**DL\_4**

**import pandas as pd**

Pandas is a powerful Python library for data manipulation and analysis. It is widely used for data preprocessing tasks such as cleaning, transforming, and analyzing data.

**import numpy as np**

It provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays

**import tensorflow as tf**

TensorFlow is an open-source machine learning framework developed by Google

It provides a comprehensive set of tools and libraries for building and deploying machine learning models, especially deep learning models. Used for NPL and Images Processing

**import matplotlib.pyplot as plt**

Matplotlib is a comprehensive library for creating static, interactive, and animated visualizations in Python. including line plots, scatter plots, histograms, and more. It is commonly used for data visualization and graphical representation of data.

**import seaborn as sns**

Seaborn is a Python data visualization library based on matplotlib

**from sklearn.model\_selection import train\_test\_split**

This is a function from the Scikit-learn library. function is used to split data arrays into two subsets: one for training data and another for testing data

**from sklearn.preprocessing import StandardScaler**

used for standardizing features by removing the mean and scaling to unit variance. It is a common preprocessing step in many machine learning algorithms to standardize the range of independent variables or features.

**TEST\_PCT = 0.3**

This variable represents the percentage of the dataset that will be allocated for testing

30% of the data will be used for testing, while the remaining 70% will be used for training the model.

**LABELS = ["Normal","Fraud"]**

This variable represents the labels or categories in your dataset.

**print("Any nulls in the dataset",dataset.isnull().values.any())**

#check for any null values.

**print("No. of unique labels",len(dataset['Class'].unique()))**

unique() function to extract the unique values from the 'Class' column and then calculates the length of the resulting list of unique value

**print("Label values",dataset.Class.unique())**

Diff in two line

both line are display same result first line display length of unique values and second display what that values are

**count\_classes = pd.value\_counts(dataset['Class'],sort=True)**

**count\_classes.plot(kind='bar',rot=0)**

**plt.xticks(range(len(dataset['Class'].unique())),dataset.Class.unique())**

**plt.title("Frequency by observation number")**

**plt.xlabel("Class")**

**plt.ylabel("Number of Observations")**

code generates a bar plot that visualizes the frequency of each class in the 'Class' column of the dataset. Providing a clear overview of the class distribution in the imbalanced dataset.

**bins = np.linspace(200,2500,100)**

**plt.hist(normal\_dataset.Amount,bins=bins,alpha=1,density=True,label='Normal')**

**plt.hist(fraud\_dataset.Amount,bins=bins,alpha=0.5,density=True,label='Fraud')**

**plt.legend(loc='upper right')**

**plt.title("Transcation Amount vs Percentage of Transcations")**

**plt.xlabel("Transcation Amount (USD)")**

**plt.ylabel("Percentage of Transcations")**

**plt.show()**

alpha

This parameter controls the transparency of the histogram bars. It takes a value between 0 and 1

bins

This parameter specifies the number of bins or the specific bin edges for the histogram.

parameter is set using bins = np.linspace(200,2500,100) which creates 100 evenly spaced bins between the values 200 and 2500.

**sc = StandardScaler()** #instance of StandardScaler which scale the data to mean 0 and deviation 1

**dataset['Time'] = sc.fit\_transform(dataset['Time'].values.reshape(-1,1))**

**dataset['Amount'] = sc.fit\_transform(dataset['Amount'].values.reshape(-1,1))**

#both line transform data in standard form in 2D array format

**raw\_data = dataset.value()**

**labels = raw\_data[:,-1]** extracts the last column of array raw\_data

**data = raw\_data[:,0:-1]** extracts all the columns except the last one of raw\_data

**min\_val = tf.reduce\_min(train\_data)**

**max\_val = tf.reduce\_max(train\_data)**

**train\_data = (train\_data - min\_val) / (max\_val - min\_val)**

**test\_data = (test\_data - min\_val) / (max\_val - min\_val)**

**train\_data = tf.cast(train\_data,tf.float32)**

**test\_data = tf.cast(test\_data,tf.float32)** this line convert test\_data to float32 data type why we convert ….to incre consistency and compatibility with the TensorFlow framework.and in rest code we calculate min and max using tensorflow lib

**train\_labels = train\_labels.astype(bool)** #convert in bool datatype True and False

**test\_labels = test\_labels.astype(bool)**

**#Creating normal and fraud datasets**

**normal\_train\_data = train\_data[~train\_labels]**

#[~train\_lable] means ~True =false values only (Normal trx)

**normal\_test\_data = test\_data[~test\_labels]**

**fraud\_train\_data = train\_data[train\_labels]**  #here only True trx

**fraud\_test\_data = test\_data[test\_labels]**

**print("No. of records in Fraud Train Data=",len(fraud\_train\_data))**

**print("No. of records in Normal Train Data=",len(normal\_train\_data))**

**print("No. of records in Fraud Test Data=",len(fraud\_test\_data))**

**print("No. of records in Normal Test Data=",len(normal\_test\_data))**

**nb\_epoch = 50**

which is the number of times the learning algorithm will work through the entire training dataset just like loop

**batch\_size = 64** # used 64 data point in each iteration

**input\_dim = normal\_train\_data.shape[1]** #number of input neuron in NN

**#num of columns,30**

**encoding\_dim = 14** # number of neuron in middle or hidden layer of NN

**hidden\_dim1 = int(encoding\_dim / 2)** #First hidden layer size

**hidden\_dim2 = 4** #second hidden layer size

**learning\_rate = 1e-7**

which determines the step size at each iteration while moving toward a minimum of the loss function during training. It is set to a small value of 1e-7 to ensure stability during the training process.

**#input layer**

**input\_layer = tf.keras.layers.Input(shape=(input\_dim,))**

**#Encoder**

**encoder = tf.keras.layers.Dense(encoding\_dim,activation="tanh",activity\_regularizer = tf.keras.regularizers.l2(learning\_rate))(input\_layer)**

**encoder = tf.keras.layers.Dropout(0.2)(encoder)**

**encoder = tf.keras.layers.Dense(hidden\_dim1,activation='relu')(encoder)**

**encoder = tf.keras.layers.Dense(hidden\_dim2,activation=tf.nn.leaky\_relu)(encoder)**

**#Decoder**

**decoder = tf.keras.layers.Dense(hidden\_dim1,activation='relu')(encoder)**

**decoder = tf.keras.layers.Dropout(0.2)(decoder)**

**decoder = tf.keras.layers.Dense(encoding\_dim,activation='relu')(decoder)**

**decoder = tf.keras.layers.Dense(input\_dim,activation='tanh')(decoder)**

tanh- non-linear activation fun .range -1 to 1 it is symmetric around origin .mostly used for hidden layer

L2 Regularization – tech to prevent overfitting and improve generalization .also know as weight decay

Envolve penalty term to loss fun during training penalty is cal by adding sum of sq. of all weight by lambda (Regularization parameter or constant )

Dropout – randomly dropping the unit neuron during training to improve robustness ahe efficiency

ReLu – Rectified Linear unit .used in ANN f(x)=max(0,input) ability to alleviate vanishing gradient problem

Why we used Relu before leaky Relu   
ReLu is simple and incre efficiency and non-linear natural in input layer and avoid vanishing gradiant problem while Relu suffer from dead neuron problem to avoid this in hidden layer we used leaky Relu for next layers

why Relu before tanh in decoder

lly reason Relu in first decoder layer and tanh is used in final layer allow model to give output in specific range of -1 to 1.

**#Autoencoder**

**autoencoder = tf.keras.Model(inputs = input\_layer,outputs = decoder)**

**autoencoder.summary()**

**cp=tf.keras.callbacks.ModelCheckpoint(filepath="autoencoder\_fraud.h5",mode='min',monitor='val\_loss',verbose=2,save\_best\_only=True)**

Modelcheckpoint – used to save model weight during training

filepath where the model weights will be saved

**monitor='val\_loss'**: This specifies the quantity to be monitored during training. Here, it monitors the validation loss.

verbose=2 verbosity mode. A value of 2 means that the progress update is displayed in a more detailed way.

**#Define our early stopping**

**early\_stop = tf.keras.callbacks.EarlyStopping(**

**monitor='val\_loss',**

**min\_delta=0.0001,**

**patience=10,**

**verbose=11,**

**mode='min',**

**restore\_best\_weights=True**

**)**

Earlystopping is used to stop training process if performance matric stops improving   
when to stop parameter are given   
min\_delta=0.0001 min change in monitored

Patience =10 -number of epochs to stop if no improvement is noticed

**autoencoder.compile(metrics=['accuracy'],loss= 'mean\_squared\_error',optimizer='adam')**

#compile the autoencoder

Adam optimizer- Adam is different to classical stochastic gradient descent.Stochastic gradient descent maintains a single [learning rate](https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/) (termed alpha) for all weight updates and the learning rate does not change during training.in adam its changes durning training.algo is combination of AdaGrad and RMSProp.

**history = autoencoder.fit(normal\_train\_data,normal\_train\_data,epochs = nb\_epoch,**

**batch\_size = batch\_size,shuffle = True,**

**validation\_data = (test\_data,test\_data),**

**verbose=1,**

**callbacks = [cp,early\_stop]).history**

history- it store training and validation loss value and others parameter

**plt.plot(history['loss'],linewidth = 2,label = 'Train')**

**plt.plot(history['val\_loss'],linewidth = 2,label = 'Test')**

**plt.legend(loc='upper right')**

**plt.title('Model Loss')**

**plt.ylabel('Loss')**

**plt.xlabel('Epoch')**

**#plt.ylim(ymin=0.70,ymax=1**

**plt.show()**

**test\_x\_predictions = autoencoder.predict(test\_data)**

**mse = np.mean(np.power(test\_data - test\_x\_predictions, 2),axis = 1)**

**error\_df = pd.DataFrame({'Reconstruction\_error':mse,**

**'True\_class':test\_labels})**

Mse-Mean Sq error

The purpose of this code is to find error between original test data and reconstructed data

**threshold\_fixed = 50**

**groups = error\_df.groupby('True\_class')**

**fig,ax = plt.subplots()**

**for name,group in groups:**

**ax.plot(group.index,group.Reconstruction\_error,marker='o',ms = 3.5,linestyle='',**

**label = "Fraud" if name==1 else "Normal")**

**ax.hlines(threshold\_fixed,ax.get\_xlim()[0],ax.get\_xlim()[1],colors="r",zorder=100,label="Threshold")**

**ax.legend()**

**plt.title("Reconstructions error for normal and fraud data")**

**plt.ylabel("Reconstruction error")**

**plt.xlabel("Data point index")**

**plt.show()**

marker =’o’ – means circular marker is used and ms=3.5 means size of marker

**linestyle=''** ensures that the data points are only marked without lines connecting them.

**threshold\_fixed = 52**

**pred\_y = [1 if e > threshold\_fixed else 0**

**for e in**

**error\_df.Reconstruction\_error.values]**

**error\_df['pred'] = pred\_y**

**conf\_matrix = confusion\_matrix(error\_df.True\_class,pred\_y)**

**plt.figure(figsize = (4,4))**

**sns.heatmap(conf\_matrix,xticklabels = LABELS,yticklabels = LABELS,annot = True,fmt="d")**

**plt.title("Confusion matrix")**

**plt.ylabel("True class")**

**plt.xlabel("Predicted class")**

**plt.show()**

fmt=’d’ - This indicates that the annotations in the heatmap should be formatted as integers (decimal format)

**#Print Accuracy,Precision and Recall**

**print("Accuracy :",accuracy\_score(error\_df['True\_class'],error\_df['pred']))**

**print("Recall :",recall\_score(error\_df['True\_class'],error\_df['pred']))**

**print("Precision :",precision\_score(error\_df['True\_class'],error\_df['pred']))**