**Classification using Deep neural network (Any One from the following):**

Binary classification using Deep Neural Networks Example: Classify movie reviews into positive" reviews and "negative" reviews, just based on the text content of the reviews. Use IMDB dataset

The first line, import tensorflow as tf, brings in the TensorFlow library, a powerful open-source library for numerical computation and large-scale machine learning. We are giving it a shorter alias tf for easier use throughout the code.

Next, from sklearn.model\_selection import train\_test\_split imports a specific function, train\_test\_split, from the scikit-learn (often shortened to sklearn) library. This function is used to split datasets into training and testing sets, which is crucial for evaluating the performance of machine learning models on unseen data.

Similarly, from sklearn.metrics import classification\_report, confusion\_matrix imports two more functions from scikit-learn's metrics module. classification\_report provides a detailed summary of the model's classification performance, including precision, recall, F1-score, and support for each class. confusion\_matrix creates a table that visualizes the performance of a classification model by showing the counts of true positives, true negatives, false positives, and false negatives.

The line import pandas as pd imports the pandas library, which is widely used for data manipulation and analysis. We're giving it the alias pd.

import numpy as np imports the NumPy library, the fundamental package for numerical computation in Python. It provides support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. We use the alias np.

import seaborn as sns imports the seaborn library, which is built on top of matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics. We use the alias sns.

import matplotlib.pyplot as plt imports the pyplot module from the matplotlib library. Matplotlib is a comprehensive library for creating static, interactive, and animated visualizations in Python. pyplot provides a collection of functions that make matplotlib work like MATLAB.

The line vocab\_size = 10000 defines a variable named vocab\_size and sets its value to 10000. This variable will likely be used to specify the size of the vocabulary for a text processing task, indicating that we will consider the 10,000 most frequent words in our dataset.

(x\_train, y\_train), (x\_test, y\_test) = tf.keras.datasets.imdb.load\_data(num\_words=vocab\_size) uses TensorFlow Keras' built-in dataset loading functionality. Specifically, it loads the IMDb movie reviews dataset, which is a binary sentiment classification dataset (positive or negative reviews). The load\_data() function, with the num\_words parameter set to vocab\_size, retrieves the dataset where each movie review is represented as a sequence of integers. These integers correspond to the indices of words in the vocabulary, and only the top vocab\_size most frequent words are considered. The function returns two tuples: (x\_train, y\_train) for the training data (sequences of word indices and their corresponding sentiment labels) and (x\_test, y\_test) for the testing data.

(x\_train.shape, y\_train.shape), (x\_test.shape, y\_test.shape) displays the shapes (dimensions) of the training and testing data. x\_train.shape and x\_test.shape will show the number of samples (movie reviews) and the length of each sequence (which might vary at this point). y\_train.shape and y\_test.shape will show the number of labels (sentiments) in the training and testing sets.

x\_train[0] displays the first movie review in the training set. This will be a list of integers, where each integer represents a word in the vocabulary.

int(np.mean(np.array([len(row) for row in x\_train])).round()) calculates the average length of the movie reviews in the training set. It first creates a NumPy array of the lengths of each review in x\_train, then calculates the mean of these lengths using np.mean(), and finally rounds the result to the nearest integer using .round() and converts it to an integer using int().

max\_length = int(np.mean(np.array([len(row) for row in x\_train])).round()) assigns the calculated average length of the training sequences to a variable named max\_length. This value will likely be used to pad or truncate sequences to a uniform length.

x\_train = tf.keras.preprocessing.sequence.pad\_sequences(x\_train, maxlen=max\_length) uses the pad\_sequences function from TensorFlow Keras' preprocessing module. This function transforms the list of sequences (movie reviews) in x\_train into a 2D NumPy array of shape (num\_samples, maxlen). Sequences that are shorter than maxlen are padded with zeros at the end (by default), and sequences longer than maxlen are truncated.

x\_test = tf.keras.preprocessing.sequence.pad\_sequences(x\_test, maxlen=max\_length) does the same padding or truncation for the test set x\_test, ensuring that all sequences in the test set also have the same length max\_length.

(x\_train.shape, y\_train.shape), (x\_test.shape, y\_test.shape) again displays the shapes of the training and testing data after padding. Now, x\_train.shape and x\_test.shape will show the number of samples and the fixed length max\_length of each sequence.

The following block defines a sequential neural network model using TensorFlow Keras: model = tf.keras.Sequential([ ... ]) initializes a sequential model, where layers are added in a linear stack. tf.keras.layers.Embedding(vocab\_size, 512, input\_length=max\_length) adds an embedding layer. This layer takes integer-encoded words as input and maps each word to a dense vector of fixed size (here, 512 dimensions). input\_length=max\_length specifies the expected length of the input sequences. The embedding layer helps to capture semantic relationships between words. tf.keras.layers.Flatten() flattens the output of the embedding layer into a 1D tensor. This is necessary to connect the embedding layer to the dense layers that follow. tf.keras.layers.Dense(16, activation=tf.keras.activations.relu) adds a densely connected (fully connected) layer with 16 neurons. The relu (Rectified Linear Unit) activation function is applied to the output of this layer. tf.keras.layers.Dense(1, activation='sigmoid') adds another dense layer with a single neuron. The sigmoid activation function is used here, which outputs a value between 0 and 1, making it suitable for binary classification tasks (like sentiment analysis, where the output can be interpreted as the probability of the review being positive).

model.summary() prints a summary of the model architecture, including the number of layers, the output shape of each layer, and the number of trainable parameters.

model.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=0.0001), loss='binary\_crossentropy', metrics=['accuracy']) configures the model for training. optimizer=tf.keras.optimizers.Adam(learning\_rate=0.0001) specifies the optimization algorithm to be used for updating the model's weights during training. Adam is a popular optimization algorithm, and learning\_rate=0.0001 sets the step size at each iteration. loss='binary\_crossentropy' defines the loss function to be minimized during training. Binary cross-entropy is a standard loss function for binary classification problems. metrics=['accuracy'] specifies that the accuracy should be calculated and reported during training and evaluation.

history = model.fit(x\_train, y\_train, batch\_size=128, epochs=5, validation\_data=(x\_test, y\_test)) trains the model. x\_train and y\_train are the training data and labels, respectively. batch\_size=128 specifies the number of samples per gradient update during training. epochs=5 indicates that the training process will iterate over the entire training dataset 5 times. validation\_data=(x\_test, y\_test) provides the test set to be used for evaluating the model's performance on unseen data after each epoch. The results of this validation are stored in the history object.

results = model.evaluate(x\_test, y\_test) evaluates the trained model on the test data (x\_test, y\_test) and returns the loss and any specified metrics (in this case, accuracy). The results are stored in the results variable.

results displays the evaluation results (loss and accuracy) on the test set.

pd.DataFrame(history.history)[['loss', 'val\_loss']].plot(figsize=(10,7)) creates a pandas DataFrame from the history.history dictionary (which contains the training loss, validation loss, training accuracy, and validation accuracy recorded during training). It then selects the 'loss' and 'val\_loss' columns and generates a line plot of these values over the epochs. figsize=(10,7) sets the size of the plot.

plt.title("Loss Curves") sets the title of the plot to "Loss Curves".

plt.xticks([0, 1, 2, 3, 4]) sets the x-axis ticks to represent the epochs (0 to 4, as we trained for 5 epochs).

plt.show() displays the generated loss curve plot.

pd.DataFrame(history.history)[['accuracy', 'val\_accuracy']].plot(figsize=(10,7)) similarly creates a pandas DataFrame from the training history, selects the 'accuracy' and 'val\_accuracy' columns, and generates a line plot of these values over the epochs. figsize=(10,7) sets the size of this plot as well.

plt.title("Accuracy Curves") sets the title of this plot to "Accuracy Curves".

plt.xticks([0, 1, 2, 3, 4]) sets the x-axis ticks for the accuracy plot to represent the epochs.

plt.show() displays the generated accuracy curve plot.

y\_pred = model.predict(x\_test) uses the trained model to make predictions on the test data (x\_test). The output y\_pred will be an array of probabilities (between 0 and 1) for each test sample, representing the model's confidence that the review is positive.

predicted\_labels = (y\_pred > 0.5).astype(int) converts the probabilities in y\_pred into binary labels (0 or 1). If the probability is greater than 0.5, the label is assigned as 1 (positive); otherwise, it's 0 (negative). .astype(int) converts the boolean results (True/False) to integers (1/0).

report = classification\_report(y\_test, predicted\_labels, target\_names=['Negative', 'Positive']) generates a classification report using the true labels (y\_test) and the model's predicted labels (predicted\_labels). target\_names=['Negative', 'Positive'] provides meaningful names for the classes in the report.

print(report) displays the classification report, which includes precision, recall, F1-score, and support for each class (Negative and Positive), as well as overall accuracy.

cm = confusion\_matrix(y\_test, predicted\_labels) calculates the confusion matrix using the true labels and the predicted labels.

sns.heatmap(cm, cmap='crest', annot=True, fmt=".0f") creates a heatmap visualization of the confusion matrix using the seaborn library. cm is the confusion matrix data, cmap='crest' sets the color map, annot=True displays the values in each cell of the heatmap, and fmt=".0f" formats the annotations as integers.

plt.title("Confusion Matrix") sets the title of the heatmap plot.

plt.show() displays the confusion matrix heatmap.