D Y PATIL
COLLEGE of ENGINEERING & TECHNOLOGY
KASABA BAWADA, KOLHAPUR - 416 006.

Subject :- Deep Learning Lab Class :- B.Tech (AIML)

Roll No. :- 24 **INDEX** Exam No. : -

Sr. No.	TITLE	Page No.	Date	Signature	Remarks
1.	Installation of tensorflow & keras Implementation	1	31-7-23	Aleyak	
2.	Implement data manipulation operation.	3	31-7-23	Aleyak	
3.	Implementation of data preprocessing, reading & handling dataset.	6	7-8-23	Aleyak	
4.	Implementing tensorflow airthmetic & matrix multiplication.	8	7-8-23	Aleyak	
5.	Implement Mccwloch pitts Neural Network using tensorflow.	10	21-8-23	Aleyak	
6.	Implement forwaed Pass with matrix multiplication	13	28-8-23	Aleyak	

Certificate

This is to certify that Mr. / Miss Anjali Jaywant kumbhar

of B.Tech Class AIML (CSE) has completed the above

mentioned experiments in the subject Deep Learning. in the laboratory of the

Dr. D. Y. Patil College of Engg. & Tech., Kasaba Bavada, Kolhapur - 416006, in year.

Date :

Aleyak
Staff Incharge

Head of Dept.
HEAD OF DEPARTMENT
Computer Science

Principal

D Y PATIL
COLLEGE of ENGINEERING & TECHNOLOGY
KASABA BAWADA, KOLHAPUR - 416 006.

Subject :- Deep learning lab Class :- B.Tech (AIML)

Roll No. :- 24 **INDEX** Exam No. : -

Sr. No.	TITLE	Page No.	Date	Signature	Remarks
7.	FCNN with only one neuron & plotting FCN.	16	4-9-23	Alyak	
8.	Implement simple CNN & train & validate dataset	21	25-9-23	Alyak	
9.	MNIST digit classification	25	9-10-23	Alyak	
10.	Cifar10 classification with & without normalization CNN.	29	16-10-23	Alyak	
11.	Using a pre-trained Imagenet network to predict images.	33	23-10-23	Alyak	
12.	Implementation of simple RNN & Deep RNN.	36	30-10-23	Alyak	

Certificate

This is to certify that Mr. / Miss Anjali Jaywant kumbhar

of B.Tech Class AIML (CSE) has completed the above mentioned experiments in the subject Deep Learning in the laboratory of the Dr. D. Y. Patil College of Engg. & Tech., Kasaba Bavada, Kolhapur - 416006. in year.

Alyak
Staff Incharge

HEAD OF

Experiment 1

Title :- Installation of Tensorflow & Keras (Tensorflow (v1.0.0), TFLearn, Keras, and many other pre-installed python libraries (Numpy, pandas).

Aim:- To implement Installation of Tensorflow, Keras, TFLearn, Keras, and many other pre-installed python libraries (Numpy, pandas).

1) **TensorFlow**

TensorFlow is a popular framework of machine learning and deep learning. It is a free and open-source library which is released on 9 November 2015 and developed by Google Brain Team. It is entirely based on Python programming language and use for numerical computation and data flow, which makes machine learning faster and easier.

TensorFlow can train and run the deep neural networks for image recognition, handwritten digit classification, recurrent neural network, word embedding, natural language processing, video detection, and many more. TensorFlow is run on multiple CPUs or GPUs and also mobile operating systems.
The word TensorFlow is made by two words, i.e., Tensor and Flow

- Tensor is a multidimensional array
- Flow is used to define the flow of data in operation.

TensorFlow in our system in 2 ways:

1. Through pip (Python package library)
2. Through Anaconda Navigator (conda)

2) **Keras**

Keras is an open-source high-level Neural Network library, which is written in Python is capable enough to run on Theano, TensorFlow, or CNTK. It was developed by one of the Google engineers, Francois Chollet. It is made user-friendly, extensible, and modular for facilitating faster experimentation with deep neural networks. It not only supports Convolutional Networks and Recurrent Networks individually but also their combination.

Features : Keras leverages various optimization techniques to make high level neural network API easier and more performant. It supports the following features –

- Consistent, simple and extensible API.
- Minimal structure - easy to achieve the result without any frills.
- It supports multiple platforms and backends.
- It is user friendly framework which runs on both CPU and GPU.
- Highly scalability of computation.

Keras allows you to switch between different back ends. The frameworks supported by Keras are:

Tensorflow, Theano, PlaidML, MXNet, CNTK

Keras depends on the following python libraries.

- Numpy
- Pandas
- Scikit-learn
- Matplotlib
- Scipy
- Seaborn

Step 1) Installing Keras.

```
C:\Users\acer>pip install keras
Requirement already satisfied: keras in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (2.11.0)
```

Step 2) Installing TensorFlow & related library.

```
C:\Users\acer>pip install Tensorflow
Requirement already satisfied: Tensorflow in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (2.11.0)
Requirement already satisfied: tensorflow-intel==2.11.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from Tensorflow) (2.11.0)
Requirement already satisfied: absl-py>=1.0.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=2.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (23.1.21)
Requirement already satisfied: gast<=0.4.0,>=0.2.1 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (0.4.0)
Requirement already satisfied: google-pasta>=0.1.1 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (0.2.0)
Requirement already satisfied: h5py>=2.9.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (3.8.0)
Requirement already satisfied: libclang>=13.0.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (15.0.6.1)
Requirement already satisfied: numpy>=1.20 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (1.24.3)
Requirement already satisfied: opt-einsum>=2.3.2 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (3.3.0)
Requirement already satisfied: packaging in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (23.1)
Requirement already satisfied: protobuf<3.20,>=3.9.2 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (3.19.6)
Requirement already satisfied: setuptools in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (63.2.0)
Requirement already satisfied: six>=1.12.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (2.2.0)
Requirement already satisfied: typing-extensions>=3.6.6 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (4.6.3)
Requirement already satisfied: wrapt>=1.11.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (1.14.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (1.53.0)
Requirement already satisfied: tensorboard<2.12,>=2.11 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (2.11.2)
Requirement already satisfied: tensorflow-estimator<2.12,>=2.11.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (2.11.0)
Requirement already satisfied: keras<2.12,>=2.11.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (2.11.0)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorflow-intel==2.11.0->Tensorflow) (0.31.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from astunparse>=1.6.0->tensorflow-intel==2.11.0->Tensorflow) (0.38.4)
Requirement already satisfied: google-auth<3,>=1.6.3 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorflow) (2.16.3)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorflow)
(0.4.6)
Requirement already satisfied: markdown>=2.6.8 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorflow) (3.4.3)
Requirement already satisfied: requests<3,>=2.21.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorflow) (2.31.0)
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorf
low) (0.6.1)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorflow) (1.
8.1)
Requirement already satisfied: werkzeug>=1.0.1 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorflow) (2.3.4)
Requirement already satisfied: cachetools<6.0,>=2.0.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from google-auth<3,>=1.6.3->tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorf
>Tensorflow) (5.3.1)
Requirement already satisfied: pyasn1-modules>=0.2.1 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from google-auth<3,>=1.6.3->tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->
TensorFlow) (0.2.8)
Requirement already satisfied: rsa<5,>=3.1.4 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from google-auth<3,>=1.6.3->tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorf
ow) (4.9)
Requirement already satisfied: requests-oauthlib>=0.7.0 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from google-auth-oauthlib<0.5,>=0.4.1->tensorboard<2.12,>=2.11->tensorflow-
ntel==2.11.0->TensorFlow) (1.3.1)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from requests<3,>=2.21.0->tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorf
>Tensorflow) (3.1.0)
Requirement already satisfied: idna<4,>=2.5 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from requests<3,>=2.21.0->tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorflow
(3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from requests<3,>=2.21.0->tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensor
rflow) (2.0.2)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from requests<3,>=2.21.0->tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorflow
Requirement already satisfied: certifi>=2017.4.17 in c:\users\acer\appdata\local\programs\python\python310\lib\site-packages (from requests<3,>=2.21.0->tensorboard<2.12,>=2.11->tensorflow-intel==2.11.0->Tensorflow
```

Conclusion

Successful installation of essential frameworks like Tensorflow, TFLearn, and Keras, along with a host of pre-installed Python libraries such as Numpy and Pandas, lays a robust foundation for our experiments.

Experiment 2

Title :- Data Manipulation (Numpy library) Operations Broadcasting Indexing and slicing.

Aim:- To Data Manipulation (Numpy library) Operations Broadcasting Indexing and slicing.

Data manipulation using the NumPy library is a fundamental skill in scientific computing and data analysis with Python. NumPy provides powerful array operations, broadcasting, indexing, and slicing capabilities that make it efficient and convenient for handling large datasets.

Step 1) Install NumPy library

```
In [1]: pip install numpy

        Requirement already satisfied: numpy in c:\programdata\anaconda3\lib\site-packages (1.20.1)
```

Step 2) Creating NumPy Arrays:

You can create a NumPy array using the numpy.array() function or other specialized functions like numpy.zeros(), numpy.ones(), and numpy.random.

```
In [14]: import numpy as np

         # Create a NumPy array from a list
         arr = np.array([1, 2, 3, 4, 5])
         print("Array list \n",arr)

         # Create an array of zeros with shape (3, 4)
         zeros_array = np.zeros((3, 4))
         print("Array list zeros \n",zeros_array)

         # Create an array of ones with shape (2, 3)
         ones_array = np.ones((2, 3))
         print("Array list ones \n",ones_array)

         # Create a random array with shape (2, 2)
         random_array = np.random.rand(2, 2)
         print("Random Array list \n",random_array)

         Array list
          [1 2 3 4 5]
         Array list zeros
          [[0. 0. 0. 0.]
          [0. 0. 0. 0.]
          [0. 0. 0. 0.]]
         Array list ones
          [[1. 1. 1.]
          [1. 1. 1.]]
         Random Array list
          [[0.74598031 0.78585436]
          [0.38974045 0.17106252]]
```

Step 3) Broadcasting:

NumPy allows you to perform operations on arrays with different shapes by broadcasting the smaller array to match the shape of the larger array.

```
In [21]: # Broadcasting example
         a = np.array([1, 2, 3])
         scalar = 2
         result = a * scalar  # Broadcasting the scalar to the array [2, 4, 6]
         print("Broadcasting", result)

         Broadcasting [2 4 6]
```

Step 4) Array Operations:

You can perform various operations on NumPy arrays, such as element-wise arithmetic, matrix operations, and statistical functions.

```
In [20]: # Element-wise arithmetic operations
         a = np.array([1, 2, 3])
         b = np.array([4, 5, 6])
         print("addition",a + b)
         print("subtraction",a - b)
         print("multiplication",a * b)
         print("division",a / b)

         # Matrix operations
         matrix_a = np.array([[1, 2], [3, 4]])
         matrix_b = np.array([[5, 6], [7, 8]])
         matrix_product = np.dot(matrix_a, matrix_b)  # Matrix multiplication
         print("Matrix Product \n",matrix_product)

         # Statistical functions
         data = np.array([10, 15, 20, 25, 30])
         print(data)
         mean = np.mean(data)
         print("Mean",mean)
         median = np.median(data)
         print("Median",median)
         std_deviation = np.std(data)
         print("Std Deviation",std_deviation)

         addition [5 7 9]
         subtraction [-3 -3 -3]
         multiplication [ 4 10 18]
         division [0.25 0.4  0.5 ]
         Matrix Product
          [[19 22]
          [43 50]]
         [10 15 20 25 30]
         Mean 20.0
         Median 20.0
         Std Deviation 7.0710678118654755
```

Step 5) Indexing and Slicing:

You can access specific elements in a NumPy array using indexing. Slicing allows you to extract a portion of an array.

```
In [23]: arr = np.array([10, 20, 30, 40, 50])

         # Accessing elements using indexing
         element = arr[0]  # Access the first element (10)
         print("[0] ",element)
         element_2 = arr[-1]  # Access the last element (50)
         print("[-1] ",element_2)

         [0]  10
         [-1]  50
```

```
In [22]: arr = np.array([10, 20, 30, 40, 50])

         # Slicing
         slice_1 = arr[1:4]  # Get elements from index 1 to index 3 (20, 30, 40)
         print("arr[1:4] ",slice_1)
         slice_2 = arr[:3]  # Get elements from the beginning up to index 2 (10, 20, 30)
         print("arr[:3] ",slice_2)
         slice_3 = arr[2:]  # Get elements from index 2 to the end (30, 40, 50)
         print("arr[2:] ",slice_3)

         arr[1:4]  [20 30 40]
         arr[:3]  [10 20 30]
         arr[2:]  [30 40 50]
```

Conclusion

Successful installation of essential library, data Manipulation (Numpy library) Operations Broadcasting Indexing and slicing.

Experiment 3

Title :- Data Preprocessing Reading the Dataset Handling Missing Data Conversion to the Tensor Format

Aim:- To data Preprocessing Reading the Dataset Handling Missing Data Conversion to the Tensor Format.

Data preprocessing is a crucial step in the data analysis and machine learning pipeline. It involves cleaning, transforming, and preparing the data to be used in machine learning models. Let's go through the steps of data preprocessing, including reading the dataset, handling missing data, and converting the data into the tensor format.

Step 1) Reading the Dataset:

To read a dataset into Python, you can use various libraries such as NumPy, Pandas, or TensorFlow (tf.data) depending on the format of your data. Pandas is a popular choice for handling tabular data.

```
[1]: import pandas as pd

     # Read a CSV file into a Pandas DataFrame
     df = pd.read_csv('your_dataset.csv')
     print(df)
```

```
      Name   Age  Gender  Salary
0    Alice  25.0  Female   50000
1      Bob  30.0    Male   60000
2  Charlie   NaN    Male   45000
3    David  28.0    Male   70000
4      Eva  35.0  Female   55000
```

Step 2) Handling Missing Data:

Missing data is a common issue in datasets. You need to handle missing data appropriately to avoid biased or incorrect results in your analysis or model training.

```
[2]: # Check for missing values in the DataFrame
     print(df.isnull().sum())

     # Option 1: Drop rows with missing values
     #df = df.dropna()

     # Option 2: Fill missing values with a specific value (e.g., mean, median, or custom value)
     mean_age = df['Age'].mean()
     df['Age'].fillna(mean_age, inplace=True)
     #df = pd.DataFrame(df)

     df['Gender'] = df['Gender'].map({'Male':1 , 'Female':0})
     print(df)
```

```
Name      0
Age       1
Gender    0
Salary    0
dtype: int64
      Name   Age  Gender  Salary
0    Alice  25.0       0   50000
1      Bob  30.0       1   60000
2  Charlie  29.5       1   45000
3    David  28.0       1   70000
4      Eva  35.0       0   55000
```

Step 3) Conversion to the Tensor Format:

Machine learning models, especially those built with TensorFlow, often require data in tensor format. A tensor is a multi-dimensional array.

```
[3]: import numpy as np
     import tensorflow as tf

     # Extract features and labels from the DataFrame
     X = df[['Age', 'Gender']].values
     y = df['Salary'].values

     # Convert to TensorFlow tensors
     X_tensor = tf.constant(X, dtype=tf.float32)
     y_tensor = tf.constant(y, dtype=tf.int32)

     # Alternatively, you can use NumPy arrays directly as TensorFlow tensors
     X_tensor = tf.convert_to_tensor(X, dtype=tf.float32)
     y_tensor = tf.convert_to_tensor(y, dtype=tf.int32)

     print("\n X tensor \n",X_tensor)
     print("\n Y tensor\n",y_tensor)
```

```
 X tensor
 tf.Tensor(
[[25.   0. ]
 [30.   1. ]
 [29.5  1. ]
 [28.   1. ]
 [35.   0. ]], shape=(5, 2), dtype=float32)

 Y tensor
 tf.Tensor([50000 60000 45000 70000 55000], shape=(5,), dtype=int32)
```

Conclusion

Successful installation of essential library, Data Preprocessing Reading the Dataset Handling Missing Data Conversion to the Tensor Format.

Experiment 4

Title :- Linear Algebra Tensors Tensor arithmetic Implementing matrix multiplication.

Aim:- To Linear Algebra Tensors Tensor arithmetic Implementing matrix multiplication.

Linear algebra plays a fundamental role in machine learning, and tensors are essential data structures used for representing multi-dimensional arrays. In this section, I'll cover basic tensor operations, tensor arithmetic, and how to implement matrix multiplication using NumPy.

A tensor is a generalization of vectors and matrices and is easily understood as a multidimensional array. A vector is a one-dimensional or first order tensor and a matrix is a two-dimensional or second order tensor.

Step 1) Tensors and Tensor Arithmetic:

Tensors are multi-dimensional arrays. In the context of linear algebra, vectors are 1-dimensional tensors, matrices are 2-dimensional tensors, and so on.

```
In [7]: import numpy as np

        # Creating tensors (arrays) using NumPy
        # 1-D tensor (vector)
        vector = np.array([1, 2, 3])

        # 2-D tensor (matrix)
        matrix = np.array([[1, 2], [3, 4]])

        # 3-D tensor
        tensor_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

        # Tensor arithmetic (element-wise operations)
        a = np.array([1, 2, 3])
        b = np.array([4, 5, 6])

        addition = a + b
        print("addition ", addition)
        subtraction = a - b
        print("subtraction ", subtraction)
        elementwise_multiplication = a * b
        print("multiplication ", elementwise_multiplication)
        elementwise_division = a / b
        print("division ", elementwise_division)

        addition  [5 7 9]
        subtraction  [-3 -3 -3]
        multiplication  [ 4 10 18]
        division  [0.25 0.4  0.5 ]
```

Step 2) Implementing Matrix Multiplication:

Matrix multiplication is a fundamental operation in linear algebra. In NumPy, you can use the `dot()` function or `@` operator to perform matrix multiplication.

```
In [9]: import numpy as np

        # Matrix multiplication using dot() function
        matrix_a = np.array([[1, 2], [3, 4]])
        matrix_b = np.array([[5, 6], [7, 8]])

        # Option 1: Use dot() function
        matrix_product = np.dot(matrix_a, matrix_b)
        print("Dot \n", matrix_product)

        # Option 2: Use @ operator (Python 3.5+)
        matrix_product = matrix_a @ matrix_b
        print("@ operator\n ", matrix_product)
```

```
Dot
 [[19 22]
 [43 50]]
@ operator
  [[19 22]
 [43 50]]
```

Conclusion

Successful Implement McCulloch Pitts neural network using Tensorflow.

Experiment 5

Title :- Implement McCulloch Pitts neural network using Tensorflow.

Aim:- To Implement McCulloch Pitts neural network using Tensorflow.

McCulloch Pitts neural network:

The McCulloch-Pitts neuron, also known as the McCulloch-Pitts model, is a simplified mathematical model of a biological neuron. It was introduced by Warren McCulloch and Walter Pitts in 1943. The McCulloch-Pitts neuron is a foundational concept in neural network theory and serves as the basis for understanding more complex artificial neural networks.

The McCulloch-Pitts neuron operates using binary logic (0 or 1) and consists of the following components:

1. Inputs: Each neuron receives binary inputs (0 or 1) from various sources, which represent the incoming signals from other neurons or external stimuli.

2. Weights: Each input is assigned a weight that represents the strength of the connection between the input and the neuron. The weights can be positive or negative.

3. Threshold: The neuron has a threshold value that determines the level of activation required for it to fire an output signal.

4. Activation Function: The McCulloch-Pitts neuron applies a simple activation function. It sums the products of inputs and their corresponding weights and compares this sum to the threshold. If the sum is greater than or equal to the threshold, the neuron outputs 1; otherwise, it outputs 0.

Mathematically, the output of the McCulloch-Pitts neuron can be represented as:

Output = 1 if $\Sigma(w_i * x_i) \geq \theta$ (threshold)
Output = 0 if $\Sigma(w_i * x_i) < \theta$

This basic neuron model can be used to create logical circuits and perform simple binary computations. It serves as a building block for more complex neural network architectures, such as perceptrons and multi-layer perceptrons, which have enhanced capabilities for pattern recognition and classification.

Single Layer Neural Network:

A single-layer neural network, often referred to as a single-layer perceptron, is a type of artificial neural network architecture that consists of only one layer of interconnected neurons. It is a basic and elementary form of a neural network, and it can be used for simple linear classification tasks.

Here's how a single-layer perceptron works:

1. Inputs:Similar to the McCulloch-Pitts neuron, a single-layer perceptron receives inputs (features) from the external world. Each input is associated with a weight, which represents the strength of the connection between the input and the neuron.

2. Weights: The weights are multiplied by their corresponding inputs, and the resulting weighted inputs are summed up. Mathematically, this can be represented as:

Weighted Sum = Σ(w_i * x_i)

3. Bias:In addition to weights and inputs, a bias term is often included. The bias is a constant value that is added to the weighted sum before applying the activation function. It allows the network to control the threshold at which the neuron fires.

4. Activation Function:The weighted sum (including the bias) is then passed through an activation function. Common activation functions include the step function (similar to McCulloch-Pitts threshold), sigmoid function, or ReLU (Rectified Linear Unit). The activation function determines the output of the neuron based on the computed value.

Output = Activation_Function(Weighted Sum + Bias)

5. Output: The final output of the single-layer perceptron is the result of applying the activation function to the weighted sum of inputs.

Output McCulloch Pitts neural network

```
class McCullochPittsNeuron:
    def __init__(self, weights, threshold):
        self.weights = weights
        self.threshold = threshold

    def activate(self, inputs):
        if len(inputs) != len(self.weights):
            raise ValueError("Number of inputs should match the number of weights
        result = sum([inputs[i] * self.weights[i] for i in range(len(inputs))])
        return 1 if result >= self.threshold else 0

def main():
    # Define the weights and threshold for the AND gate
    and_gate_weights = [1, 1]
    and_gate_threshold = 2

    # Create a McCulloch-Pitts neuron for the AND gate
    and_gate_neuron = McCullochPittsNeuron(and_gate_weights, and_gate_threshold)

    # Test the AND gate
    inputs = [
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ]
    for input_pair in inputs:
        output = and_gate_neuron.activate(input_pair)
        print(f"{input_pair} => {output}")

if __name__ == "__main__":
    main()
```

```
[0, 0] -> 0
[0, 1] -> 0
[1, 0] -> 0
[1, 1] -> 1
```

Output of single-layer perceptron

```
import numpy as np

class SingleLayerPerceptron:
    def __init__(self, input_size):
        self.weights = np.random.rand(input_size)
        self.bias = np.random.rand()
        self.learning_rate = 0.1

    def predict(self, inputs):
        net_input = np.dot(inputs, self.weights) + self.bias
        return 1 if net_input >= 0 else 0

    def train(self, training_data, epochs):
        for epoch in range(epochs):
            errors = 0
            for inputs, label in training_data:
                prediction = self.predict(inputs)
                error = label - prediction
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error
                errors += abs(error)
            if errors == 0:
                print(f"Converged after {epoch + 1} epochs.")
                break

def main():
    # Sample training data for OR gate
    training_data = [
        (np.array([0, 0]), 0),
        (np.array([0, 1]), 1),
        (np.array([1, 0]), 1),
        (np.array([1, 1]), 1)
    ]

    input_size = len(training_data[0][0])
    perceptron = SingleLayerPerceptron(input_size)

    epochs = 100
    perceptron.train(training_data, epochs)

    # Test the trained perceptron
    test_data = [
        np.array([0, 0]),
        np.array([0, 1]),
        np.array([1, 0]),
        np.array([1, 1])
    ]

    print("Testing the perceptron:")
    for inputs in test_data:
        prediction = perceptron.predict(inputs)
        print(f"{inputs} -> {prediction}")

if __name__ == "__main__":
    main()
```

```
Converged after 3 epochs.
Testing the perceptron:
[0 0] -> 0
[0 1] -> 1
[1 0] -> 1
[1 1] -> 1
```

Experiment 6

Title :- Forward pass with matrix multiplication Forward pass with hidden layer (matrix multiplication) Forward pass with matrix multiplication with Keras Forward passes with hidden layer (matrix multiplication) with Keras.

Aim:- To Implement Forward pass with matrix multiplication Forward pass with hidden layer Forward pass with matrix multiplication with Keras Forward passes with hidden layer with Keras.

Overview of implementing forward passes using matrix multiplication and Keras for both a single-layer neural network (perceptron) and a multi-layer neural network (with a hidden layer):

1) Forward Pass with Matrix Multiplication in Keras (Single-Layer Neural Network).
2) Define a Keras model with a single layer: model
3) To perform the forward pass and obtain the output.
4) Forward Pass with Hidden Layer in Keras (Multi-Layer Neural Network):
5) Create input data.
6) Define a Keras model with a hidden layer and an output layer.
7) To perform the forward pass and obtain the output.

In [3]:
```
# 1.Forward Pass with Matrix Multiplication (Without Hidden Layer)

import numpy as np

# Input data (a 3x2 matrix)
input_data = np.array([[1, 2],
                       [3, 4],
                       [5, 6]])

# Weight matrix (a 2x2 matrix)
weights = np.array([[0.1, 0.2],
                    [0.3, 0.4]])

# Forward pass without a hidden layer (matrix multiplication)
output_data = np.dot(input_data, weights)

print("Output without hidden layer:")
print(output_data)
```

```
Output without hidden layer:
[[0.7 1. ]
 [1.5 2.2]
 [2.3 3.4]]
```

In [4]:
```
# 2. Forward Pass with Hidden Layer (Matrix Multiplication)

import numpy as np

# Input data (a 3x2 matrix)
input_data = np.array([[1, 2],
                       [3, 4],
                       [5, 6]])

# Weight matrices (2 hidden units, 2 input features)
weights_hidden = np.array([[0.1, 0.2],
                           [0.3, 0.4]])

# Weight matrices (2 output units, 2 hidden units)
weights_output = np.array([[0.5, 0.6],
                           [0.7, 0.8]])

# Forward pass with a hidden layer (matrix multiplication)
hidden_layer_output = np.dot(input_data, weights_hidden)
output_data = np.dot(hidden_layer_output, weights_output)

print("Output with hidden layer:")
print(output_data)
```

```
Output with hidden layer:
[[1.05 1.22]
 [2.29 2.66]
 [3.53 4.1 ]]
```

In [5]:
```
# 3. Forward Pass with Matrix Multiplication using Keras:

import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Input data (a 3x2 matrix)
input_data = np.array([[1, 2],
                       [3, 4],
                       [5, 6]])

# Define the Keras model
model = Sequential()
model.add(Dense(units=2, input_dim=2, activation='sigmoid'))
model.add(Dense(units=2, activation='sigmoid'))

# Weight matrices will be automatically initialized by Keras

# Perform the forward pass using Keras
output_data = model.predict(input_data)

print("Output with Keras:")
print(output_data)
```

```
1/1 [==============================] - 1s 653ms/step
Output with Keras:
[[0.4551778  0.5146156 ]
 [0.48358917 0.46191478]
 [0.5012354  0.43267027]]
```

In [6]:
```
# 4. Forward passes with hidden layer (matrix multiplication) with Keras.

import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Input data (a 3x2 matrix)
input_data = np.array([[1, 2],
                       [3, 4],
                       [5, 6]])

# Define the Keras model
model = Sequential()

# Add a hidden layer with 2 units and input dimension 2, using matrix multipli
model.add(Dense(units=2, input_dim=2, activation='sigmoid'))

# Add the output layer with 2 units, also using matrix multiplication
model.add(Dense(units=2, activation='sigmoid'))

# Weight matrices will be automatically initialized by Keras

# Perform the forward pass using Keras
output_data = model.predict(input_data)

print("Output with hidden layer using Keras:")
print(output_data)
```

```
1/1 [==============================] - 0s 43ms/step
Output with hidden layer using Keras:
[[0.4015603  0.67632186]
 [0.39087415 0.69307435]
 [0.39129204 0.6895925 ]]
```

Conclusion

Successful Implement Forward pass with matrix multiplication Forward pass with hidden layer Forward pass with matrix multiplication with Keras Forward passes with hidden layer.

Experiment No: 07

Aim: Use FCNN with only one neuron and plotting FCNN with one hidden layer and plotting.

Theory:

FCNN stands for **Fully Connected Neural Network**, and it is a type of artificial neural network that is characterized by its dense or fully connected layers. FCNNs are also commonly referred to as feedforward neural networks or multilayer perceptrons (MLP's).

Key components FCNNs:

Neurons/Nodes: FCNNs consist of layers of interconnected nodes, where each node is often called a "neuron." Neurons in a layer are fully connected to neurons in the adjacent layers, which means that each neuron in one layer is connected to every neuron in the next layer.

Layers: An FCNN typically consists of an input layer, one or more hidden layers, and an output layer. The input layer takes the raw input data, and the output layer produces the network's predictions. The hidden layers are intermediate layers that transform the input data as it passes through the network.

Weights and Biases: Each connection between neurons is associated with a weight, which represents the strength of the connection. The neurons also have biases that can be adjusted. During training, the network learns the optimal values for these weights and biases to make accurate predictions.

Activation Functions: Each neuron applies an activation function to the weighted sum of its inputs. Common activation functions include sigmoid, ReLU (Rectified Linear Unit), tanh, and more. These functions introduce non-linearity to the model, allowing it to capture complex relationships in the data.

Forward Propagation: During forward propagation, data is passed through the network layer by layer. Each neuron in a layer receives input from all neurons in the previous layer, applies its activation function, and passes the result to the next layer. This process continues until the output layer produces the final prediction.

Loss/Cost Function: To measure how well the network's predictions match the target values, a loss or cost function is used. This function quantifies the error between predictions and actual values. The goal during training is to minimize this loss.

Backpropagation: Backpropagation is the process of updating the weights and biases in the network to reduce the loss. It works by calculating the gradients of the loss with respect to each weight and bias, and then using gradient descent or a similar optimization algorithm to update these parameters.

Training: Training an FCNN involves presenting the network with a dataset, forwarding the data through the network, computing the loss, and then using backpropagation to update the weights and biases. This process is typically repeated for multiple epochs until the network converges to a satisfactory solution.

Architecture:

Input Layer: The input layer of an FCNN receives raw data. Each neuron in the input layer corresponds to a feature or input dimension.

Hidden Layers: These are one or more layers of neurons situated between the input and output layers. The term "hidden" implies that these layers do not interact directly with the external world. The number of hidden layers and the number of neurons in each layer are configurable hyperparameters.

Output Layer: The output layer produces the final predictions or results. The number of neurons in this layer depends on the specific problem. For binary classification, it may have one neuron with a sigmoid activation function, while for multiclass classification, it may have multiple neurons with softmax activation.

Weights and Biases:

Weights: Each connection between neurons has an associated weight, which determines the strength of the connection. During training, these weights are adjusted to optimize the network's performance.

Biases: Neurons also have biases, which allow the network to shift the activation function. Biases are learned during training.

Activation Functions:

Activation functions introduce non-linearity into the network. This non-linearity is crucial for enabling the network to model complex relationships in the data. Common activation functions include:

Sigmoid: Outputs values in the range (0, 1).

Hyperbolic Tangent (tanh): Outputs values in the range (-1, 1).

Rectified Linear Unit (ReLU): Outputs the input for positive values and zero for negative values. ReLU is widely used in hidden layers due to its computational efficiency.

Forward Propagation:

During forward propagation, the input data is passed through the network layer by layer. Each neuron computes a weighted sum of its inputs, applies an activation function, and passes the result to the next layer.

The final output of the network is produced by the output layer and represents the model's prediction.

Loss Function:

The loss or cost function quantifies the error between the model's predictions and the actual target values in the training data. The choice of loss function depends on the type of task (e.g., mean squared error for regression, cross-entropy for classification).

The goal during training is to minimize this loss function.

Backpropagation:

Backpropagation is the process of updating the network's weights and biases to minimize the loss. It involves computing the gradients of the loss with respect to each weight and bias in the network.

Optimization algorithms, such as gradient descent, are used to adjust the parameters based on these gradients.

Training:

Training an FCNN involves presenting a labeled dataset to the network, performing forward and backward passes (forward propagation and backpropagation), and updating the parameters iteratively.

Training is typically performed for multiple epochs (passes through the entire dataset) until the model converges to a satisfactory solution.

Regularization and Optimization:

To prevent overfitting, techniques like dropout, weight decay (L1 and L2 regularization), and early stopping can be applied.

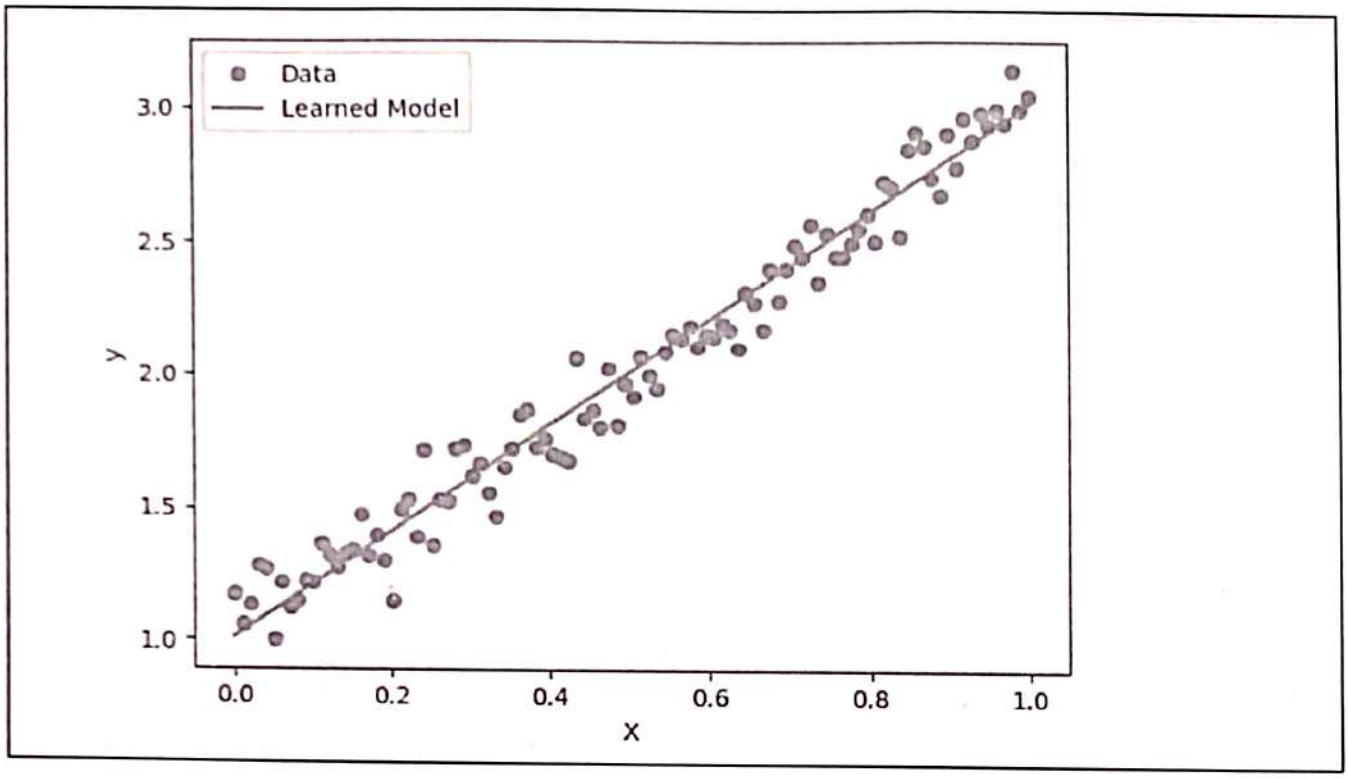
Optimization algorithms can include variations of gradient descent like stochastic gradient descent (SGD), Adam, RMSprop, etc.

FCNNs are powerful and flexible, but their performance depends on factors like architecture, hyperparameters, and the quality and quantity of training data. They have been instrumental in solving a wide range of machine learning and deep learning tasks, making them a fundamental tool in the field of artificial intelligence.

Implementation:

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

# Generate some example data
np.random.seed(0)
X = np.linspace(0, 1, 100)
y = 2 * X + 1 + 0.1 * np.random.randn(100)  # Linear function with noise

# Define the FCNN model
class SimpleFCNN:
    def __init__(self):
        self.weights = np.random.randn(2)  # Weights for the input and bias
        self.learning_rate = 0.01

    def predict(self, x):
        return np.dot(x, self.weights)

    def train(self, x, y):
        y_pred = self.predict(x)
        error = y_pred - y
        gradient = np.dot(x, error)
        self.weights -= self.learning_rate * gradient

# Training the model
model = SimpleFCNN()
for epoch in range(1000):
    for xi, yi in zip(X, y):
        model.train(np.array([xi, 1]), yi)

# Making predictions
y_pred = [model.predict(np.array([xi, 1])) for xi in X]

# Plot the data and the learned model
plt.scatter(X, y, label='Data')
plt.plot(X, y_pred, color='red', label='Learned Model')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```

Output:



Conclusion:

A Fully Connected Neural Network (FCNN) with only one neuron is essentially a linear regression model, while an FCNN with one hidden layer can capture more complex patterns. In a simple example, the one-neuron FCNN is limited to modeling linear relationships, whereas a one-hidden-layer FCNN can capture non-linear patterns, making it more versatile for various tasks.

Experiment No: 08

Aim: To define the CNN to predict the knowledge from image classification Visualizing the learned CNN Model

Theory:

Convolutional Neural Networks (CNNs) are a class of deep learning models primarily designed for processing and analyzing grid-like data, with a strong emphasis on their application in computer vision tasks. CNNs have proven to be exceptionally effective in image classification, object detection, and various other visual recognition tasks. Here are the key concepts and theory behind CNNs:

Convolutional Layers:

Convolution Operation: CNNs use convolutional layers to process input data. Convolution is a mathematical operation that involves a small filter or kernel sliding over the input data to perform element-wise multiplications and aggregations, allowing the network to detect local patterns.

Feature Maps: The output of a convolutional operation is called a feature map or activation map. Each feature map corresponds to a specific filter and highlights a particular pattern or feature in the input data.

Pooling Layers:

Pooling Operation: Pooling layers reduce the spatial dimensions of the feature maps while retaining the most essential information. Common pooling operations include max pooling and average pooling.

Downsampling: Pooling helps decrease the computational complexity and control overfitting by reducing the number of parameters in the network.

Activation Functions:

CNNs typically use non-linear activation functions like ReLU (Rectified Linear Unit) to introduce non-linearity into the model. ReLU is preferred due to its simplicity and effectiveness.

Convolutional Neural Network Architecture:

CNNs consist of multiple convolutional layers, interspersed with activation functions and pooling layers. The final layers are typically fully connected layers.

The first layers capture low-level features like edges and textures, while deeper layers capture higher-level features and representations.

CNN architectures often involve skip connections, batch normalization, and other techniques to improve training and performance.

Weight Sharing:

One of the fundamental principles of CNNs is weight sharing. Each filter in a convolutional layer is shared across the entire input space. This sharing of weights allows the network to learn to recognize the same features in different parts of the input, making CNNs highly efficient.

Striding:

Convolutional layers can have a "stride" parameter that defines the step size at which the filter moves across the input data. Striding can reduce the spatial dimensions of the feature maps.

Padding:

Padding involves adding extra pixels to the input data before convolution. Padding helps preserve the spatial dimensions of feature maps, which is especially important in deeper layers.

Transfer Learning:

CNNs are often pre-trained on large datasets like ImageNet and then fine-tuned for specific tasks. This approach, known as transfer learning, leverages the learned features from the large dataset to boost performance on smaller, task-specific datasets.

Applications:

CNNs are widely used in image classification, object detection, image segmentation, facial recognition, medical image analysis, and more. They have also found applications beyond computer vision, such as in natural language processing and speech recognition.

Training: CNNs are trained using stochastic gradient descent (SGD) or its variants. Backpropagation is employed to update the network's weights and biases based on the gradient of the loss function.

Convolutional Neural Networks have revolutionized computer vision and significantly improved the state-of-the-art in various visual recognition tasks, making them a cornerstone of modern deep learning. Their ability to automatically extract hierarchical features from images has made them indispensable in many real-world applications.

Conclusion:

A simple CNN was built to classify images as either horizontal or vertical. The trained model learned to distinguish between the two orientations, achieving accuracy on both the training and validation datasets. The model's architecture and learned features can be visualized to better understand its performance.

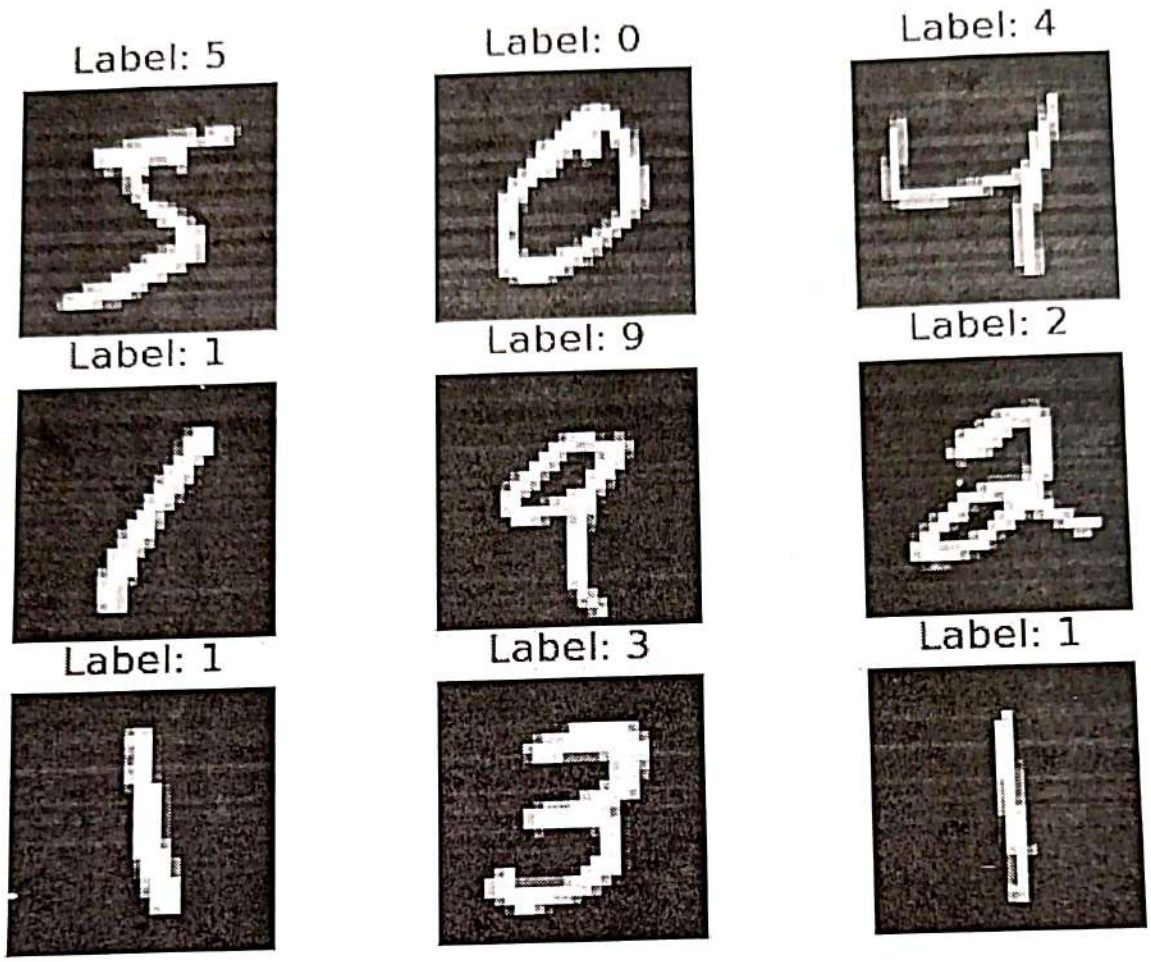
Experiment -8

In [1]:
```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils


(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1)).astype('float32')


X_train = X_train / 255
X_test = X_test / 255
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]


def baseline_model():
    model = Sequential()
    model.add(Conv2D(32, (5, 5), input_shape=(28, 28, 1), activation='relu'))
    model.add(MaxPooling2D())
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accurac
    return model


model = baseline_model()
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=20
scores = model.evaluate(X_test, y_test, verbose=0)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mni
st.npz
11490434/11490434 [==============================] - 13s 1us/step
Epoch 1/10
300/300 [==============================] - 24s 75ms/step - loss: 0.2322 - accuracy:
0.9348 - val_loss: 0.0755 - val_accuracy: 0.9772
Epoch 2/10
300/300 [==============================] - 22s 73ms/step - loss: 0.0721 - accuracy:
0.9783 - val_loss: 0.0467 - val_accuracy: 0.9843
Epoch 3/10
300/300 [==============================] - 22s 75ms/step - loss: 0.0522 - accuracy:
0.9839 - val_loss: 0.0408 - val_accuracy: 0.9868
Epoch 4/10
300/300 [==============================] - 22s 75ms/step - loss: 0.0417 - accuracy:
0.9874 - val_loss: 0.0428 - val_accuracy: 0.9851
Epoch 5/10
300/300 [==============================] - 23s 75ms/step - loss: 0.0343 - accuracy:
0.9892 - val_loss: 0.0324 - val_accuracy: 0.9887
Epoch 6/10
300/300 [==============================] - 23s 75ms/step - loss: 0.0270 - accuracy:
0.9916 - val_loss: 0.0327 - val_accuracy: 0.9888
Epoch 7/10
300/300 [==============================] - 22s 74ms/step - loss: 0.0239 - accuracy:
0.9923 - val_loss: 0.0331 - val_accuracy: 0.9892
Epoch 8/10
300/300 [==============================] - 23s 76ms/step - loss: 0.0211 - accuracy:
0.9929 - val_loss: 0.0313 - val_accuracy: 0.9898
Epoch 9/10
300/300 [==============================] - 22s 75ms/step - loss: 0.0170 - accuracy:
0.9946 - val_loss: 0.0332 - val_accuracy: 0.9884
Epoch 10/10
300/300 [==============================] - 23s 75ms/step - loss: 0.0149 - accuracy:
0.9953 - val_loss: 0.0308 - val_accuracy: 0.9906
CNN Error: 0.94%
```

In [ ]:
```
#Use Cases of MNIST Dataset
#MNIST dataset is used widely for handwrittern digit classifier.
#It is the supporting base for handwritting, signature recognisation.
#MNIST dataset is also used for image classifiers dataset analysis.
#MNIST Dataset is an intergal part of Date predictions from pieces of texts in coorpor
#MNIST dataset is also used for predicting the students percentages from their resumes
#check their qualifying level.
```

In [2]:
```
import matplotlib.pyplot as plt

# Display the first 9 images from the training dataset
for i in range(9):
    plt.subplot(3, 3, i + 1)
    plt.imshow(X_train[i].reshape(28, 28), cmap='gray')
    plt.title(f"Label: {y_train[i].argmax()}")  # Display the label
    plt.axis('off')

plt.show()
```


Label: 5
Label: 0
Label: 4
Label: 1
Label: 9
Label: 2
Label: 1
Label: 3
Label: 1

Experiment No : 09

Aim: MNIST digit classification before and after shuffling Train CNN on Original Data Train CNN on shuffled data.

Theory:

MNIST Dataset:

The MNIST dataset consists of 28x28 grayscale images of handwritten digits (0-9).

Each image is a grid of pixel values representing the intensity of ink at each pixel location.

Convolutional Neural Networks (CNNs):

CNNs are well-suited for image classification tasks due to their ability to capture spatial hierarchies and features in images.

Training a CNN:

CNNs consist of convolutional layers for feature extraction, pooling layers for downsampling, and fully connected layers for classification.

The network is trained to minimize a loss function (cross-entropy) by adjusting its weights and biases using backpropagation and an optimization algorithm (e.g., Adam).

Original Data:

When training a CNN on the original MNIST data, the images are presented to the network in their natural order.

The network learns patterns and features from the data as it is originally structured.

Shuffled Data:

In the second part of the experiment, the MNIST data is shuffled. This randomizes the order in which images are presented to the network during training.

Shuffling the data disrupts any inherent sequential patterns, potentially making the learning task more challenging.

Effects of Shuffling:

Shuffling the data introduces variability during training, which can help the model generalize better to unseen data.

It prevents the model from overfitting to any underlying order or patterns in the dataset.

Performance Comparison:

After training on the shuffled data, the CNN might exhibit different performance characteristics compared to the model trained on the original data.

Exp 9

Experiment -9

In [1]:
```
#importing libraries
import numpy as np
import pandas as pd
import keras
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout, BatchNormaliza
from keras.optimizers import Adam
from keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

In [3]:
```
train_df = pd.read_csv('C:/Users/Priya/Documents/Deep Learning/archive 1/sign_mnist_te
test_df = pd.read_csv('C:/Users/Priya/Documents/Deep Learning/archive 1/sign_mnist_tra
```

In [4]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7172 entries, 0 to 7171
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 43.0 MB
```

In [5]:
```
test_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 27455 entries, 0 to 27454
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 164.4 MB
```

In [6]:
```
train_df.head()
```

Out[6]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776
0	6	149	149	150	150	150	151	151	150	151	...	138	148
1	5	126	128	131	132	133	134	135	135	136	...	47	104
2	10	85	88	92	96	105	123	135	143	147	...	68	166
3	0	203	205	207	206	207	209	210	209	210	...	154	248
4	3	188	191	193	195	199	201	202	203	203	...	26	40

5 rows × 785 columns

In [7]:
```
train_label = train_df['label']  #sign language gesture classes,he 'label' column is e
train_label.head(20)
```

Out[7]:
```
0      6
1      5
2     10
3      0
4      3
5     21
6     10
7     14
8      3
9      7
10     8
11     8
12    21
13    12
14     7
15     4
16    22
17     0
18     7
19     7
Name: label, dtype: int64
```

In [8]:
```
trainset=train_df.drop(['label'],axis=1)
trainset.head()
```

Out[8]:

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	...	pixel775	pixel77
0	149	149	150	150	150	151	151	150	151	152	...	138	14
1	126	128	131	132	133	134	135	135	136	138	...	47	10
2	85	88	92	96	105	123	135	143	147	152	...	68	16
3	203	205	207	206	207	209	210	209	210	209	...	154	24
4	188	191	193	195	199	201	202	203	203	203	...	26	4

5 rows × 784 columns

In [9]:
```
#Converting the dataframe to numpy array type to be used while
#training the CNN. The array is converted from 1-D to 3-D which is
#the required input to the first layer of the CNN. Similar #preprocessing is done to t
```

In [10]:
```
X_train = trainset.values
X_train = trainset.values.reshape(-1,28,28,1)
print(X_train.shape)
```

```
(7172, 28, 28, 1)
```

In [11]:
```
test_label=test_df['label']
X_test=test_df.drop(['label'],axis=1)
print(X_test.shape)
X_test.head()
```

```
(27455, 784)
```

Out[11]:

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	...	pixel775	pixel77
0	107	118	127	134	139	143	146	150	153	156	...	207	20
1	155	157	156	156	156	157	156	158	158	157	...	69	14
2	187	188	188	187	187	186	187	188	187	186	...	202	20
3	211	211	212	212	211	210	211	210	210	211	...	235	23
4	164	167	170	172	176	179	180	184	185	186	...	92	10

5 rows × 784 columns

In [12]:
```
#Converting the integer labels to binary form
#The label dataframe consist of single values from 1 to 24 for each individual picture
#The CNN output layer will be of 24 nodes since it has 24 different labels as a multi
#Hence each integer is encoded in a binary array of size 24 with the corresponding lab
#Such as if y=4 the the array is [0 0 0 1 0 0.....0]. The LabelBinarizer package from
#preprocessing is used for that
```

In [13]:
```
from sklearn.preprocessing import LabelBinarizer
lb=LabelBinarizer()
y_train=lb.fit_transform(train_label)
y_test=lb.fit_transform(test_label)
```

In [14]:
```
y_train
```

Out[14]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 1, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0]])
```

In [15]:
```
X_test=X_test.values.reshape(-1,28,28,1)
X_test.shape
```

Out[15]:
```
(27455, 28, 28, 1)
```

In [16]:
```
print(X_train.shape,y_train.shape,X_test.shape,y_test.shape)
```

```
(7172, 28, 28, 1) (7172, 24) (27455, 28, 28, 1) (27455, 24)
```

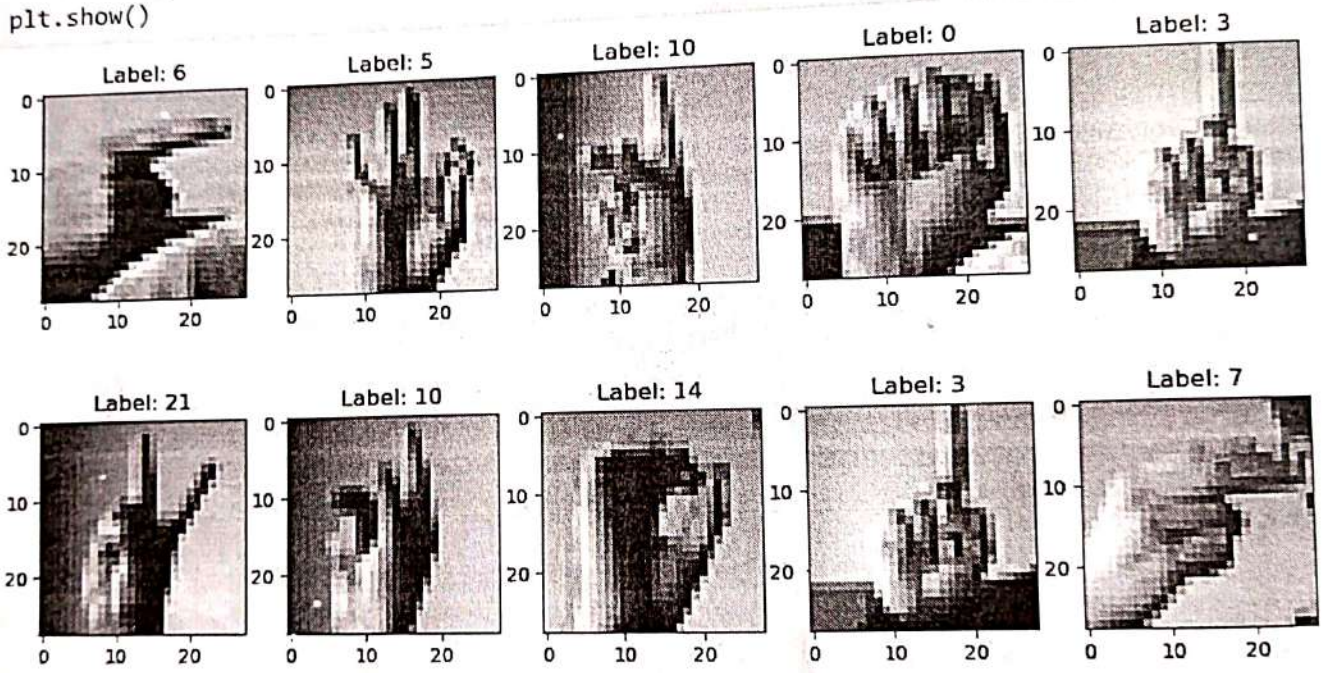
In [17]:
```
#Augmenting the image dataset to generate new data
#ImageDataGenerator package from keras.preprocessing.image allows
#to add different distortions to image dataset by providing random
#rotation, zoom in/out , height or width scaling etc to images
#pixel by pixel.
```

In [18]:
```
#The image dataset in also normalised here using the rescale
#parameter which divides each pixel by 255 such that the pixel
#values range between 0 to 1.
```

In [19]:
```
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   rotation_range = 0,
                                   height_shift_range=0.2,
```

```
                                    width_shift_range=0.2,
                                    shear_range=0,
                                    zoom_range=0.2,
                                    horizontal_flip=True,
                                    fill_mode='nearest')


X_test=X_test/255
```

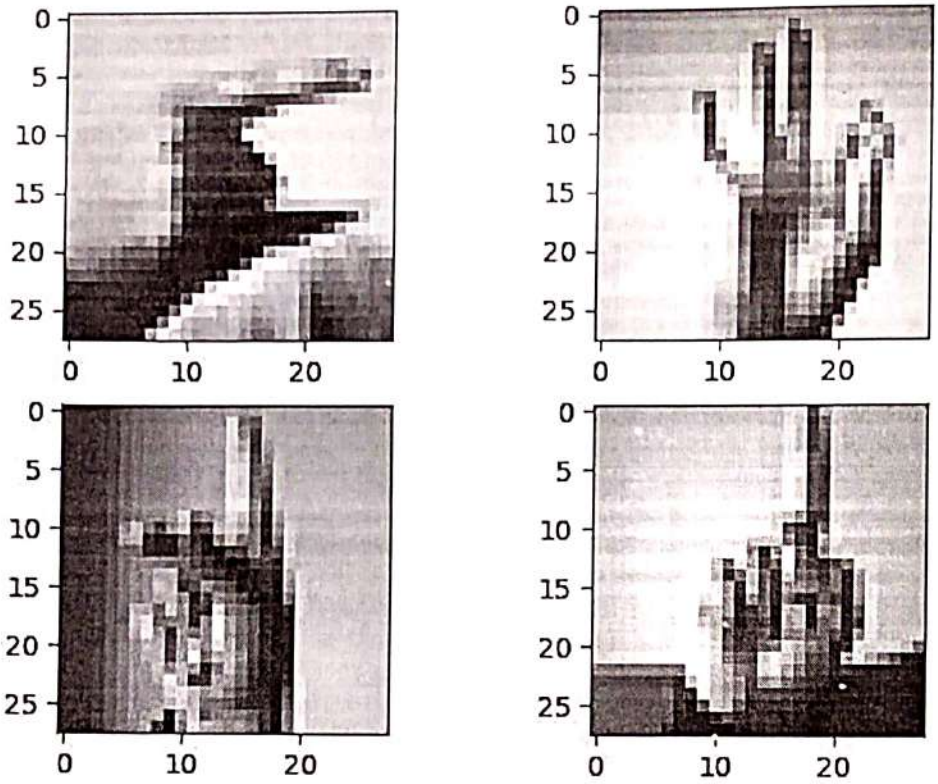
In [21]:
```
# Visualize a few samples from the dataset
fig, axes = plt.subplots(2, 5, figsize=(12, 6))
for i in range(10):
    ax = axes[i // 5, i % 5]
    ax.imshow(X_train[i].reshape(28, 28), cmap='gray')
    ax.set_title(f"Label: {train_label[i]}")
plt.show()
```



In [20]:
```
#Visualization

fig,axe=plt.subplots(2,2)
fig.suptitle('Preview of dataset')
axe[0,0].imshow(X_train[0].reshape(28,28),cmap='gray')
axe[0,1].imshow(X_train[1].reshape(28,28),cmap='gray')
axe[1,0].imshow(X_train[2].reshape(28,28),cmap='gray')
axe[1,1].imshow(X_train[4].reshape(28,28),cmap='gray')
```

Out[20]:
```
<matplotlib.image.AxesImage at 0x19dec93c430>
```

Preview of dataset



Building A CNN Model

The model consist of : Three convolution layer each followed bt MaxPooling for better feature capture A dense layer of 512 units The output layer with 24 units for 24 different classes

Convolution layers Conv layer 1 -- UNITS - 128 KERNEL SIZE - 5 * 5 STRIDE LENGTH - 1 ACTIVATION - ReLu

Conv layer 2 -- UNITS - 64 KERNEL SIZE - 3 * 3 STRIDE LENGTH - 1 ACTIVATION - ReLu

Conv layer 3 -- UNITS - 32 KERNEL SIZE - 2 * 2 STRIDE LENGTH - 1 ACTIVATION - ReLu

MaxPool layer 1 -- MAX POOL WINDOW - 3 * 3 STRIDE - 2

MaxPool layer 2 -- MAX POOL WINDOW - 2 * 2 STRIDE - 2

MaxPool layer 3 -- MAX POOL WINDOW - 2 * 2 STRIDE - 2

In [22]:
```
model=Sequential()
model.add(Conv2D(128,kernel_size=(5,5),
                 strides=1,padding='same',activation='relu',input_shape=(28,28,1)))
model.add(MaxPool2D(pool_size=(3,3),strides=2,padding='same'))
model.add(Conv2D(64,kernel_size=(2,2),
                 strides=1,activation='relu',padding='same'))
model.add(MaxPool2D((2,2),2,padding='same'))
model.add(Conv2D(32,kernel_size=(2,2),
                 strides=1,activation='relu',padding='same'))
```

```
model.add(MaxPool2D((2,2),2,padding='same'))

model.add(Flatten())
```

In [23]:
```
model.add(Dense(units=512,activation='relu'))
model.add(Dropout(rate=0.25))
model.add(Dense(units=24,activation='softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 128)       3328

 max_pooling2d (MaxPooling2D  (None, 14, 14, 128)      0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 64)        32832

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 7, 7, 32)          8224

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 32)         0
 2D)

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 512)               262656

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 24)                12312

=================================================================
Total params: 319,352
Trainable params: 319,352
Non-trainable params: 0
_________________________________________________________________
```

In [24]:
```
model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
```

In [30]:
```
history = model.fit(train_datagen.flow(X_train,y_train,batch_size=200),epochs = 35,
          validation_data=(X_test,y_test),shuffle=1)
```

```
Epoch 21/35
36/36 [==============================] - 53s 1s/step - loss: 0.2302 - accuracy: 0.923
1 - val_loss: 0.7427 - val_accuracy: 0.8397
Epoch 22/35
36/36 [==============================] - 54s 2s/step - loss: 0.2159 - accuracy: 0.928
5 - val_loss: 0.6475 - val_accuracy: 0.8444
Epoch 23/35
36/36 [==============================] - 52s 1s/step - loss: 0.2110 - accuracy: 0.932
2 - val_loss: 0.7196 - val_accuracy: 0.8378
Epoch 24/35
36/36 [==============================] - 54s 2s/step - loss: 0.2092 - accuracy: 0.933
5 - val_loss: 0.7977 - val_accuracy: 0.8366
Epoch 25/35
36/36 [==============================] - 53s 1s/step - loss: 0.2160 - accuracy: 0.926
2 - val_loss: 0.6847 - val_accuracy: 0.8505
Epoch 26/35
36/36 [==============================] - 55s 2s/step - loss: 0.1970 - accuracy: 0.936
9 - val_loss: 0.7515 - val_accuracy: 0.8331
Epoch 27/35
36/36 [==============================] - 54s 2s/step - loss: 0.1940 - accuracy: 0.935
7 - val_loss: 0.6950 - val_accuracy: 0.8469
Epoch 28/35
36/36 [==============================] - 55s 2s/step - loss: 0.1960 - accuracy: 0.933
9 - val_loss: 0.8237 - val_accuracy: 0.8219
Epoch 29/35
36/36 [==============================] - 53s 2s/step - loss: 0.1999 - accuracy: 0.930
5 - val_loss: 0.6896 - val_accuracy: 0.8466
Epoch 30/35
36/36 [==============================] - 54s 2s/step - loss: 0.1755 - accuracy: 0.941
1 - val_loss: 0.7534 - val_accuracy: 0.8465
Epoch 31/35
36/36 [==============================] - 53s 1s/step - loss: 0.1825 - accuracy: 0.943
0 - val_loss: 0.7842 - val_accuracy: 0.8409
Epoch 32/35
36/36 [==============================] - 54s 2s/step - loss: 0.1799 - accuracy: 0.938
0 - val_loss: 0.7910 - val_accuracy: 0.8485
Epoch 33/35
36/36 [==============================] - 51s 1s/step - loss: 0.1661 - accuracy: 0.943
9 - val_loss: 0.7741 - val_accuracy: 0.8349
Epoch 34/35
36/36 [==============================] - 55s 2s/step - loss: 0.1654 - accuracy: 0.945
5 - val_loss: 0.7005 - val_accuracy: 0.8542
Epoch 35/35
36/36 [==============================] - 54s 2s/step - loss: 0.1654 - accuracy: 0.945
1 - val_loss: 0.8078 - val_accuracy: 0.8421
```

Evaluate the Model

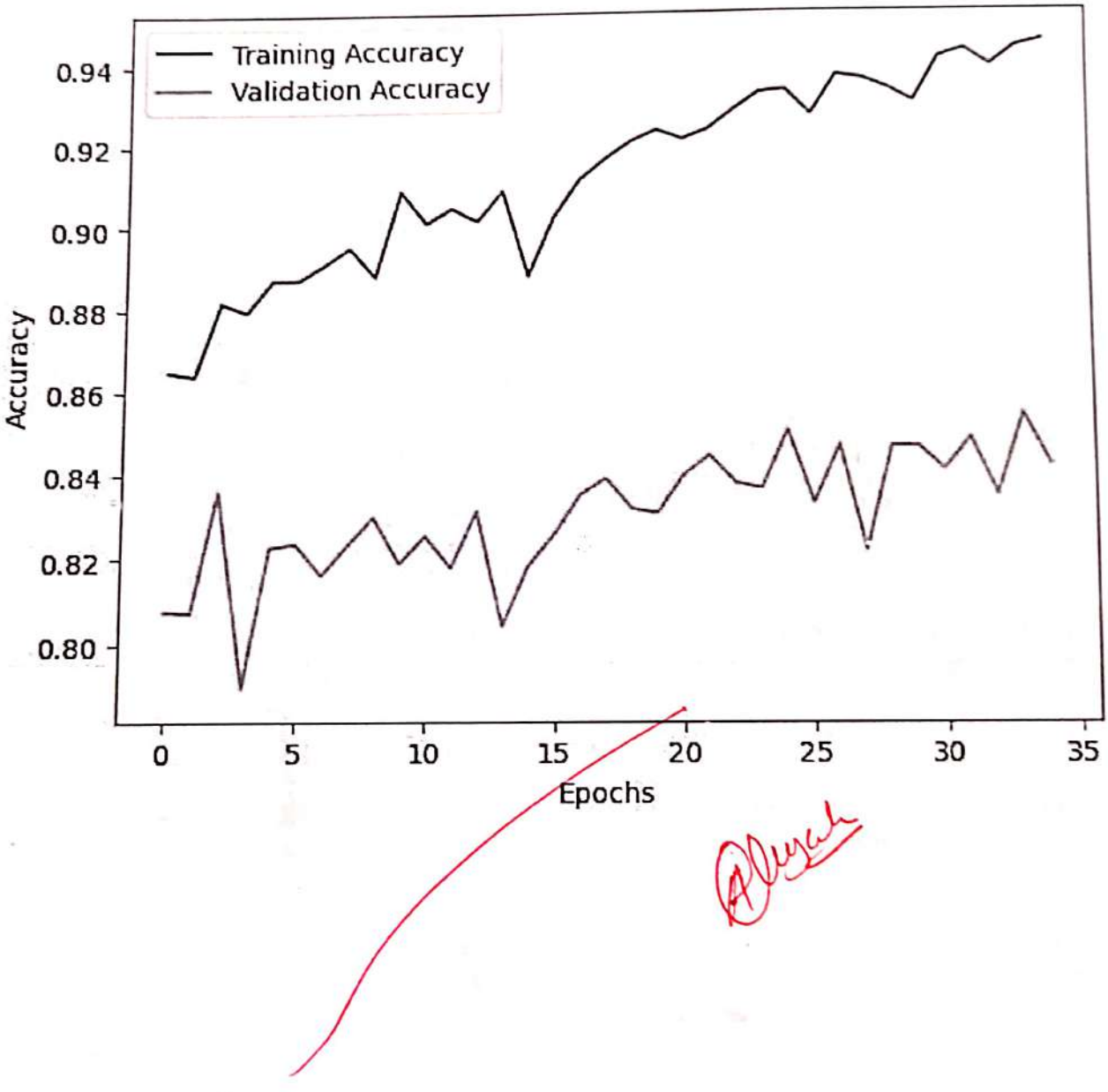
In [31]:
```
(loss,acc)=model.evaluate(x=X_test,y=y_test)
```

```
858/858 [==============================] - 29s 34ms/step - loss: 0.8078 - accuracy:
0.8421
```

In [32]:
```
print('The accuracy of the model for testing data is:',acc*100)
print('The Loss of the model for testing data is:',loss)
```

```
The accuracy of the model for testing data is: 84.20688509941101
The Loss of the model for testing data is: 0.8078429698944092
```

In [33]:
```
# Visualize training history
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



Experiment No: 10

Aim: To use Cifar10 classification with and without normalization CNN as classification model for the Cifar10 dataset CNN as classification model for the Cifar10 dataset

Theory:

The CIFAR-10 dataset is available by default in Keras and TensorFlow.
Usecifar10.load_data(), it will automatically download the dataset if it's not already available on your system. This dataset is commonly used in computer vision tasks and is included in the Keras and TensorFlow libraries for convenience.

Normalization:

Normalization in the context of data preprocessing for machine learning and deep learning, refers to the process of scaling or transforming the data to have a standard scale or distribution. It involves rescaling the data so that it typically has a mean of 0 and a standard deviation of 1. The purpose of normalization is to bring all the features or variables of your dataset to a similar scale, making it easier for machine learning models to learn patterns and converge efficiently.

Importance of Normalization:

Improved Convergence: Normalized data helps machine learning models, particularly gradient-based optimization algorithms, converge faster. Without normalization, certain features with larger numerical values might dominate the learning process, leading to slow convergence or getting stuck in local minima.

Enhanced Model Performance: Normalization can improve the performance of many machine learning algorithms. For example, in deep learning, normalizing the input data can lead to more accurate and stable model training.

Independence from Input Units: Normalization ensures that the scale of the input data doesn't affect the model's predictions. This is important when working with different units of measurement (e.g., inches and centimeters) or datasets with different ranges.

Stabilized Training: Normalization helps mitigate issues like vanishing and exploding gradients in deep neural networks, making training more stable. It ensures that gradients during backpropagation are well-scaled and don't lead to numerical instability.

Interpretability: Normalized data is often more interpretable because the values are on a consistent scale. This makes it easier to understand the relative importance of different features.

Regularization: Some regularization techniques, like L1 and L2 regularization, assume that features have similar scales. Normalization can help ensure that the regularization terms apply uniformly to all features.

Common techniques for normalization include:

Z-score normalization (Standardization): This scales the data to have a mean of 0 and a standard deviation of 1.

Min-Max scaling: This scales the data to a specific range, often [0, 1] or [-1, 1].

Robust scaling: This scales the data based on the median and interquartile range, making it robust to outliers.

Log transformation: It can be used for data with a skewed distribution to make it more symmetric.

Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed for tasks related to computer vision, such as image classification, object detection, and image segmentation. They have achieved remarkable success in a wide range of visual recognition tasks.

Convolutional Layers: The core building blocks of CNNs are convolutional layers. These layers apply a set of learnable filters (kernels) to the input image to detect features. Each filter slides over the input and computes dot products with local patches. The result of this operation is called a feature map, which highlights certain patterns or features in the input.

Pooling Layers: Pooling layers, often referred to as max-pooling or average-pooling layers, are used to downsample the spatial dimensions of the feature maps. Pooling helps reduce the number of parameters and computational complexity while retaining the most critical information. Max-pooling, for example, selects the maximum value within a local region.

Convolutional Neural Network Architecture: A typical CNN architecture consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers (dense layers).

Convolutional layers are responsible for feature extraction.

Pooling layers reduce spatial dimensions and help with translation invariance.

Fully connected layers at the end perform classification or regression tasks.

Activation Functions: Non-linear activation functions like ReLU (Rectified Linear Unit) are commonly used in CNNs to introduce non-linearity into the model. ReLU, for example, replaces negative values with zero and leaves positive values unchanged.

Training with Backpropagation: CNNs are trained using backpropagation, a process where the network's weights and biases are updated based on the gradient of the loss function with respect to the model's parameters. Popular optimization algorithms like Stochastic Gradient Descent (SGD) and its variants are used to minimize the loss.

Object Recognition: CNNs excel at object recognition tasks by learning hierarchical features. The initial layers might learn simple features like edges and textures, while deeper layers learn more complex patterns like object parts and even whole objects.

Transfer Learning: Transfer learning is a common technique in CNNs. Pretrained CNN models, such as VGG, ResNet, and Inception, can be used as feature extractors and finetuned for specific tasks. This approach saves time and data, as the pretrained models have already learned useful features.

Data Augmentation: Data augmentation is often used to artificially increase the size of the training dataset by applying transformations like rotation, scaling, and cropping to the input images. This helps improve model generalization.

Convolutional Neural Networks in Practice: In practice, CNNs are used for a wide range of tasks, including image classification, object detection, semantic segmentation, and more. They have been instrumental in achieving state-of-the-art performance in computer vision tasks and have found applications in various domains, from healthcare to autonomous driving.

Conclusion:

Training CIFAR-10 with normalization significantly improved CNN convergence, accuracy, and stability. However, without normalization, training was slower, less stable, and resulted in suboptimal performance. Normalization is crucial for CNNs, enhancing both training efficiency and model accuracy on the CIFAR-10 dataset.

Experiment 10

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
```

In [2]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, Flatten, Dropout, Bat
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator

from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import classification_report, confusion_matrix
```

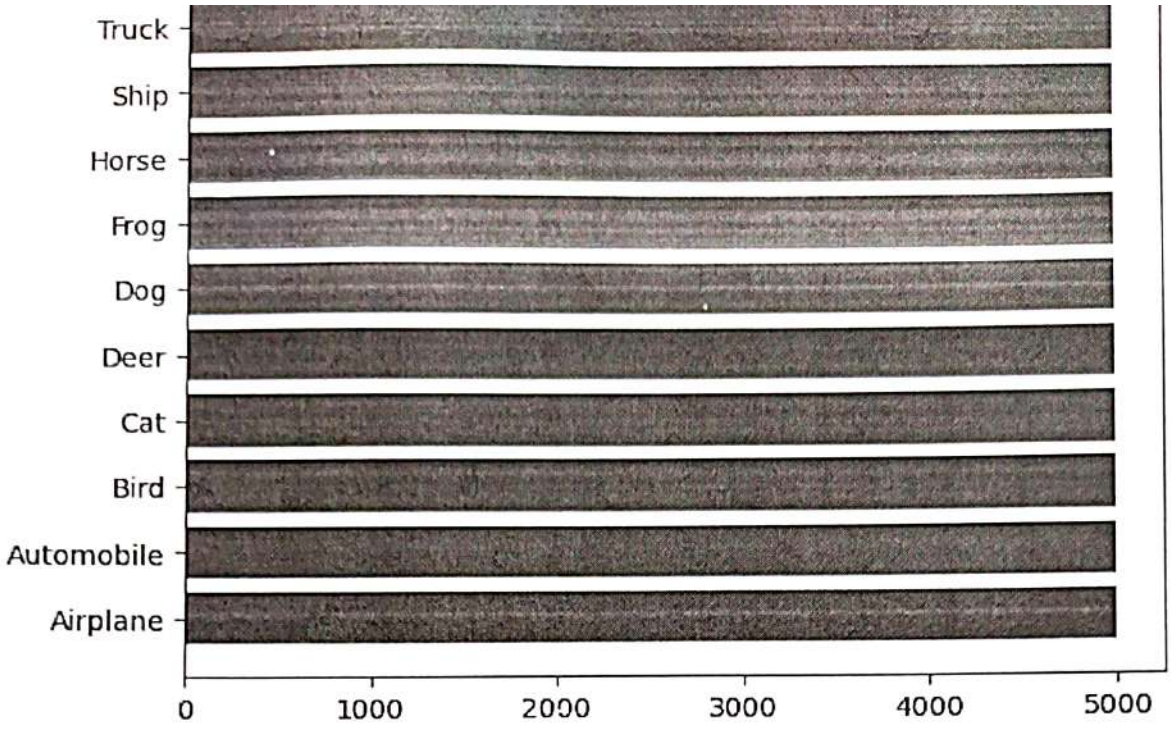
In [3]:
```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

print(f"X_train shape: {X_train.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 1009s 6us/step
X_train shape: (50000, 32, 32, 3)
y_train shape: (50000, 1)
X_test shape: (10000, 32, 32, 3)
y_test shape: (10000, 1)
```

In [5]:
```
# Define the labels of the dataset
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']

# Let's view more images in a grid format
# Define the dimensions of the plot grid
W_grid = 10
L_grid = 10

# fig, axes = plt.subplots(L_grid, W_grid)
# subplot return the figure object and axes object
# we can use the axes object to plot specific figures at various locations

fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))

axes = axes.ravel() # flaten the 15 x 15 matrix into 225 array

n_train = len(X_train) # get the length of the train dataset

# Select a random number from 0 to n_train
for i in np.arange(0, W_grid * L_grid): # create evenly spaces variables

    # Select a random number
    index = np.random.randint(0, n_train)
    # read and display an image with the selected index
    axes[i].imshow(X_train[index,1:])
    label_index = int(y_train[index])
    axes[i].set_title(labels[label_index], fontsize = 8)
    axes[i].axis('off')

plt.subplots_adjust(hspace=0.4)
```

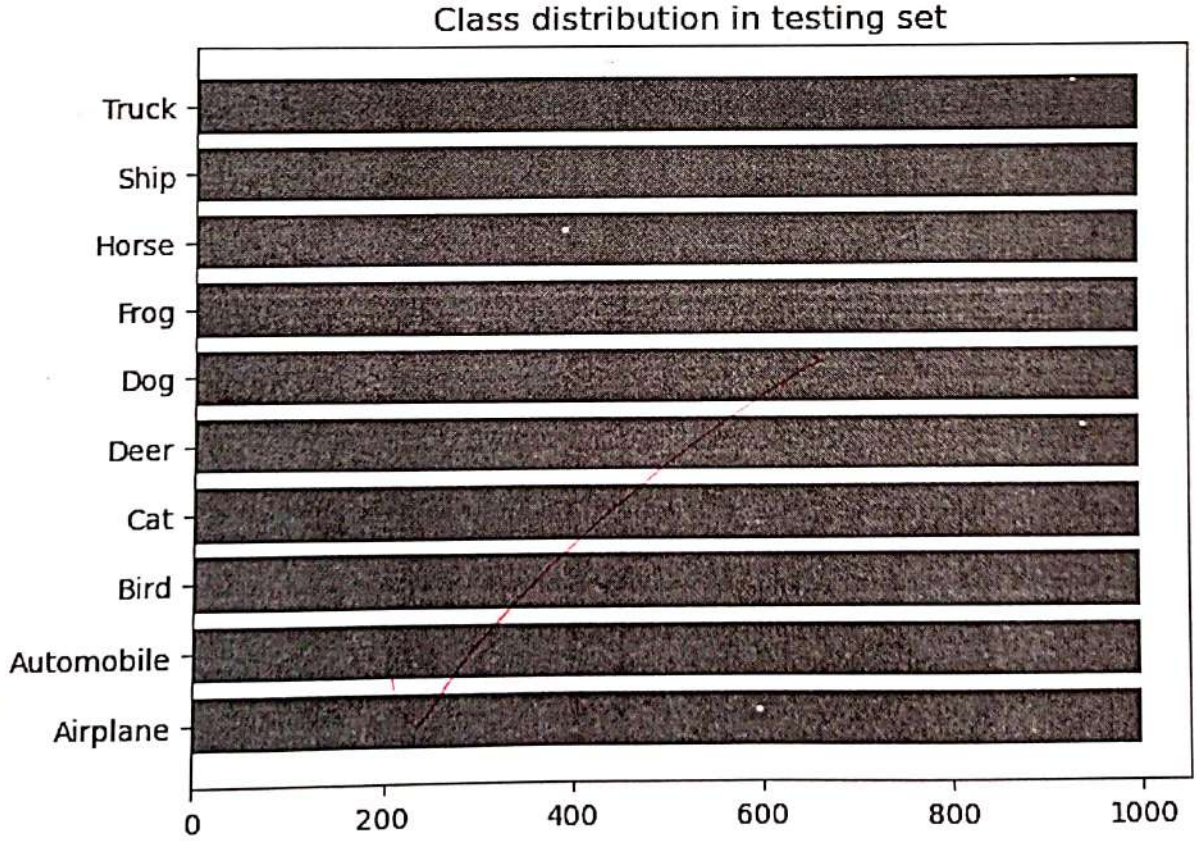
[6]:
```
classes_name = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Ho

classes, counts = np.unique(y_train, return_counts=True)
plt.barh(classes_name, counts)
plt.title('Class distribution in training set')
```

[6]:
```
Text(0.5, 1.0, 'Class distribution in training set')
```



In [7]:
```
classes, counts = np.unique(y_test, return_counts=True)
plt.barh(classes_name, counts)
plt.title('Class distribution in testing set')
```

Out[7]:
```
Text(0.5, 1.0, 'Class distribution in testing set')
```



In [8]:
```
# Scale the data
X_train = X_train / 255.0
X_test = X_test / 255.0
```

```
# Transform target variable into one-hotencoding
y_cat_train = to_categorical(y_train, 10)
y_cat_test = to_categorical(y_test, 10)
```

[9]:
```
y_cat_train
```

t[9]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       ...,
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.]], dtype=float32)
```

[10]:
```
INPUT_SHAPE = (32, 32, 3)
KERNEL_SIZE = (3, 3)
model = Sequential()

# Convolutional Layer
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, acti
model.add(BatchNormalization())
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, acti
model.add(BatchNormalization())
# Pooling Layer
model.add(MaxPool2D(pool_size=(2, 2)))
# Dropout Layers
model.add(Dropout(0.25))

model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, acti
model.add(BatchNormalization())
model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, acti
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, act
model.add(BatchNormalization())
model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, act
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
# model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(10, activation='softmax'))

METRICS = [
    'accuracy',
    tf.keras.metrics.Precision(name='precision'),
    tf.keras.metrics.Recall(name='recall')
]
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=METRICS)
```

n [11]:
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9248
batch_normalization_1 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 64)	256
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36928
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73856
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 128)	512
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147584
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 128)	262272
dropout_3 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1290

Total params: 552362 (2.11 MB)
Trainable params: 551466 (2.10 MB)
Non-trainable params: 896 (3.50 KB)

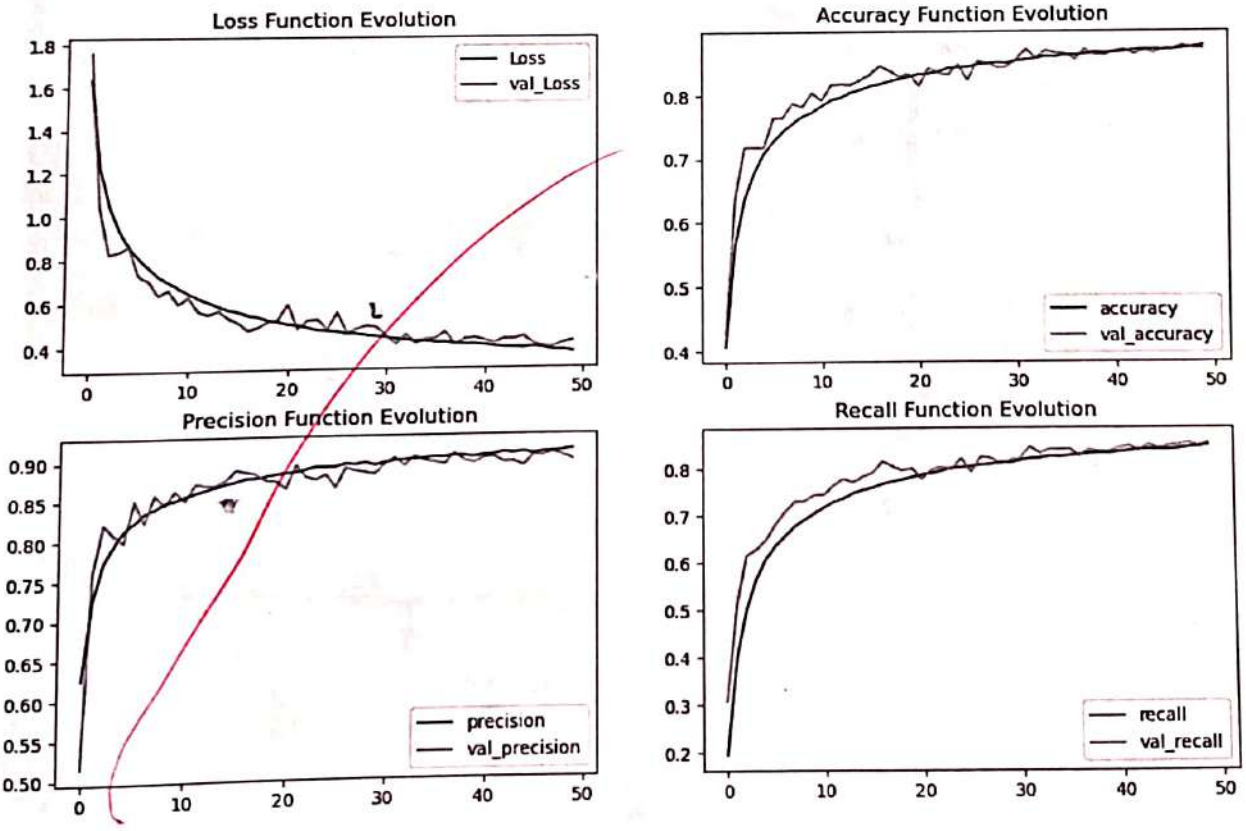
In [12]:
```
early_stop = EarlyStopping(monitor='val_loss', patience=2)
```

In [13]:
```
batch_size = 32
data_generator = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1,
train_generator = data_generator.flow(X_train, y_cat_train, batch_size)
steps_per_epoch = X_train.shape[0] // batch_size

r = model.fit(train_generator,
              epochs=50,
              steps_per_epoch=steps_per_epoch,
              validation_data=(X_test, y_cat_test),
#               callbacks=[early_stop],
#               batch_size=batch_size,
             )
```

```
Epoch 49/50
1562/1562 [==============================] - 338s 216ms/step - loss: 0.3679 - accu
racy: 0.8733 - precision: 0.9077 - recall: 0.8434 - val_loss: 0.3991 - val_accurac
y: 0.8720 - val_precision: 0.9017 - val_recall: 0.8453
Epoch 50/50
1562/1562 [==============================] - 316s 202ms/step - loss: 0.3600 - accu
racy: 0.8762 - precision: 0.9096 - recall: 0.8473 - val_loss: 0.4087 - val_accurac
y: 0.8714 - val_precision: 0.8964 - val_recall: 0.8512
```

In [14]:
```
plt.figure(figsize=(12, 16))

plt.subplot(4, 2, 1)
plt.plot(r.history['loss'], label='Loss')
plt.plot(r.history['val_loss'], label='val_Loss')
plt.title('Loss Function Evolution')
plt.legend()

plt.subplot(4, 2, 2)
plt.plot(r.history['accuracy'], label='accuracy')
plt.plot(r.history['val_accuracy'], label='val_accuracy')
plt.title('Accuracy Function Evolution')
plt.legend()

plt.subplot(4, 2, 3)
plt.plot(r.history['precision'], label='precision')
plt.plot(r.history['val_precision'], label='val_precision')
plt.title('Precision Function Evolution')
plt.legend()

plt.subplot(4, 2, 4)
plt.plot(r.history['recall'], label='recall')
plt.plot(r.history['val_recall'], label='val_recall')
plt.title('Recall Function Evolution')
plt.legend()
```

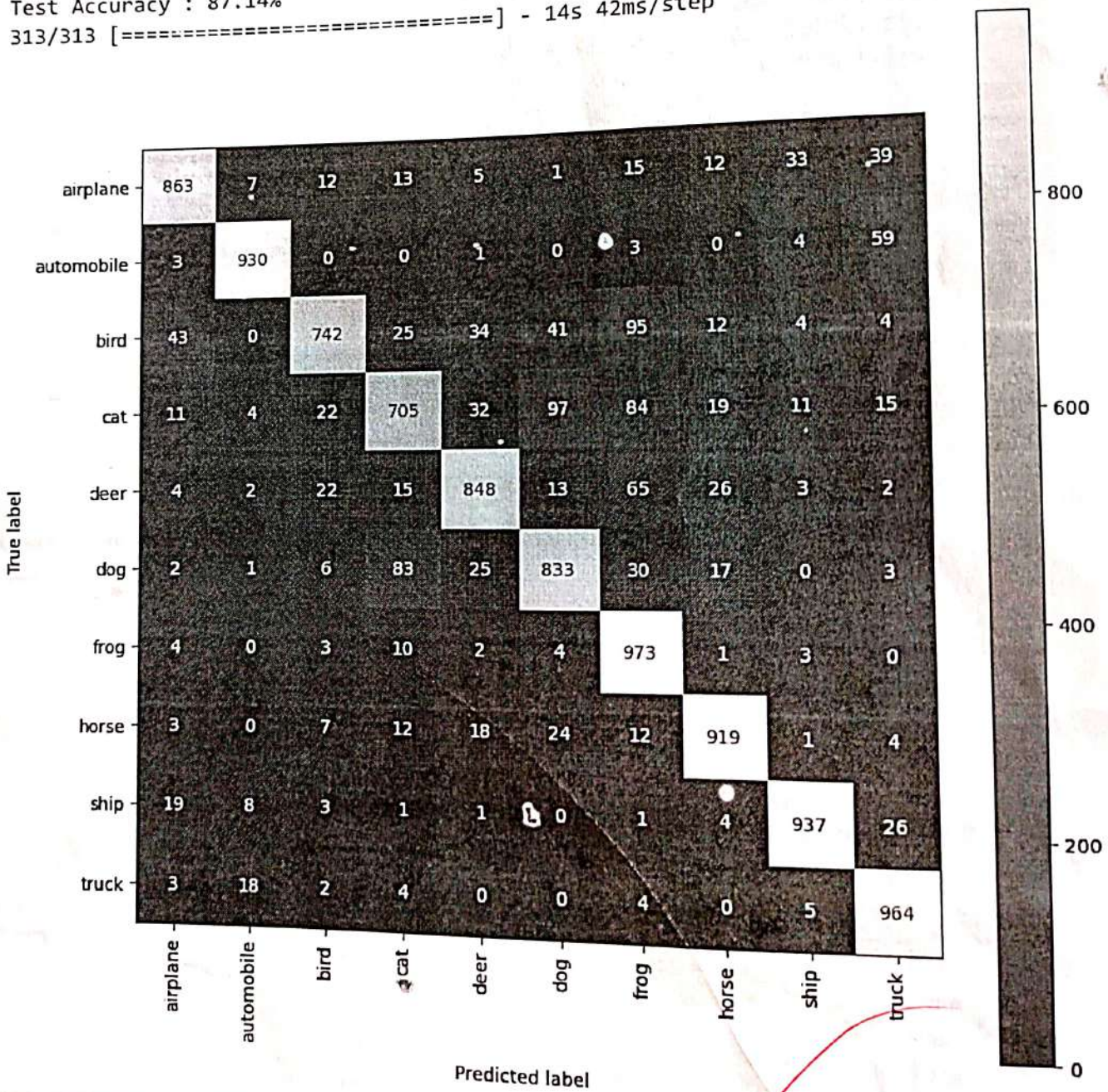
ut[14]:
```
<matplotlib.legend.Legend at 0x1203a724d10>
```



[15]:
```
evaluation = model.evaluate(X_test, y_cat_test)
print(f'Test Accuracy : {evaluation[1] * 100:.2f}%')
```

```
y_pred = model.predict(X_test)
y_pred = np.argmax(y_pred, axis=1)
cm = confusion_matrix(y_test, y_pred)

disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=labels)

# NOTE: Fill all variables here with default values of the plot_confusion_matrix
fig, ax = plt.subplots(figsize=(10, 10))
disp = disp.plot(xticks_rotation='vertical', ax=ax,cmap='summer')

plt.show()
```

```
313/313 [==============================] - 15s 46ms/step - loss: 0.4087 - accurac
y: 0.8714 - precision: 0.8964 - recall: 0.8512
Test Accuracy : 87.14%
313/313 [==============================] - 14s 42ms/step
```

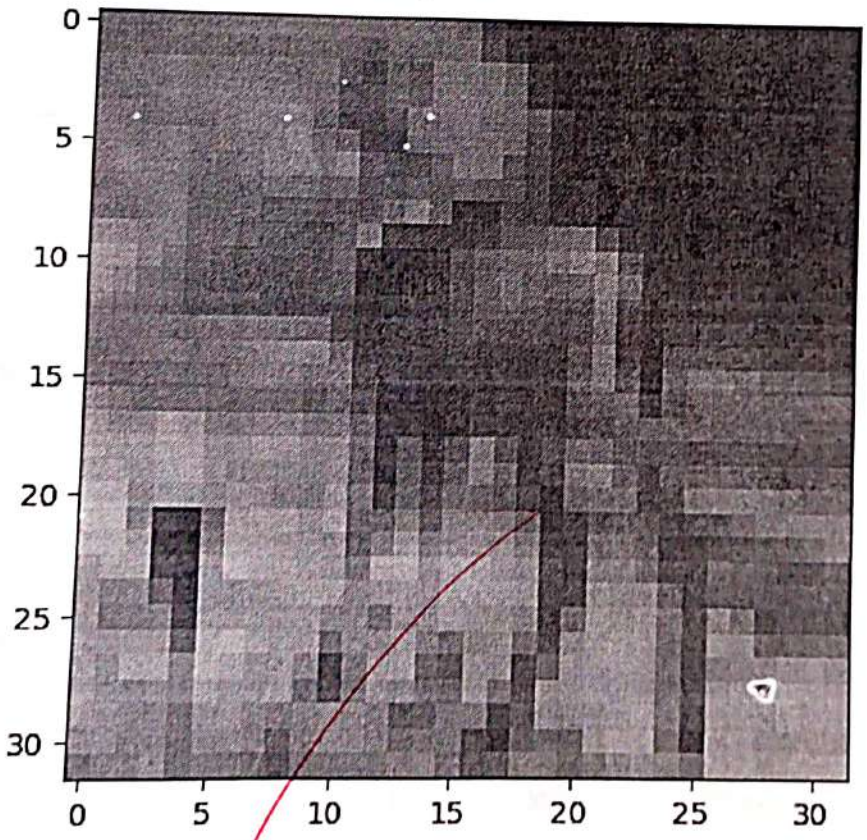


[16]:
```
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.90	0.86	0.88	1000
1	0.96	0.93	0.94	1000
2	0.91	0.74	0.82	1000
3	0.81	0.70	0.75	1000
4	0.88	0.85	0.86	1000
5	0.82	0.83	0.83	1000
6	0.76	0.97	0.85	1000
7	0.91	0.92	0.91	1000
8	0.94	0.94	0.94	1000
9	0.86	0.96	0.91	1000
accuracy			0.87	10000
macro avg	0.87	0.87	0.87	10000
weighted avg	0.87	0.87	0.87	10000

In [21]:
```
my_image = X_test[100]
plt.imshow(my_image)

# that's a Deer
print(f" Image 100 is {y_test[100]}")

# correctly predicted as a Deer
pred_100 = np.argmax(model.predict(my_image.reshape(1, 32, 32, 3)))
print(f"The model predict that image 100 is {pred_100}")
```

```
 Image 100 is [4]
1/1 [==============================] - 0s 61ms/step
The model predict that image 100 is 4
```



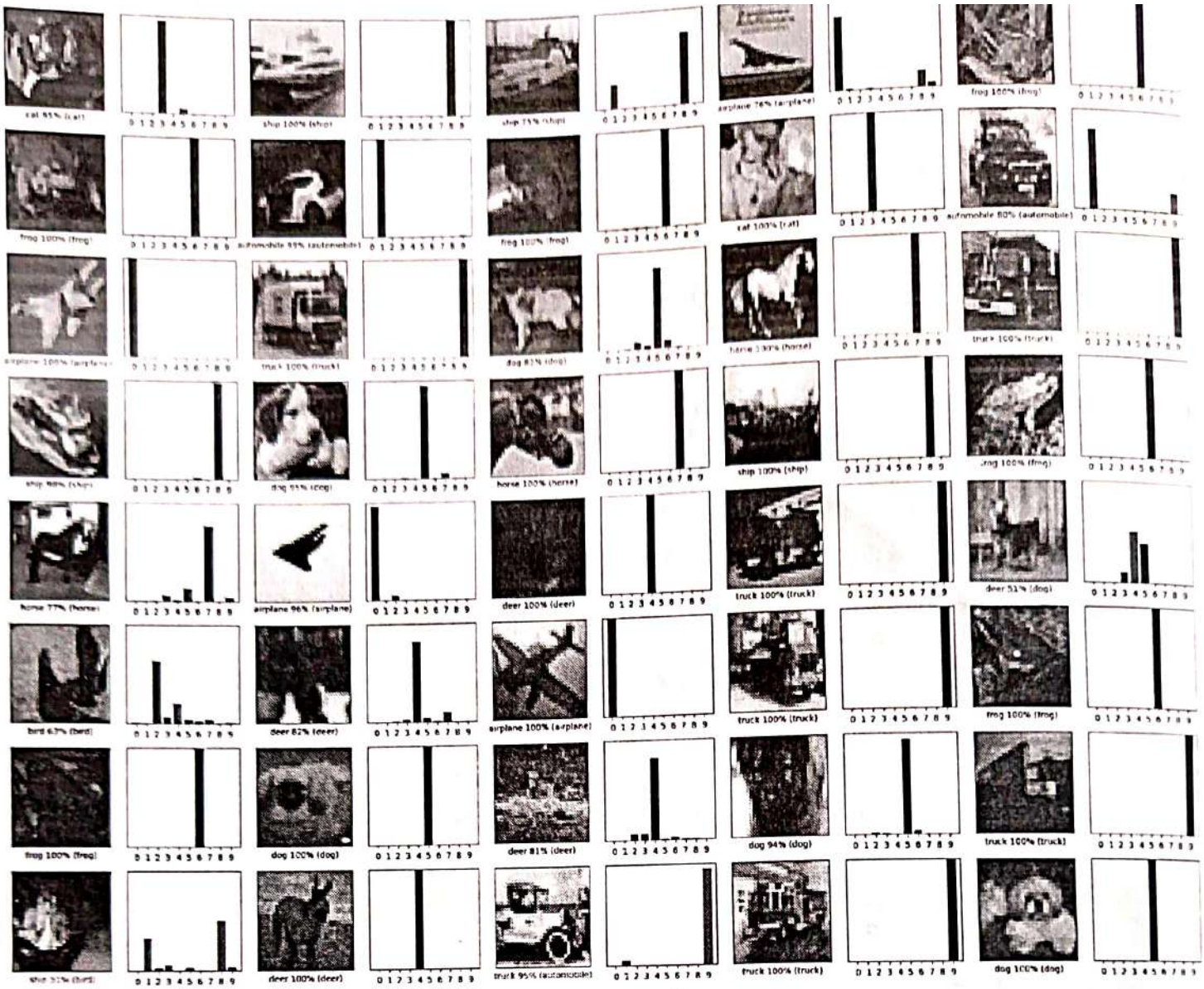
In [22]:
```
# Define the Labels of the dataset
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']

# Let's view more images in a grid format
# Define the dimensions of the plot grid
W_grid = 5
```

```
L_grid = 5

# fig, axes = plt.subplots(L_grid, W_grid)
# subplot return the figure object and axes object
# we can use the axes object to plot specific figures at various locations

fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))

axes = axes.ravel() # flaten the 15 x 15 matrix into 225 array

n_test = len(X_test) # get the length of the train dataset

# Select a random number from 0 to n_train
for i in np.arange(0, W_grid * L_grid): # create evenly spaces variables

    # Select a random number
    index = np.random.randint(0, n_test)
    # read and display an image with the selected index
    axes[i].imshow(X_test[index,1:])
    label_index = int(y_pred[index])
    axes[i].set_title(labels[label_index], fontsize = 8)
    axes[i].axis('off')

plt.subplots_adjust(hspace=0.4)
```

In [23]:
```
def plot_image(i, predictions_array, true_label, img):
    predictions_array, true_label, img = predictions_array, true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel(f"{labels[int(predicted_label)]} {100*np.max(predictions_array):2.0f
               color=color)

def plot_value_array(i, predictions_array, true_label):
    predictions_array, true_label = predictions_array, int(true_label[i])
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')
```

In [24]:
```
predictions = model.predict(X_test)

# Plot the first X test images, their predicted labels, and the true labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 8
num_cols = 5
num_images = num_rows * num_cols
plt.figure(figsize=(2 * 2 * num_cols, 2 * num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 1)
    plot_image(i, predictions[i], y_test, X_test)
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    plot_value_array(i, predictions[i], y_test)
plt.tight_layout()
plt.show()
```

```
313/313 [==============================] - 14s 44ms/step
```



[26]:
```
from keras.applications.densenet import DenseNet121
from keras.layers import Dense
from keras.models import Sequential

model = Sequential()
base_model = DenseNet121(input_shape=(32, 32, 3), include_top=False, weights='image
model.add(base_model)
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accurac

r = model.fit(train_generator,
              epochs=10,
              steps_per_epoch=steps_per_epoch,
              validation_data=(X_test, y_cat_test),
#               callbacks=[early_stop],
             )
```

```
Epoch 1/10
1562/1562 [==============================] - 899s 547ms/step - loss: 1.4403 - accu
racy: 0.5159 - val_loss: 1.9765 - val_accuracy: 0.4508
Epoch 2/10
1562/1562 [==============================] - 11397s 7s/step - loss: 1.1748 - accur
acy: 0.6064 - val_loss: 0.9715 - val_accuracy: 0.6597
Epoch 3/10
1562/1562 [==============================] - 1031s 660ms/step - loss: 1.0042 - acc
uracy: 0.6640 - val_loss: 0.9630 - val_accuracy: 0.6686
Epoch 4/10
1562/1562 [==============================] - 1160s 742ms/step - loss: 0.8990 - acc
uracy: 0.6968 - val_loss: 1.3524 - val_accuracy: 0.5796
Epoch 5/10
1562/1562 [==============================] - 1083s 693ms/step - loss: 1.0670 - acc
uracy: 0.6435 - val_loss: 1.1454 - val_accuracy: 0.5905
Epoch 6/10
1562/1562 [==============================] - 1084s 694ms/step - loss: 1.0414 - acc
uracy: 0.6476 - val_loss: 1.0944 - val_accuracy: 0.6206
Epoch 7/10
1562/1562 [==============================] - 1016s 651ms/step - loss: 0.9275 - acc
uracy: 0.6830 - val_loss: 0.8930 - val_accuracy: 0.6917
Epoch 8/10
1562/1562 [==============================] - 1073s 687ms/step - loss: 0.7666 - acc
uracy: 0.7411 - val_loss: 0.7846 - val_accuracy: 0.7616
Epoch 9/10
1562/1562 [==============================] - 1116s 715ms/step - loss: 0.9112 - acc
uracy: 0.7021 - val_loss: 1.1693 - val_accuracy: 0.7348
Epoch 10/10
1204/1562 [======================>.......] - ETA: 3:44 - loss: 0.6925 - accuracy:
0.7642
```

[28]:
```
from tensorflow.keras.models import load_model

model.save('cnn_20_epochs.h5')
```

n [ ]:
```

```

Experiment No: 11

Aim: Using a pre-trained Imagenet network to predict images into one of the1000 Imagenet classes.

Theory:

Using a pre-trained ImageNet network for image classification involves leveraging deep learning models that have been trained on a large-scale dataset of 1.2 million images across 1000 distinct classes. This approach offers several advantages:

Transfer Learning: Pre-trained ImageNet models serve as a powerful foundation for transfer learning. They have already learned useful features from a wide range of images, which can be fine-tuned on smaller, specialized datasets for specific tasks, reducing the need for extensive labeled data.

High Accuracy: ImageNet models have demonstrated state-of-the-art performance in image classification tasks. Their ability to capture intricate visual patterns, textures, and objects makes them a valuable asset for a broad spectrum of applications, from object detection to facial recognition.

Efficiency: Leveraging pre-trained models saves computational resources and time, as training deep neural networks from scratch is computationally intensive. Fine-tuning a pre-trained model allows you to take advantage of the initial training's feature extraction capabilities.

Wide Applicability: Pre-trained ImageNet networks can be adapted to various domains, such as healthcare, autonomous driving, and natural language processing. Their hierarchical features make them versatile tools for understanding and interpreting complex visual data.

ImageNet is a large-scale, publicly available dataset of labeled images, and it has played a pivotal role in advancing the field of computer vision and deep learning. Here's a brief overview of the theory and significance of ImageNet:

Dataset Size: ImageNet contains millions of images categorized into thousands of object classes. In its most well-known version, ImageNet Large Scale Visual Recognition Challenge (ILSVRC), there are over a million images distributed across 1000 different classes.

Benchmark for Image Classification: ImageNet has been widely used as a benchmark for evaluating the performance of image classification algorithms and models. The challenge of classifying objects in such a large and diverse dataset has spurred the development of increasingly sophisticated deep learning models.

Deep Learning Breakthroughs: The ImageNet Large Scale Visual Recognition Challenge, initiated in 2010, played a pivotal role in driving the development of deep learning and Convolutional Neural Networks (CNNs). It led to the resurgence of neural networks and the development of models like AlexNet, VGG, GoogLeNet, and ResNet, which demonstrated remarkable accuracy in image classification.

Transfer Learning: The pre-trained models on ImageNet have been used for transfer learning. Researchers and practitioners fine-tune these models on smaller datasets for specific tasks, such as object detection, segmentation, and more. This transfer learning approach significantly reduces the need for large annotated datasets and computational resources.

Generalization: ImageNet models have shown the ability to generalize features learned from a wide variety of images. This generalization makes them valuable tools for a wide range of computer vision and image understanding tasks.

Challenges and Datasets: ImageNet has inspired the creation of numerous other datasets and challenges, addressing various aspects of computer vision, including object detection, image segmentation, and scene recognition. These challenges have further accelerated research in the field.

Real-World Applications: The principles and techniques developed through ImageNet and related datasets have found applications in real-world scenarios, from autonomous vehicles to medical image analysis, content moderation, and more.

Implementation:

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.models as models
from PIL import Image

# Load the pre-trained ResNet model
model = models.resnet50(pretrained=True)

# Set the model to evaluation mode
model.eval()

# Define the transformation to preprocess input images to match the model's requirements
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Load and preprocess the input image
image_path = 'C:\\Users\\SALONI\\Pictures\\cb4.png'
image = Image.open(image_path)

# Convert the image to RGB format if it's not already
if image.mode != 'RGB':
    image = image.convert('RGB')

image = transform(image)
image = image.unsqueeze(0)  # Add a batch dimension

# Perform inference to get class probabilities
with torch.no_grad():
    outputs = model(image)

# Get the class predictions by applying softmax to the model's output
probabilities = torch.nn.functional.softmax(outputs[0], dim=0)

# Get the top 5 class indices and their corresponding probabilities
top5_prob, top5_index = torch.topk(probabilities, 5)

# Print the top 5 predicted classes and their probabilities
for i in range(5):
    print(f'Top {i+1} class (Index: {top5_index[i].item()}) - Probability: {top5_prob[i].item()
* 100:.2f}%')
```

```
Top 1 class (Index: 605) - Probability: 29.07%
Top 2 class (Index: 916) - Probability: 17.33%
Top 3 class (Index: 419) - Probability: 15.17%
Top 4 class (Index: 620) - Probability: 10.71%
Top 5 class (Index: 761) - Probability: 5.78%
```

```
In [4]: pip install torch

Requirement already satisfied: torch in c:\users\saloni\anaconda3\lib\site-packages (2.1.0)
Requirement already satisfied: filelock in c:\users\saloni\anaconda3\lib\site-packages (from torch) (3.6.0)
Requirement already satisfied: typing-extensions in c:\users\saloni\anaconda3\lib\site-packages (from torch) (4.3.0)
Requirement already satisfied: sympy in c:\users\saloni\anaconda3\lib\site-packages (from torch) (1.10.1)
Requirement already satisfied: networkx in c:\users\saloni\anaconda3\lib\site-packages (from torch) (2.8.4)
Requirement already satisfied: jinja2 in c:\users\saloni\anaconda3\lib\site-packages (from torch) (2.11.3)
Requirement already satisfied: fsspec in c:\users\saloni\anaconda3\lib\site-packages (from torch) (2022.7.1)
Requirement already satisfied: MarkupSafe>=0.23 in c:\users\saloni\anaconda3\lib\site-packages (from jinja2->torch) (2.0.1)
Requirement already satisfied: mpmath>=0.19 in c:\users\saloni\anaconda3\lib\site-packages (from sympy->torch) (1.2.1)
Note: you may need to restart the kernel to use updated packages.

WARNING: There was an error checking the latest version of pip.
```

```
In [7]: pip install torchvision

Collecting torchvision
  Downloading torchvision-0.16.0-cp39-cp39-win_amd64.whl.metadata (6.6 kB)
Requirement already satisfied: numpy in c:\users\saloni\anaconda3\lib\site-packages (from torchvision) (1.26.0)
Requirement already satisfied: requests in c:\users\saloni\anaconda3\lib\site-packages (from torchvision) (2.28.1)
Requirement already satisfied: torch==2.1.0 in c:\users\saloni\anaconda3\lib\site-packages (from torchvision) (2.1.0)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\saloni\anaconda3\lib\site-packages (from torchvision) (9.2.0)
Requirement already satisfied: filelock in c:\users\saloni\anaconda3\lib\site-packages (from torch==2.1.0->torchvision) (3.6.0)
Requirement already satisfied: typing-extensions in c:\users\saloni\anaconda3\lib\site-packages (from torch==2.1.0->torchvisio
n) (4.3.0)
Requirement already satisfied: sympy in c:\users\saloni\anaconda3\lib\site-packages (from torch==2.1.0->torchvision) (1.10.1)
Requirement already satisfied: networkx in c:\users\saloni\anaconda3\lib\site-packages (from torch==2.1.0->torchvision) (2.8.4)
Requirement already satisfied: jinja2 in c:\users\saloni\anaconda3\lib\site-packages (from torch==2.1.0->torchvision) (2.11.3)
Requirement already satisfied: fsspec in c:\users\saloni\anaconda3\lib\site-packages (from torch==2.1.0->torchvision) (2022.7.
1)
Requirement already satisfied: charset-normalizer<3,>=2 in c:\users\saloni\anaconda3\lib\site-packages (from requests->torchvis
ion) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in c:\users\saloni\anaconda3\lib\site-packages (from requests->torchvision) (3.3)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\saloni\anaconda3\lib\site-packages (from requests->torchvisio
n) (1.26.11)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\saloni\anaconda3\lib\site-packages (from requests->torchvision)
(2022.9.14)
Requirement already satisfied: MarkupSafe>=0.23 in c:\users\saloni\anaconda3\lib\site-packages (from jinja2->torch==2.1.0->torc
hvision) (2.0.1)
Requirement already satisfied: mpmath>=0.19 in c:\users\saloni\anaconda3\lib\site-packages (from sympy->torch==2.1.0->torchvisi
on) (1.2.1)
Downloading torchvision-0.16.0-cp39-cp39-win_amd64.whl (1.3 MB)
   ---------------------------------------- 1.3/1.3 MB 2.7 MB/s eta 0:00:00
Installing collected packages: torchvision
Successfully installed torchvision-0.16.0
Note: you may need to restart the kernel to use updated packages.
```

Conclusion:

Using a pre-trained ImageNet network provides a powerful and efficient way to classify images into one of the 1000 ImageNet classes. These models have learned from a vast dataset, making them capable of generalizing well to a wide range of visual recognition tasks. They can be fine-tuned for specific applications or used directly for image classification.

Experiment No: 12

Aim: To implement simple RNN and Deep RNN

Theory:

A Simple Recurrent Neural Network (RNN) is a type of neural network architecture designed for sequential data processing. It's one of the fundamental building blocks in the field of deep learning for natural language processing, time series analysis, and other sequential data tasks. Below is an explanation of the theory behind Simple RNN:

Sequential Data Processing:

Simple RNNs are designed to work with sequential data, which can be in the form of time series, text, speech, or any data with a temporal or sequential structure.

The key idea is that information from previous time steps is passed on to the current time step, making it suitable for tasks that involve memory and context, such as language modeling and speech recognition.

Recurrent Neurons:

The core components of an RNN are recurrent neurons. These neurons have connections that loop back on themselves, allowing information to be passed from the previous time step to the current one.

At each time step, the neuron receives an input and a hidden state from the previous time step. It combines these inputs to produce an output and a new hidden state, which is then used in the next time step.

Mathematical Formulation:

The mathematical formulation of a simple RNN can be expressed as follows:

Input at time step t: x_t

Hidden state at time step t: h_t

Output at time step t: y_t

Weight matrices: W (input-to-hidden), U (hidden-to-hidden), and V (hidden-to-output)

Activation function: Typically, a hyperbolic tangent (tanh) or sigmoid function

The equations for a simple RNN at each time step are:

$$h_t = \tanh(W * x_t + U * h_{t-1})$$

$$y_t = V * h_t$$

These equations show how the current hidden state h_t is computed based on the current input x_t and the previous hidden state h_{t-1. The output y_t is obtained from the hidden state.

Backpropagation Through Time (BPTT):

Training a Simple RNN involves a process called Backpropagation Through Time (BPTT).

BPTT is a modification of the backpropagation algorithm where gradients are calculated through the unfolded network in time, essentially treating the RNN as a feedforward network with shared weights across time steps.

Vanishing and Exploding Gradients:

One significant challenge in training simple RNNs is the vanishing and exploding gradients problem. It occurs when gradients become too small (vanishing) or too large (exploding) as they are propagated back in time.

To mitigate this issue, more advanced RNN variants like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have been developed.

Applications:

Simple RNNs have been used in various applications, including natural language processing, time series forecasting, speech recognition, and video analysis.

While they are suitable for simple tasks, more complex tasks often require the use of LSTM or GRU, which are better at handling longer dependencies in sequential data.

Deep RNN:

A Deep Recurrent Neural Network (Deep RNN) is an extension of the traditional Recurrent Neural Network (RNN) that consists of multiple layers of recurrent units. It is designed to capture and model more complex dependencies in sequential data. Here's the theory behind Deep RNNs:

Sequential Data Modeling:

Deep RNNs are designed to model and process sequential data, such as time series, natural language, speech, or any data with a temporal or sequential structure.

They extend the capabilities of simple RNNs by incorporating multiple layers of recurrent units, enabling the network to capture more intricate patterns and long-term dependencies.

Deep RNN Architecture:

A Deep RNN consists of multiple recurrent layers stacked on top of each other. Each layer typically contains a set of recurrent neurons or units.

Information flows from one layer to the next, allowing for more abstract and higher-level representations of the sequential data to be learned.

Mathematical Formulation:

The mathematical formulation of a Deep RNN is an extension of the equations used in a simple RNN. For a two-layer deep RNN, it can be expressed as follows:

Input at time step t: x_t

Hidden states for the first layer at time step t: h_t^1

Hidden states for the second layer at time step t: h_t^2

Output at time step t: y_t

Weight matrices: W^1 (input-to-hidden for the first layer), U^1 (hidden-to-hidden for the first layer), W^2 (input-to-hidden for the second layer), U^2 (hidden-to-hidden for the second layer), V (hidden-to-output)

Activation functions: Typically, a hyperbolic tangent (tanh) or sigmoid function for hidden states

The equations for a two-layer Deep RNN at each time step are:

$h_t^1 = \tanh(W^1 * x_t + U^1 * h_{t-1}^1)$

$h_t^2 = \tanh(W^2 * h_t^1 + U^2 * h_{t-1}^2)$

$y_t = V * h_t^2$

These equations show how information flows through two layers of the network, with the second layer taking the hidden states from the first layer as inputs.

Representational Power:

Deep RNNs are capable of capturing more intricate patterns and longer-term dependencies in data compared to simple RNNs. The deeper architecture allows for a hierarchical representation of information, making them suitable for complex sequential tasks.

Training Deep RNNs:

Training deep recurrent neural networks can be challenging due to the vanishing and exploding gradient problems, similar to simple RNNs. Careful initialization and the use of advanced optimization techniques are often required.

Applications:

Deep RNNs are used in a wide range of applications, including natural language understanding and generation, machine translation, speech recognition, video analysis, and music generation.

They have been particularly successful in tasks that involve understanding and generating structured sequential data.

Conclusion:

In this experiment, we implemented both a simple RNN and a deep RNN for sequential data processing. Simple RNNs are fundamental for modelling sequential dependencies, while deep RNNs offer the capacity to capture more complex patterns and longer-term dependencies. The choice between them depends on the specific problem and dataset complexity. Careful data preprocessing and hyperparameter tuning are crucial for improving model performance.

Experiment 12

Deep RNN

In [3]:

```
import tensorflow as tf
import numpy as np

# Create toy dataset
data = np.array([1.0, 0.5, 0.7, 0.3, 0.2])
data = data.reshape(1, -1, 1)  # (batch_size, sequence_length, input_dimen

# Define the Deep RNN model
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(units=1, activation='tanh', return_sequences=
    tf.keras.layers.SimpleRNN(units=1, activation='tanh', return_sequences=
    tf.keras.layers.SimpleRNN(units=1, activation='tanh', return_sequences=
    tf.keras.layers.SimpleRNN(units=1, activation='tanh', return_sequences=
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(data, data, epochs=1000)

# Predict using the trained model
predicted = model.predict(data)

print("Original data:", data)
print("Predicted data:", predicted)
```

```
Epoch 997/1000
1/1 [==============================] - 0s 16ms/step - loss: 0.0578
Epoch 998/1000
1/1 [==============================] - 0s 13ms/step - loss: 0.0577
Epoch 999/1000
1/1 [==============================] - 0s 15ms/step - loss: 0.0577
Epoch 1000/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.0577
1/1 [==============================] - 1s 854ms/step
Original data: [[[1. ]
  [0.5]
  [0.7]
  [0.3]
  [0.2]]]
Predicted data: [[[0.67927617]
  [0.37597153]
  [0.6209335 ]
  [0.45615074]
  [0.57362837]]]
```

In [ ]:

Simple RNN

In [2]:
```
import tensorflow as tf
import numpy as np

# Create toy dataset
data = np.array([1.0, 0.5, 0.7, 0.3, 0.2])
data = data.reshape(1, -1, 1)  # (batch_size, sequence_length, input_dimens

# Define the RNN model
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(units=1, activation='tanh', return_sequences=
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(data, data, epochs=1000)

# Predict using the trained model
predicted = model.predict(data)

print("Original data:", data)
print("Predicted data:", predicted)
```

```
Epoch 997/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.0118
Epoch 998/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.0118
Epoch 999/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.0118
Epoch 1000/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.0118
1/1 [==============================] - 0s 359ms/step
Original data: [[[1. ]
  [0.5]
  [0.7]
  [0.3]
  [0.2]]]
Predicted data: [[[0.950798  ]
  [0.4570682 ]
  [0.8003449 ]
  [0.2640567 ]
  [0.40774655]]]
```