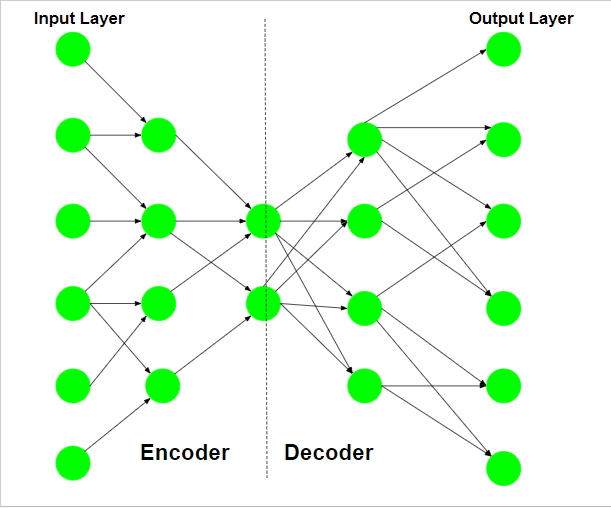
|  |  |
| --- | --- |
| **Ex no : 2** | **Outlier Detection using Autoencoder** |
| **Date :** |

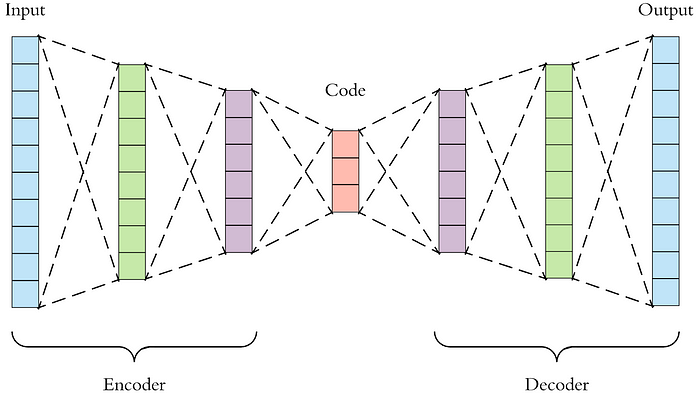
**Aim**

To study about Autoencoder and implement it for outlier detection for random data.

**Basic Theory of Autoencoder.**

* Autoencoders are a powerful tool for data compression and analysis. They can be used to discover hidden patterns within your data and then use those patterns to create a compressed representation of the original data.
* Autoencoder is an unsupervised neural network model that uses reconstruction error to detect anomalies or outliers. The reconstruction error is the difference between the reconstructed data and the input data.
* Autoencoder uses only normal data to train the model and all data to make predictions. Therefore, we expect outliers to have higher reconstruction errors because they are different from the regular data.
* Some of the architectures of encoder-decoder models are given below.





**PROCEDURE AND CODE**

1. **Import the following libraries**

**Code**

# Synthetic dataset

from sklearn.datasets import make\_classification

# Data processing

import pandas as pd

import numpy as np

from collections import Counter

# Visualization

import matplotlib.pyplot as plt

import seaborn as sns

# Model and performance

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

import tensorflow as tf

from tensorflow.keras import layers, losses

from sklearn.metrics import classification\_report

1. **Create Dataset with Outliers.**

Using make\_classification from the sklearn library and create two classes with the ratio between the majority class and the minority class being 0.995:0.005. 32 informative features were made as predictors. Donot include any redundant or repeated features in this dataset.

**Code**

X, y = make\_classification(n\_samples=100000, n\_features=32, n\_informative=32,n\_redundant=0, n\_repeated=0, n\_classes=2,

                           n\_clusters\_per\_class=1,

                           weights=[0.995, 0.005],

                           class\_sep=0.5, random\_state=0)

1. **Split the dataset into 80% training data and 20% validation data. random\_state ensures that we have the same train test split every time and check the data.**

**Code**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Check the number of records

print('The number of records in the training dataset is', X\_train.shape[0])

print('The number of records in the test dataset is', X\_test.shape[0])

print(f"The training dataset has {sorted(Counter(y\_train).items())[0][1]} records for the majority class and {sorted(Counter(y\_train).items())[1][1]} records for the minority class."

* The autoencoder model trains on the normal dataset, so first separate the expected data from the anomaly data.
* Then create the input layer, encoder layers, and decoder layers.
* In the input layer, Specify the shape of the dataset. Because the modeling dataset has 32 features, the shape is 32 here.
* The encoder consists of 3 layers with 16, 8, and 4 neurons, respectively. Note that the encoder requires the number of neurons to decrease with the layers. The last layer in the encoder is the size of the encoded representation, and it is also called the bottleneck.
* The decoder consists of 3 layers with 8, 16, and 32 neurons, respectively. Opposite from the encoder, the decoder requires the number of neurons to increase with the layers. The output layer in the decoder has the same size as the input layer.
* The relu activation function is used for each layer except for the decoder output layer. relu is a popular activation function, but you can try other activation functions and compare the model performance.
* After defining the input, encoder, and decoder layers, create the autoencoder model to combine the layers.

1. **Create Autoencoder model for training**

**Code**

# Keep only the normal data for the training dataset

X\_train\_normal = X\_train[np.where(y\_train == 0)]

# Input layer

input = tf.keras.layers.Input(shape=(32,))

# Encoder layers

encoder = tf.keras.Sequential([

  layers.Dense(16, activation='relu'),

  layers.Dense(8, activation='relu'),

  layers.Dense(4, activation='relu')])(input)

# Decoder layers

decoder = tf.keras.Sequential([

      layers.Dense(8, activation="relu"),

      layers.Dense(16, activation="relu"),

      layers.Dense(32, activation="sigmoid")])(encoder)

# Create the autoencoder

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

* After creating the autoencoder model, Compile the model with the optimizer of adam and the loss of mae (Mean Absolute Error).
* When fitting the autoencoder model, we can see that the input and output datasets are the same, which is the dataset that contains only the normal data points.
* The validation data is the testing dataset that contains both normal and anomaly data points.
* The epochs of 20 and batch\_size of 64 mean the model uses 64 datapoints to update the weights in each iteration, and the model will go through the whole training dataset 20 times. shuffle=True will shuffle the dataset before each epoch.

1. **Compile the model**

**Code**

# Compile the autoencoder

autoencoder.compile(optimizer='adam', loss='mae')

# Fit the autoencoder

history = autoencoder.fit(X\_train\_normal, X\_train\_normal,

          epochs=20,

          batch\_size=64,

          validation\_data=(X\_test, X\_test),

          shuffle=True)

1. **Visualize the Training and validation Loss**

**Code**

plt.plot(history.history["loss"], label="Training Loss")

plt.plot(history.history["val\_loss"], label="Validation Loss")

plt.legend()

1. **Setting Threshold to identify the anomalies**

* let's use autoencoder model to predict the outliers.
* Firstly, use .predict to get the reconstruction value for the testing data set containing the usual data points and the outliers.
* Then calculate the loss value between actual and reconstruction using mean absolute error.
* After that, a threshold is set to identify the outliers. This threshold can be based on percentile, standard deviation, or other methods. Let’s use 98% loss as the threshold to identify 2% of the data as outliers.

**Code**

# Predict anomalies/outliers in the training dataset

prediction = autoencoder.predict(X\_test)

# Get the mean absolute error between actual and reconstruction/prediction

prediction\_loss = tf.keras.losses.mae(prediction, X\_test)

# Check the prediction loss threshold for 2% of outliers

loss\_threshold = np.percentile(prediction\_loss, 98)

print(f'The prediction loss threshold for 2% of outliers is {loss\_threshold:.2f}')

# Visualize the threshold

sns.histplot(prediction\_loss, bins=30, alpha=0.8)

plt.axvline(x=loss\_threshold, color='orange')

* The visualization chart shows that the prediction loss is close to a normal distribution with a mean of around 2.5. The prediction loss threshold for 2% of outliers is about 3.5.

1. **Performance evaluation of the encoder model**

* Sometimes the dataset has the ground truth label for the anomalies, and the dataset often does not. When there is a label for anomalies then we can evaluate the model performance.
* Based on the threshold identified in the previous step, Let us predict normal data points if the prediction loss is less than the threshold. Otherwise, predict the data point to be an outlier or anomaly and label the normal prediction 0 and outlier prediction 1 to be consistent with the ground truth label.

**Code**

# Check the model performance at 2% threshold

threshold\_prediction = [0 if i < loss\_threshold else 1 for i in prediction\_loss]

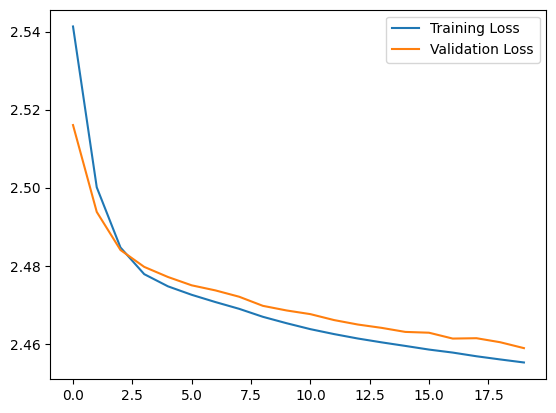
# # Check the prediction performance

print(classification\_report(y\_test, threshold\_prediction))

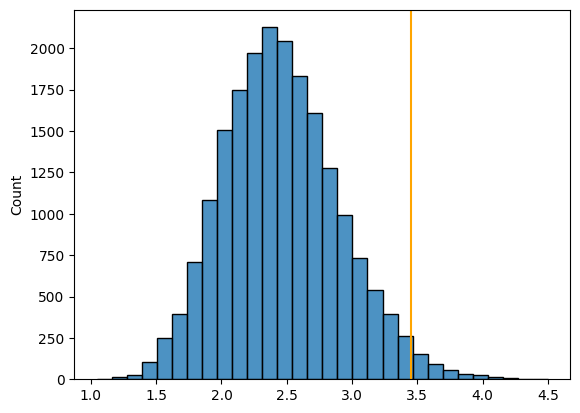
* The recall value of 0.01 shows that around 1% of the outliers were captured by the autoencoder.

**OUTPUT**

Loss during training and validation



Prediction Loss and Threshold Range



**RESULT**

Thus autoencoder was implemented for outlier detection for random dataset successfully.