EXPERIMENT NO - 1

1. Implement multilayer perceptron algorithm for MNIST Hand written Digit Classification.

Software and Hardware requirements:

Procedure:

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.

# 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** :

Epoch 1/10

469/469 [==============================] - 6s 9ms/step - loss: 0.2501 - accuracy: 0.9255 - val\_loss: 0.1090 - val\_accuracy: 0.9660

Epoch 2/10

469/469 [==============================] - 4s 9ms/step - loss: 0.1027 - accuracy: 0.9678 - val\_loss: 0.0902 - val\_accuracy: 0.9714

Epoch 3/10

469/469 [==============================] - 4s 8ms/step - loss: 0.0700 - accuracy: 0.9784 - val\_loss: 0.0668 - val\_accuracy: 0.9795

Epoch 4/10

469/469 [==============================] - 4s 8ms/step - loss: 0.0542 - accuracy: 0.9829 - val\_loss: 0.0699 - val\_accuracy: 0.9785

Epoch 5/10

469/469 [==============================] - 5s 10ms/step - loss: 0.0450 - accuracy: 0.9853 - val\_loss: 0.0640 - val\_accuracy: 0.9802

Epoch 6/10

469/469 [==============================] - 5s 11ms/step - loss: 0.0407 - accuracy: 0.9865 - val\_loss: 0.0742 - val\_accuracy: 0.9776

Epoch 7/10

469/469 [==============================] - 4s 9ms/step - loss: 0.0340 - accuracy: 0.9889 - val\_loss: 0.0726 - val\_accuracy: 0.9786

Epoch 8/10

469/469 [==============================] - 4s 9ms/step - loss: 0.0309 - accuracy: 0.9897 - val\_loss: 0.0655 - val\_accuracy: 0.9823

Epoch 9/10

469/469 [==============================] - 5s 11ms/step - loss: 0.0288 - accuracy: 0.9903 - val\_loss: 0.0695 - val\_accuracy: 0.9817

Epoch 10/10

469/469 [==============================] - 5s 10ms/step - loss: 0.0248 - accuracy: 0.9916 - val\_loss: 0.0815 - val\_accuracy: 0.9802

Accuracy: 9.80%

EXPERIMENT NO -2

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

Procedure :

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.

# 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.

Downloading data from [https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz](https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz" \t "_blank)

17464789/17464789 [==============================] - 1s 0us/step

Epoch 1/10

196/196 [==============================] - 2s 7ms/step - loss: 257.8831 - accuracy: 0.4990 - val\_loss: 3.1933 - val\_accuracy: 0.5036

Epoch 2/10

196/196 [==============================] - 1s 6ms/step - loss: 18.6795 - accuracy: 0.5094 - val\_loss: 0.7013 - val\_accuracy: 0.5010

Epoch 3/10

196/196 [==============================] - 1s 6ms/step - loss: 4.7280 - accuracy: 0.4982 - val\_loss: 0.6967 - val\_accuracy: 0.4960

Epoch 4/10

196/196 [==============================] - 1s 6ms/step - loss: 2.4193 - accuracy: 0.5012 - val\_loss: 0.6952 - val\_accuracy: 0.4974

Epoch 5/10

196/196 [==============================] - 1s 5ms/step - loss: 1.5178 - accuracy: 0.5043 - val\_loss: 0.6936 - val\_accuracy: 0.4989

Epoch 6/10

196/196 [==============================] - 1s 6ms/step - loss: 1.2867 - accuracy: 0.5026 - val\_loss: 0.6937 - val\_accuracy: 0.5003

Epoch 7/10

196/196 [==============================] - 1s 6ms/step - loss: 1.1111 - accuracy: 0.5014 - val\_loss: 0.6932 - val\_accuracy: 0.5002

Epoch 8/10

196/196 [==============================] - 2s 8ms/step - loss: 1.0110 - accuracy: 0.4982 - val\_loss: 0.6932 - val\_accuracy: 0.5002

Epoch 9/10

196/196 [==============================] - 1s 7ms/step - loss: 0.8932 - accuracy: 0.4972 - val\_loss: 0.6932 - val\_accuracy: 0.5004

Epoch 10/10

196/196 [==============================] - 1s 6ms/step - loss: 0.8888 - accuracy: 0.4971 - val\_loss: 0.6931 - val\_accuracy: 0.5002

Accuracy: 50.02%

EXPERIMENT NO -3

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

Procedure :

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.

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='adam', 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.

Epoch 1/20

18/18 [==============================] - 2s 53ms/step - loss: 3.5441 - accuracy: 0.1880 - val\_loss: 2.9635 - val\_accuracy: 0.4809

Epoch 2/20

18/18 [==============================] - 1s 38ms/step - loss: 2.6812 - accuracy: 0.4036 - val\_loss: 1.9865 - val\_accuracy: 0.5610

Epoch 3/20

18/18 [==============================] - 1s 44ms/step - loss: 2.0019 - accuracy: 0.5345 - val\_loss: 1.6289 - val\_accuracy: 0.6394

Epoch 4/20

18/18 [==============================] - 1s 38ms/step - loss: 1.6804 - accuracy: 0.6021 - val\_loss: 1.4734 - val\_accuracy: 0.6772

Epoch 5/20

18/18 [==============================] - 1s 39ms/step - loss: 1.4998 - accuracy: 0.6431 - val\_loss: 1.3653 - val\_accuracy: 0.6874

Epoch 6/20

18/18 [==============================] - 1s 39ms/step - loss: 1.3755 - accuracy: 0.6711 - val\_loss: 1.2938 - val\_accuracy: 0.6968

Epoch 7/20

18/18 [==============================] - 1s 44ms/step - loss: 1.2737 - accuracy: 0.6915 - val\_loss: 1.2457 - val\_accuracy: 0.7070

Epoch 8/20

18/18 [==============================] - 1s 38ms/step - loss: 1.1878 - accuracy: 0.7150 - val\_loss: 1.2033 - val\_accuracy: 0.7168

Epoch 9/20

18/18 [==============================] - 1s 66ms/step - loss: 1.0933 - accuracy: 0.7280 - val\_loss: 1.1682 - val\_accuracy: 0.7320

Epoch 10/20

18/18 [==============================] - 1s 61ms/step - loss: 1.0417 - accuracy: 0.7455 - val\_loss: 1.1414 - val\_accuracy: 0.7480

Epoch 11/20

18/18 [==============================] - 1s 38ms/step - loss: 0.9840 - accuracy: 0.7562 - val\_loss: 1.1134 - val\_accuracy: 0.7547

Epoch 12/20

18/18 [==============================] - 1s 38ms/step - loss: 0.9252 - accuracy: 0.7701 - val\_loss: 1.0935 - val\_accuracy: 0.7631

Epoch 13/20

18/18 [==============================] - 1s 38ms/step - loss: 0.8799 - accuracy: 0.7859 - val\_loss: 1.0739 - val\_accuracy: 0.7694

Epoch 14/20

18/18 [==============================] - 1s 38ms/step - loss: 0.8282 - accuracy: 0.7931 - val\_loss: 1.0629 - val\_accuracy: 0.7703

Epoch 15/20

18/18 [==============================] - 1s 42ms/step - loss: 0.7916 - accuracy: 0.7973 - val\_loss: 1.0557 - val\_accuracy: 0.7720

Epoch 16/20

18/18 [==============================] - 1s 38ms/step - loss: 0.7636 - accuracy: 0.8055 - val\_loss: 1.0515 - val\_accuracy: 0.7787

Epoch 17/20

18/18 [==============================] - 1s 38ms/step - loss: 0.7293 - accuracy: 0.8127 - val\_loss: 1.0626 - val\_accuracy: 0.7743

Epoch 18/20

18/18 [==============================] - 1s 38ms/step - loss: 0.7039 - accuracy: 0.8233 - val\_loss: 1.0591 - val\_accuracy: 0.7765

Epoch 19/20

18/18 [==============================] - 1s 39ms/step - loss: 0.6743 - accuracy: 0.8253 - val\_loss: 1.0495 - val\_accuracy: 0.7809

Epoch 20/20

18/18 [==============================] - 1s 38ms/step - loss: 0.6579 - accuracy: 0.8272 - val\_loss: 1.0723 - val\_accuracy: 0.7769

71/71 [==============================] - 0s 4ms/step - loss: 1.0723 - accuracy: 0.7769

Test accuracy: 0.7769367694854736

EXPERIMENT NO -4

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

Procedure:

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.

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.

# Normalize the data

x\_train = normalize(x\_train, axis=0)

x\_test = normalize(x\_test, axis=0)

# 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.

Downloading data from [https://storage.googleapis.com/tensorflow/tf-keras-datasets/boston\_housing.npz](https://storage.googleapis.com/tensorflow/tf-keras-datasets/boston_housing.npz" \t "_blank)

57026/57026 [==============================] - 0s 0us/step

Epoch 1/100

13/13 [==============================] - 3s 71ms/step - loss: 581.6057 - val\_loss: 600.8628

Epoch 2/100

13/13 [==============================] - 0s 10ms/step - loss: 568.9142 - val\_loss: 578.0995

Epoch 3/100

13/13 [==============================] - 0s 11ms/step - loss: 547.0474 - val\_loss: 539.9392

Epoch 4/100

13/13 [==============================] - 0s 9ms/step - loss: 510.9485 - val\_loss: 479.1323

Epoch 5/100

13/13 [==============================] - 0s 6ms/step - loss: 455.0125 - val\_loss: 393.1474

Epoch 6/100

13/13 [==============================] - 0s 7ms/step - loss: 379.5060 - val\_loss: 288.2769

Epoch 7/100

13/13 [==============================] - 0s 6ms/step - loss: 289.0050 - val\_loss: 187.7191

Epoch 8/100

13/13 [==============================] - 0s 6ms/step - loss: 198.9047 - val\_loss: 123.9099

Epoch 9/100

13/13 [==============================] - 0s 5ms/step - loss: 133.4827 - val\_loss: 118.9333

Epoch 10/100

13/13 [==============================] - 0s 5ms/step - loss: 103.5629 - val\_loss: 148.6580

Epoch 11/100

13/13 [==============================] - 0s 5ms/step - loss: 96.0384 - val\_loss: 157.2076

Epoch 12/100

13/13 [==============================] - 0s 6ms/step - loss: 94.3371 - val\_loss: 151.7262

Epoch 13/100

13/13 [==============================] - 0s 6ms/step - loss: 91.8126 - val\_loss: 139.2459

Epoch 14/100

13/13 [==============================] - 0s 5ms/step - loss: 89.3874 - val\_loss: 128.9134

Epoch 15/100

13/13 [==============================] - 0s 7ms/step - loss: 87.6433 - val\_loss: 122.3881

Epoch 16/100

13/13 [==============================] - 0s 7ms/step - loss: 85.5203 - val\_loss: 118.5746

Epoch 17/100

13/13 [==============================] - 0s 6ms/step - loss: 83.6930 - val\_loss: 117.3677

Epoch 18/100

13/13 [==============================] - 0s 6ms/step - loss: 81.8004 - val\_loss: 107.7317

Epoch 19/100

13/13 [==============================] - 0s 6ms/step - loss: 80.0276 - val\_loss: 108.8664

Epoch 20/100

13/13 [==============================] - 0s 6ms/step - loss: 78.1314 - val\_loss: 101.8108

Epoch 21/100

13/13 [==============================] - 0s 5ms/step - loss: 76.4282 - val\_loss: 97.4463

Epoch 22/100

13/13 [==============================] - 0s 7ms/step - loss: 74.9360 - val\_loss: 96.3559

Epoch 23/100

13/13 [==============================] - 0s 6ms/step - loss: 73.3877 - val\_loss: 87.7806

Epoch 24/100

13/13 [==============================] - 0s 7ms/step - loss: 71.7510 - val\_loss: 89.5797

Epoch 25/100

13/13 [==============================] - 0s 8ms/step - loss: 70.0859 - val\_loss: 84.9000

Epoch 26/100

13/13 [==============================] - 0s 9ms/step - loss: 68.7180 - val\_loss: 81.1061

Epoch 27/100

13/13 [==============================] - 0s 8ms/step - loss: 67.2199 - val\_loss: 80.6916

Epoch 28/100

13/13 [==============================] - 0s 6ms/step - loss: 65.8538 - val\_loss: 78.3895

Epoch 29/100

13/13 [==============================] - 0s 6ms/step - loss: 64.5018 - val\_loss: 75.4445

Epoch 30/100

13/13 [==============================] - 0s 9ms/step - loss: 63.2297 - val\_loss: 75.0658

Epoch 31/100

13/13 [==============================] - 0s 6ms/step - loss: 62.0135 - val\_loss: 72.5331

Epoch 32/100

13/13 [==============================] - 0s 6ms/step - loss: 61.0150 - val\_loss: 72.5535

Epoch 33/100

13/13 [==============================] - 0s 6ms/step - loss: 59.7378 - val\_loss: 70.0550

Epoch 34/100

13/13 [==============================] - 0s 6ms/step - loss: 58.8055 - val\_loss: 71.9888

Epoch 35/100

13/13 [==============================] - 0s 6ms/step - loss: 57.8878 - val\_loss: 70.7422

Epoch 36/100

13/13 [==============================] - 0s 6ms/step - loss: 57.4425 - val\_loss: 68.0706

Epoch 37/100

13/13 [==============================] - 0s 6ms/step - loss: 56.2824 - val\_loss: 73.5046

Epoch 38/100

13/13 [==============================] - 0s 6ms/step - loss: 55.7446 - val\_loss: 72.7915

Epoch 39/100

13/13 [==============================] - 0s 6ms/step - loss: 55.0651 - val\_loss: 71.6527

Epoch 40/100

13/13 [==============================] - 0s 7ms/step - loss: 54.5890 - val\_loss: 71.9477

Epoch 41/100

13/13 [==============================] - 0s 5ms/step - loss: 54.0848 - val\_loss: 74.4059

Epoch 42/100

13/13 [==============================] - 0s 6ms/step - loss: 53.6057 - val\_loss: 72.6524

Epoch 43/100

13/13 [==============================] - 0s 5ms/step - loss: 53.2400 - val\_loss: 72.7631

Epoch 44/100

13/13 [==============================] - 0s 7ms/step - loss: 52.7545 - val\_loss: 74.9595

Epoch 45/100

13/13 [==============================] - 0s 7ms/step - loss: 52.3954 - val\_loss: 74.4425

Epoch 46/100

13/13 [==============================] - 0s 8ms/step - loss: 51.8754 - val\_loss: 75.4689

Epoch 47/100

13/13 [==============================] - 0s 6ms/step - loss: 51.5840 - val\_loss: 75.5107

Epoch 48/100

13/13 [==============================] - 0s 7ms/step - loss: 51.3794 - val\_loss: 74.1811

Epoch 49/100

13/13 [==============================] - 0s 7ms/step - loss: 50.7736 - val\_loss: 77.1227

Epoch 50/100

13/13 [==============================] - 0s 6ms/step - loss: 50.6973 - val\_loss: 75.3571

Epoch 51/100

13/13 [==============================] - 0s 7ms/step - loss: 50.3545 - val\_loss: 75.5459

Epoch 52/100

13/13 [==============================] - 0s 9ms/step - loss: 50.0110 - val\_loss: 74.9539

Epoch 53/100

13/13 [==============================] - 0s 7ms/step - loss: 49.5859 - val\_loss: 75.7610

Epoch 54/100

13/13 [==============================] - 0s 7ms/step - loss: 49.2332 - val\_loss: 75.3416

Epoch 55/100

13/13 [==============================] - 0s 7ms/step - loss: 48.9296 - val\_loss: 74.7842

Epoch 56/100

13/13 [==============================] - 0s 7ms/step - loss: 48.5693 - val\_loss: 73.6319

Epoch 57/100

13/13 [==============================] - 0s 7ms/step - loss: 48.2237 - val\_loss: 74.5521

Epoch 58/100

13/13 [==============================] - 0s 5ms/step - loss: 47.9942 - val\_loss: 74.2557

Epoch 59/100

13/13 [==============================] - 0s 5ms/step - loss: 47.6148 - val\_loss: 72.9914

Epoch 60/100

13/13 [==============================] - 0s 7ms/step - loss: 47.3916 - val\_loss: 73.7292

Epoch 61/100

13/13 [==============================] - 0s 5ms/step - loss: 47.1164 - val\_loss: 71.3105

Epoch 62/100

13/13 [==============================] - 0s 7ms/step - loss: 46.8377 - val\_loss: 73.0056

Epoch 63/100

13/13 [==============================] - 0s 7ms/step - loss: 46.1684 - val\_loss: 71.6649

Epoch 64/100

13/13 [==============================] - 0s 7ms/step - loss: 45.9518 - val\_loss: 71.3532

Epoch 65/100

13/13 [==============================] - 0s 6ms/step - loss: 45.8064 - val\_loss: 71.3245

Epoch 66/100

13/13 [==============================] - 0s 6ms/step - loss: 45.3031 - val\_loss: 70.0585

Epoch 67/100

13/13 [==============================] - 0s 7ms/step - loss: 44.8652 - val\_loss: 70.1172

Epoch 68/100

13/13 [==============================] - 0s 6ms/step - loss: 44.7074 - val\_loss: 69.8758

Epoch 69/100

13/13 [==============================] - 0s 5ms/step - loss: 44.4409 - val\_loss: 69.5317

Epoch 70/100

13/13 [==============================] - 0s 7ms/step - loss: 43.9173 - val\_loss: 68.3756

Epoch 71/100

13/13 [==============================] - 0s 7ms/step - loss: 43.6754 - val\_loss: 68.1564

Epoch 72/100

13/13 [==============================] - 0s 7ms/step - loss: 43.0529 - val\_loss: 67.8396

Epoch 73/100

13/13 [==============================] - 0s 6ms/step - loss: 42.8460 - val\_loss: 67.5689

Epoch 74/100

13/13 [==============================] - 0s 8ms/step - loss: 42.5708 - val\_loss: 67.1981

Epoch 75/100

13/13 [==============================] - 0s 7ms/step - loss: 42.2138 - val\_loss: 66.7805

Epoch 76/100

13/13 [==============================] - 0s 7ms/step - loss: 41.8676 - val\_loss: 66.0953

Epoch 77/100

13/13 [==============================] - 0s 7ms/step - loss: 41.4192 - val\_loss: 65.9270

Epoch 78/100

13/13 [==============================] - 0s 8ms/step - loss: 40.9319 - val\_loss: 65.6209

Epoch 79/100

13/13 [==============================] - 0s 6ms/step - loss: 40.6032 - val\_loss: 64.8652

Epoch 80/100

13/13 [==============================] - 0s 8ms/step - loss: 40.3409 - val\_loss: 65.0239

Epoch 81/100

13/13 [==============================] - 0s 6ms/step - loss: 40.0193 - val\_loss: 64.4650

Epoch 82/100

13/13 [==============================] - 0s 6ms/step - loss: 39.5756 - val\_loss: 64.0286

Epoch 83/100

13/13 [==============================] - 0s 6ms/step - loss: 39.0019 - val\_loss: 63.1940

Epoch 84/100

13/13 [==============================] - 0s 6ms/step - loss: 39.1355 - val\_loss: 63.2680

Epoch 85/100

13/13 [==============================] - 0s 6ms/step - loss: 38.4614 - val\_loss: 63.4833

Epoch 86/100

13/13 [==============================] - 0s 6ms/step - loss: 38.2256 - val\_loss: 62.5658

Epoch 87/100

13/13 [==============================] - 0s 7ms/step - loss: 37.5265 - val\_loss: 63.4185

Epoch 88/100

13/13 [==============================] - 0s 7ms/step - loss: 37.2877 - val\_loss: 62.3244

Epoch 89/100

13/13 [==============================] - 0s 5ms/step - loss: 37.2992 - val\_loss: 61.4046

Epoch 90/100

13/13 [==============================] - 0s 4ms/step - loss: 36.6447 - val\_loss: 62.3672

Epoch 91/100

13/13 [==============================] - 0s 4ms/step - loss: 36.2271 - val\_loss: 60.2370

Epoch 92/100

13/13 [==============================] - 0s 6ms/step - loss: 36.5574 - val\_loss: 61.8120

Epoch 93/100

13/13 [==============================] - 0s 5ms/step - loss: 35.8656 - val\_loss: 60.5489

Epoch 94/100

13/13 [==============================] - 0s 4ms/step - loss: 35.2197 - val\_loss: 62.9568

Epoch 95/100

13/13 [==============================] - 0s 5ms/step - loss: 34.7053 - val\_loss: 61.1308

Epoch 96/100

13/13 [==============================] - 0s 4ms/step - loss: 34.6180 - val\_loss: 62.5150

Epoch 97/100

13/13 [==============================] - 0s 5ms/step - loss: 34.1021 - val\_loss: 62.0751

Epoch 98/100

13/13 [==============================] - 0s 5ms/step - loss: 33.7709 - val\_loss: 61.6955

Epoch 99/100

13/13 [==============================] - 0s 5ms/step - loss: 33.4811 - val\_loss: 61.1410

Epoch 100/100

13/13 [==============================] - 0s 6ms/step - loss: 33.0442 - val\_loss: 61.5967

4/4 [==============================] - 0s 3ms/step - loss: 61.5967

Test loss: 61.5966682434082

EXPERIMENT NO – 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.

from tensorflow.keras.datasets import mnist

# Load the MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

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

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.

# Normalize the pixel values to be between 0 and 1

x\_train = x\_train / 255.0

x\_test = x\_test / 255.0

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.

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz

11490434/11490434 [==============================] - 0s 0us/step

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)

from tensorflow.keras.models import Sequential

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

# Define the CNN architecture

model = Sequential()

model.add(Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)))

model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(Flatten())

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

model.add(Dropout(0.5))

model.add(Dense(10, activation='softmax'))

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.

# Compile the model

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

We compile the model with the Adam optimizer, sparse\_categorical\_crossentropy loss, and accuracy metric.

# Train the model

history = model.fit(x\_train, y\_train, epochs=10, validation\_data=(x\_test, y\_test))

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.

Epoch 1/10

1875/1875 [==============================] - 72s 38ms/step - loss: 0.2613 - accuracy: 0.9220 - val\_loss: 0.0447 - val\_accuracy: 0.9859

Epoch 2/10

1875/1875 [==============================] - 66s 35ms/step - loss: 0.0927 - accuracy: 0.9752 - val\_loss: 0.0491 - val\_accuracy: 0.9852

Epoch 3/10

1875/1875 [==============================] - 69s 37ms/step - loss: 0.0650 - accuracy: 0.9821 - val\_loss: 0.0339 - val\_accuracy: 0.9892

Epoch 4/10

1875/1875 [==============================] - 68s 36ms/step - loss: 0.0516 - accuracy: 0.9855 - val\_loss: 0.0328 - val\_accuracy: 0.9895

Epoch 5/10

1875/1875 [==============================] - 68s 36ms/step - loss: 0.0413 - accuracy: 0.9887 - val\_loss: 0.0315 - val\_accuracy: 0.9907

Epoch 6/10

1875/1875 [==============================] - 65s 35ms/step - loss: 0.0359 - accuracy: 0.9898 - val\_loss: 0.0270 - val\_accuracy: 0.9925

Epoch 7/10

1875/1875 [==============================] - 65s 35ms/step - loss: 0.0312 - accuracy: 0.9910 - val\_loss: 0.0278 - val\_accuracy: 0.9920

Epoch 8/10

1875/1875 [==============================] - 68s 36ms/step - loss: 0.0283 - accuracy: 0.9920 - val\_loss: 0.0365 - val\_accuracy: 0.9908

Epoch 9/10

1875/1875 [==============================] - 65s 35ms/step - loss: 0.0233 - accuracy: 0.9931 - val\_loss: 0.0324 - val\_accuracy: 0.9927

Epoch 10/10

1875/1875 [==============================] - 67s 36ms/step - loss: 0.0197 - accuracy: 0.9938 - val\_loss: 0.0346 - val\_accuracy: 0.9941

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print('Test accuracy:', test\_acc)

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.

313/313 [==============================] - 4s 11ms/step - loss: 0.0346 - accuracy: 0.9941

Test accuracy: 0.9940999746322632

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.

# import the libraries as shown below

from tensorflow.keras.layers import Input, Lambda, Dense, Flatten

from tensorflow.keras.models import Model

from tensorflow.keras.applications.vgg16 import VGG16

from tensorflow.keras.preprocessing import image

from tensorflow.keras.preprocessing.image import ImageDataGenerator,load\_img

from tensorflow.keras.models import Sequential

import numpy as np

from glob import glob

in this step we are importing the required modules from keras and tensorflow . Here we are using the pre-defined neural network called VGG16 .

from google.colab import drive

drive.mount('/content/drive')

In this step we are mounting our Google colab with our drive

Mounted at /content/drive

ROOT\_PATH = '/content/drive/MyDrive/cat-dog-project-20230412T173959Z-001/cat-dog-project'

!pwd

/content

import os

os.chdir(ROOT\_PATH)

os.getcwd()

/content/drive/MyDrive/cat-dog-project-20230412T173959Z-001/cat-dog-project

# re-size all the images to this

IMAGE\_SIZE = [224, 224]

train\_path = 'PetImages/train'

valid\_path = 'PetImages/validation'

# Import the VGG16 library as shown below and add preprocessing layer to the front of VGG

# Here we will be using imagenet weights

vgg16 = VGG16(input\_shape=IMAGE\_SIZE + [3], weights='imagenet', include\_top=False)

Downloading data from [https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5](https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5" \t "_blank)

58889256/58889256 [==============================] - 0s 0us/step

# don't train existing weights

for layer in vgg16.layers:

  print(layer)

<keras.engine.input\_layer.InputLayer object at 0x7fa764e92190>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa764e92b20>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7645e4310>

<keras.layers.pooling.max\_pooling2d.MaxPooling2D object at 0x7fa7645ae6d0>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7645e4910>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7645c6880>

<keras.layers.pooling.max\_pooling2d.MaxPooling2D object at 0x7fa7640cf4f0>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7645c61f0>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7640d2970>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7640d9610>

<keras.layers.pooling.max\_pooling2d.MaxPooling2D object at 0x7fa7640df700>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7640e5190>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7640e5d30>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7640df8b0>

<keras.layers.pooling.max\_pooling2d.MaxPooling2D object at 0x7fa7640efca0>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7640ef760>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa7640d9970>

<keras.layers.convolutional.conv2d.Conv2D object at 0x7fa76407c520>

<keras.layers.pooling.max\_pooling2d.MaxPooling2D object at 0x7fa7640f57f0>

# don't train existing weights

for layer in vgg16.layers:

    layer.trainable = False

for layer in vgg16.layers:

  print(layer.name,layer.trainable)

input\_1 False

block1\_conv1 False

block1\_conv2 False

block1\_pool False

block2\_conv1 False

block2\_conv2 False

block2\_pool False

block3\_conv1 False

block3\_conv2 False

block3\_conv3 False

block3\_pool False

block4\_conv1 False

block4\_conv2 False

block4\_conv3 False

block4\_pool False

block5\_conv1 False

block5\_conv2 False

block5\_conv3 False

block5\_pool False

vgg16.summary()

Model: "vgg16"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) [(None, 224, 224, 3)] 0

block1\_conv1 (Conv2D) (None, 224, 224, 64) 1792

block1\_conv2 (Conv2D) (None, 224, 224, 64) 36928

block1\_pool (MaxPooling2D) (None, 112, 112, 64) 0

block2\_conv1 (Conv2D) (None, 112, 112, 128) 73856

block2\_conv2 (Conv2D) (None, 112, 112, 128) 147584

block2\_pool (MaxPooling2D) (None, 56, 56, 128) 0

block3\_conv1 (Conv2D) (None, 56, 56, 256) 295168

block3\_conv2 (Conv2D) (None, 56, 56, 256) 590080

block3\_conv3 (Conv2D) (None, 56, 56, 256) 590080

block3\_pool (MaxPooling2D) (None, 28, 28, 256) 0

block4\_conv1 (Conv2D) (None, 28, 28, 512) 1180160

block4\_conv2 (Conv2D) (None, 28, 28, 512) 2359808

block4\_conv3 (Conv2D) (None, 28, 28, 512) 2359808

block4\_pool (MaxPooling2D) (None, 14, 14, 512) 0

block5\_conv1 (Conv2D) (None, 14, 14, 512) 2359808

block5\_conv2 (Conv2D) (None, 14, 14, 512) 2359808

block5\_conv3 (Conv2D) (None, 14, 14, 512) 2359808

block5\_pool (MaxPooling2D) (None, 7, 7, 512) 0

=================================================================

Total params: 14,714,688

Trainable params: 0

Non-trainable params: 14,714,688

  # useful for getting number of output classes

folders = glob('PetImages/train/\*')

len(folders)

model = Sequential()

model.add(vgg16)

model.add(Flatten())

model.add(Dense(256,activation='relu'))

model.add(Dense(2,activation='softmax'))

# view the structure of the model

model.summary()

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

vgg16 (Functional) (None, 7, 7, 512) 14714688

flatten (Flatten) (None, 25088) 0

dense (Dense) (None, 256) 6422784

dense\_1 (Dense) (None, 2) 514

=================================================================

Total params: 21,137,986

Trainable params: 6,423,298

Non-trainable params: 14,714,688

# tell the model what cost and optimization method to use

model.compile(

  loss='categorical\_crossentropy',

  optimizer='adam',

  metrics=['accuracy']

)

# Use the Image Data Generator to import the images from the dataset

from tensorflow.keras.preprocessing.image import ImageDataGenerator

train\_datagen = ImageDataGenerator(rescale = 1./255,

                                   shear\_range = 0.2,

                                   zoom\_range = 0.2,

                                   horizontal\_flip = True)

test\_datagen = ImageDataGenerator(rescale = 1./255)

# Make sure you provide the same target size as initialied for the image size

training\_set = train\_datagen.flow\_from\_directory('PetImages/train',

                                                 target\_size = (224, 224),

                                                 batch\_size = 32,

                                                 class\_mode = 'categorical')

test\_set = test\_datagen.flow\_from\_directory('PetImages/validation',

                                            target\_size = (224, 224),

                                            batch\_size = 32,

                                            class\_mode = 'categorical')

# fit the model

# Run the cell. It will take some time to execute

r = model.fit(

  training\_set,

  validation\_data=test\_set,

  epochs=10,

  steps\_per\_epoch=len(training\_set),

  validation\_steps=len(test\_set)

)

import matplotlib.pyplot as plt

# plot the loss

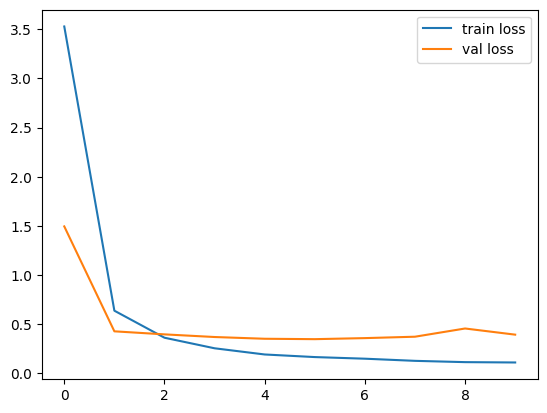
plt.plot(r.history['loss'], label='train loss')

plt.plot(r.history['val\_loss'], label='val loss')

plt.legend()

plt.show()

plt.savefig('LossVal\_loss')



# plot the accuracy

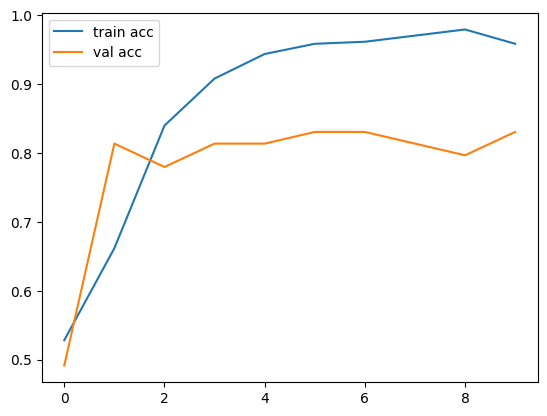
plt.plot(r.history['accuracy'], label='train acc')

plt.plot(r.history['val\_accuracy'], label='val acc')

plt.legend()

plt.show()

plt.savefig('AccVal\_acc')



# save it as a h5 file

from tensorflow.keras.models import load\_model

model.save('model\_vgg16.h5')

y\_pred = model.predict(test\_set)

y\_pred

import numpy as np

y\_pred = np.argmax(y\_pred, axis=1)

y\_pred

from tensorflow.keras.models import load\_model

from tensorflow.keras.preprocessing import image

model=load\_model('model\_vgg16.h5')

img=image.load\_img('cat.jpg',target\_size=(224,224))

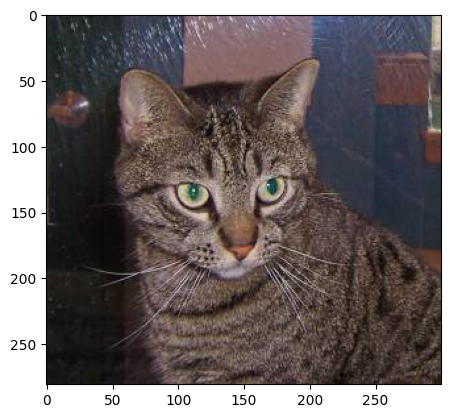
x=image.img\_to\_array(img)

x

Z = plt.imread('cat.jpg')

plt.imshow(Z)

<matplotlib.image.AxesImage at 0x7fa6ca436130>



x.shape

x=x/255

from keras.applications.vgg16 import preprocess\_input

import numpy as np

x=np.expand\_dims(x,axis=0)

img\_data=preprocess\_input(x)

img\_data.shape

model.predict(img\_data)

result = np.argmax(model.predict(img\_data), axis=1)

result[0]

if result[0] == 1:

    prediction = 'dog'

    print(prediction)

else:

    prediction = 'cat'

    print(prediction)

cat

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.

from keras.applications.vgg16 import VGG16

##from keras.preprocessing import image

from keras.applications.vgg16 import preprocess\_input, decode\_predictions

import numpy as np

from tensorflow.keras.preprocessing import image

Import the necessary libraries: You'll need to import Keras and other required libraries for this task.

model = VGG16(weights='imagenet', include\_top=True)

Downloading data from [https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16\_weights\_tf\_dim\_ordering\_tf\_kernels.h5](https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels.h5" \t "_blank)

553467096/553467096 [==============================] - 20s 0us/step

Load the pre-trained VGG16 model: You can load the pre-trained VGG16 model by calling the VGG16() function from the Keras library.

img\_path = '/cat.jpg'

img = image.load\_img(img\_path, target\_size=(224, 224))

Load an image for classification: You can use any image that you want to classify.

from google.colab import drive

drive.mount('/content/drive')

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force\_remount=True).

x = image.img\_to\_array(img)

x = np.expand\_dims(x, axis=0)

x = preprocess\_input(x)

You'll need to preprocess the image before feeding it to the VGG16 model. You can do this using the preprocess\_input() function from the Keras library.

preds = model.predict(x)

Predict the class of the image: You can predict the class of the image using the predict() function of the VGG16 model.

pred\_classes = decode\_predictions(preds, top=3)[0]

for i, pred\_class in enumerate(pred\_classes):

    print("{}. {}: {:.2f}%".format(i+1, pred\_class[1], pred\_class[2]\*100))

Decode the predictions: You can decode the predictions using the decode\_predictions() function from the Keras library.

Result:

Downloading data from [https://storage.googleapis.com/download.tensorflow.org/data/imagenet\_class\_index.json](https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_index.json" \t "_blank)

35363/35363 [==============================] - 0s 0us/step

1. tabby: 44.72%

2. tiger\_cat: 42.62%

3. Egyptian\_cat: 6.74%

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.

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)

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

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)

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

EXPERIMENT NO – 9

Implement word embeddings for IMDB dataset

Procedure :

Word embeddings are a type of representation learning that can be used to convert words into numerical vectors. In natural language processing (NLP), word embeddings are commonly used to represent words as dense vectors in a high-dimensional space. This allows us to perform various NLP tasks such as text classification, sentiment analysis, and language translation.

In this example, we will implement word embeddings for the IMDB dataset, which consists of movie reviews labeled as positive or negative. We will use the Keras library to implement the word embeddings.

First, we will load the IMDB dataset using Keras. The dataset consists of 50,000 movie reviews, with 25,000 reviews for training and 25,000 reviews for testing. Each review is a sequence of words, and the label is either 0 (negative) or 1 (positive).

from keras.datasets import imdb

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from keras.models import Sequential

from keras.layers import Dense, Flatten, Embedding

# Set the vocabulary size and the maximum length of a sequence

vocab\_size = 5000

maxlen = 100

# Load the IMDB dataset and split it into training and testing sets

(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=vocab\_size)

We set num\_words to 5000, which means we will only use the top 5000 most frequent words in the dataset.

Next, we will preprocess the data by padding the sequences to a fixed length and truncating sequences that are longer than the fixed length.

# Pad the sequences to ensure that they all have the same length

X\_train = pad\_sequences(X\_train, maxlen=maxlen)

X\_test = pad\_sequences(X\_test, maxlen=maxlen)

The pad\_sequences function pads the sequences with zeros to ensure they are all the same length. Here, we set maxlen to 100, which means that all sequences will be truncated or padded to 100 words.

# Define the embedding dimension

embedding\_dim = 50

# Define the model architecture

model = Sequential()

model.add(Embedding(input\_dim=vocab\_size, output\_dim=embedding\_dim, input\_length=maxlen))

model.add(Flatten())

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

We set the input\_dim to 5000, which is the size of the vocabulary, and output\_dim to 50, which is the dimensionality of the word embeddings. The GlobalMaxPooling1D layer takes the maximum value of each dimension in the embedding vectors, which allows us to obtain a fixed-length representation of each review. The final Dense layer uses a sigmoid activation function, which outputs a probability score between 0 and 1.

# Compile the model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

We compile the model with the Adam optimizer, binary\_crossentropy loss, and accuracy metric.

# Train the model

model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test), epochs=10, batch\_size=32)

We train the model on the training data for 10 epochs with a batch size of 32. Finally, we evaluate the model on the test data and print the accuracy score.

# Evaluate the model on the test set

test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test)

print('Test accuracy:', test\_accuracy)

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.

Downloading data from [https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz](https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz" \t "_blank)

17464789/17464789 [==============================] - 0s 0us/step

Epoch 1/10

782/782 [==============================] - 7s 8ms/step - loss: 0.4746 - accuracy: 0.7688 - val\_loss: 0.3402 - val\_accuracy: 0.8478

Epoch 2/10

782/782 [==============================] - 7s 9ms/step - loss: 0.2593 - accuracy: 0.8955 - val\_loss: 0.3311 - val\_accuracy: 0.8531

Epoch 3/10

782/782 [==============================] - 6s 8ms/step - loss: 0.1545 - accuracy: 0.9526 - val\_loss: 0.3563 - val\_accuracy: 0.8437

Epoch 4/10

782/782 [==============================] - 7s 9ms/step - loss: 0.0689 - accuracy: 0.9909 - val\_loss: 0.3924 - val\_accuracy: 0.8410

Epoch 5/10

782/782 [==============================] - 6s 8ms/step - loss: 0.0281 - accuracy: 0.9990 - val\_loss: 0.4303 - val\_accuracy: 0.8392

Epoch 6/10

782/782 [==============================] - 7s 9ms/step - loss: 0.0128 - accuracy: 0.9997 - val\_loss: 0.4633 - val\_accuracy: 0.8410

Epoch 7/10

782/782 [==============================] - 6s 8ms/step - loss: 0.0066 - accuracy: 0.9999 - val\_loss: 0.4977 - val\_accuracy: 0.8404

Epoch 8/10

782/782 [==============================] - 7s 9ms/step - loss: 0.0036 - accuracy: 1.0000 - val\_loss: 0.5372 - val\_accuracy: 0.8384

Epoch 9/10

782/782 [==============================] - 6s 8ms/step - loss: 0.0021 - accuracy: 1.0000 - val\_loss: 0.5627 - val\_accuracy: 0.8386

Epoch 10/10

782/782 [==============================] - 8s 11ms/step - loss: 0.0013 - accuracy: 1.0000 - val\_loss: 0.5935 - val\_accuracy: 0.8391

782/782 [==============================] - 2s 2ms/step - loss: 0.5935 - accuracy: 0.8391

Test accuracy: 0.83911997079849

EXPERIMENT NO – 10

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

Procedure :

To implement a Recurrent Neural Network (RNN) for the IMDB movie review classification problem, we will use the Keras deep learning library, which provides a simple and intuitive interface for building and training neural networks.

The IMDB movie review classification problem is a binary classification task, where the goal is to classify movie reviews as either positive or negative. The dataset contains 50,000 movie reviews, split into 25,000 for training and 25,000 for testing. Each review is a sequence of words, and the task is to predict whether the overall sentiment of the review is positive or negative.

To build our RNN, we will use an architecture called Long Short-Term Memory (LSTM), which is a type of RNN that is particularly good at processing sequential data. The basic idea behind LSTMs is to allow the network to selectively remember or forget information from previous time steps, which makes them well-suited for tasks like natural language processing.

from tensorflow.keras.datasets import imdb

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, LSTM, Dense

We start by importing the necessary modules from Keras

# Set the vocabulary size and maximum sequence length

vocab\_size = 10000

maxlen = 200

We set the maximum number of words to use from the IMDB dataset to 10,000.

# Load the IMDB dataset

(x\_train, y\_train), (x\_test, y\_test) = imdb.load\_data(num\_words=vocab\_size)

We load the IMDB dataset using the imdb.load\_data() function, which returns the training and testing data as a tuple of (x\_train, y\_train) and (x\_test, y\_test), where x\_train and x\_test are arrays of sequences, and y\_train and y\_test are the corresponding labels.

# Pad the sequences to have the same length

x\_train = pad\_sequences(x\_train, maxlen=maxlen)

x\_test = pad\_sequences(x\_test, maxlen=maxlen)

We pad the sequences to a fixed length of 200 using the pad\_sequences() function from Keras. This is necessary because the input sequences to an RNN must be of the same length.

# Define the model architecture

model = Sequential()

model.add(Embedding(input\_dim=vocab\_size, output\_dim=32))

model.add(LSTM(units=64))

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

We build the model using a Sequential() model from Keras. The model consists of an Embedding() layer, an LSTM() layer, and a Dense() layer. The Embedding() layer maps the input sequences of integers to vectors of fixed size, which allows the model to learn meaningful representations of the words in the sequences. The LSTM() layer processes the sequences, using the information from previous time steps to make predictions. The Dense() layer outputs a single value between 0 and 1, which represents the probability that the sentiment of the input sequence is positive.

# Compile the model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

We compile the model using the compile() function, specifying the loss function, optimizer, and metrics to use during training.

# Train the model

model.fit(x\_train, y\_train, validation\_data=(x\_test, y\_test), epochs=5, batch\_size=64)

We train the model using the fit() function

# Evaluate the model

test\_loss, test\_accuracy = model.evaluate(x\_test, y\_test)

print('Test accuracy:', test\_accuracy)

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.

Epoch 1/5

391/391 [==============================] - 90s 224ms/step - loss: 0.4158 - accuracy: 0.8017 - val\_loss: 0.2988 - val\_accuracy: 0.8741

Epoch 2/5

391/391 [==============================] - 85s 216ms/step - loss: 0.2428 - accuracy: 0.9062 - val\_loss: 0.3196 - val\_accuracy: 0.8702

Epoch 3/5

391/391 [==============================] - 85s 217ms/step - loss: 0.1863 - accuracy: 0.9306 - val\_loss: 0.3453 - val\_accuracy: 0.8642

Epoch 4/5

391/391 [==============================] - 86s 220ms/step - loss: 0.1401 - accuracy: 0.9495 - val\_loss: 0.3542 - val\_accuracy: 0.8557

Epoch 5/5

391/391 [==============================] - 86s 220ms/step - loss: 0.1171 - accuracy: 0.9586 - val\_loss: 0.3992 - val\_accuracy: 0.8589

782/782 [==============================] - 27s 34ms/step - loss: 0.3992 - accuracy: 0.8589

Test accuracy: 0.8588799834251404