Deep Learning with Tensorflow

LAB MANUAL

DEPARTMENT OF CSE(AIML)
ADITYA COLLEGE OF ENGINEERING AND TECHNOLOGY

VISION & MISSION OF THE INSTITUTE

VISION

To induce higher planes of learning by imparting technical education with

- International standards
- Applied research
- Creative Ability
- Value based instruction and to emerge as a premiere institute.

MISSION

Achieving academic excellence by providing globally acceptable technical education by forecasting technology through

- Innovative Research and development
- Industry Institute Interaction
- Empowered Manpower

VISION & MISSION OF THE DEPARTMENT

VISION

To be a recognized Computer Science and Engineering hub striving to meet the growing needs of the Industry and Society.

MISSION

M1: Imparting Quality Education through state-of-the-art infrastructure with industry Collaboration

M2: Enhance Teaching Learning Process to disseminate knowledge.

M3: Organize Skill based, Industrial and Societal Events for overall Development.

Department of Computer Science & Engineering-AIML

Course Outcome mapping with PO's and PSO's

Course Name:	Deep Learning with Tensorflow	**Class:**	III B.Tech CSE-AIML
Faculty Name:	K N S K SANTHOSH	**Regulation:**	R20
Academic Year:	2024-25	**Semester:**	II

COURSE OUTCOMES (COs):

Upon completion of the course, students will be able to:

CO's	Description	Bloom's Taxonomy Level
CO1	Implement deep neural networks to solve real world problems	Apply
CO2	Design a neural network for classifying Movie reviews,news wires ,houses prices	Create
CO3	Choose appropriate pre-trained model to solve real time problem	Evaluate
CO4	Build a Convolution Neural Network for Hand written Digit Classification.,simple image Classification	Apply
CO5	Interpret the results of two different deep learning models	Analyze
CO6	Build natural language processing systems using TensorFlow	Create

CO-PO/PSO MATRIX:

Course Outcomes	Program Outcomes													
	PO 1	PO2	PO 3	PO 4	PO 5	PO 6	PO7	PO8	PO9	PO10	PO 11	PO12	PSO 1	PSO 2
CO1	2	2	3	2	2				1	1				2
CO2	2	3	2		2				1	1				2
CO3	2	3	2	2					1					2
CO4	3	2	3	2	2									2
CO5	2	3		2					1	2				2
CO6	3	2	3	3	3				1	1				2
Course	2.33	2.5	2.6	2.2	2.25				1	1				2

PO1	Engineering Knowledge	PO7	Environment & Sustainability
PO2	Problem Analysis	PO8	Ethics
PO3	Design / Development of Solutions	PO9	Individual & Team Work
PO4	Conduct Investigations of complex problems	PO10	Communication Skills
PO5	Modern Tool usage	PO11	Project Management & Finance
PO6	Engineer & Society	PO12	Life-long Learning

Faculty Signature

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY: KAKINADA
KAKINADA – 533 003, Andhra Pradesh, India

DEPARTMENT OF CSE - ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

III B Tech II Sem		L	T	P	C
		0	0	3	1.5
DEEP LEARNING WITH TENSORFLOW					

Course Outcomes:
On completion of this course, the student will be able to
- Implement deep neural networks to solve real world problems
- Choose appropriate pre-trained model to solve real time problem
- Interpret the results of two different deep learning models

Software Packages required:
- Keras
- Tensorflow
- PyTorch

List of Experiments:
1. Implement multilayer perceptron algorithm for MNIST Hand written Digit Classification.
2. Design a neural network for classifying movie reviews (Binary Classification) using IMDB dataset.
3. Design a neural Network for classifying news wires (Multi class classification) using Reuters dataset.
4. Design a neural network for predicting house prices using Boston Housing Price dataset.
5. Build a Convolution Neural Network for MNIST Hand written Digit Classification.
6. Build a Convolution Neural Network for simple image (dogs and Cats) Classification
7. Use a pre-trained convolution neural network (VGG16) for image classification.
8. Implement one hot encoding of words or characters.
9. Implement word embeddings for IMDB dataset.
10. Implement a Recurrent Neural Network for IMDB movie review classification problem.

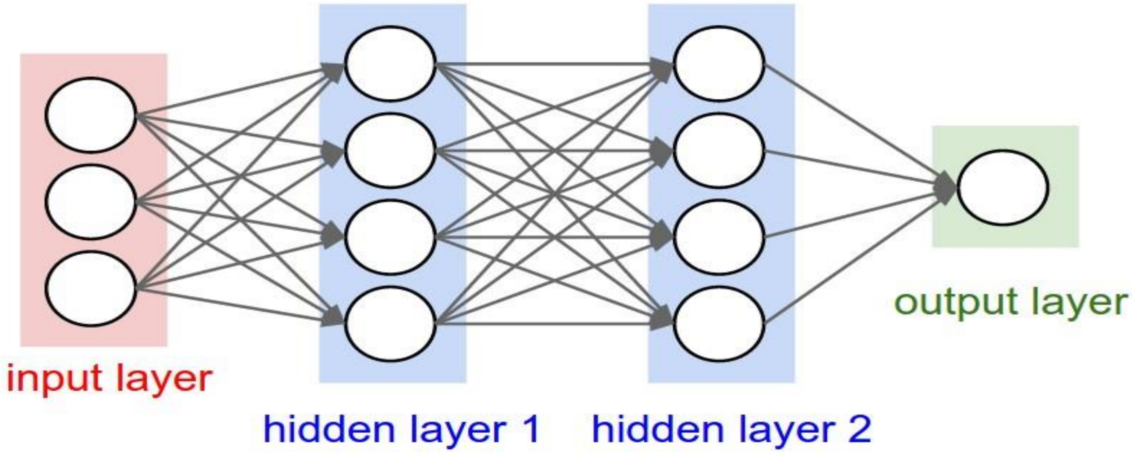
Text Books:
1. Reza Zadeh and BharathRamsundar, “Tensorflow for Deep Learning”, O’Reilly publishers, 2018

References:
1. https://github.com/fchollet/deep-learning-with-python-notebooks

Introduction: Artificial intelligence, machine learning, and deep learning

- **Artificial intelligence** is a broad term that refers to the ability of machines to perform tasks that are typically associated with human intelligence, such as learning, reasoning, and problem-solving.
- **Machine learning** is a subset of AI that involves the development of algorithms that can learn from data without being explicitly programmed. Machine learning algorithms are trained on large datasets, and they can then be used to make predictions or decisions about new data.
- **Deep learning** is a subset of machine learning that uses artificial neural networks to learn from data. Neural networks are inspired by the human brain, and they can be used to solve complex problems that would be difficult or impossible to solve with traditional machine learning algorithms.

- **Artificial neural networks:** are built on the principles of the structure and operation of human neurons. It is also known as neural networks or neural nets. An artificial neural network's input layer, which is the first layer, receives input from external sources and passes it on to the hidden layer, which is the second layer. Each neuron in the hidden layer gets information from the neurons in the previous layer, computes the weighted total, and then transfers it to the neurons in the next layer.



- **Keras**, Keras is a high-level, deep learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy. It also supports multiple backend neural network computation. Keras is relatively easy to learn and work with because it provides a python frontend with a high level of abstraction while having the option of multiple back-ends for computation purposes. This makes Keras slower than other deep learning frameworks, but extremely beginner-friendly.

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

- [Tensorflow]
- Theano

- PlaidML
- MXNet
- CNTK (Microsoft Cognitive Toolkit)

- **TensorFlow**: Is an open-source library developed by Google primarily for deep learning applications. It also supports traditional machine learning. Tensor Flow was originally developed for large numerical computations without keeping deep learning in mind.

Process of running the project in Deep Learning:

1. Import the libraries and load the dataset

First, we are going to import all the modules that we are going to need for training our model. The Keras library already contains some datasets and MNIST is one of them. So we can easily import the dataset and start working with it. The **mnist.load_data()** method returns us the training data, its labels and also the testing data and its labels.

2. Preprocess the data

The image data cannot be fed directly into the model so we need to **perform some operations and process the data** to make it ready for our neural network. The dimension of the training data is (60000,28,28). The CNN model will require one more dimension so we reshape the matrix to shape (60000,28,28,1).

3. Create the model

Now we will **create our CNN model** in Python data science project. A CNN model generally consists of convolutional and pooling layers. It works better for data that are represented as grid structures, this is the reason why CNN works well for image classification problems. The dropout layer is used to deactivate some of the neurons and while training, it reduces offer fitting of the model. We will then compile the model with the Adadelta optimizer.

4. Train the model

The **model.fit() function** of Keras will start the training of the model. It **takes the training data, validation data, epochs, and batch size.**

It takes some time to train the model. After training, we save the weights and model definition in the ‘mnist.h5’ file.

5. Evaluate the model

We have 10,000 images in our dataset which will be used to **evaluate how good our model works**. The testing data was not involved in the training of the data therefore, it is new data for our model. The MNIST dataset is well balanced so we can get around 99% accuracy.

6. Create GUI to predict digits

Now for the GUI, we have created a new file in which we **build an interactive window to draw digits on canvas** and with a button, we can recognize the digit. The Tkinter library comes in the Python standard library. We have created a function **predict_digit()** that takes the image as input and then uses the trained model to predict the digit.

Then we **create the App class** which is responsible for building the GUI for our app. We create a canvas where we can draw by capturing the mouse event and with a button, we trigger the predict_digit() function and display the results.

EXPERIMENT NO - 1

Implement multilayer perceptron algorithm for MNIST Hand written Digit Classification.

MNIST Handwritten Digit Classification DataSet :

The MNIST dataset is a popular benchmark dataset for image classification tasks. It consists of 60,000 grayscale images of handwritten digits (0 to 9) for training and 10,000 images for testing. Each image is 28 x 28 pixels in size, and each pixel value ranges from 0 to 255. The goal of the task is to correctly classify each image into one of the 10 possible digit classes.

Multilayer Perceptron (MLP) Algorithm :

The MLP algorithm is a type of artificial neural network that consists of multiple layers of interconnected nodes or neurons. It is a feedforward neural network, meaning that the data flows from the input layer to the output layer through one or more hidden layers, with each layer performing a nonlinear transformation on the input.

The basic building block of an MLP is the perceptron, which is a mathematical model of a neuron that takes a set of inputs, computes a weighted sum of the inputs, and applies a nonlinear activation function to produce an output. The MLP is called a multilayer perceptron because it contains multiple layers of perceptrons.

To train an MLP for a classification task like the MNIST digit classification task, we need to define the architecture of the network, the loss function to optimize, and the optimization algorithm to use. Typically, the architecture of an MLP for image classification consists of an input layer, one or more hidden layers, and an output layer. The number of neurons in the input layer is equal to the number of features in the input data, and the number of neurons in the output layer is equal to the number of classes in the classification task.

PROGRAM:

```
# Import necessary libraries
import numpy as np
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.utils import to_categorical
```

In this implementation, we first load the MNIST dataset using the mnist.load_data() function from Keras

```
# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

In this step, we use the mnist.load_data() function from Keras to load the MNIST dataset. The training data consists of the x_train images and their corresponding y_train labels, while the test data consists of the x_test images and their corresponding y_test labels.

```
# Reshape input data
X_train = X_train.reshape(X_train.shape[0], 28*28)
X_test = X_test.reshape(X_test.shape[0], 28*28)
```

In this step, we preprocess the data by reshaping the images to 1D arrays, normalizing the pixel values to be between 0 and 1, and

```
# Normalize input data
X_train = X_train / 255
X_test = X_test / 255
```

. We then preprocess the data by flattening the input images into 1D arrays of size 784 (28x28), scaling the pixel values to the range of 0 to 1, and dividing by 255.0 to normalize the data.

```
# One-hot encode target variables
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

converting the labels to one-hot encoding using the to_categorical() function from Keras.

```
# Define MLP model
model = Sequential()
model.add(Dense(512, input_shape=(784,), activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

Next, we define the neural network model with three fully connected (dense) layers. The first two hidden layers have 256 and 128 units, respectively, and use ReLU activation functions. The dropout layers randomly drop out 20% of the input units during training to prevent overfitting. The output layer has 10 units with softmax activation for multi-class classification

```
# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=128)
```

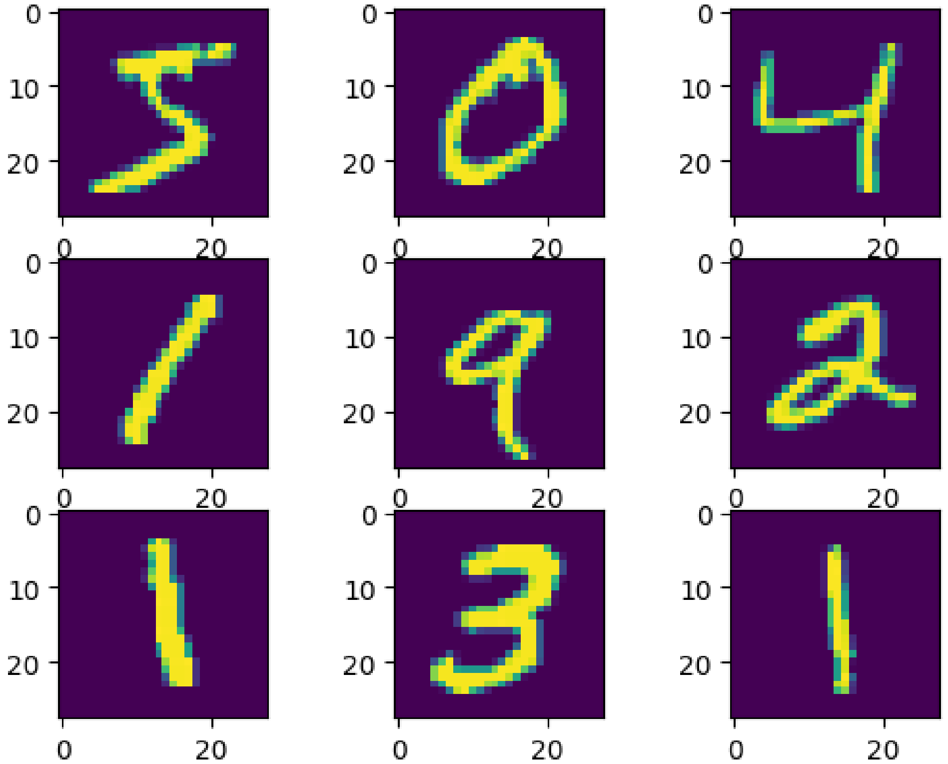
We compile the model with the Adam optimizer, sparse categorical cross-entropy loss, and accuracy metric. We train the model on the training data for 10 epochs with a batch size of 128. Finally, we evaluate the model on the test data and print the accuracy score.

```
# Evaluate model on test data
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

OUTPUT:

```
[  0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 2
53
 225 172 253 242 195  64   0   0   0   0]
[  0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251
  93  82  82  56  39   0   0   0   0   0]
[  0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0   0   0
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0   0   0
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0  11 190 253  70   0   0   0
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0  35 241 225 160 108   1
   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 253 119
  25   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 253 253
 150  27   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252
 253 187   0   0   0   0   0   0   0   0]
```

```
[   0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251
   93  82  82  56  39   0   0   0   0   0]
 [   0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0  11 190 253  70   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0  35 241 225 160 108   1
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 253 119
   25   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 253 253
  150  27   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252
  253 187   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 249
  253 249  64   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253
  253 207   2   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 253 253
  250 182   0   0   0   0   0   0   0   0]
[   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252
  253 187   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 249
  253 249  64   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253
  253 207   2   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 253 253
  250 182   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253 253 201
   78   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0  23  66 213 253 253 253 253 198  81   2
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0  18 171 219 253 253 253 253 195  80   9   0   0
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0  55 172 226 253 253 253 253 244 133  11   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0 136 253 253 253 212 135 132  16   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]]
```



```
Reshaped training data shape: (60000, 784)
Reshaped testing data shape: (10000, 784)
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   3  18  18  18 126 136 175  26 166 255
  247 127   0   0   0   0   0   0   0   0   0   0   0   0  30  36  94 154
  170 253 253 253 253 253 225 172 253 242 195  64   0   0   0   0   0   0
    0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93  82
   82  56  39   0   0   0   0   0   0   0   0   0   0   0   0  18 219 253
  253 253 253 253 198 182 247 241   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0  14   1 154 253  90   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0  11 190 253  70   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  35 241
  225 160 108   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0  81 240 253 253 119  25   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0  45 186 253 253 150  27   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253 187
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0 249 253 249  64   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253
  253 207   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0  39 148 229 253 253 253 250 182   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253
```

```
 253 201  78   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0  23  66 213 253 253 253 253 198  81   2   0   0   0   0   0   0
     0   0   0   0   0   0   0   0   0   0  18 171 219 253 253 253 253 195
    80   9   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    55 172 226 253 253 253 253 244 133  11   0   0   0   0   0   0   0   0
     0   0   0   0   0   0   0   0   0   0 136 253 253 253 212 135 132  16
     0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0   0   0   0   0   0   0   0   0]
Normalizing the input data...
Normalization complete. Pixel values are now between 0 and 1.
One-hot encoding the target variables...
Encoded training labels shape: (60000, 10)
Encoded testing labels shape: (10000, 10)
Compiling the model...
Model compiled with categorical cross-entropy loss and Adam optimizer.
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 18ms/step  - accuracy: 0.8729 - los
s: 0.4403 - val_accuracy: 0.9659 - val_loss: 0.1055
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 19ms/step  - accuracy: 0.9658 - los
s: 0.1089 - val_accuracy: 0.9736 - val_loss: 0.0806
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 21ms/step  - accuracy: 0.9784 - los
s: 0.0703 - val_accuracy: 0.9773 - val_loss: 0.0708
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step  - accuracy: 0.9819 - los
s: 0.0563 - val_accuracy: 0.9810 - val_loss: 0.0599
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 8s 18ms/step - accuracy: 0.9850 - loss
: 0.0458 - val_accuracy: 0.9816 - val_loss: 0.0594
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 19ms/step - accuracy: 0.9880 - los
s: 0.0351 - val_accuracy: 0.9819 - val_loss: 0.0610
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 20ms/step - accuracy: 0.9887 - los
s: 0.0330 - val_accuracy: 0.9809 - val_loss: 0.0638
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.9892 - los
s: 0.0303 - val_accuracy: 0.9818 - val_loss: 0.0667
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 8s 17ms/step - accuracy: 0.9911 - loss
: 0.0266 - val_accuracy: 0.9833 - val_loss: 0.0622
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.9920 - los
s: 0.0234 - val_accuracy: 0.9819 - val_loss: 0.0662
Training complete.
Evaluating the model on test data...
Test Accuracy: 98.19%
Training history:
Training accuracy: 0.99
Validation accuracy: 0.98
```

RESULT: multilayer perceptron algorithm for MNIST Hand written Digit Classification is successfully executed

EXPERIMENT NO -2

Design a neural network for classifying movie reviews (Binary Classification) using IMDB dataset.

IMDB DataSet :

The IMDB (Internet Movie Database) dataset is a popular benchmark dataset for sentiment analysis, which is the task of classifying text into positive or negative categories. The dataset consists of 50,000 movie reviews, where 25,000 are used for training and 25,000 are used for testing. Each review is already preprocessed and encoded as a sequence of integers, where each integer represents a word in the review.

The goal of designing a neural network for binary classification of movie reviews using the IMDB dataset is to build a model that can classify a given movie review as either positive or negative based on the sentiment expressed in the review.

Program

```
# Import necessary libraries

from tensorflow.keras.datasets import imdb

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout

from tensorflow.keras.preprocessing.sequence import pad_sequences

# Load the dataset

(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000)
```

In this step, we load the IMDB dataset using the imdb.load_data() function from Keras. We set the num_words parameter to 10000 to limit the number of words in each review to 10,000, which helps to reduce the dimensionality of the input data and improve model performance.

```
# Preprocess the data

maxlen = 200

X_train = pad_sequences(X_train, maxlen=maxlen)

X_test = pad_sequences(X_test, maxlen=maxlen)
```

In this step, we preprocess the data by padding the sequences with zeros to a maximum length of 200 using the pad_sequences() function from Keras. This ensures that all input sequences have the same length and can be fed into the neural network.

```
# Define the model

model = Sequential()

model.add(Dense(128, activation='relu', input_shape=(maxlen,)))

model.add(Dropout(0.5))

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(1, activation='sigmoid'))
```

In this step, we define the neural network architecture using the Sequential() class from Keras. Next, we define the neural network model with three fully connected layers. The first layer has 128 units with ReLU activation, the second layer has 64 units with ReLU activation, and the final layer has a single unit with sigmoid activation for binary classification.

```
# Compile the model

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In this step, we compile the model using the compile() method from Keras. We set the loss function to binary cross-entropy, which is appropriate for binary classification problems. We use the adam optimizer and track the accuracy metric during training.

```
# Train the model

model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=128)
```

In this step, we train the model on the training data using the fit() method from Keras. We set the number of epochs to 10 and the batch size to 128. We also pass in the test data as the validation data to monitor the performance of the model on unseen data during training.

```
# Evaluate the model on test data

scores = model.evaluate(X_test, y_test, verbose=0)

print("Accuracy: %.2f%%" % (scores[1]*100))
```

Finally, we can evaluate the performance of the model on the test data using the evaluate() function from Keras.

```
from sklearn.metrics import classification_report

# Predict class labels for the test set

y_pred = (model.predict(X_test) > 0.5).astype("int32")

# Generate and print classification report

print(classification_report(y_test, y_pred, target_names=["Negative", "Positive"]))

from sklearn.metrics import confusion_matrix

import seaborn as sns

import matplotlib.pyplot as plt

# Generate the confusion matrix

cm = confusion_matrix(y_test, y_pred)

# Visualize the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=["Negative", "Positive"],
yticklabels=["Negative", "Positive"])

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix")

plt.show()
```

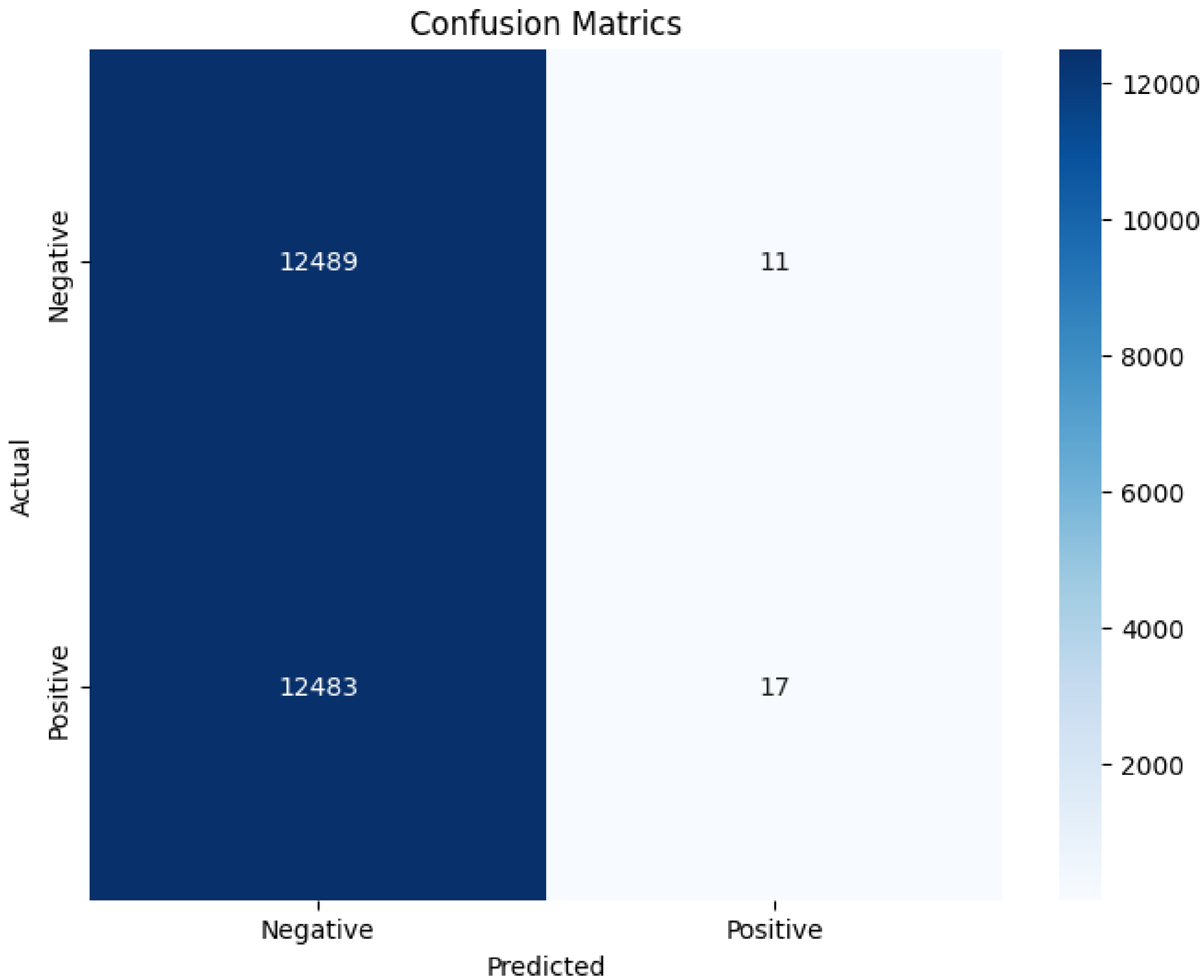
OUTPUT:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets
/imdb.npz
17464789/17464789 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

```
Epoch 1/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - accuracy: 0.5047 - loss: 813.3904 - val_accuracy: 0.4982 - val_loss: 29.5431
Epoch 2/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.4931 - loss: 100.0582 - val_accuracy: 0.5006 - val_loss: 0.9354
Epoch 3/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.5018 - loss: 14.6608 - val_accuracy: 0.5022 - val_loss: 0.7642
Epoch 4/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.5058 - loss: 5.7007 - val_accuracy: 0.4961 - val_loss: 0.7721
Epoch 5/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.5046 - loss: 3.9950 - val_accuracy: 0.4980 - val_loss: 0.7758
Epoch 6/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.4979 - loss: 2.3970 - val_accuracy: 0.5016 - val_loss: 0.7028
Epoch 7/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.5017 - loss: 1.9570 - val_accuracy: 0.5050 - val_loss: 0.7120
Epoch 8/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.5001 - loss: 1.8261 - val_accuracy: 0.5040 - val_loss: 0.7083
Epoch 9/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.5052 - loss: 1.5838 - val_accuracy: 0.5048 - val_loss: 0.6991
Epoch 10/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.4993 - loss: 1.2770 - val_accuracy: 0.5009 - val_loss: 0.6952
Epoch 11/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.5026 - loss: 1.3109 - val_accuracy: 0.5017 - val_loss: 0.6950
Epoch 12/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.5040 - loss: 1.3620 - val_accuracy: 0.4975 - val_loss: 0.6955
Epoch 13/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.4981 - loss: 1.0187 - val_accuracy: 0.4979 - val_loss: 0.6947
Epoch 14/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.4968 - loss: 1.0124 - val_accuracy: 0.5000 - val_loss: 0.6937
Epoch 15/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.4994 - loss: 0.9437 - val_accuracy: 0.4960 - val_loss: 0.6939
Epoch 16/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.4983 - loss: 0.8723 - val_accuracy: 0.4982 - val_loss: 0.6938
Epoch 17/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.4933 - loss: 0.8976 - val_accuracy: 0.4976 - val_loss: 0.6938
Epoch 18/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - accuracy: 0.4977 - loss: 0.9048 - val_accuracy: 0.4993 - val_loss: 0.6935
Epoch 19/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.4949 - loss: 0.9958 - val_accuracy: 0.4998 - val_loss: 0.6937
```

```
Epoch 20/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.5029 - loss: 0
.8430 - val_accuracy: 0.4997 - val_loss: 0.6933
Epoch 21/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.5046 - loss:
0.8882 - val_accuracy: 0.5004 - val_loss: 0.6932
Epoch 22/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.4933 - loss:
0.7644 - val_accuracy: 0.5003 - val_loss: 0.6932
Epoch 23/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.4975 - loss:
0.7671 - val_accuracy: 0.4999 - val_loss: 0.6932
Epoch 24/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.5005 - loss: 0
.7883 - val_accuracy: 0.4994 - val_loss: 0.6932
Epoch 25/25
98/98 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.5003 - loss: 0
.8115 - val_accuracy: 0.5002 - val_loss: 0.6932
Accuracy: 50.02%
782/782 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step
              precision    recall  f1-score   support

    negative       0.50      1.00      0.67     12500
    positive       0.61      0.00      0.00     12500

    accuracy                           0.50     25000
   macro avg       0.55      0.50      0.33     25000
weighted avg       0.55      0.50      0.33     25000
```

•



RESULT: Neural network for classifying movie reviews (Binary Classification) using IMDB dataset is executed

EXPERIMENT NO -3

Design a neural Network for classifying news wires (Multi class classification) using Reuters dataset

Reuters DataSet :

The Reuters dataset is a collection of newswire articles and their categories. It consists of 11,228 newswire articles that are classified into 46 different topics or categories. The goal of this task is to train a neural network to accurately classify newswire articles into their respective categories.

Input layer: This layer will take in the vectorized representation of the news articles in the Reuters dataset.

Hidden layers: You can use one or more hidden layers with varying number of neurons in each layer. You can experiment with the number of layers and neurons to find the optimal configuration for your specific problem.

Output layer: This layer will output a probability distribution over the possible categories for each input news article. Since this is a multi-class classification problem, you can use a softmax activation function in the output layer to ensure that the predicted probabilities sum to 1.

Program

```
import numpy as np

from tensorflow.keras.datasets import reuters

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout

from tensorflow.keras.utils import to_categorical
```

We will import all the necessary libraries for the model and We will use the Keras library to load the dataset and preprocess it.

```
# Load the Reuters dataset

(x_train, y_train), (x_test, y_test) = reuters.load_data(num_words=10000)
```

The first step is to load the Reuters dataset and preprocess it for training. We will also split the dataset into train and test sets.

In this step, we load the IMDB dataset using the reuters.load_data() function from Keras. We set the num_words parameter to 10000 to limit the number of words in each review to 10,000, which helps to reduce the dimensionality of the input data and improve model performance.

```
# Vectorize the data using one-hot encoding
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
      results[i, sequence] = 1
    return results
x_train = vectorize_sequences(x_train)
x_test = vectorize_sequences(x_test)
# Convert the labels to one-hot vectors
num_classes = max(y_train) + 1
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
# Define the neural network architecture
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(10000,)))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

The next step is to design the neural network architecture. For this task, we will use a fully connected neural network with an input layer, multiple hidden layers, and an output layer. We will use the Dense class in Keras to add the layers to our model. Since we have 46 categories, the output layer will have 46 neurons, and we will use the softmax activation function to ensure that the output of the model represents a probability distribution over the 46 categories.

```
# Compile the model
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Once we have defined the model architecture, the next step is to compile the model. We need to specify the loss function, optimizer, and evaluation metrics for the model. Since this is a multi-class classification problem, we will use the categorical_crossentropy loss function. We will use the adam optimizer and accuracy as the evaluation metric.

```
# Train the model on the training set
history = model.fit(x_train, y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_test, y_test))
```

After compiling the model, the next step is to train it on the training data. We will use the fit method in Keras to train the model. We will also specify the validation data and the batch size.

```
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test accuracy:', test_acc)
```

Evaluate the performance of the neural network on the validation set and tune the hyperparameters such as learning rate, number of layers, number of neurons, etc., based on the validation performance.

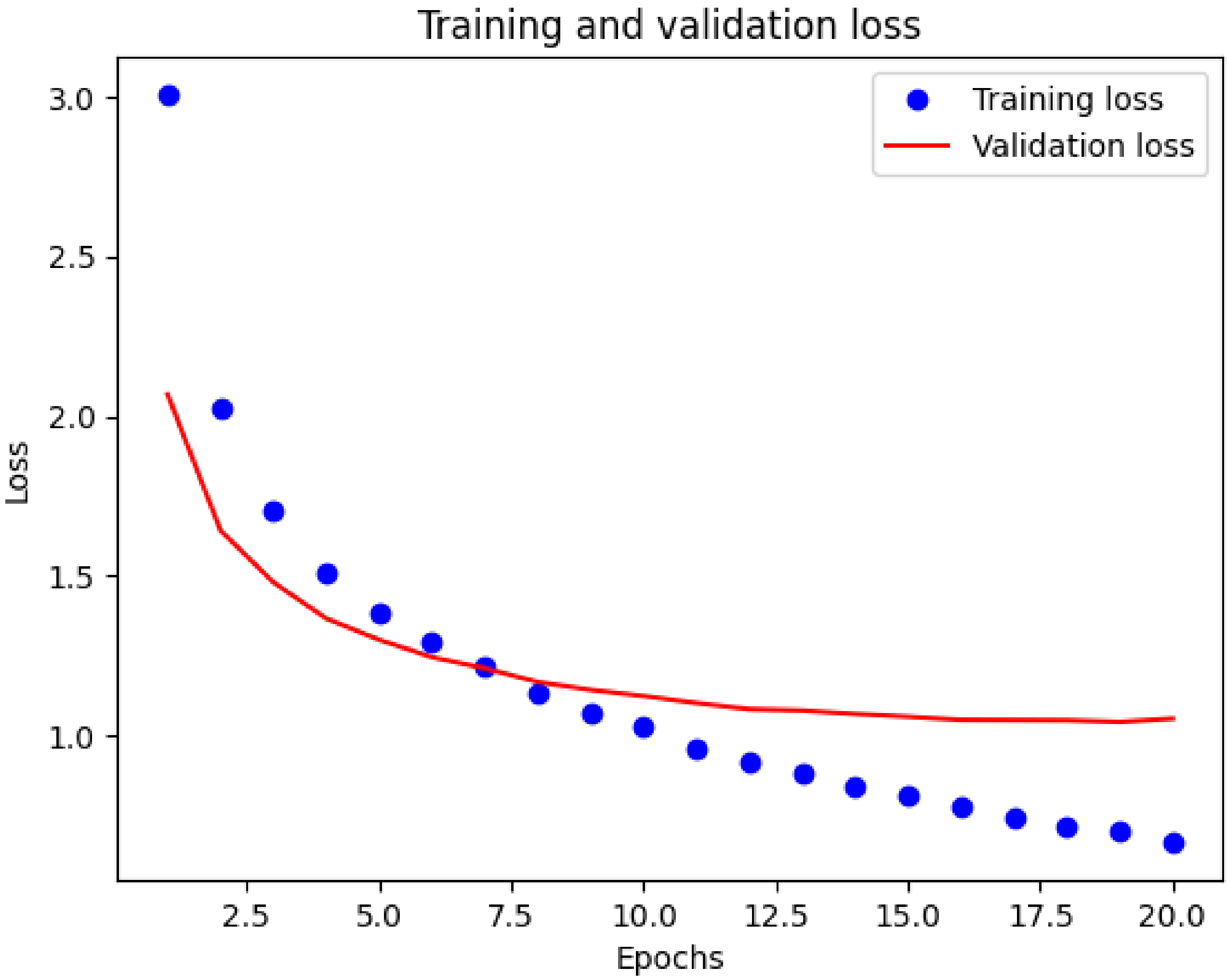
```
import matplotlib.pyplot as plt
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
```

```
plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

acc = history.history['accuracy']

val_acc = history.history['val_accuracy']

plt.plot(epochs, acc, 'bo', label='Training acc')

plt.plot(epochs, val_acc, 'b', label='Validation acc')

plt.title('Training and validation accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

from sklearn.metrics import confusion_matrix, classification_report

# Predict the classes for the test set

y_pred = model.predict(x_test)

y_pred_classes = np.argmax(y_pred, axis=1)

y_true_classes = np.argmax(y_test, axis=1)

# Generate the confusion matrix

conf_matrix = confusion_matrix(y_true_classes, y_pred_classes)

print("Confusion Matrix:\n", conf_matrix)


# Generate a classification report

report = classification_report(y_true_classes, y_pred_classes)
```

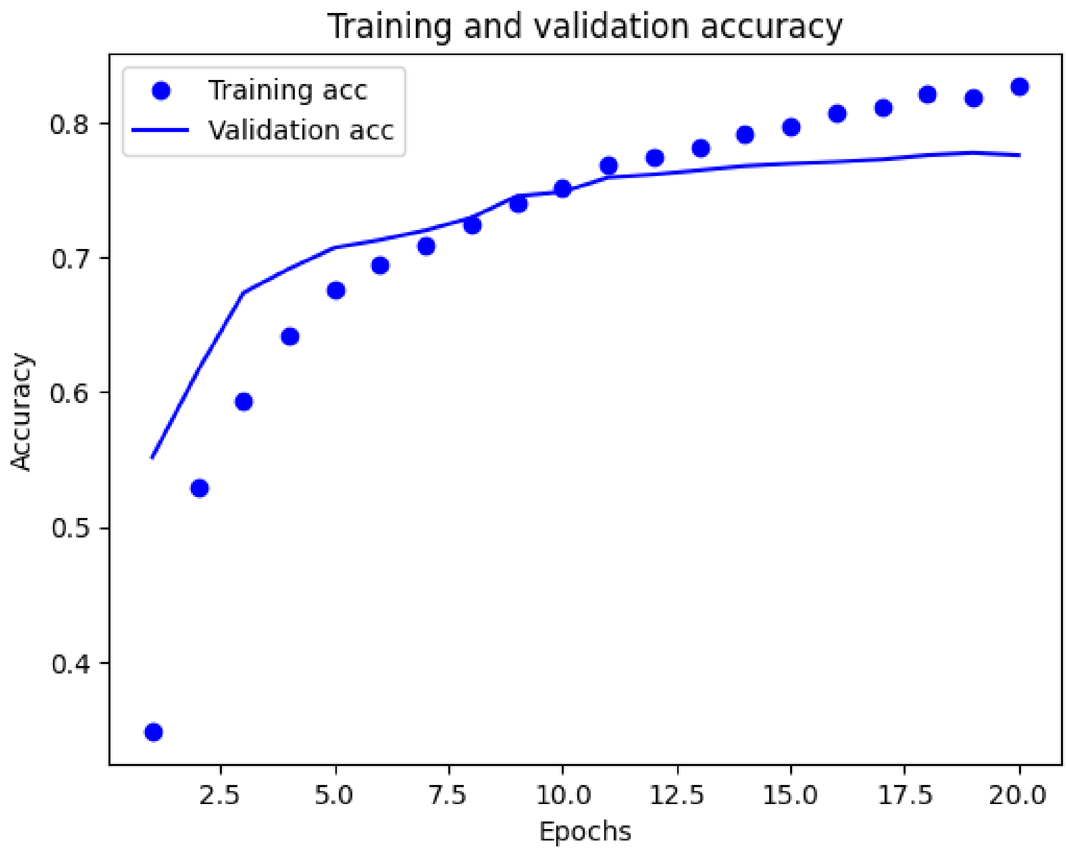
print("Classification Report:\n", report)

```
OUTPUT: Epoch 1/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 3s 87ms/step - accuracy: 0.2438 - loss: 3.3648 - val_accuracy: 0.5516 - val_loss: 2.0680
Epoch 2/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - accuracy: 0.5036 - loss: 2.1498 - val_accuracy: 0.6158 - val_loss: 1.6425
Epoch 3/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.5795 - loss: 1.7620 - val_accuracy: 0.6736 - val_loss: 1.4801
Epoch 4/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - accuracy: 0.6356 - loss: 1.5305 - val_accuracy: 0.6915 - val_loss: 1.3667
Epoch 5/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.6726 - loss: 1.3851 - val_accuracy: 0.7070 - val_loss: 1.3001
Epoch 6/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - accuracy: 0.6940 - loss: 1.2903 - val_accuracy: 0.7128 - val_loss: 1.2456
Epoch 7/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - accuracy: 0.7061 - loss: 1.2262 - val_accuracy: 0.7199 - val_loss: 1.2106
Epoch 8/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.7253 - loss: 1.1311 - val_accuracy: 0.7293 - val_loss: 1.1679
Epoch 9/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.7387 - loss: 1.0797 - val_accuracy: 0.7453 - val_loss: 1.1435
Epoch 10/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 2s 85ms/step - accuracy: 0.7486 - loss: 1.0493 - val_accuracy: 0.7484 - val_loss: 1.1240
Epoch 11/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 2s 56ms/step - accuracy: 0.7658 - loss: 0.9609 - val_accuracy: 0.7591 - val_loss: 1.1019
Epoch 12/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 47ms/step - accuracy: 0.7746 - loss: 0.9224 - val_accuracy: 0.7614 - val_loss: 1.0830
Epoch 13/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - accuracy: 0.7772 - loss: 0.8995 - val_accuracy: 0.7645 - val_loss: 1.0787
Epoch 14/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.7896 - loss: 0.8483 - val_accuracy: 0.7676 - val_loss: 1.0680
Epoch 15/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.7924 - loss: 0.8421 - val_accuracy: 0.7694 - val_loss: 1.0594
Epoch 16/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - accuracy: 0.8044 - loss: 0.7778 - val_accuracy: 0.7707 - val_loss: 1.0492
Epoch 17/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - accuracy: 0.8184 - loss: 0.7259 - val_accuracy: 0.7725 - val_loss: 1.0486
Epoch 18/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.8169 - loss: 0.7153 - val_accuracy: 0.7756 - val_loss: 1.0479
Epoch 19/20
```

```
.
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 63ms/step - accuracy: 0.8183 - loss: 0.7031 - val_accuracy: 0.7774 - val_loss: 1.0426
Epoch 20/20
18/18 ━━━━━━━━━━━━━━━━━━━━ 2s 84ms/step - accuracy: 0.8298 - loss: 0.6594 - val_accuracy: 0.7756 - val_loss: 1.0532
71/71 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7850 - loss: 1.0388
Test accuracy: 0.7756010890007019
```



```
71/71 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
Confusion Matrix:
 [[ 6  3  0 ...  0  0  0]
 [ 0 89  0 ...  0  0  0]
 [ 0  5 11 ...  0  0  0]
 ...
 [ 0  0  0 ...  0  0  0]
 [ 0  1  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]]
Classification Report:
              precision    recall  f1-score   support

           0       0.67      0.50      0.57        12
           1       0.57      0.85      0.68       105
           2       0.73      0.55      0.63        20
           3       0.91      0.95      0.93       813
           4       0.83      0.87      0.85       474
           5       0.00      0.00      0.00         5
           6       1.00      0.71      0.83        14
           7       0.00      0.00      0.00         3
           8       0.72      0.68      0.70        38
           9       0.71      0.68      0.69        25
          10       0.84      0.90      0.87        30
          11       0.56      0.77      0.65        83
          12       0.33      0.08      0.12        13
          13       0.57      0.68      0.62        37
          14       0.00      0.00      0.00         2
          15       0.00      0.00      0.00         9
          16       0.60      0.79      0.68        99
          17       0.00      0.00      0.00        12
          18       0.46      0.60      0.52        20
          19       0.66      0.73      0.69       133
          20       0.60      0.53      0.56        70
          21       0.58      0.70      0.63        27
          22       0.00      0.00      0.00         7
          23       0.25      0.08      0.12        12
          24       0.33      0.05      0.09        19
          25       0.68      0.68      0.68        31
          26       0.00      0.00      0.00         8
          27       0.00      0.00      0.00         4
          28       0.50      0.10      0.17        10
          29       0.00      0.00      0.00         4
          30       0.20      0.08      0.12        12
          31       1.00      0.31      0.47        13
          32       0.83      0.50      0.62        10
          33       0.00      0.00      0.00         5
          34       0.57      0.57      0.57         7
          35       0.00      0.00      0.00         6
          36       0.00      0.00      0.00        11
          37       0.00      0.00      0.00         2
```



RESULT: A neural Network for classifying news wires (Multi class classification) using Reuters dataset is executed.

EXPERIMENT NO -4

Design a neural network for predicting house prices using Boston Housing Price dataset.

The Boston Housing Price dataset is a collection of 506 samples of housing prices in the Boston area, where each sample has 13 features such as crime rate, average number of rooms per dwelling, and others. The goal of this task is to train a neural network to accurately predict the median value of owner-occupied homes in $1000's.

Input layer: This layer will take in the 13 features of each house.

Hidden layers: You can use one or more hidden layers with varying number of neurons in each layer. You can experiment with the number of layers and neurons to find the optimal configuration for your specific problem.

Output layer: This layer will output a single numerical value, which is the predicted price of the house.

We have 404 training samples and 102 test samples. The data comprises 13 features. The 13 features in the input data are as follow:

1. Per capita crime rate.
2. Proportion of residential land zoned for lots over 25,000 square feet.
3. Proportion of non-retail business acres per town.
4. Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).
5. Nitric oxides concentration (parts per 10 million).
6. Average number of rooms per dwelling.
7. Proportion of owner-occupied units built prior to 1940.
8. Weighted distances to five Boston employment centres.
9. Index of accessibility to radial highways.
10. Full-value property-tax rate per $10,000.
11. Pupil-teacher ratio by town.
12. 1000 * (Bk - 0.63) ** 2 where Bk is the proportion of Black people by town.
13. % lower status of the population.

PROGRAM:

```
from tensorflow.keras.datasets import boston_housing

from tensorflow.keras.models import Sequential
```

from tensorflow.keras.layers import Dense

from tensorflow.keras.utils import normalize

#We will import all the necessary libraries for the model and We will use the Keras library to load the dataset and preprocess it.

Load the Boston Housing Price dataset

(x_train, y_train), (x_test, y_test) = boston_housing.load_data()

#We will also split the dataset into training and validation sets.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train = scaler.fit_transform(x_train)

x_test = scaler.transform(x_test)

Define the neural network architecture

model = Sequential()

model.add(Dense(64, activation='relu', input_shape=(13,)))

model.add(Dense(64, activation='relu'))

model.add(Dense(1))

The next step is to design the neural network architecture. For this task, we will use a fully connected neural network with an input layer, multiple hidden layers, and an output layer. We will use the Dense class in Keras to add the layers to our model. Since this is a regression problem, the output layer will have only one neuron, and we will not use any activation function

Compile the model

model.compile(optimizer='adam', loss='mse')

Once we have defined the model architecture, the next step is to compile the model. We need to specify the loss function, optimizer, and evaluation metrics for the model. Since this is a regression problem, we will use the mean_squared_error loss function. We will use the adam optimizer and mean_absolute_error as the evaluation metric. Train the model on the training set

history = model.fit(x_train, y_train,

```
        epochs=100,
        batch_size=32,
        validation_data=(x_test, y_test))
```

After compiling the model, the next step is to train it on the training data. We will use the fit method in Keras to train the model. We will also specify the validation data and the batch size.

```
# Evaluate the model on the test set
test_loss = model.evaluate(x_test, y_test)
print('Test loss:', test_loss)
```

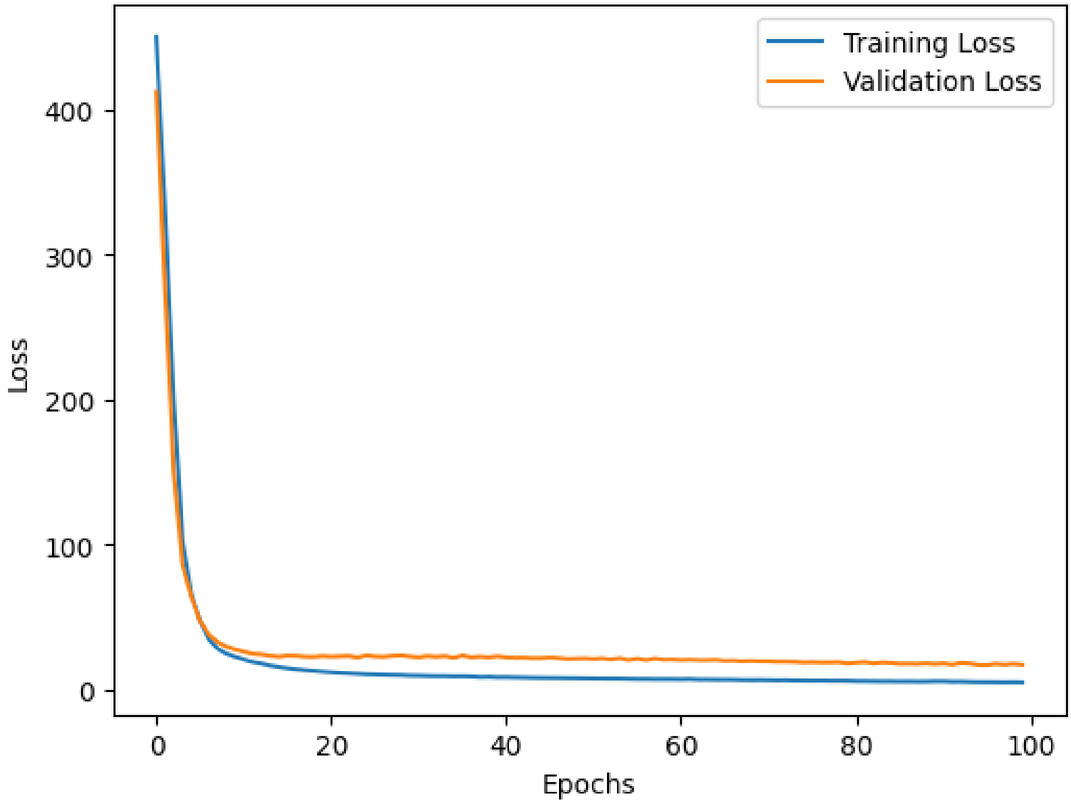
Once the model is trained, the next step is to evaluate its performance on the test data. We will use the evaluate method in Keras to evaluate the model.

```
# Evaluate the model on the test set
test_loss = model.evaluate(x_test, y_test)
print('Test loss:', test_loss)
#calculate metrics like Mean Absolute Error (MAE) to gain better insights into the model's performance:
from sklearn.metrics import mean_absolute_error
y_pred = model.predict(x_test)
mae = mean_absolute_error(y_test, y_pred)
print('Mean Absolute Error:', mae)
#Plot training and validation losses
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
OUTPUT: Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/boston_housing.npz
57026/57026 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 3s 35ms/step - loss: 488.5871 - mae: 19.9661 - val_loss: 412.4561 - val_mae: 18.2666
Epoch 2/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 356.2255 - mae: 16.6107 - val_loss: 276.6468 - val_mae: 14.5210
Epoch 3/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 238.7078 - mae: 13.0797 - val_loss: 150.6518 - val_mae: 10.4748
Epoch 4/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 110.8227 - mae: 8.3642 - val_loss: 86.0421 - val_mae: 7.4320
Epoch 5/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 73.2420 - mae: 6.7305 - val_loss: 64.4460 - val_mae: 6.1842
Epoch 6/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 50.3410 - mae: 5.3574 - val_loss: 47.2811 - val_mae: 5.3485
Epoch 7/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 30.6437 - mae: 4.2194 - val_loss: 38.1929 - val_mae: 4.8505
Epoch 8/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 24.4160 - mae: 3.8042 - val_loss: 32.7840 - val_mae: 4.5304
Epoch 9/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 21.8807 - mae: 3.4462 - val_loss: 29.7112 - val_mae: 4.3402
Epoch 10/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 25.7278 - mae: 3.5729 - val_loss: 27.8523 - val_mae: 4.1437
Epoch 11/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 24.4294 - mae: 3.4755 - val_loss: 26.2419 - val_mae: 3.9780
Epoch 12/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 17.6161 - mae: 3.1725 - val_loss: 24.8224 - val_mae: 3.8308
Epoch 13/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 20.3679 - mae: 3.0424 - val_loss: 24.3796 - val_mae: 3.7765
Epoch 14/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 22.2973 - mae: 3.1543 - val_loss: 23.3870 - val_mae: 3.6363
Epoch 15/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 15.1733 - mae: 2.7444 - val_loss: 22.8251 - val_mae: 3.5574
Epoch 16/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 12.7597 - mae: 2.5518 - val_loss: 23.4714 - val_mae: 3.5819
Epoch 17/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 14.8287 - mae: 2.7242 - val_loss: 23.3582 - val_mae: 3.5419
Epoch 18/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 17.8166 - mae: 2.7778 - val_loss: 22.8030 - val_mae: 3.4448
Epoch 19/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 12.6734 - mae: 2.
```

```
Epoch 20/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 12.5439 - mae: 2.436
8 - val_loss: 23.1905 - val_mae: 3.4094
Epoch 21/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 10.4825 - mae: 2.394
1 - val_loss: 22.8585 - val_mae: 3.3454
Epoch 22/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 13.0886 - mae: 2.442
7 - val_loss: 23.0387 - val_mae: 3.3022
Epoch 23/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 10.6802 - mae: 2.363
7 - val_loss: 23.2831 - val_mae: 3.3042
Epoch 25/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 10.3403 - mae: 2.372
0 - val_loss: 23.6411 - val_mae: 3.2925
Epoch 26/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 10.0937 - mae: 2.267
9 - val_loss: 22.9478 - val_mae: 3.2070
Epoch 27/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 8.8602 - mae: 2.2062
- val_loss: 22.7643 - val_mae: 3.1868
Epoch 28/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 8.7893 - mae: 2.1755
- val_loss: 23.3269 - val_mae: 3.2083
Epoch 29/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 8.4871 - mae: 2.1653
- val_loss: 23.6982 - val_mae: 3.1967
Epoch 30/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 9.5347 - mae: 2.2231
- val_loss: 22.9556 - val_mae: 3.1389
Epoch 91/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 6.3610 - mae: 1.7426
- val_loss: 18.0887 - val_mae: 2.6157
Epoch 92/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 5.5882 - mae: 1.7113
- val_loss: 17.2416 - val_mae: 2.5923
Epoch 93/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 5.3523 - mae: 1.6432
- val_loss: 18.3930 - val_mae: 2.6622
Epoch 94/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 6.5573 - mae: 1.8276
- val_loss: 18.0743 - val_mae: 2.6563
Epoch 95/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 5.9543 - mae: 1.7180
- val_loss: 16.9853 - val_mae: 2.5752
Epoch 96/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 5.4186 - mae: 1.6574
- val_loss: 16.9465 - val_mae: 2.5560
Epoch 97/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.8210 - mae: 1.5839
- val_loss: 17.7486 - val_mae: 2.6270
Epoch 98/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 5.3396 - mae: 1.7000
- val_loss: 17.1378 - val_mae: 2.6081
Epoch 99/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.8006 - mae: 1.5789
- val_loss: 17.8201 - val_mae: 2.6619
Epoch 100/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.3653 - mae: 1.5546
- val_loss: 17.0820 - val_mae: 2.5868
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 12.4587 - mae: 2.3914
Test Loss: [17.082021713256836, 2.5867905616760254]

4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
Mean Absolute Error: 2.5867906645232557
```



RESULT: Neural network for predicting house prices using Boston Housing Price dataset is exceuted.

EXPERIMENT – 5

Build a Convolution Neural Network for MNIST Hand written Digit Classification.

MNIST Handwritten Digit Classification DataSet :

The MNIST dataset is a popular benchmark dataset for image classification tasks. It consists of 60,000 grayscale images of handwritten digits (0 to 9) for training and 10,000 images for testing. Each image is 28 x 28 pixels in size, and each pixel value ranges from 0 to 255. The goal of the task is to correctly classify each image into one of the 10 possible digit classes.

Program:

In this implementation, we first load the MNIST dataset using the mnist.load_data() function from Keras.

```
from tensorflow.keras.datasets import mnist

import pandas as pd
```

In this step, we use the mnist.load_data() function from Keras to load the MNIST dataset. The training data consists of the x_train images and their corresponding y_train labels, while the test data consists of the x_test images and their corresponding y_test labels.

```
# Load the MNIST dataset

(x_train, y_train), (x_test, y_test) = mnist.load_data()

print('x_train:', x_train.shape)

print('y_train:', y_train.shape)

print('x_test:', x_test.shape)

print('y_test:', y_test.shape)

# Save image parameters to the constants that we will use later for data re-shaping and for model traning.

(_, IMAGE_WIDTH, IMAGE_HEIGHT) = x_train.shape

IMAGE_CHANNELS = 1

print('IMAGE_WIDTH:', IMAGE_WIDTH);

print('IMAGE_HEIGHT:', IMAGE_HEIGHT);
```

```
print('IMAGE_CHANNELS:', IMAGE_CHANNELS);

pd.DataFrame(x_train[0])

plt.imshow(x_train[0], cmap=plt.cm.binary)

plt.show()

# Normalize the pixel values to be between 0 and 1

x_train = x_train / 255.0

x_test = x_test / 255.0

pd.DataFrame(x_train[0])
```

In this step, we preprocess the data by reshaping the images to 1D arrays, normalizing the pixel values to be between 0 and 1, and. . We then preprocess the data by flattening the input images into 1D arrays of size 784 (28x28), scaling the pixel values to the range of 0 to 1, and dividing by 255.0 to normalize the data.

```
# Normalize the pixel values to be between 0 and 1

x_train = x_train / 255.0

x_test = x_test / 255.0

import numpy as np

# Reshape the data to add a channel dimension

x_train = np.expand_dims(x_train, axis=-1)

x_test = np.expand_dims(x_test, axis=-1)
```

The next step is to define the CNN architecture. For this task, we will use a simple CNN architecture with three convolutional layers with ‘relu’ activation function and followed by two max pooling layers, then a flatten layer and two fully connected (dense) layers. The final output layer will have 10 neurons, one for each digit class, and we will use the softmax activation function to produce probabilities for each class.

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# Define CNN architecture with stride, pooling, and filter details
```

```
model = Sequential()
# Convolutional Layer 1
model.add(Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), activation='relu',
         input_shape=(28, 28, 1), padding="valid", name="Conv1"))
# Max Pooling Layer 1
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding="valid", name="Pool1"))
# Convolutional Layer 2
model.add(Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation='relu',
         padding="same", name="Conv2"))
# Max Pooling Layer 2
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding="valid", name="Pool2"))
# Convolutional Layer 3
model.add(Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation='relu',
         padding="same", name="Conv3"))
# Flatten Layer
model.add(Flatten(name="Flatten"))
# Fully Connected Layer
model.add(Dense(64, activation='relu', name="Dense1"))
# Dropout Layer (Regularization)
model.add(Dropout(0.5, name="Dropout"))
# Output Layer (10 classes for digits 0-9)
model.add(Dense(10, activation='softmax', name="Output"))
model.summary()
```

We compile the model with the Adam optimizer, sparse_categorical_crossentropy loss, and accuracy metric.

```
# Compile the model
```

```
model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
```

We train the model on the training data for 10 epochs with a batch size of 128. Finally, we evaluate the model on the test data and print the accuracy score.

```
# Train the model
history = model.fit(x_train, y_train, epochs=10,batch_size=128, validation_data=(x_test, y_test))
```

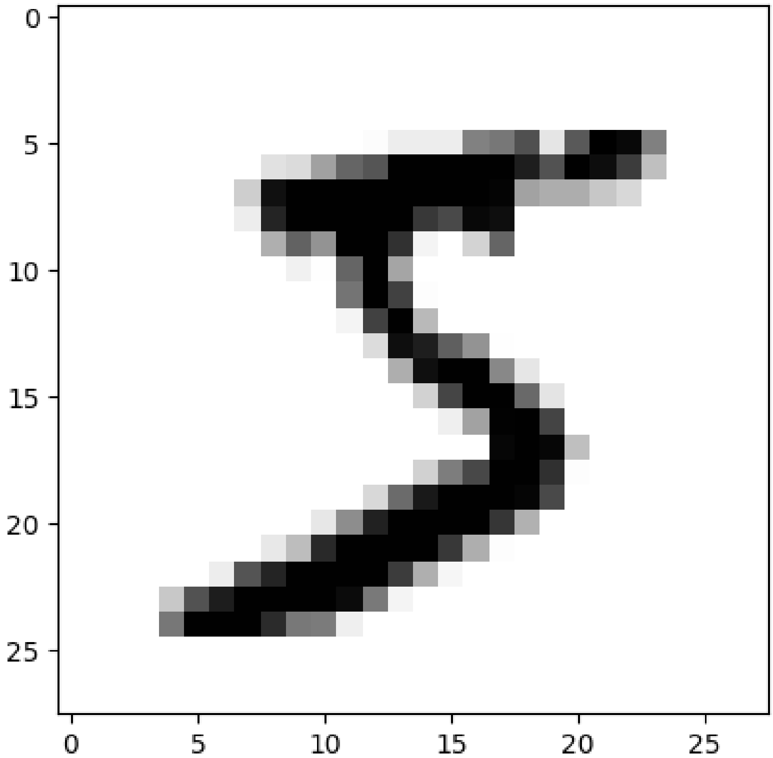
Once the model is trained, the next step is to evaluate its performance on the test data. We will use the evaluate method in Keras to evaluate the model.

```
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test accuracy:', test_acc)
plt.xlabel('Epoch Number')
plt.ylabel('Accuracy')
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Training vs Validation Accuracy')
plt.show()
```

OUTPUT:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets
/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
X_train: (60000, 28, 28)
y_train: (60000,)
X_test: (10000, 28, 28)
y_test: (10000,)
IMAGE_WIDTH: 28
IMAGE_HEIGHT: 28
IMAGE_CHANNELS: 1
```

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**...**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**
0	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	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	...	1	2	1	255	2	12	0	0	0	0



Layer (type)	Output Shape	Param #
conv1 (Conv2D)	(None, 26, 26, 32)	320
pool1 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2 (Conv2D)	(None, 13, 13, 64)	18,496
pool2 (MaxPooling2D)	(None, 6, 6, 64)	0
conv3 (Conv2D)	(None, 6, 6, 64)	36,928
flatten (Flatten)	(None, 2304)	0
dense1 (Dense)	(None, 64)	147,520

Total params: 203,914 (796.54 KB)

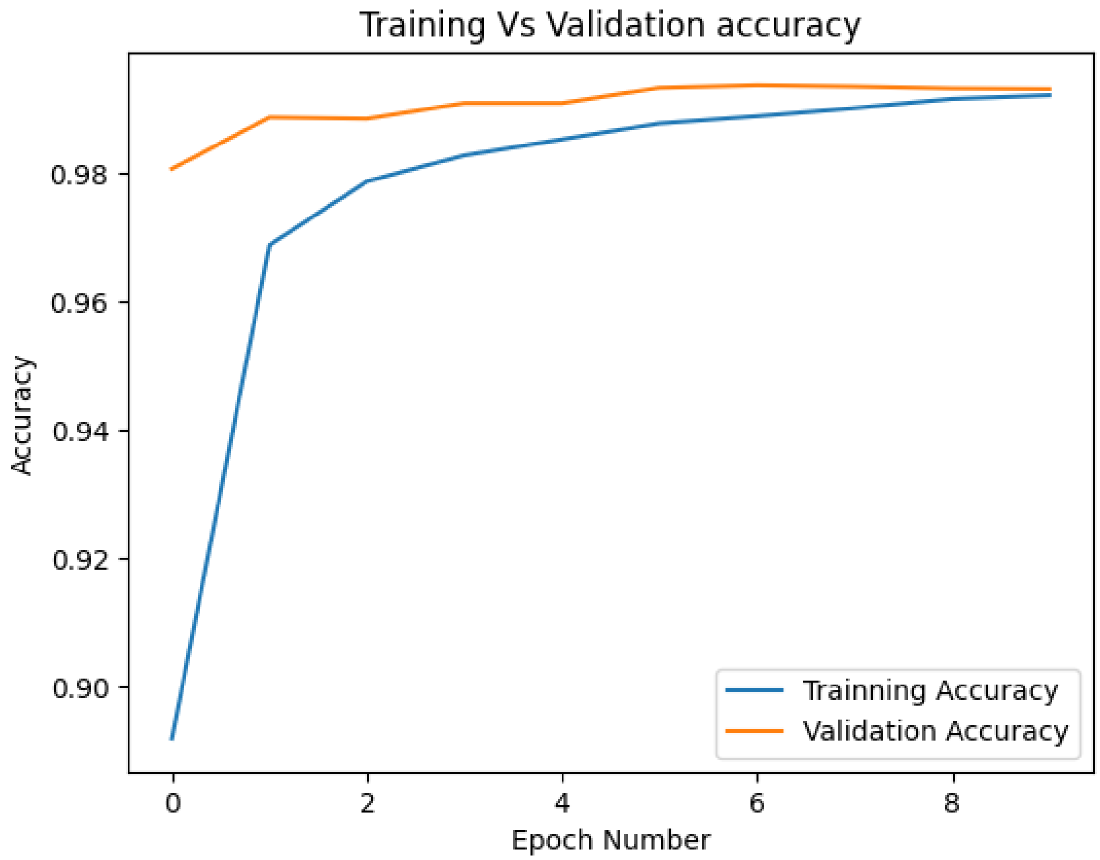
Trainable params: 203,914 (796.54 KB)

Non-trainable params: 0 (0.00 B)

In [21]:

```
Epoch 1/10

469/469 ━━━━━━━━━━━━━━━━━━━━ 86s 178ms/step - accuracy: 0.7741 - lo
ss: 0.6802 - val_accuracy: 0.9806 - val_loss: 0.0564
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 137s 168ms/step - accuracy: 0.9658 - l
oss: 0.1174 - val_accuracy: 0.9886 - val_loss: 0.0350
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 74s 158ms/step - accuracy: 0.9787 - lo
ss: 0.0758 - val_accuracy: 0.9884 - val_loss: 0.0348
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 83s 160ms/step - accuracy: 0.9818 - lo
ss: 0.0630 - val_accuracy: 0.9908 - val_loss: 0.0312
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 81s 152ms/step - accuracy: 0.9921 - lo
ss: 0.0247 - val_accuracy: 0.9930 - val_loss: 0.0260

313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 12ms/step - accuracy: 0.9902 - loss
: 0.0320
Test Loss: 0.025982458144426346
Test Accuracy: 0.9929999709129333
```

.



RESULT: Convolution Neural Network for MNIST Hand written Digit Classification is executed.

EXPERIMENT NO – 6

Build a Convolution Neural Network for simple image (dogs and Cats) Classification

Image classification is the task of categorizing images into different classes based on their content. In this case, we want to build a model that can distinguish between images of dogs and cats.

Program:

```
# Install unrar if not installed

!apt-get install unrar -y

# Extract the dataset

!unrar x "/content/dogs-vs-cats.rar" "/content/dogs-vs-cats/"

# Verify extraction

!ls "/content/dogs-vs-cats/"

import tensorflow as tf

from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Define dataset directory

dataset_dir = "/content/dogs-vs-cats/"

# Define image size & batch size

IMG_SIZE = (150, 150)

BATCH_SIZE = 32

# Data augmentation for training set

train_datagen = ImageDataGenerator(

    rescale=1.0/255,

    rotation_range=20,

    width_shift_range=0.2,

    height_shift_range=0.2,

    shear_range=0.2,
```

```
    zoom_range=0.2,
    horizontal_flip=True,
    validation_split=0.2  # 80% train, 20% validation
)
# Load training & validation datasets
train_generator = train_datagen.flow_from_directory(
    dataset_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary', # Since it's a binary classification (dogs vs cats)
    subset='training'
)
val_generator = train_datagen.flow_from_directory(
    dataset_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary',
    subset='validation'
)
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
# Build the CNN Model
model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(150, 150, 3), strides=(1,1),
padding="same"),
    MaxPooling2D((2,2), strides=(2,2)),
```

```
    Conv2D(64, (3,3), activation='relu', padding="same"),
    MaxPooling2D((2,2), strides=(2,2)),
    Conv2D(128, (3,3), activation='relu', padding="same"),
    MaxPooling2D((2,2), strides=(2,2)),
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')  # Binary classification (dog or cat)
])
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Model Summary
model.summary()
# Train the model
history = model.fit(
    train_generator,
    validation_data=val_generator,
    epochs=10
)
import matplotlib.pyplot as plt
# Plot Accuracy & Loss
plt.figure(figsize=(12, 4))
# Accuracy
plt.subplot(1, 2, 1)
```

```
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Model Accuracy')
# Loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Model Loss')
plt.show()
import numpy as np
from tensorflow.keras.preprocessing import image
import matplotlib.pyplot as plt # Import matplotlib.pyplot
def predict_image(img_path):
    img = image.load_img(img_path, target_size=(150, 150))
    img_array = image.img_to_array(img) / 255.0
    img_array = np.expand_dims(img_array, axis=0)
    prediction = model.predict(img_array)[0][0]
    if prediction > 0.5:
        label = "Dog"  # Define label
```

```
        confidence = prediction # Define confidence
        print(f"The image is a Dog ({prediction:.2f})")
    else:
        label = "Cat" # Define label
        confidence = 1 - prediction # Define confidence
        print(f"The image is a Cat ({1 - prediction:.2f})")
    # Display the image, indented correctly
    plt.imshow(image.load_img(img_path))
    plt.axis("off")
    plt.title(f"Prediction: {label} ({confidence:.2f})")
    plt.show()
# Example Usage
predict_image("/content/240_F_97589769_t45CqXyzjz0KXwoBZT9PRaWGHRk5hQqQ.jpg
")
```

OUTPUT:

```
Found 4 images belonging to 1 classes.
Found 1 images belonging to 1 classes.
```

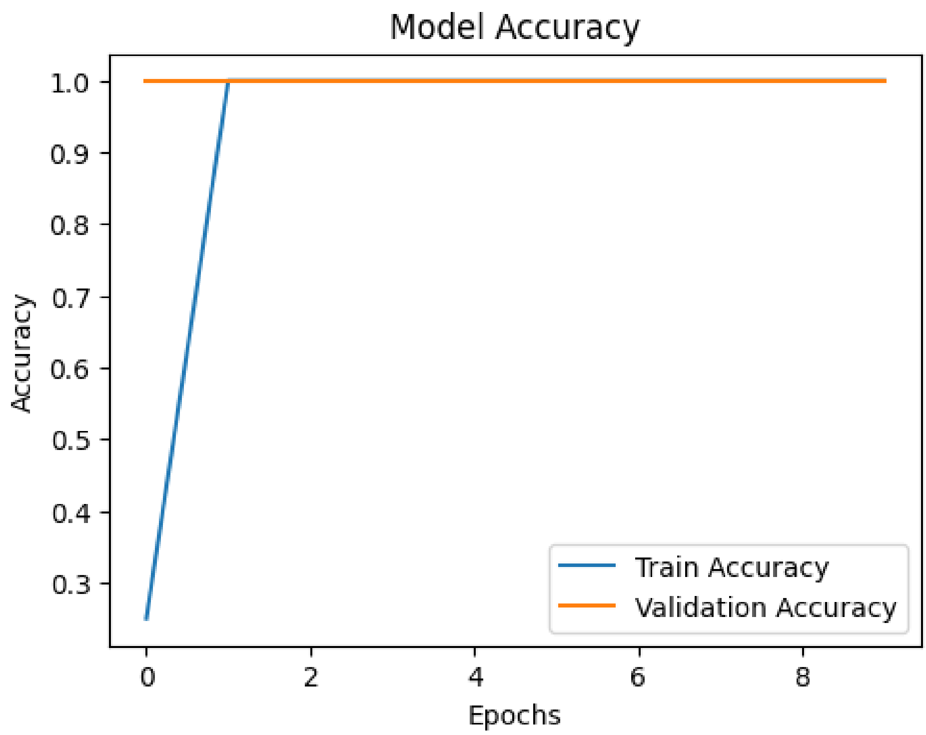
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 150, 150, 32)	896
max_pooling2d (MaxPooling2D)	(None, 75, 75, 32)	0
conv2d_1 (Conv2D)	(None, 75, 75, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 37, 37, 64)	0
conv2d_2 (Conv2D	(None, 37, 37, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 18, 18, 128)	0
dense (Dense)	dense (Dense)	21,234,176

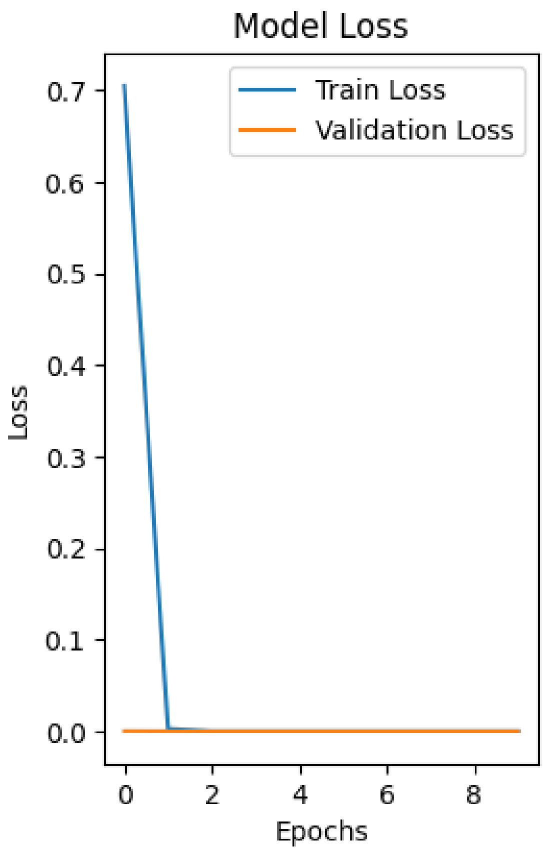
Total params: 21,327,937 (81.36 MB)

Trainable params: 21,327,937 (81.36 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 3s 3s/step - accuracy: 0.2500 - loss: 0.7048 - val_accuracy: 1.0000 - val_loss: 1.2769e-04
Epoch 2/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 1.0000 - loss: 0.0026 - val_accuracy: 1.0000 - val_loss: 2.3025e-09
Epoch 3/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - accuracy: 1.0000 - loss: 5.3029e-06 - val_accuracy: 1.0000 - val_loss: 6.0686e-15
Epoch 4/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 763ms/step - accuracy: 1.0000 - loss: 7.2873e-09 - val_accuracy: 1.0000 - val_loss: 2.0287e-19
Epoch 5/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 764ms/step - accuracy: 1.0000 - loss: 1.4401e-11 - val_accuracy: 1.0000 - val_loss: 3.1340e-26
Epoch 6/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 746ms/step - accuracy: 1.0000 - loss: 1.8211e-13 - val_accuracy: 1.0000 - val_loss: 5.7284e-32
Epoch 7/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 1.0000 - loss: 5.7896e-17 - val_accuracy: 1.0000 - val_loss: 2.5094e-38
Epoch 8/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 738ms/step - accuracy: 1.0000 - loss: 2.4533e-19 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 9/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 764ms/step - accuracy: 1.0000 - loss: 8.4601e-27 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 10/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 750ms/step - accuracy: 1.0000 - loss: 2.0871e-28 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
```






```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 147ms/step
The image is a Cat
```

Prediction: Cat (1.00)

RESULT: Convolution Neural Network for simple image (dogs and Cats) Classification is executed

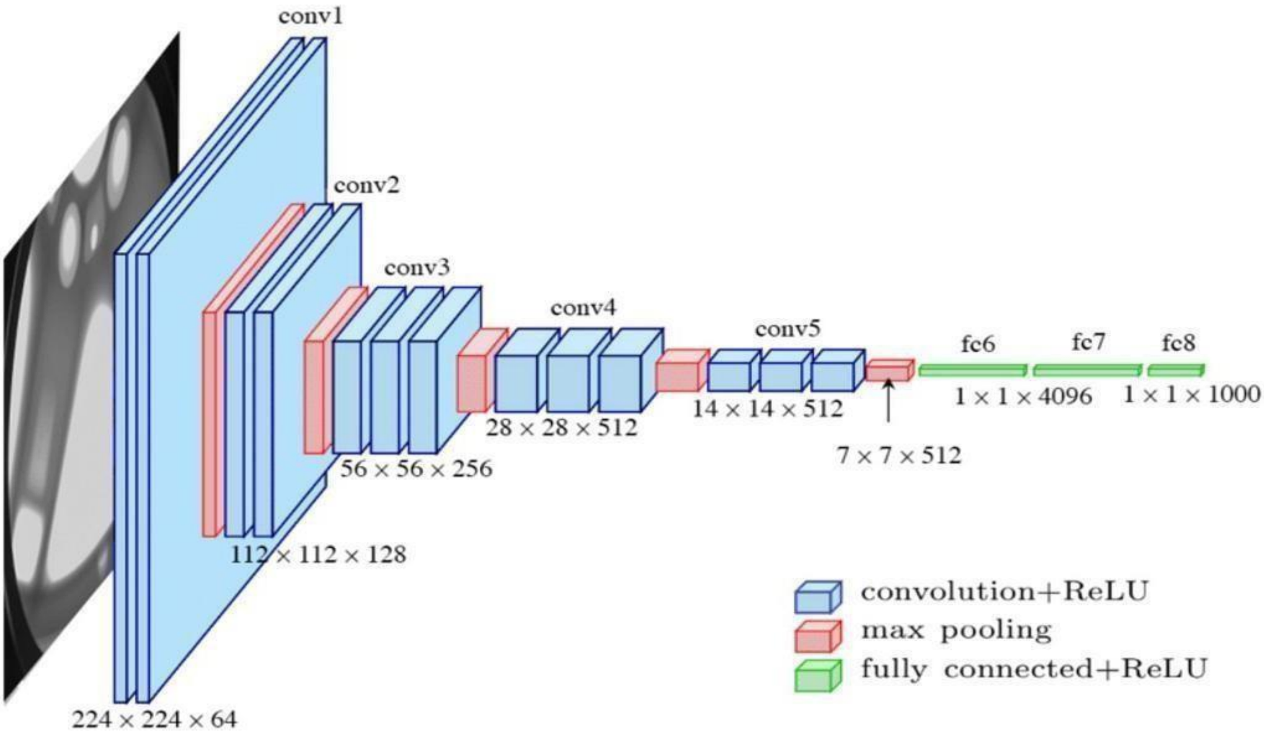
EXPERIMENT N0 – 7

Use a pre-trained convolution neural network (VGG16) for image classification.

Procedure:

VGG16 is a convolutional neural network (CNN) architecture that was developed by researchers at the Visual Geometry Group (VGG) at the University of Oxford. It was introduced in the paper titled "Very Deep Convolutional Networks for Large-Scale Image Recognition" by Karen Simonyan and Andrew Zisserman in 2014.

The VGG16 architecture consists of 16 layers, including 13 convolutional layers and 3 fully connected layers. The input to the network is an RGB image of size 224x224. The network uses small 3x3 convolutional filters throughout the network, which allows the network to learn more complex features with fewer parameters.



Program:

```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
import matplotlib.pyplot as plt

# Load dataset from TensorFlow Datasets (No manual download required)
```

```
dataset_name = "cats_vs_dogs"
dataset, info = tfds.load(dataset_name, as_supervised=True, with_info=True)
# Split dataset into training and validation
train_data = dataset['train'].take(20000)  # First 20,000 for training
val_data = dataset['train'].skip(20000).take(5000) # Next 5,000 for validation
# Function to preprocess images (resize, normalize)
def preprocess(image, label):
    image = tf.image.resize(image, (224, 224)) # Resize to VGG16 expected size
    image = image / 255.0  # Normalize to [0,1]
    return image, label
# Apply preprocessing and batching
train_data = train_data.map(preprocess).batch(32).shuffle(1000)
val_data = val_data.map(preprocess).batch(32)
# Load Pre-trained VGG16 Model (without top layers)
base_model = tf.keras.applications.VGG16(input_shape=(224, 224, 3),
                                         include_top=False, weights='imagenet')

# Freeze the base model (so pre-trained weights are not changed)
base_model.trainable = False
# Add custom classifier on top
model = tf.keras.Sequential([
    base_model,
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1, activation='sigmoid')  # Binary classification
])
# Compile Model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train Model
history = model.fit(train_data, validation_data=val_data, epochs=5)
# Evaluate Model
loss, acc = model.evaluate(val_data)
print(f"\nValidation Accuracy: {acc * 100:.2f}%")
```

```
# Function to display a predicted image
def show_prediction():
    image, label = next(iter(val_data))  # Get a batch
    img = image[0].numpy() # Convert tensor to numpy array
    true_label = label[0].numpy()

    prediction = model.predict(tf.expand_dims(image[0], axis=0))
    predicted_label = "Dog" if prediction[0][0] > 0.5 else "Cat"
    plt.imshow(img)
    plt.title(f"Predicted: {predicted_label}, Actual: {'Dog' if true_label else 'Cat'}")
    plt.axis("off")
    plt.show()
# Show a random predicted image
show_prediction()
```

Output: Epoch 1/3
40/40
Epoch 2/3
40/40
Epoch 3/3
40/40
8/8
7825 19s/step - accuracy: 0.4909 - loss: 0.7935 - val accuracy: 0.4970 - val_loss: 0.6942
778s 19s/step - accuracy: 0.5061 - Loss: 0.6936 - val_accuracy: 0.4970 - val_loss: 0.6921
7485
18s/step - accuracy: 0.5139 - loss: 0.6908 - val accuracy: 0.5730 - val loss: 0.6870
59s 5s/step - accuracy: 0.5688 - loss: 0.6870
Validation Accuracy: 57.30%
1/1- 0s 272ms/step

RESULT: pre-trained convolution neural network (VGG16) for image classification is executed.

EXPERIMENT N0 – 8

Implement one hot encoding of words or characters.

Procedure :

One-hot encoding is a technique used to represent categorical data as numerical data. In the context of natural language processing (NLP), one-hot encoding can be used to represent words or characters as vectors of numbers.

In one-hot encoding, each word or character is assigned a unique index, and a vector of zeros is created with the length equal to the total number of words or characters in the vocabulary. The index of the word or character is set to 1 in the corresponding position in the vector, and all other positions are set to 0.

For example, suppose we have a vocabulary of four words: "apple", "banana", "cherry", and "date". Each word is assigned a unique index: 0, 1, 2, and 3, respectively. The one-hot encoding of the word "banana" would be [0, 1, 0, 0], because it is in the second position in the vocabulary.

In Python, we can implement one-hot encoding using the keras.preprocessing.text.one_hot() function from the Keras library. This function takes as input a list of text strings, the size of the vocabulary, and a hash function to convert words to integers. It returns a list of one-hot encoded vectors.

Program1:

```
from tensorflow.keras.preprocessing.text import one_hot

# Define the list of words
words = ['apple', 'banana', 'cherry', 'apple', 'cherry', 'banana', 'apple']

# Create a vocabulary of unique words
vocab = set(words)

# Assign a unique integer to each word in the vocabulary
word_to_int = {word: i for i, word in enumerate(vocab)}

# Convert the list of words to a list of integers using the vocabulary
int_words = [word_to_int[word] for word in words]
```

```
# Perform one-hot encoding of the integer sequence
one_hot_words = []
for int_word in int_words:
    one_hot_word = [0] * len(vocab)
    one_hot_word[int_word] = 1
    one_hot_words.append(one_hot_word)

print(one_hot_words)
```

Program 2:

```
import string

# Define the input string
input_string = 'hello world'

# Create a vocabulary of unique characters
vocab = set(input_string)

# Assign a unique integer to each character in the vocabulary
char_to_int = {char: i for i, char in enumerate(vocab)}

# Convert the input string to a list of integers using the vocabulary
int_chars = [char_to_int[char] for char in input_string]

# Perform one-hot encoding of the integer sequence
one_hot_chars = []
for int_char in int_chars:
    one_hot_char = [0] * len(vocab)
    one_hot_char[int_char] = 1
    one_hot_chars.append(one_hot_char)

print(one_hot_chars)
```

output:

```
[[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 0, 1], [1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

0, 0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0, 0],
1. [0, 0, 0, 0,0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0],
2. [0, 1, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0]]

RESULT: one hot encoding of words or characters are executed.

EXPERIMENT-9

AIM: Implement word embeddings for IMDB dataset.

Word embedding is essential in natural language processing with deep learning. This technique allows the network to learn about the meaning of the words. In this post, we classify movie reviews in the IMDB dataset as positive or negative, and provide a visual illustration of embedding.

Today is to train a neural network to find out whether some text is globally positive or negative, a task called sentiment analysis.

The first layer of our neural network will perform an operation called word embedding, which is essential in NLP with deep learning.

The IMDB database: We will work with the IMDB dataset, which contains 25,000 movie reviews from IMDB. Each review is labeled as positive or negative from the rating provided by users together with their reviews.

PROGRAM:

```
# get reproducible results

from numpy.random import seed

seed(0xdeadbeef)

import tensorflow as tf

tf.random.set_seed(0xdeadbeef)

from tensorflow import keras

imdb = keras.datasets.imdb

num_words = 20000

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(seed=1, num_words=num_words)

print(train_data[0])

print('label:', train_labels[0])
```

```
# A dictionary mapping words to an integer index

vocabulary = imdb.get_word_index()

# The first indices are reserved

vocabulary = {k:(v+3) for k,v in vocabulary.items()}

vocabulary["<PAD>"] = 0

# See how integer 1 appears first in the review above.

vocabulary["<START>"] = 1

vocabulary["<UNK>"] = 2  # unknown

vocabulary["<UNUSED>"] = 3

# reversing the vocabulary.

# in the index, the key is an integer,

# and the value is the corresponding word.

index = dict([(value, key) for (key, value) in vocabulary.items()])

def decode_review(text):

    '''converts encoded text to human readable form. each integer in the text is looked up in the
index, and replaced by the corresponding word. '''

    return ' '.join([index.get(i, '?') for i in text])

decode_review(train_data[0])

train_data = keras.preprocessing.sequence.pad_sequences(train_data,

                                        value=vocabulary["<PAD>"],

                                        padding='post',

                                        maxlen=256)

test_data = keras.preprocessing.sequence.pad_sequences(test_data,

                                       value=vocabulary["<PAD>"],
```

```
                                padding='post',
                                maxlen=256)

train_data[1]

model = keras.Sequential()

# the first layer is the embedding layer.

# we indicate the number of possible words,

# the dimension of the embedding space,

# and the maximum size of the text.

model.add(keras.layers.Embedding(len(vocabulary), 2, input_length=256))

# the output of the embedding is multidimensional,

# with shape (256, 2)

# for each word, we obtain two values,

# the x and y coordinates

# we flatten this output to be able to

# use it in a dense layer

model.add(keras.layers.Flatten())

# dropout regularization

model.add(keras.layers.Dropout(rate=0.5))

# small dense layer. It's role is to analyze

# the distribution of points from embedding

model.add(keras.layers.Dense(5))

# final neuron, with sigmoid activation

# for binary classification

model.add(keras.layers.Dense(1, activation='sigmoid'))
```

```
model.summary()
model.compile(optimizer='adam',
       loss='binary_crossentropy',
       metrics=['accuracy'])
history = model.fit(train_data,
          train_labels,
          epochs=5,
          batch_size=100,
          validation_data=(test_data, test_labels),
          verbose=1)
import matplotlib.pyplot as plt
def plot_accuracy(history, miny=None):
  acc = history.history['accuracy']
  test_acc = history.history['val_accuracy']
  epochs = range(len(acc))
  plt.plot(epochs, acc)
  plt.plot(epochs, test_acc)
  if miny:
    plt.ylim(miny, 1.0)
plt.title('accuracy')
plt.xlabel('epoch')
plt.figure()
plot_accuracy(history)
```

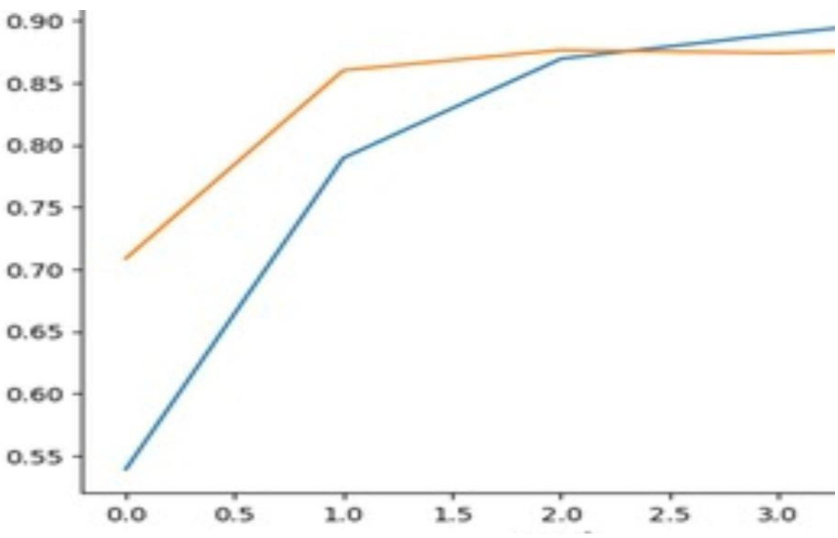
Output:1, 13, 28, 1039, 7, 14, 23, 1856, 13, 104, 1, 13, 28, 1039, 7, 14,

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	?	0
flatten_1 (Flatten)	?	0
dropout_1 (Dropout)	?	0
dense_2 (Dense)	?	0

Trainable params: 0 (0.00 B)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/5
250/250 -5s 14ms/step - accuracy: 0.5138 - loss: 0.6924 - val_accuracy:
0.7088 - val_loss: 0.6578
Epoch 2/5
250/250 -4s 11ms/step - accuracy: 0.7510 - loss: 0.5559 - val_accuracy:
0.8602 - val_loss: 0.3502
Epoch 3/5
250/250 -4s 9ms/step - accuracy: 0.8602 - loss: 0.5559 - val_accuracy:
0.8602 - val_loss: 0.3502

Epoch 4/5
250/250 -4s 15ms/step - accuracy: 0.8872 - loss: 0.5559 - val_accuracy:
0.8602 - val_loss: 0.3502
Epoch 5/5
250/250 -4s 9ms/step - accuracy: 0.9076 - loss: 0.5559 - val_accuracy:
0.8602- val_loss: 0.3502
```



RESULT: word embeddings of IMDB dataset is executed.

EXPERIMENT-10

AIM: Implement a Recurrent Neural Network for IMDB movie review classification problem.

The process of building a Sentiment Analysis model using Recurrent Neural Networks (RNNs) to classify movie reviews as positive or negative. Sentiment analysis is a popular Natural Language Processing (NLP) task that involves determining the sentiment or emotion expressed in a piece of text. We will use Python, TensorFlow, and Keras to implement the RNN model and analyze the results.

Recurrent Neural Networks (RNN) are to the rescue when the sequence of information is needed to be captured (another use case may include Time Series, next word prediction, etc.). Due to its internal memory factor, it remembers past sequences along with current input which makes it capable to capture context rather than just individual words.

The IMDB dataset consists of movie reviews from the IMDb website, along with labels indicating whether each review is “positive” or “negative” based on the reviewer’s opinion. It is designed for binary sentiment classification, with 25,000 reviews for training and an additional 25,000 for testing. Both the training and testing sets have an equal number of positive and negative reviews, making them balanced for sentiment analysis tasks.

PROGRAM:

```
import numpy as np

import pandas as pd

import tensorflow as tf

import re

import matplotlib.pyplot as plt

import seaborn as sns

import pickle

from tensorflow.keras.layers import SimpleRNN, LSTM, GRU, Bidirectional, Dense, Embedding

from tensorflow.keras.datasets import imdb
```

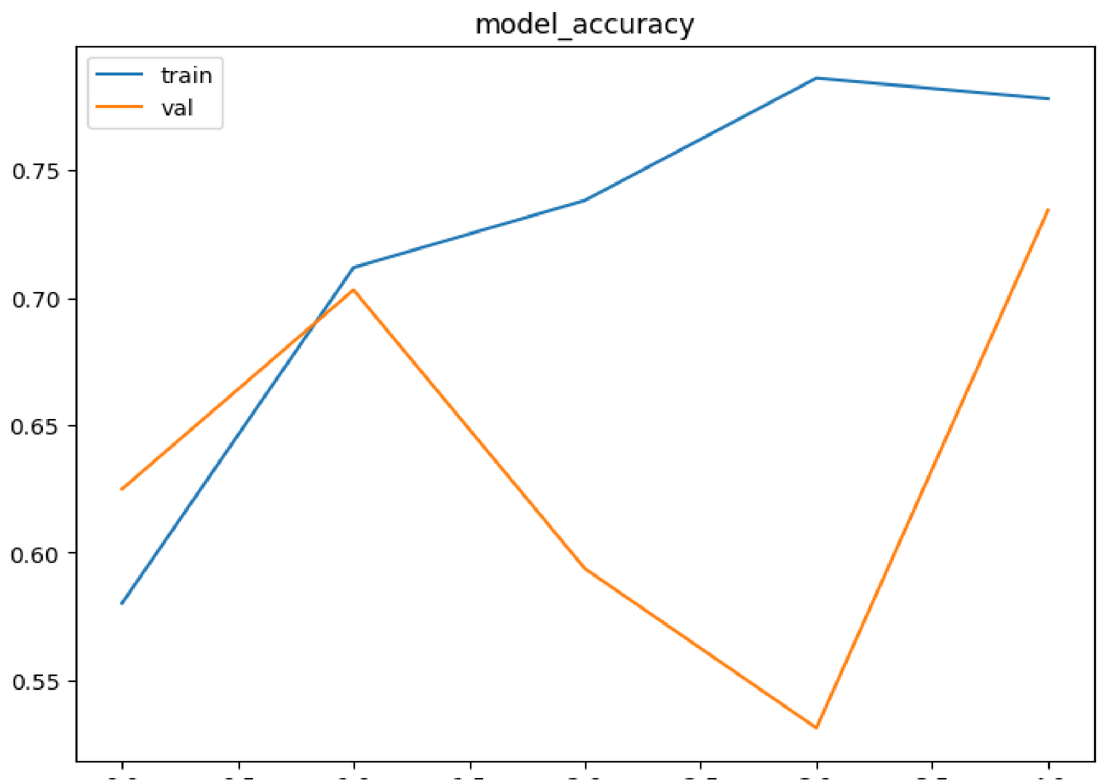
```
from tensorflow.keras.models import Sequential

# Getting reviews with words that come under 5000

# most occurring words in the entire

# corpus of textual review data

vocab_size = 5000

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)

print(x_train[0])

# Getting all the words from word_index dictionary

word_idx = imdb.get_word_index()

# Originally the index number of a value and not a key,

# hence converting the index as key and the words as values

word_idx = {i: word for word, i in word_idx.items()}

# again printing the review

print([word_idx[i] for i in x_train[0]])

# Get the minimum and the maximum length of reviews

print("Max length of a review:: ", len(max((x_train+x_test), key=len)))

print("Min length of a review:: ", len(min((x_train+x_test), key=len)))

from tensorflow.keras.preprocessing import sequence

# Keeping a fixed length of all reviews to max 400 words

max_words = 400

x_train = sequence.pad_sequences(x_train, maxlen=max_words)

x_test = sequence.pad_sequences(x_test, maxlen=max_words)


x_valid, y_valid = x_train[:64], y_train[:64]
```

```
x_train_, y_train_ = x_train[64:], y_train[64:]

# fixing every word's embedding size to be 32

embd_len = 32

# Creating a RNN model

RNN_model = Sequential(name="Simple_RNN")

RNN_model.add(Embedding(vocab_size,

                        embd_len,

                        input_length=max_words))

# In case of a stacked(more than one layer of RNN)

# use return_sequences=True

RNN_model.add(SimpleRNN(128,

                        activation='tanh',

                        return_sequences=False))

RNN_model.add(Dense(1, activation='sigmoid'))

# printing model summary

print(RNN_model.summary())

# Compiling model

RNN_model.compile(

    loss="binary_crossentropy",

    optimizer='adam',

    metrics=['accuracy']

)

# Training the model

history = RNN_model.fit(x_train_, y_train_,
```

```
                batch_size=64,
                epochs=5,
                verbose=1,
                validation_data=(x_valid, y_valid))
# Printing model score on test data
print()
print("Simple_RNN Score---> ", RNN_model.evaluate(x_test, y_test, verbose=0))
plt.title('model_accuracy')
plt.ylabel('accuracy') # Corrected the typo from ylable to ylabel
plt.xlabel('apoch') # Corrected the typo from xlable to xlabel
plt.legend(['train','val'],loc='upper left')
plt.show()
```

Output: [1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173

```
36, 256, 5, 25, 100, 43, 838, 112, 19, 178, 32]
 ['the', 'as', 'you', 'with', 'out', 'themselves', 'powerful', 'lets', 'loves',
  journalist', 'of', 'lot', 'from']

None
Epoch 1/5
 390/390 ━━━━━━━━━━━━━━━━━━━━ 75s 186ms/step - accuracy: 0.5331
   loss: 0.6869 - val_accuracy: 0.6250 - val_loss: 0.6564
Epoch 2/5
390/390 ━━━━━━━━━━━━━━━━━━━━ 83s 188ms/step - accuracy: 0.6920
   - loss: 0.5730 - val_accuracy: 0.7031 - val_los
Epoch 3/5
390/390 ━━━━━━━━━━━━━━━━━━━━ 81s 186ms/step - accuracy:
0.7355 - loss: 0.5290 - val_accuracy: 0.5938 - val_loss: 0.6919
Epoch 4/5
390/390 ━━━━━━━━━━━━━━━━━━━━ 82s 186ms/step - accuracy:
0.7705 - loss: 0.5034 - val_accuracy: 0.5312 - val_loss: 0.7140
Epoch 5/5
  390/390 ━━━━━━━━━━━━━━━━━━━━ 82s 187ms/step - accuracy: 0.7367
val_loss: 0.5816
Simple_RNN Score--->  [0.5270069241523743, 0.7359200119972229]
```



RESULT:

Recurrent Neural Network for IMDB movie review classification problem is executed.