Practical-4 Auto Encoder

```python
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
```

Explanation:
- We’re importing libraries that are essential for our anomaly detection model:
  - **pandas**: Helps with data handling and organization in tables.
  - **numpy**: Supports mathematical operations on arrays.
  - **tensorflow**: Builds and trains neural networks (like our autoencoder).
  - **matplotlib** and **seaborn**: Visualize data with plots and graphs.
  - **sklearn.model_selection**: Splits the dataset into training and testing parts.
  - **sklearn.preprocessing**: Scales (normalizes) the dataset for more consistent results.
  - **sklearn.metrics**: Measures the model's performance.

```python
RANDOM_SEED = 2021
TEST_PCT = 0.3
LABELS = ["Normal","Fraud"]
```

Explanation:
- We set up some basic configurations:
  - **RANDOM_SEED**: Ensures consistent results each time by controlling the randomness.
  - **TEST_PCT**: Tells us what percentage of data will be used for testing (30% here).
  - **LABELS**: Labels the two types of transactions as “Normal” and “Fraud.”

```python
dataset = pd.read_csv("creditcard.csv")
```

Explanation:
- This line loads the dataset named **creditcard.csv** into a **pandas DataFrame** called `dataset`, making it easy to analyze and manipulate.

```python
# Check for any null values
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())

# 0 is for normal credit card transactions
# 1 is for fraudulent credit card transactions
print('-------')
print("Breakdown of Normal and Fraud Transactions")
print(pd.value_counts(dataset['Class'], sort=True))
```

Explanation:
1. **Null Check**: Checks if there are any missing values in the dataset. This is important because missing data could affect model accuracy.
2. **Unique Labels**: Finds and displays unique values in the "Class" column. Here, `Class` is the label column, with `0` for normal and `1` for fraud.
3. **Label Breakdown**: Counts how many normal and fraudulent transactions we have, helping us understand the class distribution.

```python
# Visualizing the imbalanced dataset
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")
```

Explanation:
- This part visualizes how many normal vs. fraud transactions are present in the

dataset using a bar chart.
  - **pd.value_counts()**: Counts each label’s occurrences.
  - **plt.title()** and other settings: Label the axes and title to make the chart clear.

```python
# Save the normal and fraudulent transactions in separate DataFrames
normal_dataset = dataset[dataset.Class == 0]
fraud_dataset = dataset[dataset.Class == 1]

# Visualize transaction amounts for normal and fraudulent transactions
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()
```

Explanation:
- This code creates separate datasets for normal and fraud transactions and then plots transaction amounts for each.
  - **normal_dataset** and **fraud_dataset**: Split `dataset` based on the class (0 for normal, 1 for fraud).
  - **plt.hist()**: Plots histograms for transaction amounts to show the difference in value ranges between normal and fraud transactions.

```python
sc = StandardScaler()
dataset['Time'] = sc.fit_transform(dataset['Time'].values.reshape(-1, 1))
dataset['Amount'] = sc.fit_transform(dataset['Amount'].values.reshape(-1, 1))
```

Explanation:
- Here, we normalize `Time` and `Amount` columns for better model performance.
  - **StandardScaler**: Scales each feature so that it has a mean of 0 and a standard deviation of 1.
  - **fit_transform()**: Fits the scaler and applies it to `Time` and `Amount` to

standardize their values.

```python
raw_data = dataset.values
labels = raw_data[:, -1]  # The last element contains if the transaction is normal (0) or fraud (1)
data = raw_data[:, 0:-1]  # Other columns are the main data points
train_data, test_data, train_labels, test_labels = train_test_split(data, labels, test_size=0.2, random_state=2021)
```

Explanation:
- **raw_data**: Converts the dataset into a **NumPy array** for easier manipulation.
  - **labels**: Takes the last column (`Class`) as our target labels (0 or 1).
  - **data**: Takes the other columns as our features.
  - **train_test_split()**: Splits the data into training and test sets, with 80% for training and 20% for testing.

```python
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)
```

Explanation:
- **Min-Max Scaling**: This scales our data to fit within a 0-1 range, which stabilizes training.
  - **tf.reduce_min()** and **tf.reduce_max()**: Find the minimum and maximum values in the training data.
  - **tf.cast()**: Converts data to **float32** format, which is standard for TensorFlow.

```python
train_labels = train_labels.astype(bool)
test_labels = test_labels.astype(bool)
```

# Creating normal and fraud datasets
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))
```

### Explanation:
- **Convert Labels to Boolean**: Converts labels to `True` for fraud (1) and `False` for normal (0), which is useful for easy data separation.
- **Separate Data**: Creates `normal_train_data`, `fraud_train_data`, `normal_test_data`, and `fraud_test_data` based on labels. This way, we can focus on training with normal data first, as it is usually the majority class.

---

Absolutely! Let's continue with the remaining code cells in the same format.

---

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

Explanation:
- Here, we define key parameters for our model:
  - **nb_epoch**: Sets the number of training cycles (epochs) to 50, meaning the model will go over the training data 50 times.
  - **batch_size**: Processes the data in batches of 64 samples at a time for faster and more stable training.
  - **input_dim**: Sets the input layer's dimension to match the number of columns (features) in the data.
  - **encoding_dim**: Defines the size of the compressed (latent) representation after the encoder part.
  - **hidden_dim1** and **hidden_dim2**: Define the dimensions of hidden
```

layers in the encoder and decoder. These layers gradually reduce the size of data, making it easier to detect anomalies.
  - **learning_rate**: Controls the step size during optimization to prevent drastic changes in weights during training.

```python
# 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)

# Autoencoder
autoencoder = tf.keras.Model(inputs=input_layer, outputs=decoder)
autoencoder.summary()
```

Explanation:
- **Input Layer**: Defines the input size using `input_dim`, which matches the number of features in our dataset.
- **Encoder**: Compresses data into a smaller "latent representation."
  - **Dense layer with tanh**: Reduces data size, with tanh activation to handle both positive and negative values.
  - **Dropout**: Randomly ignores some neurons to prevent overfitting.
  - **Relu and Leaky Relu**: Further compress data. Leaky relu avoids inactive neurons by allowing small positive gradients when inputs are negative.
- **Decoder**: Reconstructs data from latent representation to original size.
  - **ReLU** layers gradually increase the data size.
  - **Final Dense layer with tanh**: Matches input size to get the output back to the original feature space.
- **Autoencoder Model**: Combines input, encoder, and decoder into a single model and prints the summary.

```python
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
)
```

Explanation:
- **Checkpoint (cp)**: Saves the model with the lowest validation loss during training to "autoencoder_fraud.keras." This way, if training stops or ends, we still have the best version saved.
- **Early Stopping**: Stops training if there’s no improvement in validation loss for 10 consecutive epochs. This avoids overfitting and reduces training time.

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

Explanation:
- **Compile**: Prepares the autoencoder for training by defining:
  - **Metrics**: Here, accuracy will track how well the model performs.
  - **Loss**: Mean squared error compares the input data with the reconstructed output, measuring the "reconstruction error."
  - **Optimizer**: Adam optimizer adjusts model weights to minimize loss in each training step.

```python
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
```

Explanation:
- **Training (fit)**: Trains the autoencoder model on the normal (non-fraud) data, aiming to learn how normal transactions should look.
  - **normal_train_data**: We train only on normal data to recognize anomalies later.
  - **epochs, batch_size**: Defined earlier, control training cycles and batch size.
  - **validation_data**: Checks performance on test data.
  - **callbacks**: Saves best model and applies early stopping if needed.
- **history**: Stores training history for plotting later.

```python
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.show()
```

Explanation:
- **Plot Loss**: Shows the model's training and validation loss over epochs, letting us check if the model is learning.
  - **history['loss']**: Training loss per epoch.
  - **history['val_loss']**: Validation loss per epoch.
  - **Labels and Title**: Labels the plot to make it easier to read.

```python
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})
```

Explanation:
- **test_x_predictions**: Generates the model's predictions for the test set.
- **Mean Squared Error (mse)**: Calculates the reconstruction error for each test data point, showing how closely the reconstructed output matches the input.
- **error_df**: Creates a DataFrame that holds each test point's reconstruction error and true label (0 or 1).

```python
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()
```

Explanation:
- **threshold_fixed**: Sets a cutoff for reconstruction error; points above it are flagged as possible frauds.
- **Plot Reconstruction Error**:
  - Groups data by `True_class` (0 for normal, 1 for fraud).
  - Plots each point's reconstruction error. Fraud cases are expected to have higher errors.
  - **ax.hlines()**: Adds a red line at the threshold value, distinguishing normal and fraud data.

```python
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()
```

```

### Explanation:
- **Threshold Adjustment**: Sets a slightly higher threshold of 52 to classify points as fraud (1) or normal (0).
- **Prediction Array**: `pred_y` contains predictions based on reconstruction error.
- **Confusion Matrix**: Compares true labels to predictions, counting correct and incorrect classifications.
  - **sns.heatmap()**: Visualizes the confusion matrix, showing counts of normal/fraud predictions versus actual classes.

---

```python
# 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']))
```

Explanation:
- **Model Performance**:
  - **Accuracy**: Percentage of correct predictions overall.
  - **Recall**: Percentage of actual frauds correctly identified.
  - **Precision**: Percentage of predicted frauds that were actually frauds.
- These metrics give a full view of how well the model handles fraud detection.

Here are important questions for Assignment 4, which is about ECG anomaly detection using Autoencoders:

Important Questions and Answers

1. **What is Anomaly Detection?**
   - Anomaly detection identifies unusual patterns or outliers in data that do not conform to expected behavior.

2. **What are Autoencoders in Deep Learning?**
   - Autoencoders are neural networks designed to learn efficient data representations by encoding input into a latent space and reconstructing it back to the original form.

3. **List some applications of Autoencoders in deep learning.**
   - Image denoising, anomaly detection, and dimensionality reduction.
```

4. **Explain the difference between Anomaly Detection and Novelty Detection.**
   - Anomaly detection identifies deviations in known data distributions, while novelty detection recognizes patterns that are entirely new or different from training data.

5. **Describe the architecture of an Autoencoder.**
   - An autoencoder has an encoder that compresses input data into a latent representation, and a decoder that reconstructs the original data from this representation.

6. **What is reconstruction error in Autoencoders?**
   - It is the difference between the original input and the reconstructed output, used to detect anomalies based on high reconstruction errors.

7. **What is MinMaxScaler from sklearn, and why is it used?**
   - MinMaxScaler scales data to a fixed range, typically [0,1], which normalizes input for better model training performance.

8. **What is the purpose of the `train_test_split` function in sklearn?**
   - This function splits data into training and testing sets, ensuring that the model can be evaluated on unseen data.

9. **What is an anomaly score?**
   - Anomaly score quantifies how unusual a data point is, based on metrics like reconstruction error in an autoencoder.

10. **Describe the ECG dataset used in this practical.**
    - The ECG dataset contains electrocardiogram readings, often with labeled anomalies, to help train models to detect abnormal heart patterns.

11. **What are some optimizers used in Keras, and what is their role?**
    - Common optimizers include Adam, SGD, and RMSprop, which adjust model weights to minimize the loss function during training.

12. **Explain Dense and Dropout layers in Keras.**
    - **Dense Layer**: A fully connected layer that contributes to learning complex patterns.
    - **Dropout Layer**: Randomly drops a fraction of neurons during training to prevent overfitting.

13. **What is the Mean Squared Logarithmic Error (MSLE) in Keras losses?**
    - MSLE is a loss function that calculates the logarithmic difference between true and predicted values, often used for regression tasks with data on different scales.

14. **What is the purpose of the ReLU activation function?**
   - ReLU (Rectified Linear Unit) introduces non-linearity by setting negative values to zero, which helps the model learn complex patterns efficiently.

15. **How is thresholding used to detect anomalies in Autoencoders?**
   - Thresholding involves setting a cut-off value for reconstruction error; data points exceeding this threshold are labeled as anomalies.

Would you like additional questions or further details on any of these topics?