Implementing an autoencoder-based anomaly detection model using TensorFlow and Python.

The code step by step:

Importing Libraries:

The code begins by importing various libraries, including Pandas for data manipulation, NumPy for numerical operations,

TensorFlow for deep learning, Matplotlib and Seaborn for data visualization, and scikit-learn for metrics related to machine learning.

Loading the Dataset:

It loads a dataset from a CSV file named "creditcard.csv" using Pandas.

Data Exploration:

The code then performs some initial data exploration by checking for null values and displaying information about the class labels and their distribution.

The dataset appears to be related to credit card transactions, with two classes: "Normal" (class 0) and "Fraud" (class 1).

Data Visualization:

It visualizes the imbalance in the dataset by plotting the distribution of the class labels.

Data Preprocessing:

The 'Time' and 'Amount' columns are standardized using scikit-learn's StandardScaler.

The data is split into training and testing sets.

Minimum and maximum values are computed for scaling the data to a range between 0 and 1.

The data types of train_labels and test_labels are converted to boolean.

Autoencoder Architecture:

The autoencoder neural network architecture is defined. An autoencoder consists of an encoder and a decoder.

The encoder compresses the input data into a lower-dimensional representation.

The decoder reconstructs the input data from the compressed representation.

Various hyperparameters for the autoencoder are specified, including the number of hidden layers, activation functions, dropout rates, and learning rate.

Model Training:

The autoencoder model is compiled with a mean squared error loss function and the Adam optimizer.

It is trained on the normal_train_data, which contains only the samples with normal class labels.

The training process is monitored, and early stopping is implemented to prevent overfitting.

Model Evaluation:

The reconstruction error is calculated for each data point in the test set.

A threshold value is defined to classify data points as normal or anomaly based on the reconstruction error.

A confusion matrix is computed to evaluate the model's performance.

Metrics such as accuracy, recall, and precision are displayed.

The primary goal of this experiment is to demonstrate the use of an autoencoder for anomaly detection in credit card transactions.

Anomalies are detected based on the reconstruction error of the autoencoder, with the assumption that fraudulent transactions will have higher reconstruction errors compared to normal transactions.

However, the code currently shows a low recall and precision, indicating room for improvement in anomaly detection performance.

Fine-tuning hyperparameters and exploring other anomaly detection techniques may be necessary for better results.

1. **Importing Libraries:**
    - `import pandas as pd`: Imports the Pandas library and assigns it the alias 'pd'. Pandas is a popular library for data manipulation and analysis.
    - `import numpy as np`: Imports the NumPy library and assigns it the alias 'np'. NumPy is used for numerical operations and working with arrays.
    - `import tensorflow as tf`: Imports the TensorFlow library and assigns it the alias 'tf'. TensorFlow is a deep learning framework commonly used for building neural networks and machine learning models.
    - `import matplotlib.pyplot as plt`: Imports the Matplotlib library and assigns it the alias 'plt'. Matplotlib is used for creating data visualizations.
    - `import seaborn as sns`: Imports the Seaborn library and assigns it the alias 'sns'. Seaborn is a data visualization library that works well with Pandas data structures.
    - `from sklearn.model_selection import train_test_split`: Imports the `train_test_split` function from scikit-learn, which is used for splitting datasets into training and testing sets.
    - `from sklearn.preprocessing import StandardScaler`: Imports the `StandardScaler` class from scikit-learn, which is used for standardizing data (mean centering and scaling).
2. **Constants and Labels:**
    - `RANDOM_SEED = 2021`: Sets a constant variable `RANDOM_SEED` to the value 2021. This seed value is typically used to ensure reproducibility in random processes, such as splitting data or initializing random number generators.
    - `TEST_PCT = 0.3`: Sets another constant variable `TEST_PCT` to the value 0.3. This variable likely represents the percentage of data to be allocated for the testing dataset when splitting the data.
    - `LABELS = ["Normal","Fraud"]`: Defines a list called `LABELS` containing two strings, "Normal" and "Fraud". These labels are used to describe the two classes in the dataset, where "Normal" typically represents non-fraudulent data and "Fraud" represents fraudulent data.

1. `count_classes = pd.value_counts(dataset['Class'], sort=True)`
    - `pd.value_counts(dataset['Class'], sort=True)`: This line uses the Pandas library to count the occurrences of each unique value in the 'Class' column of the `dataset` DataFrame. It counts how many times each class appears in the dataset.
    - `count_classes` is assigned the result, which is a Pandas Series containing the counts of each class.
2. `count_classes.plot(kind='bar', rot=0)`

- `count_classes.plot(kind='bar', rot=0)`: This line generates a bar plot of the counts using Matplotlib. It creates a bar chart where the x-axis represents the class labels ("Normal" and "Fraud"), and the y-axis represents the number of observations for each class.
- `kind='bar'` specifies that a bar plot should be created.
- `rot=0` specifies that the x-axis labels (class names) should not be rotated.

3. `plt.xticks(range(len(dataset['Class'].unique())), dataset.Class.unique())`
    - `plt.xticks(...)`: This line customizes the x-axis tick labels of the bar chart.
    - `range(len(dataset['Class'].unique()))` generates a range of numbers equal to the number of unique classes in the 'Class' column of the dataset. This is used as the x-coordinates for the tick labels.
    - `dataset.Class.unique()` retrieves the unique class labels ("Normal" and "Fraud") from the 'Class' column and assigns them as the tick labels.
    - Together, this line ensures that the x-axis labels display "Normal" and "Fraud" instead of numeric values.
4. `plt.title("Frequency by observation number")`
    - `plt.title("Frequency by observation number")`: This line sets the title of the bar chart to "Frequency by observation number." It provides a brief description of what the chart represents.
5. `plt.xlabel("Class")` and `plt.ylabel("Number of Observations")`
    - `plt.xlabel("Class")`: This line sets the x-axis label to "Class," indicating that it represents the class labels.
    - `plt.ylabel("Number of Observations")`: This line sets the y-axis label to "Number of Observations," indicating that it represents the counts of observations.

1. `normal_dataset = dataset[dataset.Class == 0]`
    - `dataset.Class == 0`: This part of the code creates a boolean mask by checking if the 'Class' column in the `dataset` DataFrame is equal to 0. This condition evaluates to `True` for rows where the class is "Normal" (class 0) and `False` for rows where the class is "Fraudulent" (class 1).
    - `dataset[...]`: This part applies the boolean mask to the entire DataFrame `dataset`. It selects rows from the original dataset where the condition `dataset.Class == 0` is `True`.
    - `normal_dataset`: This variable is assigned the resulting subset of rows from the original dataset where the class is "Normal." Therefore, `normal_dataset` contains all the rows representing normal credit card transactions.
2. `fraud_dataset = dataset[dataset.Class == 1]`

- Similarly, this line of code creates another boolean mask by checking if the 'Class' column in the `dataset` DataFrame is equal to 1. This condition evaluates to `True` for rows where the class is "Fraudulent" (class 1) and `False` for rows where the class is "Normal" (class 0).
- `dataset[...]`: The boolean mask is applied to the entire DataFrame `dataset` to select rows where the condition `dataset.Class == 1` is `True`.
- `fraud_dataset`: This variable is assigned the resulting subset of rows from the original dataset where the class is "Fraudulent." Therefore, `fraud_dataset` contains all the rows representing fraudulent credit card transactions.

1. `bins = np.linspace(200, 2500, 100)`:
    - `np.linspace(200, 2500, 100)`: This line uses NumPy (imported as `np`) to create an array of 100 evenly spaced values between 200 and 2500. These values represent the bin edges for the histograms. In this case, it means that the range of transaction amounts is divided into 100 equally sized bins.
2. `plt.hist(normal_dataset.Amount, bins=bins, alpha=1, density=True, label='Normal')`:
    - `plt.hist(...)`: This line uses Matplotlib to create a histogram.
    - `normal_dataset.Amount`: It specifies the data to be used for the histogram, which is the 'Amount' column of the `normal_dataset` DataFrame, containing transaction amounts for normal transactions.
    - `bins=bins`: It specifies the bin edges created earlier using `np.linspace`.
    - `alpha=1`: This parameter controls the transparency of the histogram bars. An alpha value of 1 makes the bars fully opaque.
    - `density=True`: When `density` is set to `True`, the histogram represents the density (probability distribution) of the data, ensuring that the area under the histogram sums to 1. This is useful when comparing distributions of different scales.
    - `label='Normal'`: This label is assigned to the histogram, which will be used in the legend later.
3. `plt.hist(fraud_dataset.Amount, bins=bins, alpha=0.5, density=True, label='Fraud')`:
    - This line is similar to the previous one but creates a histogram for the 'Amount' column of the `fraud_dataset` DataFrame, which contains transaction amounts for fraudulent transactions.
    - `alpha=0.5`: This parameter sets the transparency of the histogram bars to 0.5, making them partially transparent.
4. `plt.legend(loc='upper right')`:

- This line adds a legend to the plot. The legend contains labels 'Normal' and 'Fraud' to indicate which histogram corresponds to normal transactions and which corresponds to fraudulent transactions. The `loc='upper right'` parameter specifies the position of the legend in the upper-right corner of the plot.

5. `plt.title("Transcation Amount vs Percentage of Transcations")`:
    - This line sets the title of the plot to "Transcation Amount vs Percentage of Transcations," providing a description of what the plot represents.
6. `plt.xlabel("Transcation Amount (USD)")` and `plt.ylabel("Percentage of Transcations")`:
    - These lines set the labels for the x-axis and y-axis, respectively. The x-axis represents transaction amounts in USD, and the y-axis represents the percentage of transactions.
7. `plt.show()`:
    - This line displays the plot, making it visible to the user.

1. `sc = StandardScaler()`: This line creates an instance of the `StandardScaler` class from scikit-learn's preprocessing module. The `StandardScaler` is used to standardize (scale) features by removing the mean and scaling to unit variance. It's a common preprocessing step in machine learning to make sure that different features have similar scales, which can help improve the performance of certain algorithms.
2. `dataset['Time'] = sc.fit_transform(dataset['Time'].values.reshape(-1, 1))`:
    - `sc.fit_transform(...)`: This method takes the 'Time' column from the 'dataset' DataFrame, reshapes it into a 2D array (with one column), and then scales it using the `StandardScaler`. The `fit_transform` method computes the mean and standard deviation of the data and then scales the data accordingly.
    - After this line, the 'Time' column in the 'dataset' DataFrame is standardized, meaning it has a mean of 0 and a standard deviation of 1.
3. `dataset['Amount'] = sc.fit_transform(dataset['Amount'].values.reshape(-1, 1))`:

    - Similar to the previous step, this line scales the 'Amount' column using the **`StandardScaler`**. Afterward, the 'Amount' column is also standardized.

4. **`raw_data = dataset.values`**:
    - This line extracts the values of the 'dataset' DataFrame and assigns them to the 'raw_data' variable. 'raw_data' is now a NumPy array containing all the data, including both the features (independent variables) and the target variable (labels).
5. **`labels = raw_data[:, -1]`**:
    - This line extracts the target variable, often referred to as 'labels,' from the 'raw_data' array. It selects the last column of the 'raw_data' array, assuming that the target variable is in the last column.
6. **`data = raw_data[:, 0:-1]`**:
    - This line extracts the features (independent variables) from the 'raw_data' array. It selects all columns except the last one, effectively removing the target variable from the feature set.
7. **`train_data, test_data, train_labels, test_labels = train_test_split(data, labels, test_size=0.2, random_state=2021)`**:
    - This line splits the dataset into training and testing sets using scikit-learn's **`train_test_split`** function. The arguments are as follows:
        - **`data`**: The feature data (independent variables) stored in the 'data' variable.
        - **`labels`**: The target variable (labels) stored in the 'labels' variable.
        - **`test_size=0.2`**: This specifies that 20% of the data will be used for testing, and the remaining 80% will be used for training.
        - **`random_state=2021`**: This sets the random seed for reproducibility, ensuring that the same split is obtained every time the code is run with the same seed.

After running this code, you'll have the following variables:

- **`train_data`**: The feature data for training.
- **`test_data`**: The feature data for testing.
- **`train_labels`**: The target labels for training.
- **`test_labels`**: The target labels for testing.

These variables are typically used to train machine learning models and evaluate their performance.

1. `min_val = tf.reduce_min(train_data)`: This line uses TensorFlow's `reduce_min` function to compute the minimum value (`min_val`) from the `train_data`. `train_data` is assumed to be a TensorFlow tensor containing the training data. This step calculates the minimum value of the training data.
2. `max_val = tf.reduce_max(train_data)`: Similar to the previous step, this line uses TensorFlow's `reduce_max` function to compute the maximum value (`max_val`) from the `train_data`. It calculates the maximum value of the training data.
3. `train_data = (train_data - min_val) / (max_val - min_val)`: This line performs feature scaling on the training data. It subtracts the minimum value (`min_val`) from each element in the `train_data` and then divides the result by the range (the difference between the maximum value `max_val` and the minimum value `min_val`). This scaling process is often referred to as min-max scaling or normalization. It scales the training data to the range [0, 1].
4. `test_data = (test_data - min_val) / (max_val - min_val)`: Similar to the previous step, this line scales the testing data using the same `min_val` and `max_val` computed from the training data. It's important to use the same scaling parameters for both the training and testing data to ensure consistency.
5. `train_data = tf.cast(train_data, tf.float32)`: This line converts the data type of the `train_data` tensor to `tf.float32`. It ensures that the data is represented as 32-bit floating-point numbers, which is a common data type for deep learning models in TensorFlow.
6. `test_data = tf.cast(test_data, tf.float32)`: Similar to the previous step, this line converts the data type of the `test_data` tensor to `tf.float32`.

After executing these steps, both the training and testing data will be scaled to the range [0, 1] and represented as TensorFlow tensors with a data type of `tf.float32`.

1. `train_labels = train_labels.astype(bool)` and `test_labels = test_labels.astype(bool)`: These lines convert the data type of the training and testing labels to boolean (`bool`). Typically, labels for binary classification tasks (such as distinguishing between normal and fraudulent transactions) are represented as boolean values, where `True` represents the positive class (e.g., fraud) and `False` represents the negative class (e.g., normal).
2. `normal_train_data = train_data[~train_labels]` and `normal_test_data = test_data[~test_labels]`: These lines create datasets containing the feature data for normal transactions in both the training and testing sets. The `~` operator is used to index

the rows where `train_labels` and `test_labels` are `False`, indicating normal transactions. So, `normal_train_data` contains the features for normal transactions in the training set, and `normal_test_data` contains the features for normal transactions in the testing set.

3. `fraud_train_data = train_data[train_labels]` and `fraud_test_data = test_data[test_labels]`: Similarly, these lines create datasets containing the feature data for fraudulent transactions in both the training and testing sets. Here, we index the rows where `train_labels` and `test_labels` are `True`, indicating fraudulent transactions. So, `fraud_train_data` contains the features for fraudulent transactions in the training set, and `fraud_test_data` contains the features for fraudulent transactions in the testing set.
4. `print("No. of records in Fraud Train Data=", len(fraud_train_data))`: This line prints the number of records (i.e., samples or transactions) in the fraudulent training dataset.
5. `print("No. of records in Normal Train Data=", len(normal_train_data))`: This line prints the number of records in the normal training dataset.
6. `print("No. of records in Fraud Test Data=", len(fraud_test_data))`: This line prints the number of records in the fraudulent testing dataset.
7. `print("No. of records in Normal Test Data=", len(normal_test_data))`: Finally, this line prints the number of records in the normal testing dataset.

These steps help you organize your data into separate datasets for normal and fraudulent transactions, which is useful for training and evaluating machine learning models for anomaly detection or fraud detection tasks.

1. `nb_epoch = 20`: This variable represents the number of training epochs. An epoch is one complete pass through the entire training dataset during the training of a neural network. In this case, the training process will consist of 20 epochs, meaning that the model will see the entire training dataset 20 times during training.
2. `batch_size = 64`: Batch size refers to the number of data samples used in each iteration of training. In deep learning, it's common to train models using mini-batch gradient descent, where the training dataset is divided into smaller batches. In this case, each batch will contain 64 data samples. Training in mini-batches is computationally efficient and helps models converge faster.
3. `input_dim = normal_train_data.shape[1]`: This variable represents the dimensionality (number of features or input units) of the input data. It is determined by the number of columns in the `normal_train_data` dataset. It's important to specify the correct input dimension when defining the neural network architecture.
4. `encoding_dim = 14`: The encoding dimension represents the dimensionality of the hidden layer in the autoencoder that encodes the input data into a lower-dimensional representation. In this case, the encoding dimension is set to 14, meaning that the

autoencoder will learn to represent the input data in a lower-dimensional space of 14 dimensions.

5. `hidden_dim1 = int(encoding_dim / 2)`: This variable calculates the dimensionality of the first hidden layer in the autoencoder. It is set to half of the encoding dimension (`encoding_dim`). This choice of architecture is common in autoencoders, where the first hidden layer reduces the dimensionality and the second hidden layer increases it.
6. `hidden_dim2 = 4`: This variable represents the dimensionality of the second hidden layer in the autoencoder. It's set to 4, which is a lower-dimensional representation compared to the first hidden layer. This architecture can be useful for learning more compact representations of the data.
7. `learning_rate = 1e-7`: The learning rate is a hyperparameter that controls the step size during the optimization process (e.g., gradient descent). A smaller learning rate (in this case, 1e-7) means smaller steps, which can help the model converge more smoothly and avoid overshooting the optimal solution during training. Learning rates are often tuned during model training to find the right balance between convergence speed and stability.

1. `input_layer = tf.keras.layers.Input(shape=(input_dim,))`: This line defines the input layer of the autoencoder. It specifies the shape of the input data, which should match the `input_dim` variable you defined earlier. In this case, it creates an input layer with a shape of `(input_dim,)`.
2. Encoder Layers:
   - `encoder = tf.keras.layers.Dense(encoding_dim, activation="tanh", activity_regularizer=tf.keras.regularizers.l2(learning_rate))(input_layer)`: This line defines the first encoder layer. It's a fully connected (Dense) layer with `encoding_dim` neurons, which is the encoding dimension you defined earlier. The activation function used is hyperbolic tangent (tanh). Additionally, it applies L2 regularization to the layer with a regularization strength determined by the `learning_rate` hyperparameter.
   - `encoder = tf.keras.layers.Dropout(0.2)(encoder)`: This line applies dropout regularization to the encoder output. Dropout is a technique used to prevent overfitting by randomly setting a fraction of the input units to zero during training.
   - `encoder = tf.keras.layers.Dense(hidden_dim1, activation='relu')(encoder)`: This line defines the second encoder layer with `hidden_dim1` neurons and a ReLU activation function. This layer reduces the dimensionality further.
   - `encoder = tf.keras.layers.Dense(hidden_dim2, activation=tf.nn.leaky_relu)(encoder)`: This line defines the third encoder layer with `hidden_dim2` neurons and a leaky ReLU activation function. Leaky ReLU

allows a small gradient when the unit is not active, which can help the network avoid dead neurons.

3. Decoder Layers:
   - `decoder = tf.keras.layers.Dense(hidden_dim1, activation='relu')(encoder)`: This line defines the first decoder layer, which has the same number of neurons as `hidden_dim1` and a ReLU activation function. This layer starts the process of increasing the dimensionality back to the original input dimension.
   - `decoder = tf.keras.layers.Dropout(0.2)(decoder)`: Like in the encoder, dropout is applied to the decoder output to prevent overfitting.
   - `decoder = tf.keras.layers.Dense(encoding_dim, activation='relu')(decoder)`: This line defines the second decoder layer with `encoding_dim` neurons and a ReLU activation function. It continues increasing the dimensionality.
   - `decoder = tf.keras.layers.Dense(input_dim, activation='tanh')(decoder)`: This line defines the final decoder layer with `input_dim` neurons and a tanh activation function. It produces the output layer that attempts to reconstruct the input data.
4. `autoencoder = tf.keras.Model(inputs=input_layer, outputs=decoder)`: This line creates the autoencoder model by specifying the input and output layers. The model takes the `input_layer` as input and produces the `decoder` output as the reconstruction. This is a standard way to create a Keras model.
5. `autoencoder.summary()`: This line prints a summary of the autoencoder model, including the layers, their types, output shapes, and the number of trainable parameters.

1. `tf.keras.callbacks.ModelCheckpoint`:
   - `filepath="autoencoder_fraud.h5"`: This parameter specifies the file path where the trained model weights will be saved. In this case, the model weights will be saved in a file named "autoencoder_fraud.h5" in the current directory.
   - `mode='min'`: This parameter defines the monitoring mode, which determines when to save the model checkpoint. `'min'` indicates that the model will be saved when the monitored quantity (specified by the `monitor` parameter) reaches its minimum value.
   - `monitor='val_loss'`: This parameter specifies the quantity to be monitored during training. In this case, it's monitoring the validation loss, which is a common metric used to evaluate the performance of a neural network during training.

- `verbose=2`: The `verbose` parameter controls the verbosity of the callback's output. A value of `2` means that it will display a progress message for each epoch when saving the best model checkpoint.
- `save_best_only=True`: When set to `True`, this parameter ensures that only the best model checkpoint (based on the monitored quantity) is saved. If a new checkpoint is saved and it doesn't improve the monitored quantity, it will overwrite the previous best checkpoint.

2. `tf.keras.callbacks.EarlyStopping`:
   - `monitor='val_loss'`: Like in the ModelCheckpoint callback, this parameter specifies the quantity to be monitored during training, which is the validation loss in this case.
   - `min_delta=0.0001`: The `min_delta` parameter determines the minimum change in the monitored quantity that must be considered an improvement. If the change is smaller than this value, the training will stop.
   - `patience=10`: The `patience` parameter specifies the number of consecutive epochs with no improvement on the monitored quantity before training is stopped. In this case, if there is no improvement for 10 consecutive epochs, training will be halted.
   - `verbose=11`: Similar to the ModelCheckpoint callback, the `verbose` parameter controls the verbosity of the output. A value of `11` means that it will display a message when early stopping is triggered.
   - `mode='min'`: This parameter indicates that early stopping should be triggered when the monitored quantity (validation loss) reaches its minimum value.
   - `restore_best_weights=True`: When set to `True`, this parameter ensures that the model's weights are restored to the best checkpoint (based on validation loss) when early stopping is triggered. This helps to keep the best-performing model.

In the provided code, the `compile` method is called on the `autoencoder` model. This method is used to configure the training process for the neural network. Let's break down the parameters used in this method call:

- `metrics=['accuracy']`: This parameter specifies a list of metrics to be calculated and displayed during training. In this case, the only metric specified is `'accuracy'`. It means that during training and evaluation, the model will compute and display the accuracy of its predictions. Accuracy is a common metric used for classification tasks and represents the fraction of correctly predicted instances.
- `loss='mean_squared_error'`: This parameter specifies the loss function to be used during training. The loss function is a mathematical function that quantifies the difference between the model's predictions and the actual target values. In this case,

`'mean_squared_error'` is used as the loss function. Mean squared error (MSE) is a common loss function for autoencoders and regression tasks. It calculates the average squared difference between the predicted values and the true values. The goal during training is to minimize this loss, which means making the predicted values as close as possible to the true values.
- `optimizer='adam'`: This parameter specifies the optimization algorithm to be used during training. `'adam'` refers to the Adam optimizer, which is a popular choice for training neural networks. The Adam optimizer adapts the learning rate during training and combines techniques from both RMSprop and momentum optimization. It is known for its good convergence properties and is widely used for various deep learning tasks.

In summary, the `compile` method is setting up the autoencoder model for training. It specifies that accuracy should be tracked as a metric, uses mean squared error as the loss function to be minimized, and employs the Adam optimizer for gradient descent during training. These settings are appropriate for an autoencoder used for anomaly detection, where the goal is to reconstruct normal data accurately while detecting anomalies based on the reconstruction error.

In the provided code, the `fit` method is called on the `autoencoder` model to train the neural network. Let's break down the parameters used in this method call and what each of them does:

- `normal_train_data, normal_train_data`: The first two arguments are the input data and target data for training. In this case, `normal_train_data` is used both as input and target because the autoencoder is trained to reconstruct its input, and `normal_train_data` represents normal data examples that the model should learn to reconstruct.
- `epochs=nb_epoch`: The `epochs` parameter specifies the number of training epochs. An epoch is one complete pass through the entire training dataset. `nb_epoch` is a variable defined earlier in the code and represents the number of training epochs, which is set to 20 in this case.
- `batch_size=batch_size`: The `batch_size` parameter determines the number of training examples used in each forward and backward pass during each training iteration. It controls how often the model's weights are updated. `batch_size` is also a variable defined earlier in the code and is set to 64.
- `shuffle=True`: This parameter indicates whether the training data should be shuffled before each epoch. Shuffling the data helps prevent the model from memorizing the order of the training examples and makes training more robust.

- `validation_data=(test_data, test_data)`: The `validation_data` parameter specifies a tuple containing the validation input data (`test_data`) and validation target data (`test_data`). During training, the model's performance on this validation dataset is monitored, and metrics like loss and accuracy are calculated. This helps assess how well the model generalizes to unseen data.
- `verbose=1`: The `verbose` parameter controls the verbosity of the training output. A value of `1` means that training progress and metrics will be displayed for each epoch, allowing you to monitor the training process.
- `callbacks=[cp, early_stop]`: The `callbacks` parameter allows you to specify a list of callback functions to be executed during training. In this case, two callbacks are specified:
    - `cp` (ModelCheckpoint): This callback, as explained earlier, saves the model's weights when the validation loss reaches a minimum, ensuring that the best model is saved.
    - `early_stop` (EarlyStopping): This callback, as explained earlier, monitors the validation loss and stops training early if the loss doesn't improve for a specified number of consecutive epochs, preventing overfitting.

Finally, `.history` is used to capture the training history, which includes information about the training and validation loss, as well as any specified metrics (in this case, accuracy). This training history can be used for visualizing the training progress and evaluating the model's performance.

1. `test_x_predictions = autoencoder.predict(test_data)`: This line uses the trained autoencoder model (`autoencoder`) to make predictions on the test data (`test_data`). The autoencoder takes the test data as input and attempts to reconstruct it. `test_x_predictions` will contain the reconstructed data, which should ideally be similar to the original test data if the autoencoder has learned to represent the data effectively.
2. `mse = np.mean(np.power(test_data - test_x_predictions, 2), axis=1)`: This line calculates the Mean Squared Error (MSE) between the original test data (`test_data`) and the reconstructed data (`test_x_predictions`) for each data point along `axis=1` (along rows). MSE measures the average squared difference between the original and reconstructed data for each feature (or dimension) in the data.
    - `np.power(test_data - test_x_predictions, 2)`: This part calculates the squared difference element-wise between the original and reconstructed data.
    - `np.mean(...)`: Calculates the mean of the squared differences, resulting in the average reconstruction error for each data point.
3. `error_df = pd.DataFrame({'Reconstruction_error': mse, 'True_class': test_labels})`: Here, a DataFrame (`error_df`) is created to store the reconstruction

error and the true class labels for each data point in the test dataset. The DataFrame has two columns:

- `'Reconstruction_error'`: This column contains the calculated reconstruction errors (MSE) for each data point.
- `'True_class'`: This column contains the true class labels for each data point, which are typically '0' for normal data and '1' for fraudulent data.

The resulting `error_df` DataFrame will be used to analyze and visualize the reconstruction errors and assess how well the autoencoder distinguishes between normal and fraudulent data points based on these errors. This information is crucial for detecting anomalies or fraud in the dataset.

the following steps are performed to visualize the reconstruction errors and the chosen threshold for distinguishing between normal and fraudulent data points:

1. `threshold_fixed = 50`: This line defines a fixed threshold value of 50. This threshold will be used to classify data points as either normal or fraudulent based on their reconstruction errors. Any data point with a reconstruction error greater than or equal to 50 will be considered as a potential anomaly or fraudulent data point.
2. `groups = error_df.groupby('True_class')`: Here, the DataFrame `error_df` is grouped by the 'True_class' column, which contains the true class labels (0 for normal and 1 for fraudulent data points). This step separates the data points into two groups: one for normal data and one for fraudulent data.
3. `fig, ax = plt.subplots()`: This line initializes a Matplotlib figure and axes for creating a plot.
4. `for name, group in groups:`: This loop iterates through the two groups created earlier, one for normal data and one for fraudulent data.

a. `ax.plot(group.index, group.Reconstruction_error, marker='o', ms=3.5, linestyle='', label="Fraud" if name == 1 else "Normal")`: For each group, a scatter plot is created using `ax.plot()`. The x-axis represents the index of each data point, and the y-axis represents the reconstruction error for that data point. The `marker='o'` option specifies that circular markers should be used, and `ms=3.5` sets the marker size. The `label` parameter is set to "Fraud" if the group represents fraudulent data (True_class == 1) and "Normal" otherwise.
b. `ax.hlines(threshold_fixed, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100, label="Threshold")`: This line adds a horizontal red line to the plot at the fixed threshold value (`threshold_fixed`). The `ax.get_xlim()` function is used to determine the start and end points of the x-axis for proper placement of the threshold line.

5. `ax.legend()`: This line adds a legend to the plot, which indicates the meaning of the plotted elements (e.g., "Normal," "Fraud," and "Threshold").
6. `plt.title("Reconstruction error for normal and fraud data")`: Sets the title of the plot.
7. `plt.ylabel("Reconstruction error")`: Sets the label for the y-axis.
8. `plt.xlabel("Data point index")`: Sets the label for the x-axis, indicating the index of each data point.
9. `plt.show()`: Finally, this command displays the plot, allowing you to visualize the distribution of reconstruction errors and how they relate to the chosen threshold for classification. This type of visualization can help in understanding how well the autoencoder distinguishes between normal and fraudulent data points based on their reconstruction errors.

This code block is performing the following steps for evaluating the performance of an anomaly detection model using a fixed threshold (`threshold_fixed`) on reconstruction errors:

1. `threshold_fixed = 52`: This line sets a fixed threshold value to 52. The threshold will be used to classify data points as normal or fraudulent based on their reconstruction errors.
2. `pred_y = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error.values]`: This line creates a list `pred_y` of predicted labels (0 for normal and 1 for fraudulent) based on the reconstruction errors. It iterates through the reconstruction errors stored in `error_df.Reconstruction_error.values` and assigns 1 to `pred_y` if the error is greater than the `threshold_fixed`, otherwise assigns 0.
3. `error_df['pred'] = pred_y`: This line adds a new column named 'pred' to the `error_df` DataFrame, storing the predicted labels generated in the previous step.
4. `conf_matrix = confusion_matrix(error_df.True_class, pred_y)`: This line calculates the confusion matrix using the true class labels (`True_class` column in `error_df`) and the predicted labels (`pred_y`). The confusion matrix is a 2x2 matrix that shows the counts of true positives, false positives, true negatives, and false negatives.
5. `plt.figure(figsize=(4, 4))`: This line sets the figure size for the confusion matrix plot.
6. `sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d")`: This code uses Seaborn's heatmap function to visualize the confusion matrix. It specifies the confusion matrix (`conf_matrix`) as input data and includes axis labels (xticklabels and yticklabels) based on the class labels stored in the

`LABELS` variable. The `annot=True` parameter adds numerical annotations to the heatmap cells, and `fmt="d"` formats the annotations as integers.

7. `plt.title("Confusion matrix")`: Sets the title for the confusion matrix plot.
8. `plt.ylabel("True class")` and `plt.xlabel("Predicted class")`: Set the labels for the y-axis and x-axis of the confusion matrix plot.
9. `plt.show()`: Displays the confusion matrix plot, allowing you to visually assess the model's performance in classifying normal and fraudulent data points.
10. Following the confusion matrix plot, the code prints the following performance metrics:
    - Accuracy: This metric measures the overall correctness of the model's predictions, i.e., the ratio of correctly classified data points to the total number of data points.
    - Recall: Also known as sensitivity or true positive rate, it measures the ability of the model to correctly identify positive (fraudulent) cases among all actual positive cases.
    - Precision: This metric measures the ability of the model to correctly identify positive cases among all predicted positive cases.

These metrics provide valuable insights into the model's performance in detecting fraudulent transactions using the chosen threshold and reconstruction errors.