ASS4

```
# ============================================================
# a. IMPORT REQUIRED LIBRARIES
# ============================================================

import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, recall_score, accuracy_score, precision_score

# Constants
RANDOM_SEED = 2021
TEST_PCT = 0.3
LABELS = ["Normal", "Fraud"]

# ============================================================
# b. UPLOAD / ACCESS THE DATASET
# ============================================================

# Load dataset (ensure creditcard.csv is in your working directory)
dataset = pd.read_csv("creditcard.csv")

print("Any nulls in the dataset:", dataset.isnull().values.any())
print('-------')
print("No. of unique labels:", len(dataset['Class'].unique()))
print("Label values:", dataset.Class.unique())
print('-------')
print("Breakdown of Normal and Fraud Transactions:")
print(pd.value_counts(dataset['Class'], sort=True))

# Visualization of class distribution
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")
plt.show()

# Separate normal and fraud transactions
normal_dataset = dataset[dataset.Class == 0]
fraud_dataset = dataset[dataset.Class == 1]

# Visualize transaction amounts
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("Transaction Amount vs Percentage of Transactions")
plt.xlabel("Transaction Amount (USD)")
plt.ylabel("Percentage of Transactions")
plt.show()

# Scale Time and Amount columns
sc = StandardScaler()
dataset['Time'] = sc.fit_transform(dataset['Time'].values.reshape(-1, 1))
dataset['Amount'] = sc.fit_transform(dataset['Amount'].values.reshape(-1, 1))

# Separate features and labels
raw_data = dataset.values
labels = raw_data[:, -1]
data = raw_data[:, 0:-1]

# Split dataset
train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size=0.2, random_state=RANDOM_SEED
)

# Normalize 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)

# Convert labels to boolean
train_labels = train_labels.astype(bool)
test_labels = test_labels.astype(bool)

# Create Normal and Fraud subsets
normal_train_data = train_data[~train_labels]
normal_test_data = test_data[~test_labels]
fraud_train_data = train_data[train_labels]
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))

# ============================================================
# c. ENCODER CONVERTS INPUT INTO LATENT REPRESENTATION
# ============================================================

nb_epoch = 50
batch_size = 64
input_dim = normal_train_data.shape[1]  # Number of features (30)
encoding_dim = 14
hidden_dim1 = int(encoding_dim / 2)
hidden_dim2 = 4
learning_rate = 1e-7

# Input Layer
input_layer = tf.keras.layers.Input(shape=(input_dim,))

# Encoder layers
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)

# ============================================================
# d. DECODER NETWORK CONVERTS IT BACK TO ORIGINAL INPUT
# ============================================================

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)

# Combine Encoder + Decoder into Autoencoder
autoencoder = tf.keras.Model(inputs=input_layer, outputs=decoder)
autoencoder.summary()

# ============================================================
# e. COMPILE MODEL WITH OPTIMIZER, LOSS, AND EVALUATION METRICS
# ============================================================

# Define callbacks
cp = tf.keras.callbacks.ModelCheckpoint(
    filepath="autoencoder_fraud.keras", mode='min',
    monitor='val_loss', verbose=2, save_best_only=True
)

early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', min_delta=0.0001, patience=10,
    verbose=11, mode='min', restore_best_weights=True
)
```

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

# Train the model (on normal transactions only)
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

# Plot training vs validation loss
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('Epochs')
plt.show()

# ============================================================
# 🔍 ANOMALY DETECTION AND EVALUATION
# ============================================================

# Predict and compute reconstruction errors
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})

# Visualize reconstruction error
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("Reconstruction Error for Normal and Fraud Data")
plt.ylabel("Reconstruction Error")
plt.xlabel("Data Point Index")
plt.show()

# Apply threshold to detect anomalies
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

# Confusion matrix
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()

# Evaluation metrics
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']))
```