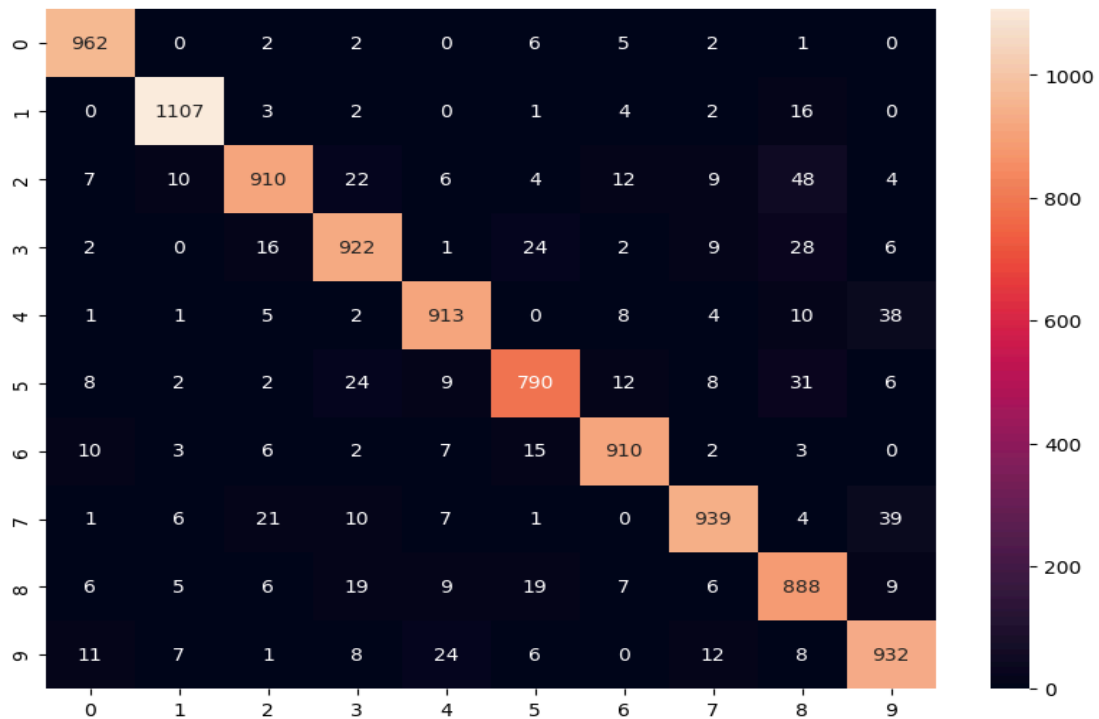
```
✓ 0s [22] #checking if the predicting is correct
      plt.imshow(X_test[3])
```

 <matplotlib.image.AxesImage at 0x7bcd1b17b20>



```
✓ 2s [23] y_predict_labels=np.argmax(y_predict,axis=1)
      #Confusion matrix
      from sklearn.metrics import confusion_matrix
      matrix=confusion_matrix(y_test,y_predict_labels)
      #visualizaing confusion matrix with heatmap
      plt.figure(figsize=(10,7))
      sns.heatmap(matrix,annot=True,fmt='d')
```

 <Axes: >



### **Learning:**

- **Baseline performance:** Despite its limitations of reduced learning capacity and susceptibility to overfitting, achieving 92.7% accuracy highlights the potential of even simple ANN architectures for specific tasks like MNIST digit recognition.
- **Importance of hidden layers:** The comparison to more complex models with hidden layers emphasizes the crucial role of hidden layers in capturing non-linear relationships and improving model performance.

# Experiment - 2

**Aim: Implementation of ANN model on MNIST dataset with hidden layer.**

## **Introduction:**

### ***ANN Model***

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of the human brain. Just like our brains learn from experience, ANNs can be trained on data to recognize patterns, make predictions, and perform various tasks.

### **Neural Network Architecture:**

At the core of ANN lies its architecture, which comprises layers of interconnected nodes, also known as neurons. These layers are categorized into three types: input layer, hidden layers, and output layer. Information is processed through these layers via weighted connections, where each connection is assigned a weight representing its importance in the learning process.

### **Neuron Functionality:**

Individual neurons in the network perform a mathematical operation on the weighted sum of their inputs, followed by the application of an activation function. This non-linear activation function introduces complexity and allows the network to capture intricate patterns and relationships within the data.

### **Training Process:**

Training an ANN involves presenting it with a labeled dataset, allowing the network to learn the underlying patterns and relationships. The learning process is facilitated through an optimization algorithm, often stochastic gradient descent, which adjusts the weights to minimize the difference between the predicted and actual outputs. This iterative process, known as backpropagation, is the backbone of ANN learning.

## **Dataset Description:**

### **MNIST: The Handwritten Digit Dataset**

The MNIST database (Modified National Institute of Standards and Technology) is a widely used benchmark dataset for image classification and machine learning tasks.

### **Key characteristics:**

**Content:** Contains handwritten digits (0-9) from various individuals.

### **Size:**

Training set: 60,000 images

Testing set: 10,000 images

**Format:** Each image is a 28x28 pixel grayscale image, representing a single handwritten digit.

**Normalization:** Images are centered and size-normalized for consistency.

### **Popularity:**

**Simplicity:** Easy to understand and work with, making it ideal for beginners in machine learning and image processing.

**Freely available:** Accessible for anyone to download and use for research or educational purposes.

**Baseline performance:** Commonly used as a baseline to compare the performance of new algorithms and models in image classification tasks.


## **Code:**

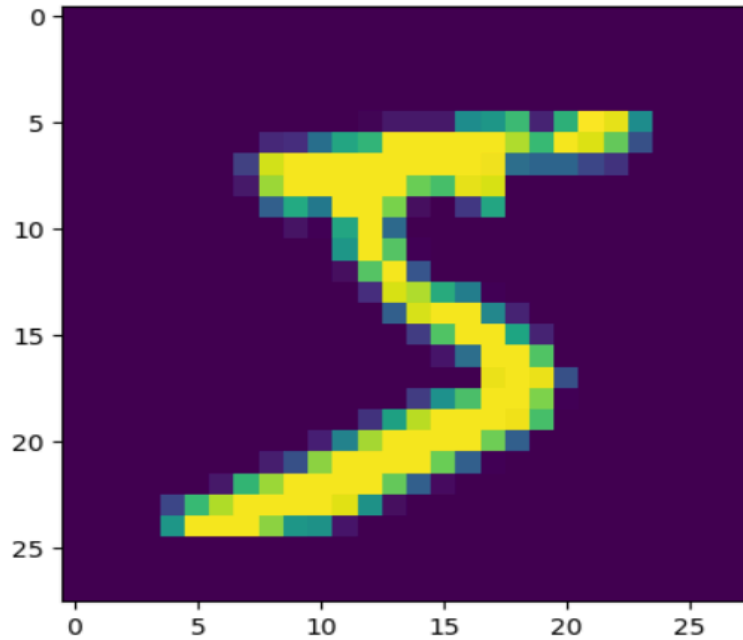
```
✓ [1] #importing libraries
      import numpy as np
      import pandas as pd
      import matplotlib.pyplot as plt
      import seaborn as sns
```

```
✓ [2] #loading MNIST dataset
      from tensorflow.keras.datasets import mnist
      (X_train,y_train) , (X_test,y_test)=mnist.load_data()
```

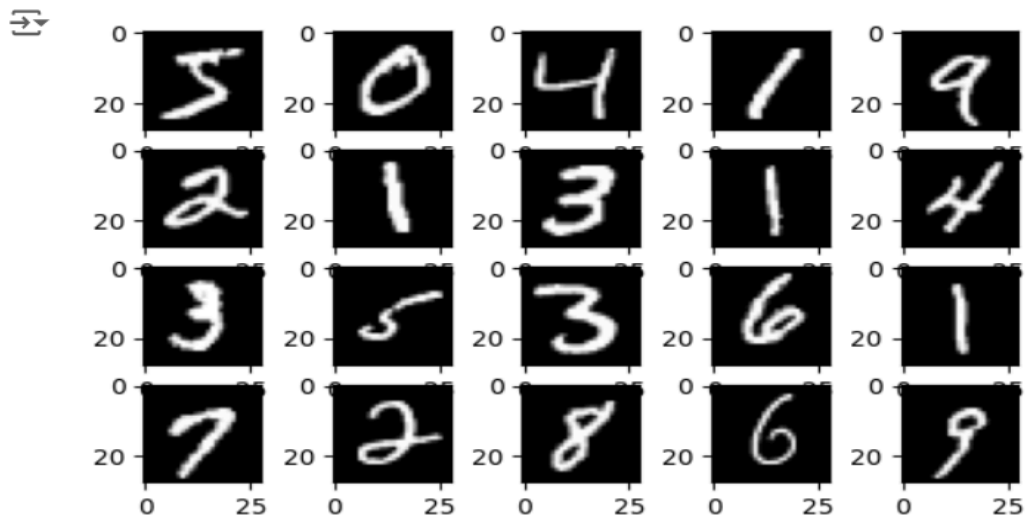
⏏ Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>  
11490434/11490434 [=====] - 0s 0us/step

```
✓ [3] #visualizing the image in train data  
0s plt.imshow(X_train[0])
```

 <matplotlib.image.AxesImage at 0x7bcde2c0b7c0>




```
[4] #visualizing the first 20 images in the dataset  
for i in range(20):  
    #subplot  
    plt.subplot(5, 5, i+1)  
    # plotting pixel data  
    plt.imshow(X_train[i], cmap=plt.get_cmap('gray'))
```



✓  
0s [6] #Checking the shape of train and test data  
print(X\_train.shape)  
print(X\_test.shape)

⇒ (60000, 28, 28)  
(10000, 28, 28)

✓  
0s [7] # the image is in pixels which ranges from 0 to 255  
X\_train[0]

⇒ ndarray (28, 28) show data  


[8] #flattening the images  
X\_train\_flat=X\_train.reshape(len(X\_train),28\*28)  
X\_test\_flat=X\_test.reshape(len(X\_test),28\*28)  
  
#checking the shape after flattening  
print(X\_train\_flat.shape)  
print(X\_test\_flat.shape)

⇒ (60000, 784)  
(10000, 784)

✓  
0s [9] #checking the representation of image after flattening  
X\_train\_flat[0]

⇒ array([ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 18, 18, 18,  
126, 136, 175, 26, 166, 255, 247, 127, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 30, 36, 94, 154, 170, 253,  
253 253 253 253 225 172 253 212 195 61 0 0 0





✓  
1m [26] #Step 3: Fitting the model  
model.fit(X\_train\_flat,y\_train,epochs=10)

⇒ Epoch 1/10  
1875/1875 [=====] - 9s 4ms/step - loss: 0.2587 - accuracy: 0.9226  
Epoch 2/10  
1875/1875 [=====] - 6s 3ms/step - loss: 0.1071 - accuracy: 0.9679  
Epoch 3/10  
1875/1875 [=====] - 8s 4ms/step - loss: 0.0786 - accuracy: 0.9755  
Epoch 4/10  
1875/1875 [=====] - 6s 3ms/step - loss: 0.0616 - accuracy: 0.9809  
Epoch 5/10  
1875/1875 [=====] - 8s 4ms/step - loss: 0.0520 - accuracy: 0.9834  
Epoch 6/10  
1875/1875 [=====] - 6s 3ms/step - loss: 0.0408 - accuracy: 0.9865  
Epoch 7/10  
1875/1875 [=====] - 8s 4ms/step - loss: 0.0366 - accuracy: 0.9880  
Epoch 8/10  
1875/1875 [=====] - 6s 3ms/step - loss: 0.0298 - accuracy: 0.9905  
Epoch 9/10  
1875/1875 [=====] - 8s 4ms/step - loss: 0.0265 - accuracy: 0.9913  
Epoch 10/10  
1875/1875 [=====] - 6s 3ms/step - loss: 0.0251 - accuracy: 0.9922  
<keras.src.callbacks.History at 0x785382fdc6d0>

✓  
1s [27] #Step 4: Evaluating the model  
model.evaluate(X\_test\_flat,y\_test)

⇒ 313/313 [=====] - 1s 3ms/step - loss: 0.1208 - accuracy: 0.9701  
[0.12083243578672409, 0.9700999855995178]


✓  
1s [28] #Step 5 :Making predictions  
y\_predict = model.predict(X\_test\_flat)  
y\_predict[3] #printing the 3rd index

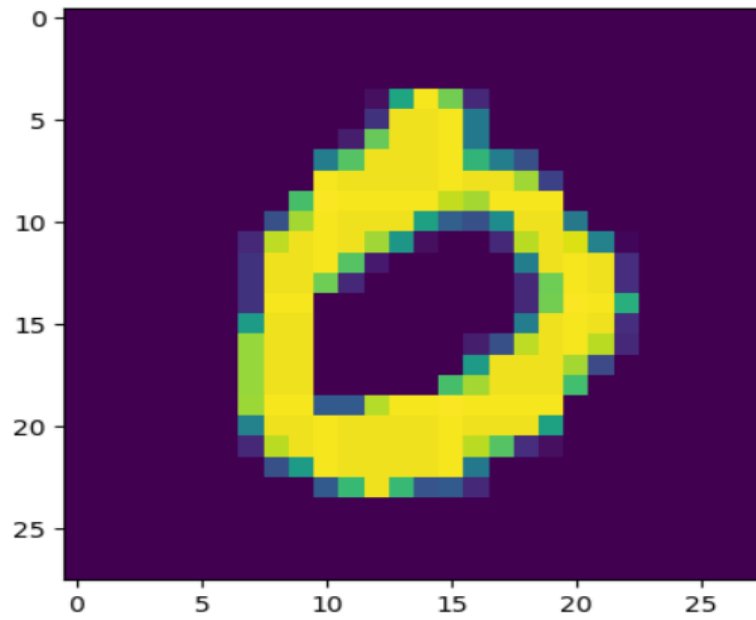
⇒ 313/313 [=====] - 1s 3ms/step  
array([9.9999994e-01, 1.3147561e-11, 8.5337121e-10, 2.4614503e-12,  
3.9837906e-09, 5.5561605e-10, 1.2005965e-09, 1.5006434e-10,  
2.2651516e-12, 2.5031555e-09], dtype=float32)

✓  
0s [29] # Here we get the index of the maximum value in the above-encoded vector.  
np.argmax(y\_predict[3])

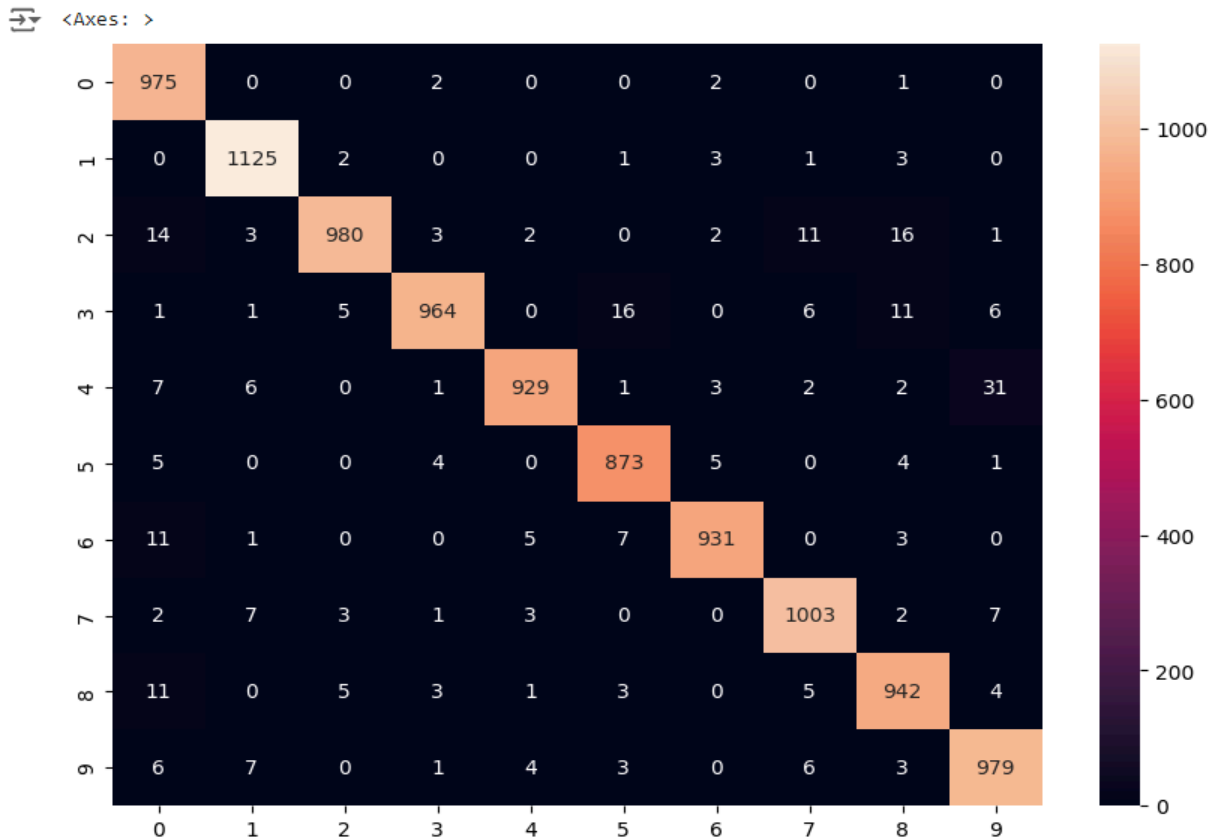
⇒ 0

```
✓ [30] #checking if the predicting is correct  
0s plt.imshow(X_test[3])
```

 <matplotlib.image.AxesImage at 0x785382da9240>



```
[31] y_predict_labels=np.argmax(y_predict,axis=1)
#Confusion matrix
from sklearn.metrics import confusion_matrix
matrix=confusion_matrix(y_test,y_predict_labels)
#visualizaing confusion matrix with heatmap
plt.figure(figsize=(10,7))
sns.heatmap(matrix,annot=True,fmt='d')
```



### Learning:

- ANNs can be effective for image classification tasks: This implementation demonstrates that a relatively simple ANN architecture with one hidden layer can achieve good accuracy on a well-known image classification benchmark (MNIST).
- Hyperparameter tuning is crucial: While a basic architecture can achieve decent results, significant accuracy improvements can be obtained by carefully tuning hyperparameters like the number of neurons in the hidden layer, learning rate, and activation function.

# Experiment - 3

**Aim:** Implementation of CNN model on MNIST dataset.

## **Introduction:**

### ***CNN Model***

Convolutional Neural Networks (CNNs) are a specialized type of deep learning neural network particularly well-suited for analyzing grid-like data, such as images and videos. They excel at extracting features and identifying patterns within these data types.

### **Key characteristics:**

**Convolutional layers:** These layers apply filters to the input data, identifying specific features like edges, shapes, and textures.

**Pooling layers:** These layers downsample the data, reducing its dimensionality and computational cost, while retaining important information.

**Fully connected layers:** Similar to traditional ANNs, these layers perform final classifications or predictions based on the extracted features.

## **Dataset Description:**

### **MNIST: The Handwritten Digit Dataset**

The MNIST database (Modified National Institute of Standards and Technology) is a widely used benchmark dataset for image classification and machine learning tasks.

### **Key characteristics:**

**Content:** Contains handwritten digits (0-9) from various individuals.

### **Size:**

Training set: 60,000 images

Testing set: 10,000 images

**Format:** Each image is a 28x28 pixel grayscale image, representing a single handwritten digit.

**Normalization:** Images are centered and size-normalized for consistency.

**Popularity:**

Simplicity: Easy to understand and work with, making it ideal for beginners in machine learning and image processing.

Freely available: Accessible for anyone to download and use for research or educational purposes.

Baseline performance: Commonly used as a baseline to compare the performance of new algorithms and models in image classification tasks.

**Code :**

```
In [1]: import tensorflow as tf
        from tensorflow.keras import layers, models
        from tensorflow.keras.datasets import mnist
        from tensorflow.keras.utils import to_categorical

In [2]: # Load and preprocess the MNIST dataset
        (train_images, train_labels), (test_images, test_labels) = mnist.load_data(path='mnist.npz')
        train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
        test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255
        train_labels = to_categorical(train_labels)
        test_labels = to_categorical(test_labels)

In [3]: # Build the CNN model
        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'))

In [4]: # Compile the model
        model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
        # Train the model
        model.fit(train_images, train_labels, epochs=5, batch_size=64, validation_data=(test_images, test_labels))

        # Evaluate the model on the test set
        test_loss, test_acc = model.evaluate(test_images, test_labels)
        print(f'Test accuracy: {test_acc}')
```

```

938/938 [=====] - 17s 17ms/step - loss: 0.1741 - accuracy: 0.9481 - val_loss: 0.0490 - val_accuracy:
0.9846
Epoch 2/5
938/938 [=====] - 21s 22ms/step - loss: 0.0469 - accuracy: 0.9858 - val_loss: 0.0369 - val_accuracy:
0.9885
Epoch 3/5
938/938 [=====] - 17s 18ms/step - loss: 0.0352 - accuracy: 0.9890 - val_loss: 0.0296 - val_accuracy:
0.9896
Epoch 4/5
938/938 [=====] - 18s 19ms/step - loss: 0.0277 - accuracy: 0.9913 - val_loss: 0.0339 - val_accuracy:
0.9886
Epoch 5/5
938/938 [=====] - 16s 17ms/step - loss: 0.0226 - accuracy: 0.9934 - val_loss: 0.0242 - val_accuracy:
0.9924
313/313 [=====] - 2s 7ms/step - loss: 0.0242 - accuracy: 0.9924
Test accuracy: 0.9923999905586243

```

## Learning:

- CNNs are effective for image classification: Achieving 99.2% accuracy on the MNIST dataset demonstrates the effectiveness of CNNs for image classification tasks, particularly for well-defined datasets like MNIST.
- Room for optimization: Even though 99.2% accuracy is good, there might be potential to improve accuracy by:
- Trying different hyperparameter combinations (learning rate, batch size, etc.)
- Experimenting with different model architectures (adding or removing layers, changing filter sizes, etc.)
- Using regularization techniques (dropout, weight decay) to prevent overfitting.

# Experiment - 4

**Aim:** Read about the CIFAR-10 dataset, implementing a simple CNN with 2 hidden layers. Show the accuracy using the confusion matrix.

## **Introduction:**

### ***CNN Model***

Convolutional Neural Networks (CNNs) are a specialized type of deep learning neural network particularly well-suited for analysing grid-like data, such as images and videos.

They excel at extracting features and identifying patterns within these data types.

### **Key characteristics:**

**Convolutional layers:** These layers apply filters to the input data, identifying specific features like edges, shapes, and textures.

**Pooling layers:** These layers down sample the data, reducing its dimensionality and computational cost, while retaining important information.

**Fully connected layers:** Similar to traditional ANNs, these layers perform final classifications or predictions based on the extracted features.

## **Dataset Description:**

**CIFAR-10:** A Classic Dataset for Image Recognition

CIFAR-10 (Canadian Institute For Advanced Research) is a widely used dataset of colour images commonly employed for training and evaluating machine learning and computer vision algorithms, particularly those focused on image classification.

### **Key characteristics:**

#### **Size:**

Total images: 60,000

Training set: 50,000 images

Testing set: 10,000 images

Classes: 10 mutually exclusive classes of everyday objects - airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck

Image format: 32x32 pixel RGB (red, green, blue) colour images

Labelling: Each image is labeled with one of the 10 object classes.

### Code :

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Load CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

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

# Define the class names
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

# Build the CNN model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
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))

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



```

▶ # Train the model
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))

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

# Make predictions
y_pred = model.predict(test_images)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = test_labels.flatten()

# Generate the confusion matrix
cm = confusion_matrix(y_true, y_pred_classes)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)

# Plot the confusion matrix
fig, ax = plt.subplots(figsize=(10, 10))
disp.plot(ax=ax)
plt.show()

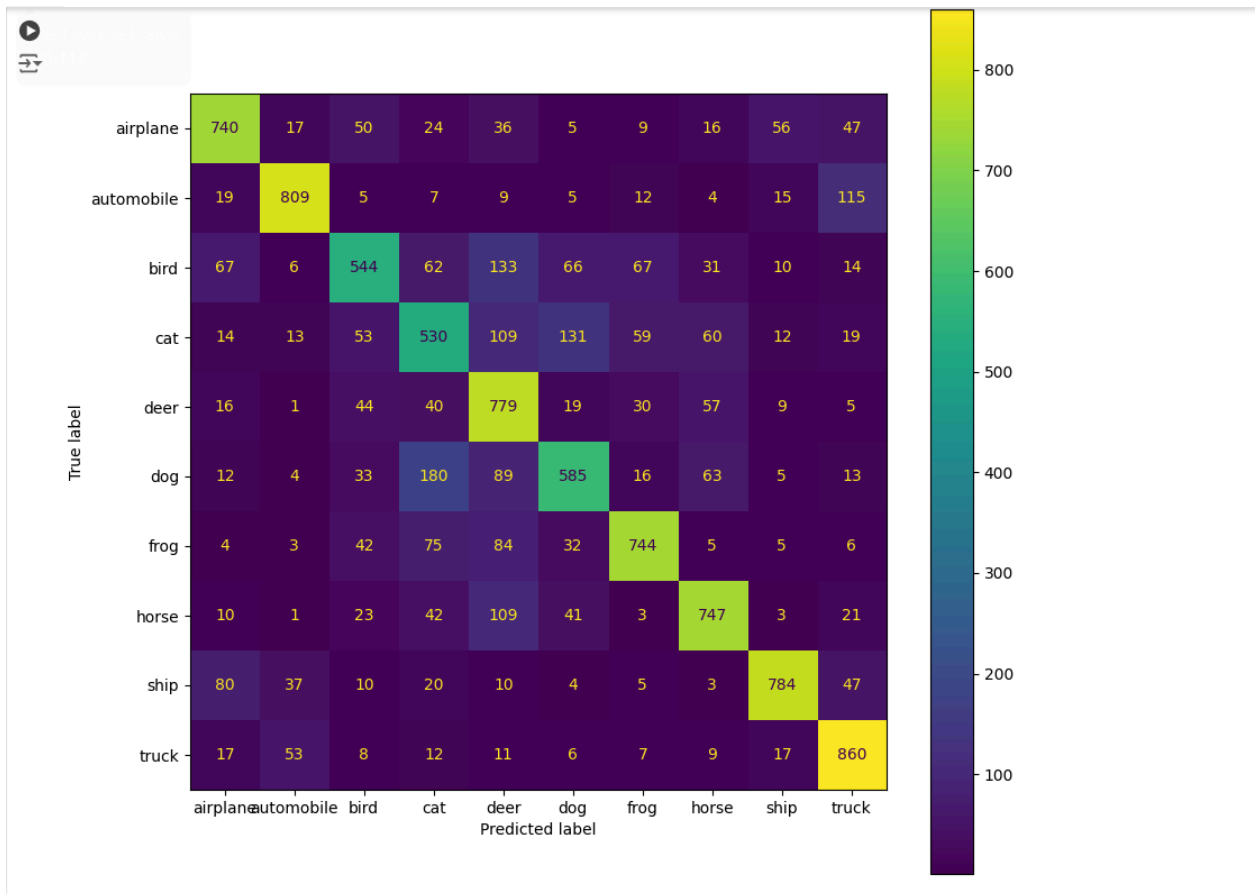
```

```

▶ Epoch 1/10
1563/1563 [=====] - 84s 53ms/step - loss: 1.5269 - accuracy: 0.4443 - val_loss: 1.2592 - val_accuracy: 0.5506
Epoch 2/10
1563/1563 [=====] - 73s 47ms/step - loss: 1.1786 - accuracy: 0.5826 - val_loss: 1.0947 - val_accuracy: 0.6160
Epoch 3/10
1563/1563 [=====] - 71s 46ms/step - loss: 1.0334 - accuracy: 0.6352 - val_loss: 1.0553 - val_accuracy: 0.6212
Epoch 4/10
1563/1563 [=====] - 69s 44ms/step - loss: 0.9504 - accuracy: 0.6652 - val_loss: 0.9676 - val_accuracy: 0.6621
Epoch 5/10
1563/1563 [=====] - 69s 44ms/step - loss: 0.8802 - accuracy: 0.6906 - val_loss: 0.9210 - val_accuracy: 0.6789
Epoch 6/10
1563/1563 [=====] - 70s 45ms/step - loss: 0.8222 - accuracy: 0.7120 - val_loss: 0.9023 - val_accuracy: 0.6888
Epoch 7/10
1563/1563 [=====] - 68s 44ms/step - loss: 0.7676 - accuracy: 0.7291 - val_loss: 0.9018 - val_accuracy: 0.6897
Epoch 8/10
1563/1563 [=====] - 69s 44ms/step - loss: 0.7272 - accuracy: 0.7426 - val_loss: 0.8699 - val_accuracy: 0.6992
Epoch 9/10
1563/1563 [=====] - 69s 44ms/step - loss: 0.6827 - accuracy: 0.7582 - val_loss: 0.8527 - val_accuracy: 0.7105
Epoch 10/10
1563/1563 [=====] - 70s 45ms/step - loss: 0.6491 - accuracy: 0.7723 - val_loss: 0.8699 - val_accuracy: 0.7122
313/313 - 5s - loss: 0.8699 - accuracy: 0.7122 - 5s/epoch - 16ms/step

Test accuracy: 0.71
313/313 [=====] - 4s 12ms/step

```



### Learning:

- The final accuracy of the model is reported to be 71%. This indicates that the model correctly classified 71% of the images in the test set.
- The provided confusion matrix shows how many images from each class were correctly classified (diagonal elements) and misclassified (off-diagonal elements).
- The high values on the diagonal indicate good performance for some classes. Conversely, low values on the diagonal and high values in specific off-diagonal entries suggest difficulty in differentiating between those classes.

# Experiment - 5

**Aim:** Write a program to implement k-Nearest Neighbour algorithm to classify IRIS dataset. Use cross validation and following metrics for performance evaluation: a)

- a. Accuracy
- b. Misclassification Rate
- c. Sensitivity
- d. Specificity
- e. Precision
- f. F-Score

## **Introduction:**

### ***KNN Model***

K-Nearest Neighbors (KNN) is a versatile and intuitive machine learning algorithm used for both classification and regression tasks. It belongs to the family of instance-based learning and is considered a simple yet powerful method for making predictions based on similarity measures.

### **Key characteristics:**

#### **Distance Metric:**

KNN relies on a distance metric (usually Euclidean distance) to measure the similarity between data points. Other distance metrics, such as Manhattan or Minkowski, can also be employed based on the nature of the data.

#### **Parameter 'k':**

The parameter 'k' represents the number of neighbors to consider when making a prediction. Selecting an appropriate 'k' is crucial; a smaller 'k' may lead to overfitting, while a larger 'k' may result in underfitting.

**Accuracy:**

- The proportion of correctly classified samples.
- Indicates the overall effectiveness of a model in making correct predictions.

**Misclassification Rate:**

- The proportion of incorrectly classified samples.
- Complementary to accuracy, highlighting the percentage of mistakes the model makes.

**Sensitivity:**

- The ability of a model to correctly identify positive cases (e.g., detecting a disease when it is present).
- Also known as the True Positive Rate (TPR) or Recall.

**Specificity:**

- The ability of a model to correctly identify negative cases (e.g., identifying healthy individuals when they are truly healthy).
- Also known as the True Negative Rate (TNR).

**Precision:**

- The proportion of positive predictions that are actually true positives.
- Measures the model's ability to avoid false positive predictions.

**F-Score:**

- A harmonic mean between precision and recall, combining their information into a single metric.
- Provides a balanced view of both precision and recall, useful when dealing with imbalanced class distributions.

**Dataset Description:****The Iris Flower Dataset: A Classic for Machine Learning**

The Iris flower dataset is a widely used and well-known dataset in the field of machine learning. It was first introduced by Ronald Fisher in 1936 and is often used as a starting point for exploring various classification algorithms.

**Key characteristics:**

Content: Measurements of four features (sepal and petal length and width) from three species of iris flowers (Iris setosa, Iris virginica, and Iris versicolor).

**Decision Rule:**

For classification tasks, the majority vote among the k-nearest neighbors determines the class of the target instance. In regression tasks, the average of the k-nearest neighbors' values is used.

Size: Contains 150 samples, with 50 samples from each of the three iris species.

Format: Each sample is represented by a five-dimensional vector, consisting of the four measurements and the corresponding flower species (represented as a label).

**Code :**

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_predict, StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score, f1_score

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Standardize the features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Initialize the KNN classifier with k=3 (you can choose other values of k as well)
knn = KNeighborsClassifier(n_neighbors=3)

# Perform cross-validation with stratified k-folds
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
y_pred = cross_val_predict(knn, X, y, cv=cv)

# Compute confusion matrix
cm = confusion_matrix(y, y_pred)
tn = cm[0, 0]
tp = np.diag(cm)
fn = np.sum(cm, axis=1) - tp
fp = np.sum(cm, axis=0) - tp
```

```

# Calculate metrics
accuracy = accuracy_score(y, y_pred)
misclassification_rate = 1 - accuracy
precision = precision_score(y, y_pred, average='weighted')
recall = recall_score(y, y_pred, average='weighted') # Sensitivity
f1 = f1_score(y, y_pred, average='weighted')

# Specificity calculation
specificity = np.mean(tn / (tn + fp))

# Print metrics
print(f'Accuracy: {accuracy:.2f}')
print(f'Misclassification Rate: {misclassification_rate:.2f}')
print(f'Sensitivity (Recall): {recall:.2f}')
print(f'Specificity: {specificity:.2f}')
print(f'Precision: {precision:.2f}')
print(f'F-Score: {f1:.2f}')

```

```

⇒ Accuracy: 0.94
Misclassification Rate: 0.06
Sensitivity (Recall): 0.94
Specificity: 0.94
Precision: 0.94
F-Score: 0.94

```

### Learning:

- The model is producing ideal Accuracy, precision, f1-score, misclassification rate, sensitivity and specificity. Which might be alarming as achieving such perfect scores shouldn't be possible.
- Still KNN can capture complex and non-linear relationships in data by relying on the similarity of points in the feature space. This can be advantageous when dealing with datasets that exhibit non-linear patterns or interactions between features.

# Experiment - 6

**Aim:** Write a python code for the following 5 activation functions:

- a. Step
- b. Sigmoid
- c. tanh
- d. ReLU (rectified linear unit)
- e. Leaky ReLU

## **Introduction:**

### ***Activation Functions***

Activation functions are the non-linear workhorses in artificial neural networks (ANNs). They play a crucial role in transforming the input received by a neuron and determining its output.

### **Key functions:**

- Introduce non-linearity: Unlike linear functions, activation functions allow ANNs to learn and model complex relationships in data, enabling them to tackle diverse tasks.
- Control output: They define the output range of a neuron, ensuring the values fall within a specific desired range.
- Gate information flow: By affecting the output based on the input, they act like gates controlling which information is passed forward in the network.

### **Step Function:**

- A simple binary function, outputting 0 for negative inputs and 1 for positive inputs.
- Not widely used in practice due to its non-differentiability at zero, hindering backpropagation in neural networks.
- Mainly used for illustration or theoretical purposes.

**Sigmoid Function (Logistic Function):**

- S-shaped curve, squishing values between 0 and 1.
- Commonly used for tasks like binary classification due to its output range.
- Suffers from vanishing gradient problem in deep networks, making it less preferred in complex architectures.

**tanh (Hyperbolic Tangent):**

- Similar to sigmoid but centered around 0, with outputs ranging from -1 to 1.
- Addresses the problem of sigmoid not being centered at zero.
- Still susceptible to vanishing gradient problems in deep networks.

**ReLU (Rectified Linear Unit):**

- Piecewise linear function, outputting the input directly if positive, otherwise zero.
- Popular choice due to its computational efficiency and ability to overcome vanishing gradients.
- Can lead to "dying ReLU" issue where neurons become permanently inactive if receiving constant negative inputs.

**Leaky ReLU:**

- Variant of ReLU that allows a small non-zero slope for negative inputs, preventing "dying ReLU" problem.
- Offers a balance between ReLU's efficiency and the ability to handle negative inputs to some extent.
- Widely used in various neural network architectures due to its effectiveness and versatility.

**Dataset Description:****Self Generated Dataset For The purpose of Visualisation**

The dataset is generated using a NumPy array `x` containing 400 evenly spaced values between -10 and 10 (inclusive). The `linspace` function is used for this purpose. The code used for the generation of dataset for the purpose of visualization is: `x = np.linspace(-10, 10, 400)` `np.linspace(start, stop, num)`: This function creates an array of `num` evenly spaced values over the interval `[start, stop]`. In this case, `start` is -10, `stop` is 10, and `num` is 400.



## Code :

```
import numpy as np
import matplotlib.pyplot as plt

# Step Function
def step_function(x):
    return np.where(x >= 0, 1, 0)

# Sigmoid Function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Tanh Function
def tanh(x):
    return np.tanh(x)

# ReLU Function
def relu(x):
    return np.maximum(0, x)

# Leaky ReLU Function
def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

# Test inputs
x = np.linspace(-10, 10, 400)

# Calculate outputs for each activation function
y_step = step_function(x)
y_sigmoid = sigmoid(x)
y_tanh = tanh(x)
y_relu = relu(x)
y_leaky_relu = leaky_relu(x)
```

```
# Plotting the activation functions
```

```
plt.figure(figsize=(12, 8))
```

```
plt.subplot(2, 3, 1)
```

```
plt.plot(x, y_step)
```

```
plt.title('Step Function')
```

```
plt.subplot(2, 3, 2)
```

```
plt.plot(x, y_sigmoid)
```

```
plt.title('Sigmoid Function')
```

```
plt.subplot(2, 3, 3)
```

```
plt.plot(x, y_tanh)
```

```
plt.title('Tanh Function')
```

```
plt.subplot(2, 3, 4)
```

```
plt.plot(x, y_relu)
```

```
plt.title('ReLU Function')
```

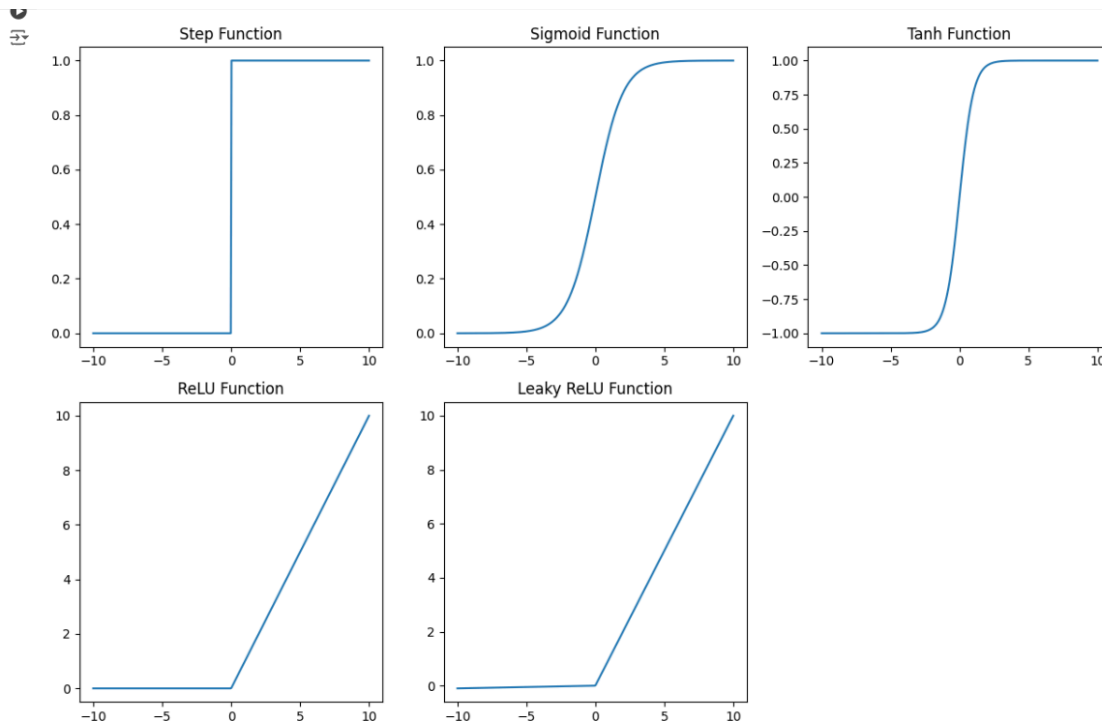
```
plt.subplot(2, 3, 5)
```

```
plt.plot(x, y_leaky_relu)
```

```
plt.title('Leaky ReLU Function')
```

```
plt.tight_layout()
```

```
plt.show()
```



**Learning:**

- The Step function created a binary output (0 or 1) based on a threshold value which is 0 in the above experiment.
- The Sigmoid function outputs a value between 0 and 1, resembling an S-shaped curve.
- The Tanh function, similar to the sigmoid function, but outputs values between -1 and 1.
- The ReLU function outputs the input directly if it's positive, otherwise outputs 0.
- The Leaky ReLU function, similar to ReLU, but with a small non-zero slope for negative inputs. It aims to address the "dying ReLU" problem by allowing a small gradient to flow through even for negative inputs.