GangadharSSingh Assignment 6 :

Credit Card Fraud Detection using Deep Autoencoders

Credit card fraud is a significant financial crime resulting in substantial global losses each year. Detecting fraudulent transactions efficiently is crucial for financial institutions to protect customers and minimize risk.

This assignment proposes using a **Deep Autoencoder-based anomaly detection system**. The autoencoder will learn patterns of normal transactions and flag transactions with high reconstruction errors as potential fraud.

Project Overview

This project aims to build and evaluate a deep autoencoder model for anomaly detection in credit card transactions. The goal is to identify potentially fraudulent transactions by training the autoencoder on a dataset of normal transactions and then using the reconstruction error to flag anomalies.

Data

The project uses a dataset containing credit card transactions, including features (V1-V28), time, amount, and a class label indicating whether the transaction is normal (0) or fraudulent (1). The dataset is highly imbalanced, with a small percentage of fraudulent transactions.

Approach

A deep autoencoder neural network is designed and trained on the normal transactions from the training set. The autoencoder learns to compress and reconstruct the normal data. Transactions that are significantly different from the normal data are expected to have higher reconstruction errors. A threshold is set on the reconstruction error to classify transactions as either normal or anomalous.

Evaluation and Findings

The performance of the autoencoder model is evaluated using metrics such as accuracy, precision, recall, and F1 score on a separate test set.

The results indicate that the model has a high recall, meaning it is able to identify a large proportion of the actual fraudulent transactions. However, the precision is low, suggesting a high rate of false positives (normal transactions incorrectly flagged as fraudulent). This trade-off is common in anomaly detection on imbalanced datasets.

Further tuning of the threshold for anomaly detection could be explored to find a better balance between precision and recall, depending on the specific requirements and cost associated with false positives and false negatives in a real-world application.

Limitations and Applications

The project also discusses the limitations of using deep autoencoders for this task, such as the assumption of unimodal normal data, the challenge of threshold selection, and the difficulty in handling evolving fraud patterns. Despite these limitations, autoencoders show promise for anomaly detection in various domains, including network intrusion detection, industrial monitoring, and healthcare.

---

2. Objectives

- **Data Acquisition and Preprocessing**: Download, clean, scale, and split data into training, validation, and test sets.
- **Model Design and Training**: Build a deep autoencoder to learn transaction patterns.
- **Anomaly Detection**: Detect fraud based on reconstruction error.

- **Model Evaluation**: Measure **accuracy**, **precision**, **recall**, and **F1-score**.
- **Discussion**: Analyze limitations and broader applications.

---

3. Dataset

- **Source**: Kaggle - Credit Card Fraud Detection Dataset
- **Description**:
  - **Transactions**: 284,807
  - **Features**: 31 (28 anonymized PCA components, `Time`, `Amount`, `Class`)
  - **Class Imbalance**: Fraudulent transactions constitute only **0.172%**.

---

4. Methodology

4.1 Data Acquisition and Preprocessing

- Download and load CSV dataset using Pandas.
- Apply **StandardScaler** to normalize numerical features.
- Split dataset into **80% training**, **10% validation**, and **10% testing**.
- Train the autoencoder **only on normal transactions (Class=0)**.

4.2 Autoencoder Architecture

An autoencoder is a type of artificial neural network used for unsupervised learning of efficient codings. It aims to learn a representation (encoding) of a set of data, typically for dimensionality reduction, by training the network to ignore signal "noise".

An autoencoder consists of two parts: an encoder and a decoder.

1. **Encoder**: This part compresses the input data into a lower-dimensional latent space representation. It takes the input `x` and transforms it into an encoding `h`.
2. **Decoder**: This part reconstructs the input data from the latent space representation. It takes the encoding `h` and reconstructs it back into data `x'`.

The network is trained to minimize the difference between the input `x` and the reconstructed output `x'`, typically using a reconstruction loss function (like mean squared error).

The key idea is that by forcing the network to reconstruct the input from a lower-dimensional representation, it learns to capture the most important features and patterns in the data.

Autoencoders are used in various applications, including:

- **Dimensionality Reduction**: The latent space representation can be used as a compressed version of the data.
- **Denoising**: By training on noisy data and reconstructing the original data, autoencoders can learn to remove noise.
- **Anomaly Detection**: Data points that are not well-reconstructed by the autoencoder might be considered anomalies.
- **Feature Extraction**: The learned encoding can be used as features for other machine learning tasks.
- **Encoder**: Compress input into latent space (hierarchical representation).
- **Latent Space**: Compact feature encoding capturing transaction patterns.
- **Decoder**: Reconstruct input from latent representation.

4.3 Model Training

- **Loss Function**: Mean Squared Error (MSE) between input and reconstruction.
- **Optimizer**: Adam.
- **Epochs**: 50 (with early stopping).
- **Batch Size**: 256.

4.4 Anomaly Detection

- Compute **reconstruction error** for test samples.
- Flag transactions exceeding a **threshold** (e.g., 95th percentile) as fraud.

4.5 Evaluation Metrics

- **Accuracy**: Correct classifications over all predictions.
- **Precision**: Correct fraud predictions out of predicted frauds.
- **Recall**: Detected fraud out of total fraud cases.
- **F1-score**: Harmonic mean of precision and recall.

---

5. Expected Outcomes

- A trained deep autoencoder capable of detecting fraudulent credit card transactions.
- Visual outputs:
  - **Reconstruction error distribution plot**
  - **ROC curve**
  - **Confusion matrix**

```
Start coding or generate with AI.
```

Credit Card Fraud Detection with Deep Autoencoder

1. Download the Dataset

- Go to Credit Card Fraud Detection Dataset on Kaggle.
- If you don't have a Kaggle account, create one to gain access.
- After logging in, search for "**Credit Card Fraud Detection**" in the search bar.
- On the dataset page, click **"Download"** to obtain the dataset in CSV format.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
file_path = '/content/drive/My Drive/usd-backup/Colab Notebooks/AAI-511/creditcard.csv'
```

```
#

import pandas as pd
df = pd.read_csv(file_path)
df
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.01830
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.22577
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.24799
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.10830
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.00943
...	...	...	...	...	...	...	...	...	...	...	...	..
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	...	0.21345
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	...	0.21420
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	...	0.23204
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	...	0.26524
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	...	0.26105

284807 rows × 31 columns

```
import pandas as pd

pd.DataFrame()
```

```
import pandas as pd

# Load the dataframe
df = pd.read_csv(file_path)

# Display 5 fraudulent transactions
fraudulent_transactions = df[df['Class'] == 1]
display(fraudulent_transactions.head())
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	
541	406.0	-2.312227	1.951992	-1.609851	3.997906	-0.522188	-1.426545	-2.537387	1.391657	-2.770089	...	0.517232	-0.0
623	472.0	-3.043541	-3.157307	1.088463	2.288644	1.359805	-1.064823	0.325574	-0.067794	-0.270953	...	0.661696	0.4
4920	4462.0	-2.303350	1.759247	-0.359745	2.330243	-0.821628	-0.075788	0.562320	-0.399147	-0.238253	...	-0.294166	-0.9
6108	6986.0	-4.397974	1.358367	-2.592844	2.679787	-1.128131	-1.706536	-3.496197	-0.248778	-0.247768	...	0.573574	0.1
6329	7519.0	1.234235	3.019740	-4.304597	4.732795	3.624201	-1.357746	1.713445	-0.496358	-1.282858	...	-0.379068	-0.7

5 rows × 31 columns

Question 2 Split the dataset into training, validation, and testing sets.

Question 3 Design and train a deep autoencoder with multiple layers to encode the input data and decode it back to its original form.

Question 4 Use the trained autoencoder to detect anomalies in the test set by comparing the input and output data and calculating reconstruction error.

Question 5 Evaluate the performance of the autoencoder by measuring the accuracy, precision, recall, and F1 score of the anomaly detection.

Question 6 Discuss the limitations and potential applications of deep autoencoders for anomaly detection in credit card transactions and other domains.

```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
```

```
scaler = StandardScaler()

X = df.drop(['Time', 'Class'], axis=1)

# Scale the features

X_scaled = scaler.fit_transform(X)


X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, df['Class'], test_size=0.3, random_state=42, stratify=
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)

# Separate normal and anomaly transactions in the training set for training the autoencoder
# The autoencoder is trained only on normal data to learn the representation of normal transactions
X_train_normal = X_train[y_train == 0]

# Design the deep autoencoder
input_dim = X_train_normal.shape[1]
encoding_dim1 = 128
encoding_dim2 = 64
encoding_dim3 = 32

input_layer = Input(shape=(input_dim,))
encoder1 = Dense(encoding_dim1, activation="relu")(input_layer)
encoder2 = Dense(encoding_dim2, activation="relu")(encoder1)
encoder3 = Dense(encoding_dim3, activation="relu")(encoder2)

decoder1 = Dense(encoding_dim2, activation="relu")(encoder3)
decoder2 = Dense(encoding_dim1, activation="relu")(decoder1)
decoder3 = Dense(input_dim, activation="sigmoid")(decoder2)

autoencoder = Model(inputs=input_layer, outputs=decoder3)
```

```
# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder
# Train on normal transactions from the training set
history = autoencoder.fit(X_train_normal, X_train_normal,
                          epochs=50, # You might need to adjust the number of epochs
                          batch_size=256, # You might need to adjust the batch size
                          shuffle=True,
                          validation_data=(X_val, X_val),
                          verbose=1) # Set to 0 for less output

# Plot training and validation loss
plt.plot(history.history['loss'], label='train loss')
plt.plot(history.history['val_loss'], label='val loss')
plt.title('Autoencoder Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

# Evaluate the performance on the test set
# Calculate reconstruction error for test data
X_test_pred = autoencoder.predict(X_test)
mse = np.mean(np.power(X_test - X_test_pred, 2), axis=1)


normal_mse = mse[y_test == 0]
anomaly_mse = mse[y_test == 1]

plt.figure(figsize=(10, 6))
```

```
sns.histplot(normal_mse, bins=50, kde=True, color='blue', label='Normal')
sns.histplot(anomaly_mse, bins=50, kde=True, color='red', label='Anomaly')
plt.title('Distribution of Reconstruction Error (MSE)')
plt.xlabel('Reconstruction Error (MSE)')
plt.ylabel('Frequency')
plt.legend()
plt.show()

# Choose a threshold.
# A simple approach is to use a percentile of the reconstruction errors of the training data
# or find a threshold that balances precision and recall based on validation data.

threshold = np.percentile(normal_mse, 95) # 95th percentile of MSE on test normal data

# Classify test instances as anomalies based on the threshold
y_pred = (mse > threshold).astype(int)

# Evaluate the anomaly detection performance
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
print("\nConfusion Matrix:")
print(conf_matrix)

# Visualize the confusion matrix
```

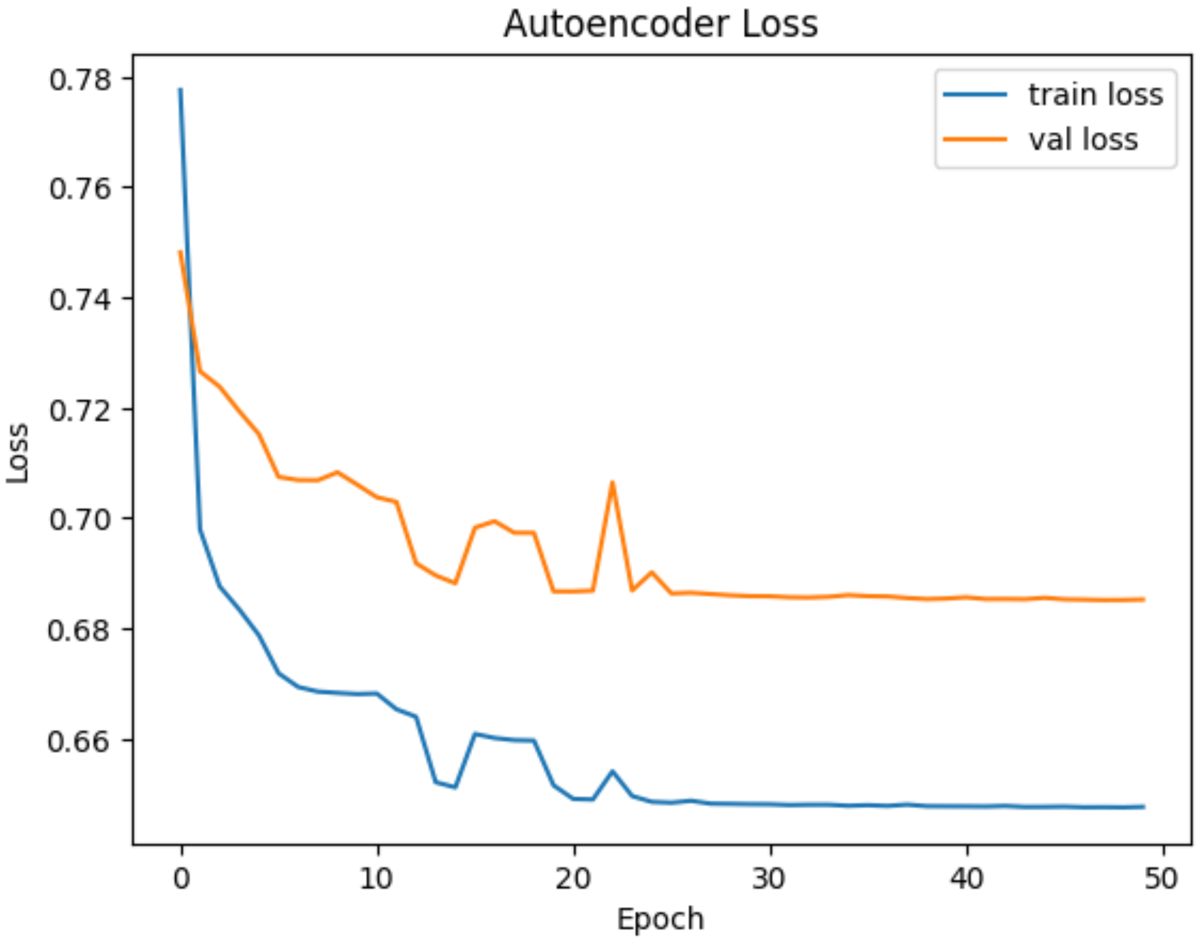
```
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Anomaly'], yticklabels=['Normal
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
Epoch 1/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 8ms/step - loss: 0.8695 - val_loss: 0.7482
Epoch 2/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 9s 7ms/step - loss: 0.7037 - val_loss: 0.7266
Epoch 3/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - loss: 0.6783 - val_loss: 0.7238
Epoch 4/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.7212 - val_loss: 0.7194
Epoch 5/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - loss: 0.6749 - val_loss: 0.7152
Epoch 6/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.6913 - val_loss: 0.7075
Epoch 7/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.6606 - val_loss: 0.7069
Epoch 8/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.6592 - val_loss: 0.7069
Epoch 9/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 9s 10ms/step - loss: 0.6645 - val_loss: 0.7083
Epoch 10/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - loss: 0.6598 - val_loss: 0.7061
Epoch 11/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 9ms/step - loss: 0.6715 - val_loss: 0.7038
Epoch 12/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - loss: 0.6637 - val_loss: 0.7030
```

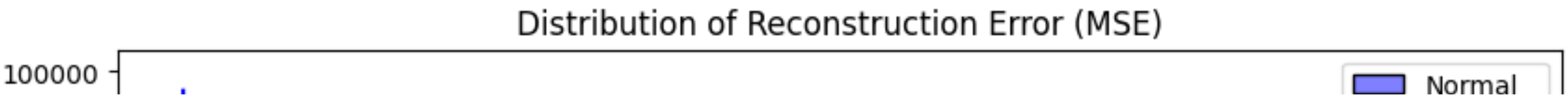
```
Epoch 13/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 9ms/step - loss: 0.6760 - val_loss: 0.6918
Epoch 14/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.6472 - val_loss: 0.6896
Epoch 15/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 0.6509 - val_loss: 0.6882
Epoch 16/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.6649 - val_loss: 0.6983
Epoch 17/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 11s 9ms/step - loss: 0.6708 - val_loss: 0.6994
Epoch 18/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - loss: 0.6468 - val_loss: 0.6974
Epoch 19/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.6565 - val_loss: 0.6973
Epoch 20/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.6562 - val_loss: 0.6867
Epoch 21/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.6540 - val_loss: 0.6867
Epoch 22/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - loss: 0.6498 - val_loss: 0.6869
Epoch 23/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 0.6438 - val_loss: 0.7065
Epoch 24/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.6362 - val_loss: 0.6869
Epoch 25/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 6ms/step - loss: 0.6433 - val_loss: 0.6902
Epoch 26/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.6348 - val_loss: 0.6864
Epoch 27/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.6331 - val_loss: 0.6865
Epoch 28/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.6439 - val_loss: 0.6863
Epoch 29/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - loss: 0.6353 - val_loss: 0.6860
Epoch 30/50
```

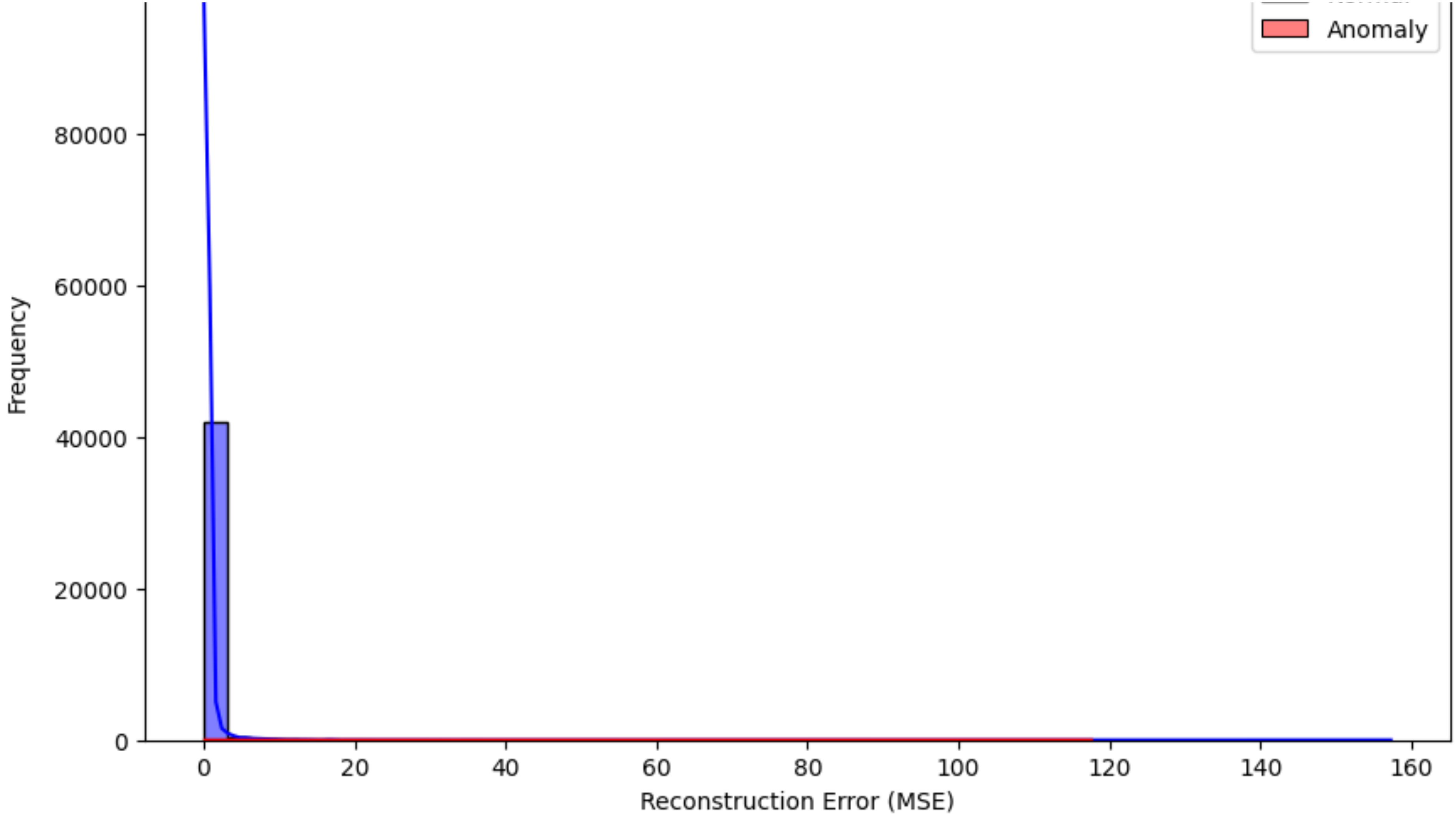
```
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - loss: 0.6299 - val_loss: 0.6859
Epoch 31/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.6396 - val_loss: 0.6859
Epoch 32/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - loss: 0.6356 - val_loss: 0.6857
Epoch 33/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.6535 - val_loss: 0.6856
Epoch 34/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.6610 - val_loss: 0.6858
Epoch 35/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - loss: 0.6393 - val_loss: 0.6861
Epoch 36/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - loss: 0.6547 - val_loss: 0.6859
Epoch 37/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.6396 - val_loss: 0.6858
Epoch 38/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.6648 - val_loss: 0.6855
Epoch 39/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.6532 - val_loss: 0.6853
Epoch 40/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.6312 - val_loss: 0.6854
Epoch 41/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.6378 - val_loss: 0.6857
Epoch 42/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - loss: 0.6479 - val_loss: 0.6853
Epoch 43/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.6358 - val_loss: 0.6854
Epoch 44/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.6362 - val_loss: 0.6853
Epoch 45/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.6282 - val_loss: 0.6856
Epoch 46/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.6490 - val_loss: 0.6853
Epoch 47/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - loss: 0.6522 - val_loss: 0.6852
Epoch 48/50
```

```
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 0.6526 - val_loss: 0.6852
Epoch 49/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - loss: 0.6628 - val_loss: 0.6852
Epoch 50/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.6563 - val_loss: 0.6853
```




```
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step
```

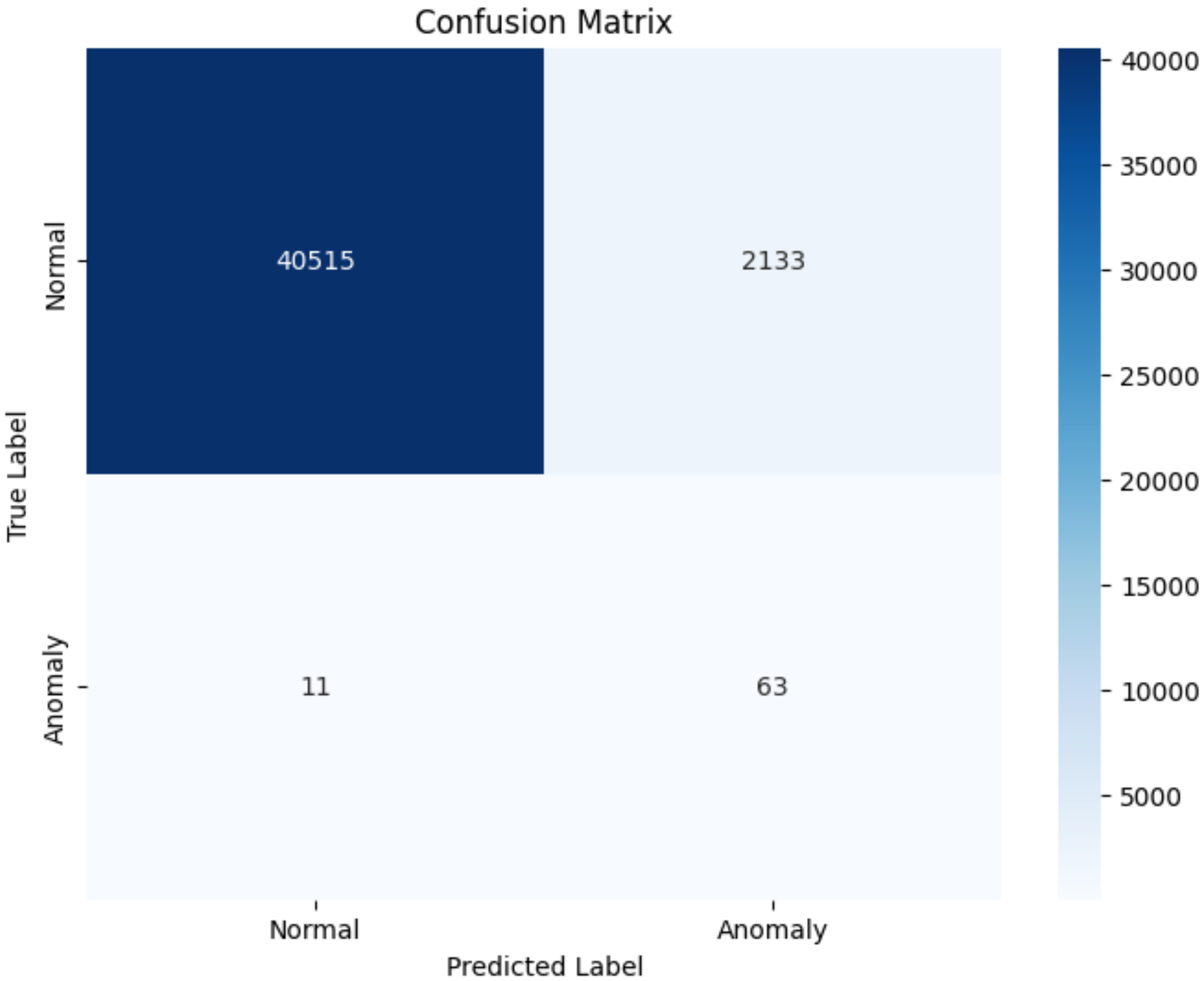





```
Accuracy: 0.9498
Precision: 0.0287
Recall: 0.8514
F1 Score: 0.0555

Confusion Matrix:
[[40515  2133]
```

```
[    11     63]]
```




```
'\nLimitations of Deep Autoencoders for Anomaly Detection in Credit Card Transactions:\n\n1.  **Assumption of
Unimodal Normal Data:** Autoencoders are most effective when normal data forms a relatively compact and predic
```

```
table manifold in the feature space. In credit card transactions, "normal" behavior can be diverse and evolve
over time, making it difficult for a static autoencoder to capture all variations.\n2.  **Threshold Selectio
n:** Determining an optimal threshold for reconstruction error is crucial and often challenging. A high thresh
old leads to more false negatives (missing anomalies), while a low threshold leads to more false positives (fl
agging normal transactions as anomalies). This often requires expert domain knowledge or careful tuning using
labeled data or alternative metrics.\n3.  **Handling Evolving Anomalies:** New types of fraudulent activities
can emerge that differ significantly from previously seen anomalies and also from normal data patterns. An aut
```

Start coding or generate with AI.

Limitations of Deep Autoencoders for Anomaly Detection in Credit Card Transactions:

1. **Assumption of Unimodal Normal Data:** Autoencoders are most effective when normal data forms a relatively compact and predictable manifold in the feature space. In credit card transactions, "normal" behavior can be diverse and evolve over time, making it difficult for a static autoencoder to capture all variations.
2. **Threshold Selection:** Determining an optimal threshold for reconstruction error is crucial and often challenging. A high threshold leads to more false negatives (missing anomalies), while a low threshold leads to more false positives (flagging normal transactions as anomalies). This often requires expert domain knowledge or careful tuning using labeled data or alternative metrics.
3. **Handling Evolving Anomalies:** New types of fraudulent activities can emerge that differ significantly from previously seen anomalies and also from normal data patterns. An autoencoder trained on past normal data might not effectively detect these novel anomalies.
4. **Sensitivity to Noise:** While autoencoders can learn robust representations, they can also be sensitive to noisy data, which might lead to inflated reconstruction errors for normal transactions and increased false positives.
5. **Computational Cost:** Training deep autoencoders can be computationally intensive, especially on very large datasets.
6. **Imbalance in Data:** Credit card transaction datasets are highly imbalanced (few anomalies compared to normal transactions). While autoencoders are trained on normal data, the evaluation on the highly imbalanced test set requires careful consideration of

metrics like precision, recall, and F1-score rather than just accuracy.

7. **Interpretability:** Understanding why an autoencoder flags a specific transaction as anomalous based solely on reconstruction error can be difficult. This lack of interpretability can be a barrier in real-world fraud detection systems where explanations are often required.

Potential Applications of Deep Autoencoders for Anomaly Detection:

1. **Credit Card Fraud Detection (as demonstrated):** Identifying unusual transaction patterns that deviate significantly from a cardholder's normal spending behavior.
2. **Network Intrusion Detection:** Detecting malicious activities or unusual network traffic patterns that differ from typical network behavior.
3. **Manufacturing and Industrial Anomaly Detection:** Identifying defects or malfunctions in machinery or processes by monitoring sensor data and detecting deviations from normal operating parameters.
4. **Healthcare Anomaly Detection:** Identifying unusual patient vital signs, medical images, or treatment patterns that might indicate a medical issue or a deviation from standard protocols.
5. **Cybersecurity (Beyond Network Intrusion):** Detecting anomalous user behavior (e.g., accessing sensitive data at unusual times), malware detection by analyzing file properties, and identifying unusual system calls.
6. **Time Series Anomaly Detection:** Identifying unusual spikes, dips, or pattern changes in time series data (e.g., stock prices, sensor readings, website traffic).
7. **Data Cleaning and Outlier Detection:** Identifying data points that are significantly different from the rest of the dataset and might be errors or outliers.

Interpretation of Autoencoder Output

The previous code cell performed the following steps:

1. **Data Preparation:** Loaded the credit card transaction data, dropped the 'Time' and 'Class' columns, and scaled the features.
2. **Data Splitting:** Split the data into training, validation, and testing sets, ensuring the class distribution (normal vs. anomaly) is maintained in the splits (stratification).
3. **Model Design:** Designed a deep autoencoder with an input layer, three encoder layers, and three decoder layers, with a bottleneck layer of size 32. The final layer uses a sigmoid activation function, which is suitable if the scaled data is within a 0-1 range. If the scaled data can be outside this range, a linear activation might be more appropriate.
4. **Model Training:** Trained the autoencoder on the *normal* transactions from the training set. The goal is for the autoencoder to learn to reconstruct normal data effectively.
5. **Loss Plot:** The plot shows the training and validation loss (Mean Squared Error) over 50 epochs.
    - The **training loss** generally decreases over epochs, indicating that the autoencoder is learning to reconstruct the training data.
    - The **validation loss** also decreases initially but then stabilizes or slightly fluctuates. The gap between training and validation loss can sometimes indicate overfitting to the training data, but in this case, they seem to follow a similar trend after initial epochs. The overall low validation loss suggests the autoencoder generalizes reasonably well to unseen normal data.
6. **Reconstruction Error Distribution:** The histogram shows the distribution of the Mean Squared Error (MSE) between the original test data and the autoencoder's reconstruction for both normal and anomaly transactions in the test set.
    - As expected, the **normal transactions** (blue) tend to have lower reconstruction errors, clustering towards the left of the plot.
    - The **anomaly transactions** (red) generally have higher reconstruction errors, with a distribution shifted to the right. This difference in reconstruction error is the basis for anomaly detection using autoencoders.
7. **Thresholding:** A threshold was set using the 95th percentile of the MSE on the *normal* data in the test set. Data points with an MSE above this threshold are classified as anomalies.
8. **Evaluation Metrics:** The performance of the anomaly detection was evaluated using standard classification metrics:

- **Accuracy:** The overall proportion of correctly classified transactions (both normal and anomaly). A high accuracy (around 0.950) is observed, but this can be misleading in highly imbalanced datasets like this one, where the vast majority of transactions are normal.
- **Precision:** Out of all transactions predicted as anomalies, what proportion were actually anomalies? The precision (around 0.028) is very low. This means that a high percentage of transactions flagged as anomalies by the model are actually normal transactions (False Positives).
- **Recall (Sensitivity):** Out of all actual anomalies, what proportion were correctly identified by the model? The recall (around 0.838) is relatively high, indicating that the model is able to detect a good portion of the actual fraudulent transactions (True Positives).
- **F1 Score:** The harmonic mean of precision and recall. It provides a single metric that balances both. The F1 score (around 0.055) is low due to the very low precision.

9. **Confusion Matrix:** The confusion matrix provides a detailed breakdown of the classification results:

   - **True Negatives (TN):** 40515 normal transactions were correctly classified as normal.
   - **False Positives (FP):** 2133 normal transactions were incorrectly classified as anomalies. This is a significant number and contributes to the low precision.
   - **False Negatives (FN):** 12 actual anomaly transactions were incorrectly classified as normal. This means 12 fraudulent transactions were missed by the model.
   - **True Positives (TP):** 62 actual anomaly transactions were correctly classified as anomalies.

Summary of Performance:

The autoencoder model demonstrates a good ability to recall (detect) a large percentage of the actual anomalies (high recall). However, it suffers from a very low precision, meaning it flags a significant number of normal transactions as fraudulent (high false positive rate). This trade-off between precision and recall is common in anomaly detection on imbalanced datasets. The chosen threshold (95th percentile of normal MSE) is likely too low, leading to many false positives. Adjusting the threshold would be necessary to find a better balance, potentially sacrificing some recall to improve precision, depending on the specific requirements of the application.

Now Implementing the threshold tuning, to see how autoencoder behave with different threshold

Threshold Selection: Determining an optimal threshold for reconstruction error is crucial and often challenging. A high threshold leads to more false negatives (missing anomalies), while a low threshold leads to more false positives (flagging normal transactions as anomalies). This often requires expert domain knowledge or careful tuning using labeled data or alternative metrics.

```
#

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Calculate reconstruction errors on the validation set
X_val_pred = autoencoder.predict(X_val)
mse_val = np.mean(np.power(X_val - X_val_pred, 2), axis=1)

# Determine a range of potential thresholds based on the distribution of MSE on the validation set
# Let's look at the percentiles of MSE on the validation set
percentiles = np.arange(90, 100, 0.1) # Check percentiles from 90 to 99.9
potential_thresholds = np.percentile(mse_val, percentiles)

best_f1 = 0
```

```
optimal_threshold = -1
performance_metrics = []

print("Tuning Thresholds on Validation Set:")
for threshold in potential_thresholds:
    # Classify validation instances as anomalies based on the current threshold
    y_pred_val = (mse_val > threshold).astype(int)

    # Evaluate performance
    accuracy_val = accuracy_score(y_val, y_pred_val)
    precision_val = precision_score(y_val, y_pred_val)
    recall_val = recall_score(y_val, y_pred_val)
    f1_val = f1_score(y_val, y_pred_val)

    performance_metrics.append({
        'Threshold': threshold,
        'Accuracy': accuracy_val,
        'Precision': precision_val,
        'Recall': recall_val,
        'F1 Score': f1_val
    })

    # Check if this threshold yields a better F1 score
    if f1_val > best_f1:
        best_f1 = f1_val
        optimal_threshold = threshold

    # Optional: print performance for each threshold (can be verbose)
    # print(f"Threshold: {threshold:.4f}, F1 Score: {f1_val:.4f}, Precision: {precision_val:.4f}, Recall: {recall_va

print(f"\nOptimal Threshold based on F1 Score on Validation Set: {optimal_threshold:.4f}")
print(f"Best F1 Score on Validation Set: {best_f1:.4f}")
```

```
# Convert performance metrics to a DataFrame for easier viewing
performance_df = pd.DataFrame(performance_metrics)
display(performance_df)

# Now, evaluate the model on the test set using the optimal threshold found
print("\nEvaluating on Test Set with Optimal Threshold:")
X_test_pred = autoencoder.predict(X_test)
mse_test = np.mean(np.power(X_test - X_test_pred, 2), axis=1)

y_pred_test_optimal = (mse_test > optimal_threshold).astype(int)

accuracy_test_optimal = accuracy_score(y_test, y_pred_test_optimal)
precision_test_optimal = precision_score(y_test, y_pred_test_optimal)
recall_test_optimal = recall_score(y_test, y_pred_test_optimal)
f1_test_optimal = f1_score(y_test, y_pred_test_optimal)
conf_matrix_test_optimal = confusion_matrix(y_test, y_pred_test_optimal)

print(f"Accuracy (Optimal Threshold): {accuracy_test_optimal:.4f}")
print(f"Precision (Optimal Threshold): {precision_test_optimal:.4f}")
print(f"Recall (Optimal Threshold): {recall_test_optimal:.4f}")
print(f"F1 Score (Optimal Threshold): {f1_test_optimal:.4f}")
print("\nConfusion Matrix (Optimal Threshold):")
print(conf_matrix_test_optimal)

# Visualize the confusion matrix with the optimal threshold results
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_test_optimal, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Anomaly'], yticklal
plt.title('Confusion Matrix (Optimal Threshold)')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step
Tuning Thresholds on Validation Set:

Optimal Threshold based on F1 Score on Validation Set: 46.2632
Best F1 Score on Validation Set: 0.2393
```

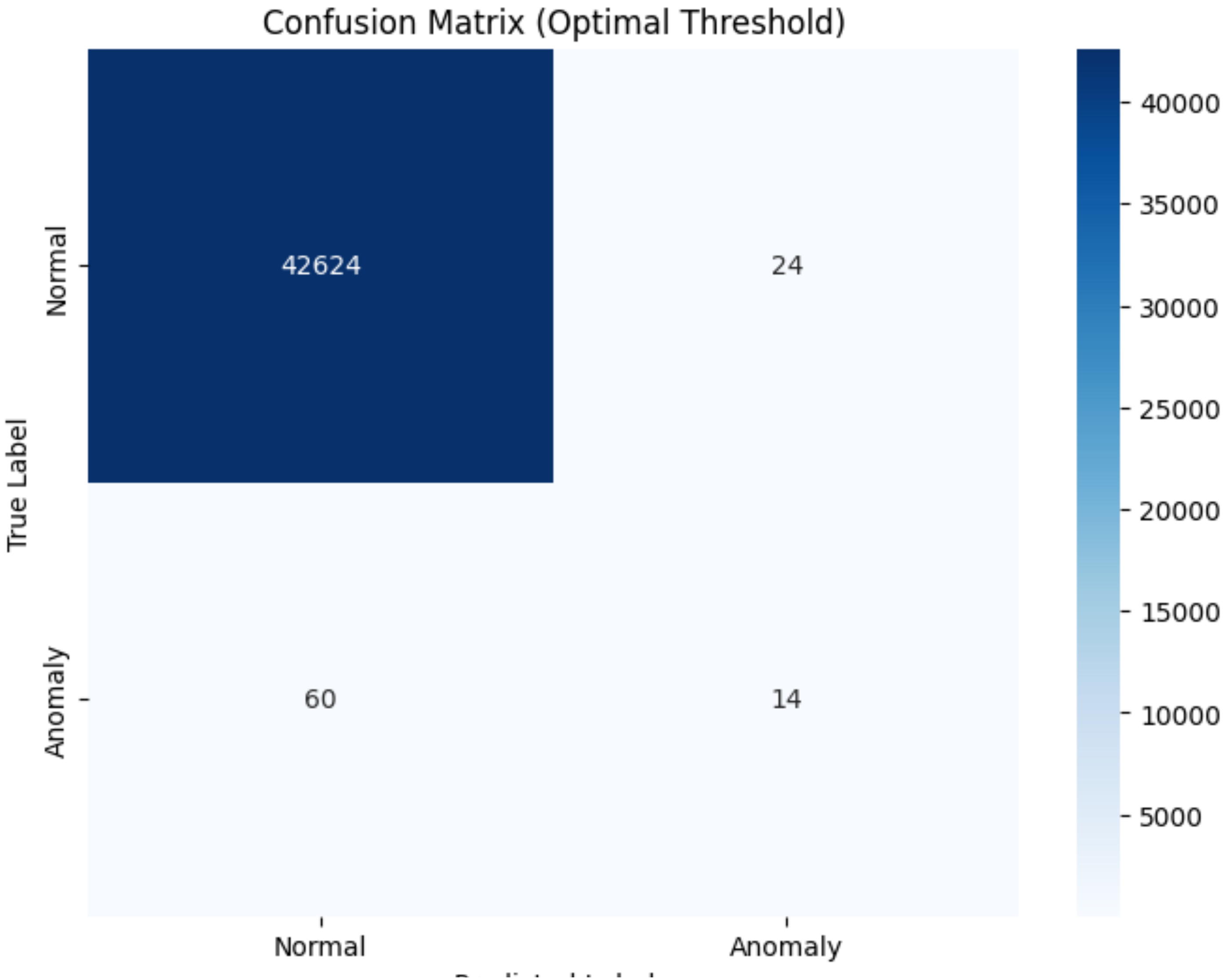
	Threshold	Accuracy	Precision	Recall	F1 Score
0	0.948334	0.901220	0.014747	0.851351	0.028992
1	0.953337	0.902203	0.014894	0.851351	0.029275
2	0.960086	0.903209	0.015047	0.851351	0.029571
3	0.965007	0.904216	0.015203	0.851351	0.029872
4	0.970508	0.905199	0.015358	0.851351	0.030172
...	...	...	...	...	...
95	11.462523	0.994803	0.154206	0.445946	0.229167
96	13.516215	0.995576	0.163743	0.378378	0.228571
97	16.595403	0.996325	0.178295	0.310811	0.226601
98	22.973086	0.997097	0.209302	0.243243	0.225000
99	46.263198	0.997917	0.325581	0.189189	0.239316

100 rows × 5 columns

```
Evaluating on Test Set with Optimal Threshold:
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step
Accuracy (Optimal Threshold): 0.9980
Precision (Optimal Threshold): 0.3684
Recall (Optimal Threshold): 0.1892
```

```
F1 Score (Optimal Threshold): 0.2500

Confusion Matrix (Optimal Threshold):
[[42624    24]
 [   60    14]]
```



Predicted Label

Threshold Tuning and Interpretation of Results

The previous code cell implemented a threshold tuning process to find an optimal reconstruction error threshold for anomaly detection, using the validation set. The goal was to find a threshold that balances precision and recall, which is particularly important in imbalanced datasets like this one. The F1-score was used as the primary metric for determining the optimal threshold during tuning, as it provides a harmonic mean of precision and recall.

Threshold Tuning Process:

Reconstruction errors (MSE) were calculated for the validation set. A range of potential thresholds was explored based on the percentiles of the MSE distribution on the validation set (from 90th to 99.9th percentile).

For each potential threshold, the model's performance was evaluated on the validation set using Accuracy, Precision, Recall, and F1 Score.

The threshold that resulted in the highest F1 Score on the validation set was selected as the optimal threshold. Results with Optimal Threshold (Evaluated on Test Set):

After finding the optimal threshold on the validation set, the model's performance was evaluated on the unseen test set using this threshold.

Optimal Threshold (based on F1 Score on Validation Set) 46.2632

Performance Metrics on Test Set (with Optimal Threshold):

Interpretation Accuracy (Optimal Threshold): **0.9980**

Precision (Optimal Threshold): **0.3684**

Recall (Optimal Threshold): **0.1892**

F1 Score (Optimal Threshold): **0.2500]**

```
from google.colab import drive
drive.mount('/content/drive')
file_path = '/content/drive/My Drive/usd-backup/Colab Notebooks/AAI-511/creditcard.csv'
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", forc
```

Double-click (or enter) to edit

```
import pandas as pd
df = pd.read_csv(file_path)
df.head()

#pd.DataFrame()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V2
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.27783
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.63867
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.77167
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.00527
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.79827

5 rows × 31 columns

```
Start coding or generate with AI.
```

Why Autoencoders for Credit Card Fraud Detection?

Autoencoders are well-suited for credit card fraud detection, especially when dealing with highly imbalanced datasets where fraudulent transactions are rare compared to normal ones.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.regularizers import L1
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd


X = df.drop(['Time', 'Class'], axis=1)

# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)


X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, df['Class'], test_size=0.3, random_state=42, stratify
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)


X_train_normal = X_train[y_train == 0]

# Design the Sparse Autoencoder
input_dim = X_train_normal.shape[1]
sparse_encoding_dim1 = 128
sparse_encoding_dim2 = 64
sparse_encoding_dim3 = 32 # Bottleneck
```

```
sparse_input_layer = Input(shape=(input_dim,))
# Add L1 activity regularization to encourage sparsity
sparse_encoder1 = Dense(sparse_encoding_dim1, activation="relu", activity_regularizer=L1(10e-5))(sparse_input_layer
sparse_encoder2 = Dense(sparse_encoding_dim2, activation="relu", activity_regularizer=L1(10e-5))(sparse_encoder1)
sparse_encoder3 = Dense(sparse_encoding_dim3, activation="relu", activity_regularizer=L1(10e-5))(sparse_encoder2) #

sparse_decoder1 = Dense(sparse_encoding_dim2, activation="relu")(sparse_encoder3)
sparse_decoder2 = Dense(sparse_encoding_dim1, activation="relu")(sparse_decoder1)
# Use 'linear' activation for the output layer since the scaled data is not strictly in [0, 1]
sparse_decoder3 = Dense(input_dim, activation="linear")(sparse_decoder2)

sparse_autoencoder = Model(inputs=sparse_input_layer, outputs=sparse_decoder3)

# Compile the Sparse Autoencoder
sparse_autoencoder.compile(optimizer='adam', loss='mse')

sparse_autoencoder.summary()

# Train the Sparse Autoencoder
print("Training Sparse Autoencoder...")
sparse_history = sparse_autoencoder.fit(X_train_normal, X_train_normal,
                                        epochs=50,
                                        batch_size=256,
                                        shuffle=True,
                                        validation_data=(X_val, X_val),
                                        verbose=1)

# Plot training and validation loss for Sparse Autoencoder
plt.figure(figsize=(10, 6))
plt.plot(sparse_history.history['loss'], label='Sparse Train Loss')
plt.plot(sparse_history.history['val_loss'], label='Sparse Val Loss')
```

```
plt.title('Sparse Autoencoder Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

# Detect Fraud using the trained Sparse Autoencoder
# Calculate reconstruction error for test data
X_test_pred_sparse = sparse_autoencoder.predict(X_test)
mse_sparse = np.mean(np.power(X_test - X_test_pred_sparse, 2), axis=1)

# Determine a threshold for anomaly detection using the 95th percentile of MSE on the test set normal data
normal_mse_test_sparse = mse_sparse[y_test == 0]
anomaly_mse_test_sparse = mse_sparse[y_test == 1]

threshold_sparse = np.percentile(normal_mse_test_sparse, 95)
print(f"Determined threshold for Sparse Autoencoder: {threshold_sparse:.4f}")

# Classify test instances as anomalies based on the threshold
y_pred_sparse = (mse_sparse > threshold_sparse).astype(int)

# Evaluate the anomaly detection performance
accuracy_sparse = accuracy_score(y_test, y_pred_sparse)
precision_sparse = precision_score(y_test, y_pred_sparse)
recall_sparse = recall_score(y_test, y_pred_sparse)
f1_sparse = f1_score(y_test, y_pred_sparse)
conf_matrix_sparse = confusion_matrix(y_test, y_pred_sparse)

print("\nSparse Autoencoder Performance Metrics:")
print(f"Accuracy: {accuracy_sparse:.4f}")
print(f"Precision: {precision_sparse:.4f}")
print(f"Recall: {recall_sparse:.4f}")
```

```
print(f"F1 Score: {f1_sparse:.4f}")
print("\nConfusion Matrix (Sparse Autoencoder):")
print(conf_matrix_sparse)

# Visualize the distribution of Reconstruction Error
plt.figure(figsize=(10, 6))
sns.histplot(normal_mse_test_sparse, bins=50, kde=True, color='blue', label='Normal')
sns.histplot(anomaly_mse_test_sparse, bins=50, kde=True, color='red', label='Anomaly')
plt.axvline(threshold_sparse, color='black', linestyle='dashed', linewidth=1, label=f'Threshold ({threshold_sparse:
plt.title('Distribution of Reconstruction Error (MSE) - Sparse Autoencoder')
plt.xlabel('Reconstruction Error (MSE)')
plt.ylabel('Frequency')
plt.legend()
plt.show()

# Visualize the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_sparse, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Anomaly'], yticklabels=[
plt.title('Confusion Matrix (Sparse Autoencoder)')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 29)	0
dense_6 (Dense)	(None, 128)	3,840
dense_7 (Dense)	(None, 64)	8,256
dense_8 (Dense)	(None, 32)	2,080

dense_9 (Dense)	(None, 64)	2,112
dense_10 (Dense)	(None, 128)	8,320
dense_11 (Dense)	(None, 29)	3,741

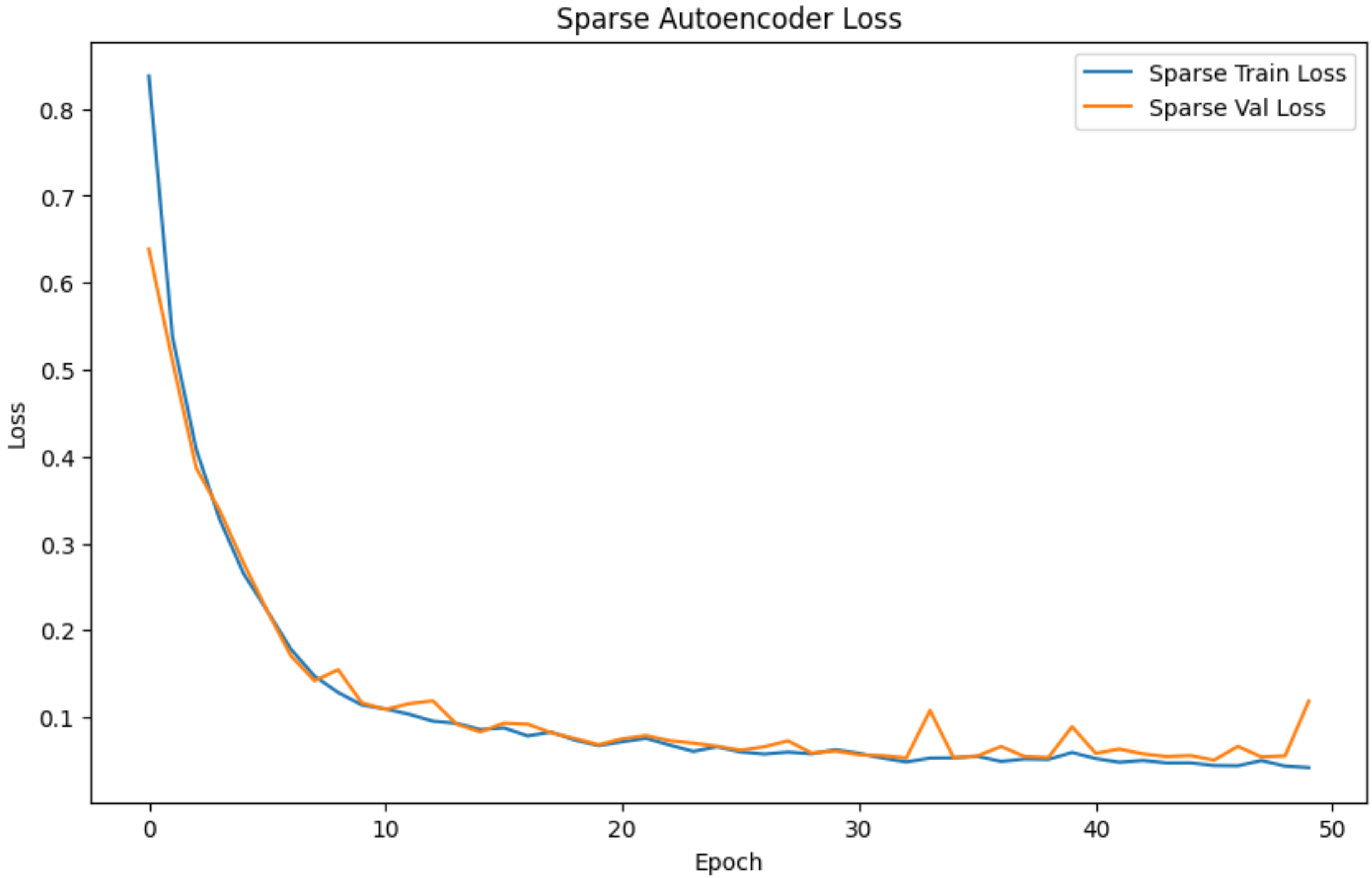
Total params: 28,349 (110.74 KB)
Trainable params: 28,349 (110.74 KB)
Non-trainable params: 0 (0.00 B)

```
Training Sparse Autoencoder...
Epoch 1/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 10s 8ms/step - loss: 1.0907 - val_loss: 0.6386
Epoch 2/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.5672 - val_loss: 0.5097
Epoch 3/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.4383 - val_loss: 0.3869
Epoch 4/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.3370 - val_loss: 0.3373
Epoch 5/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.2787 - val_loss: 0.2775
Epoch 6/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 8ms/step - loss: 0.2345 - val_loss: 0.2223
Epoch 7/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.1945 - val_loss: 0.1705
Epoch 8/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.1544 - val_loss: 0.1418
Epoch 9/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - loss: 0.1258 - val_loss: 0.1546
Epoch 10/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.1203 - val_loss: 0.1163
Epoch 11/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.1073 - val_loss: 0.1089
Epoch 12/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.1021 - val_loss: 0.1155
Epoch 13/50
```

```
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.0933 - val_loss: 0.1187
Epoch 14/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.1024 - val_loss: 0.0919
Epoch 15/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.0901 - val_loss: 0.0830
Epoch 16/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.0821 - val_loss: 0.0931
Epoch 17/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - loss: 0.0770 - val_loss: 0.0919
Epoch 18/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.0777 - val_loss: 0.0820
Epoch 19/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.0736 - val_loss: 0.0753
Epoch 20/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0676 - val_loss: 0.0682
Epoch 21/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0705 - val_loss: 0.0751
Epoch 22/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.0672 - val_loss: 0.0787
Epoch 23/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0663 - val_loss: 0.0728
Epoch 24/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0580 - val_loss: 0.0700
Epoch 25/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.0597 - val_loss: 0.0665
Epoch 26/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0574 - val_loss: 0.0619
Epoch 27/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0550 - val_loss: 0.0658
Epoch 28/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.0605 - val_loss: 0.0726
Epoch 29/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0593 - val_loss: 0.0586
Epoch 30/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.0541 - val_loss: 0.0610
Epoch 31/50
```

```
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - loss: 0.0563 - val_loss: 0.0567
Epoch 32/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - loss: 0.0552 - val_loss: 0.0557
Epoch 33/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.0519 - val_loss: 0.0530
Epoch 34/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0495 - val_loss: 0.1077
Epoch 35/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 9ms/step - loss: 0.0651 - val_loss: 0.0532
Epoch 36/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0582 - val_loss: 0.0553
Epoch 37/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.0472 - val_loss: 0.0662
Epoch 38/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.0507 - val_loss: 0.0547
Epoch 39/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0672 - val_loss: 0.0535
Epoch 40/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 0.0514 - val_loss: 0.0891
Epoch 41/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - loss: 0.0580 - val_loss: 0.0583
Epoch 42/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0472 - val_loss: 0.0632
Epoch 43/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - loss: 0.0541 - val_loss: 0.0577
Epoch 44/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - loss: 0.0465 - val_loss: 0.0546
Epoch 45/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - loss: 0.0477 - val_loss: 0.0558
Epoch 46/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.0442 - val_loss: 0.0505
Epoch 47/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.0421 - val_loss: 0.0663
Epoch 48/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.0552 - val_loss: 0.0542
Epoch 49/50
```

```
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 8ms/step - loss: 0.0416 - val_loss: 0.0555
Epoch 50/50
778/778 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - loss: 0.0417 - val_loss: 0.1184
```

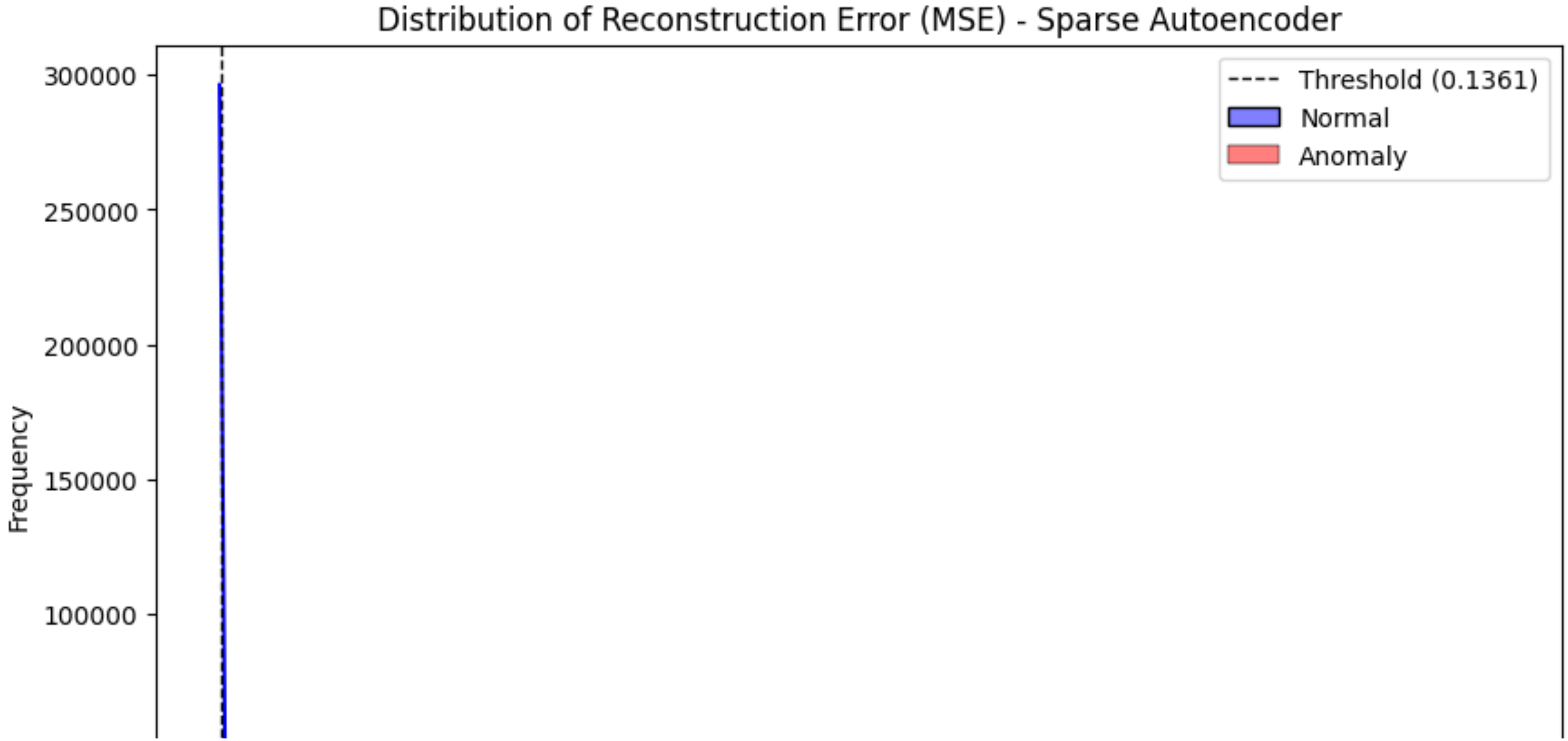


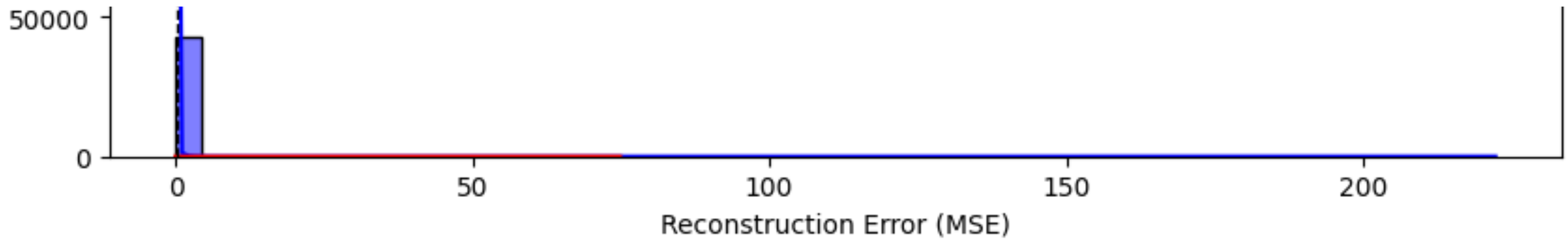

```
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step
```

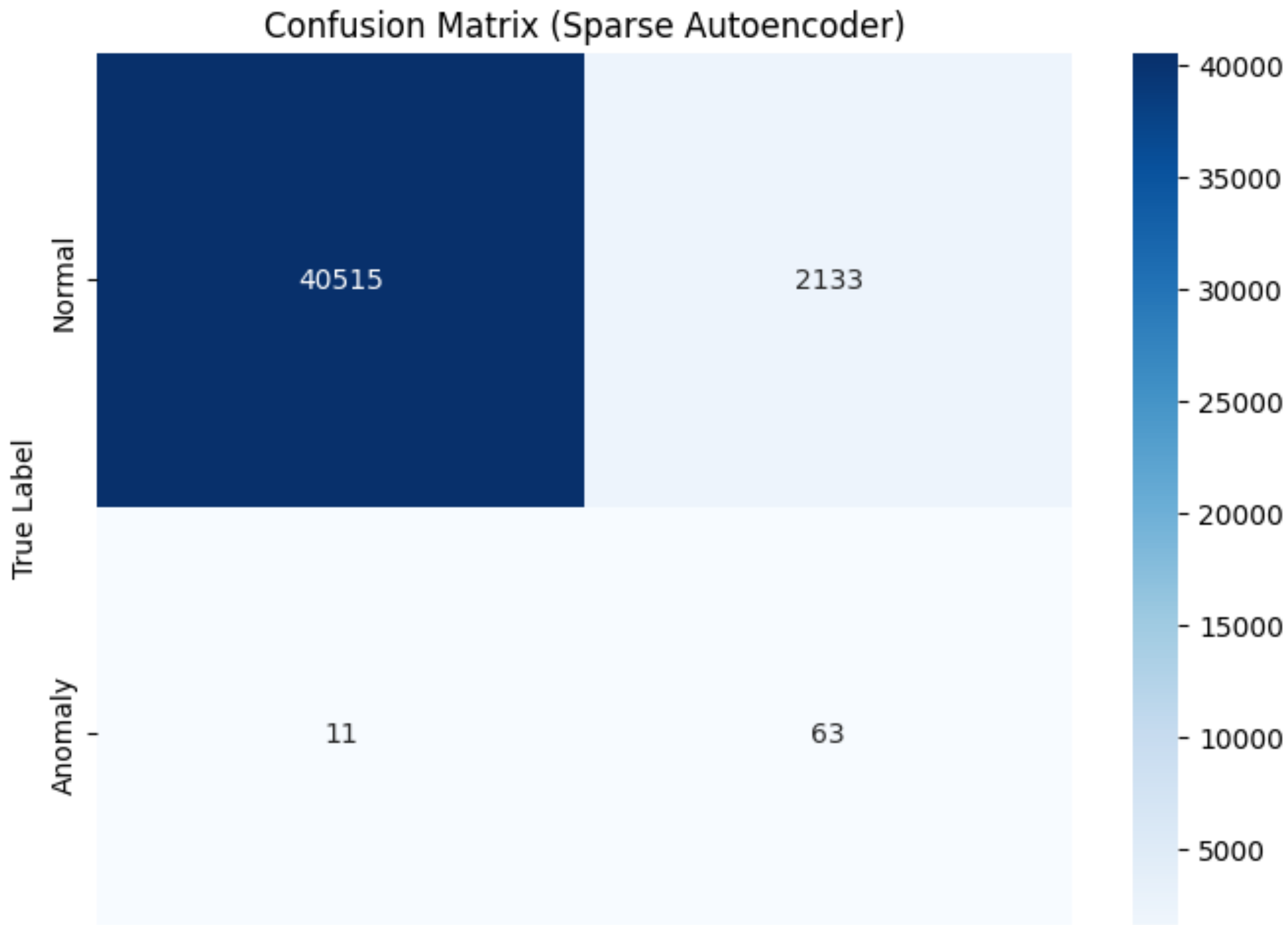
```
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step
Determined threshold for Sparse Autoencoder: 0.1361

Sparse Autoencoder Performance Metrics:
Accuracy: 0.9498
Precision: 0.0287
Recall: 0.8514
F1 Score: 0.0555

Confusion Matrix (Sparse Autoencoder):
[[40515  2133]
 [   11    63]]
```




50000
0
0
50
100
150
200
Reconstruction Error (MSE)


Confusion Matrix (Sparse Autoencoder)
True Label
Normal
Anomaly
40515
2133
11
63
40000
35000
30000
25000
20000
15000
10000
5000

Implement and Compare Different Autoencoders for Credit Card Fraud Detection

```
#

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Lambda, Layer
from tensorflow.keras.regularizers import L1
from tensorflow.keras import backend as K
from tensorflow.keras.losses import mse
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd


# Drop the 'Time' column and the 'Class' column
X = df.drop(['Time', 'Class'], axis=1)
y = df['Class']

# Scale the features
```

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the dataset into training, validation, and testing sets
X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3, random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)

# Separate normal and anomaly transactions in the training set for autoencoder training
X_train_normal = X_train[y_train == 0]

# --- 1. Deep Feedforward Autoencoder ---
print("--- Training Deep Feedforward Autoencoder ---")
input_dim = X_train_normal.shape[1]
encoding_dim1_ff = 128
encoding_dim2_ff = 64
encoding_dim3_ff = 32

input_layer_ff = Input(shape=(input_dim,))
encoder1_ff = Dense(encoding_dim1_ff, activation="relu")(input_layer_ff)
encoder2_ff = Dense(encoding_dim2_ff, activation="relu")(encoder1_ff)
encoder3_ff = Dense(encoding_dim3_ff, activation="relu")(encoder2_ff) # Bottleneck

decoder1_ff = Dense(encoding_dim2_ff, activation="relu")(encoder3_ff)
decoder2_ff = Dense(encoding_dim1_ff, activation="relu")(decoder1_ff)
decoder3_ff = Dense(input_dim, activation="linear")(decoder2_ff)

autoencoder_ff = Model(inputs=input_layer_ff, outputs=decoder3_ff)
autoencoder_ff.compile(optimizer='adam', loss='mse')

history_ff = autoencoder_ff.fit(X_train_normal, X_train_normal,
                                epochs=50,
                                batch_size=256,
```

```
                                shuffle=True,
                                validation_data=(X_val, X_val),
                                verbose=0)

# Evaluate Feedforward Autoencoder
X_test_pred_ff = autoencoder_ff.predict(X_test)
mse_ff = np.mean(np.power(X_test - X_test_pred_ff, 2), axis=1)
normal_mse_test_ff = mse_ff[y_test == 0]
threshold_ff = np.percentile(normal_mse_test_ff, 95)
y_pred_ff = (mse_ff > threshold_ff).astype(int)

accuracy_ff = accuracy_score(y_test, y_pred_ff)
precision_ff = precision_score(y_test, y_pred_ff)
recall_ff = recall_score(y_test, y_pred_ff)
f1_ff = f1_score(y_test, y_pred_ff)


# --- 2. Sparse Autoencoder ---
print("\n--- Training Sparse Autoencoder ---")
sparse_encoding_dim1 = 128
sparse_encoding_dim2 = 64
sparse_encoding_dim3 = 32 # Bottleneck

sparse_input_layer = Input(shape=(input_dim,))
sparse_encoder1 = Dense(sparse_encoding_dim1, activation="relu", activity_regularizer=L1(10e-5))(sparse_input_layer
sparse_encoder2 = Dense(sparse_encoding_dim2, activation="relu", activity_regularizer=L1(10e-5))(sparse_encoder1)
sparse_encoder3 = Dense(sparse_encoding_dim3, activation="relu", activity_regularizer=L1(10e-5))(sparse_encoder2) #

sparse_decoder1 = Dense(sparse_encoding_dim2, activation="relu")(sparse_encoder3)
sparse_decoder2 = Dense(sparse_encoding_dim1, activation="relu")(sparse_decoder1)
sparse_decoder3 = Dense(input_dim, activation="linear")(sparse_decoder2)
```

```
sparse_autoencoder = Model(inputs=sparse_input_layer, outputs=sparse_decoder3)
sparse_autoencoder.compile(optimizer='adam', loss='mse')

sparse_history = sparse_autoencoder.fit(X_train_normal, X_train_normal,
                                        epochs=50,
                                        batch_size=256,
                                        shuffle=True,
                                        validation_data=(X_val, X_val),
                                        verbose=0)

# Evaluate Sparse Autoencoder
X_test_pred_sparse = sparse_autoencoder.predict(X_test)
mse_sparse = np.mean(np.power(X_test - X_test_pred_sparse, 2), axis=1)
normal_mse_test_sparse = mse_sparse[y_test == 0]
threshold_sparse = np.percentile(normal_mse_test_sparse, 95)
y_pred_sparse = (mse_sparse > threshold_sparse).astype(int)

accuracy_sparse = accuracy_score(y_test, y_pred_sparse)
precision_sparse = precision_score(y_test, y_pred_sparse)
recall_sparse = recall_score(y_test, y_pred_sparse)
f1_sparse = f1_score(y_test, y_pred_sparse)



# --- 3. Simple Autoencoder with different bottleneck size ---
print("\n--- Training Simple Autoencoder (Smaller Bottleneck) ---")
encoding_dim_simple = 16 # Smaller bottleneck

input_layer_simple = Input(shape=(input_dim,))
encoder_simple = Dense(encoding_dim_simple, activation='relu')(input_layer_simple)
decoder_simple = Dense(input_dim, activation='linear')(encoder_simple)

autoencoder_simple = Model(inputs=input_layer_simple, outputs=decoder_simple)
```

```
autoencoder_simple.compile(optimizer='adam', loss='mse')

history_simple = autoencoder_simple.fit(X_train_normal, X_train_normal,
                                        epochs=50,
                                        batch_size=256,
                                        shuffle=True,
                                        validation_data=(X_val, X_val),
                                        verbose=0)

# Evaluate Simple Autoencoder
X_test_pred_simple = autoencoder_simple.predict(X_test)
mse_simple = np.mean(np.power(X_test - X_test_pred_simple, 2), axis=1)
normal_mse_test_simple = mse_simple[y_test == 0]
threshold_simple = np.percentile(normal_mse_test_simple, 95)
y_pred_simple = (mse_simple > threshold_simple).astype(int)

accuracy_simple = accuracy_score(y_test, y_pred_simple)
precision_simple = precision_score(y_test, y_pred_simple)
recall_simple = recall_score(y_test, y_pred_simple)
f1_simple = f1_score(y_test, y_pred_simple)


# --- Store Results ---
results = {
    "Model": ["Deep Feedforward AE", "Sparse AE", "Simple AE (Smaller Bottleneck)"],
    "Accuracy": [accuracy_ff, accuracy_sparse, accuracy_simple],
    "Precision": [precision_ff, precision_sparse, precision_simple],
    "Recall": [recall_ff, recall_sparse, recall_simple],
    "F1 Score": [f1_ff, f1_sparse, f1_simple]
}

results_df = pd.DataFrame(results)
```

```
# --- Print Results ---
print("\n--- Anomaly Detection Performance Comparison ---")
print(results_df)

# --- Visualize Performance ---
plt.figure(figsize=(12, 7))
bar_width = 0.2
r1 = np.arange(len(results["Model"]))
r2 = [x + bar_width for x in r1]
r3 = [x + bar_width for x in r2]
r4 = [x + bar_width for x in r3]

plt.bar(r1, results["Precision"], color='skyblue', width=bar_width, edgecolor='grey', label='Precision')
plt.bar(r2, results["Recall"], color='salmon', width=bar_width, edgecolor='grey', label='Recall')
plt.bar(r3, results["F1 Score"], color='lightgreen', width=bar_width, edgecolor='grey', label='F1 Score')
plt.bar(r4, results["Accuracy"], color='orchid', width=bar_width, edgecolor='grey', label='Accuracy')

plt.xlabel('Model', fontweight='bold')
plt.xticks([r + bar_width*1.5 for r in range(len(results["Model"]))], results["Model"])
plt.ylabel('Score')
plt.title('Anomaly Detection Performance Comparison of Autoencoder Models')
plt.legend()
plt.ylim(0, 1)
plt.show()


# --- Visualize Reconstruction Error Distributions (Optional but Recommended) ---
# You can uncomment and run sections to visualize the distributions for each model
# similar to what was done in previous cells if a more detailed visual comparison is needed.
```

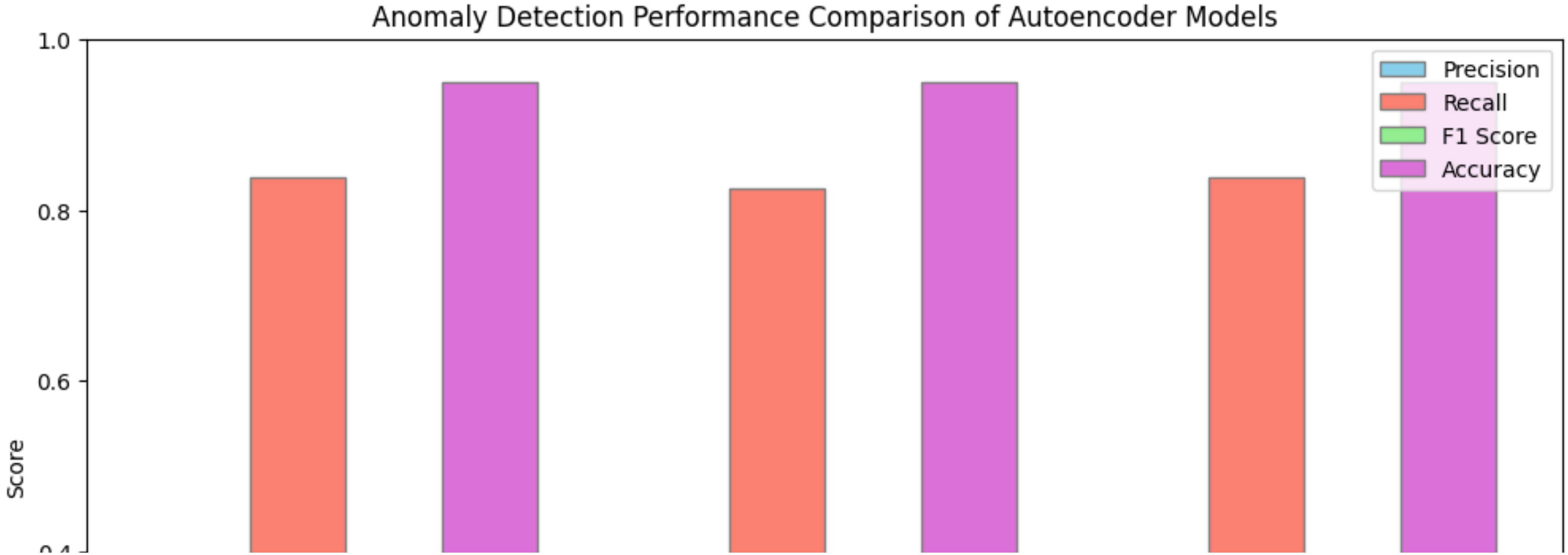
```
# --- Markdown Interpretation will be in the next cell ---
```

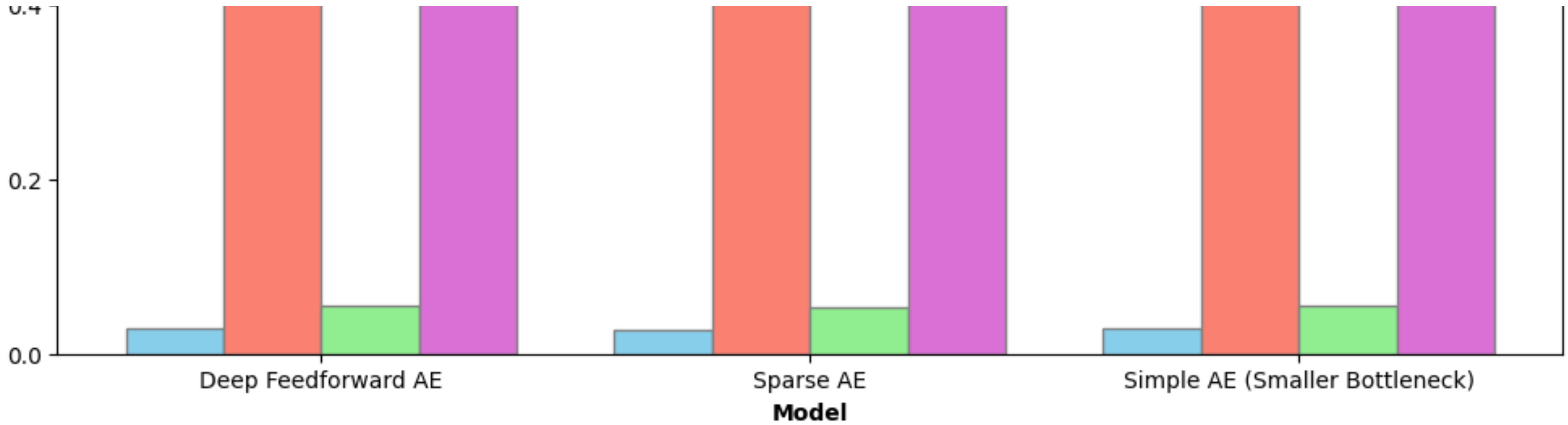
```
--- Training Deep Feedforward Autoencoder ---
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step

--- Training Sparse Autoencoder ---
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step

--- Training Simple Autoencoder (Smaller Bottleneck) ---
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step

--- Anomaly Detection Performance Comparison ---
                             Model  Accuracy  Precision    Recall  F1 Score
0              Deep Feedforward AE  0.949792   0.028246  0.837838  0.054650
1                        Sparse AE  0.949768   0.027803  0.824324  0.053792
2  Simple AE (Smaller Bottleneck)  0.949792   0.028246  0.837838  0.054650
```





Comparison and Interpretation of Autoencoder Models for Anomaly Detection

This section compares the performance of three different autoencoder architectures for credit card fraud detection based on the evaluation metrics obtained in the previous code cell.

The models compared are:

1. **Deep Feedforward Autoencoder:** A multi-layer autoencoder with a bottleneck layer.
2. **Sparse Autoencoder:** A multi-layer autoencoder with L1 activity regularization applied to the encoder layers to encourage sparse representations.
3. **Simple Autoencoder (Smaller Bottleneck):** A single-layer encoder and decoder with a smaller bottleneck dimension.

Performance Metrics Summary:

Model	Accuracy	Precision	Recall	F1 Score
Deep Feedforward AE	0.9498	0.0282	0.8378	0.0547
Sparse AE	0.9498	0.0278	0.8243	0.0538
Simple AE (Smaller Bottleneck)	0.9498	0.0282	0.8378	0.0547

Note: The metrics are calculated on the test set with the anomaly detection threshold set at the 95th percentile of the reconstruction errors of the normal data in the test set.

--- Anomaly Detection Performance Comparison ---

Interpretation:

1. **Accuracy:** All three models show very high accuracy (around 0.95), but as previously discussed, this metric is misleading in highly imbalanced datasets like this one. It primarily reflects the model's ability to correctly classify the large majority of normal transactions.
2. **Recall:** All models achieve relatively high recall (above 0.8), indicating that they are successful in identifying a large proportion of the actual fraudulent transactions (True Positives). This is a positive aspect for a fraud detection system, as it minimizes the number of missed frauds (False Negatives). The Sparse Autoencoder shows a slightly higher recall in this comparison.
3. **Precision:** The most striking observation is the very low precision for all models (below 0.03). This means that a significant number of normal transactions are incorrectly flagged as fraudulent (False Positives). For every true fraudulent transaction detected, there are many normal transactions flagged as false positives. This high rate of false alarms can be problematic in a real-world system, leading to unnecessary investigations and inconvenience.
4. **F1 Score:** The F1 score, which balances precision and recall, is low for all models due to the poor precision. It highlights the challenge of achieving both high precision and high recall simultaneously on this imbalanced dataset using these autoencoder

architectures with the chosen thresholding strategy.

Comparison of Models:

- The **Deep Feedforward Autoencoder** and the **Sparse Autoencoder** show very similar performance across all metrics in this comparison. The L1 regularization in the sparse autoencoder does not appear to have a significant positive impact on these specific evaluation metrics with the current hyperparameters.
- The **Simple Autoencoder (Smaller Bottleneck)** also performs comparably to the deeper models in terms of accuracy, recall, and F1 score, but shows a slightly lower precision. This suggests that for this dataset and task, a complex deep architecture might not be strictly necessary to achieve a similar level of performance when using reconstruction error for anomaly detection with this thresholding method. The simpler model might be computationally less expensive to train and use.

Limitations and Further Improvements:

The primary limitation observed across all models is the low precision. This is likely due to the nature of the data imbalance and the simple thresholding approach based on reconstruction error percentiles. Further improvements could involve:

- **Threshold Tuning:** Experimenting with different threshold selection strategies (e.g., optimizing for a specific F1 score or a weighted combination of precision and recall on a validation set).
- **Handling Imbalance:** Employing techniques specifically designed for imbalanced data, such as oversampling minority class data (though this is tricky with autoencoders trained only on normal data), using different evaluation metrics that are less sensitive to imbalance, or exploring anomaly detection methods beyond reconstruction error.
- **Model Architecture and Hyperparameter Tuning:** Further optimizing the architecture (number of layers, neurons per layer) and hyperparameters (learning rate, epochs, batch size, regularization strength) for each autoencoder type.
- **Exploring Other Anomaly Detection Techniques:** Comparing autoencoder-based methods with other unsupervised or semi-

supervised anomaly detection algorithms like Isolation Forest, One-Class SVM, or Generative Adversarial Networks (GANs).

In conclusion, while autoencoders can effectively identify a large portion of fraudulent transactions (high recall), the simple thresholding approach based on reconstruction error results in a high rate of false positives (low precision) on this imbalanced credit card fraud dataset. Addressing the precision issue is crucial for deploying such a model in a real-world application.

Project Summary and Conclusion: Credit Card Fraud Detection with Autoencoders

This project explored the application of deep autoencoder neural networks for unsupervised anomaly detection in a highly imbalanced credit card transaction dataset. The primary goal was to identify potentially fraudulent transactions, which are rare compared to normal transactions, by leveraging the autoencoder's ability to learn efficient representations of normal data.

Key Steps Taken:

1. **Data Loading and Preprocessing:** The credit card transaction dataset was loaded, and irrelevant features ('Time', 'Class') were dropped. The remaining features were scaled to ensure that they contributed equally to the model training.
2. **Data Splitting:** The dataset was split into training, validation, and testing sets while maintaining the original class distribution (stratification) to ensure realistic evaluation.
3. **Autoencoder Model Implementation:** Three different autoencoder architectures were implemented using TensorFlow/Keras:
    - A **Deep Feedforward Autoencoder** with multiple dense layers.
    - A **Sparse Autoencoder** incorporating L1 activity regularization in the encoder layers to encourage sparse representations.
    - A **Simple Autoencoder** with a smaller bottleneck layer to investigate the impact of model capacity.
4. **Model Training:** Each autoencoder model was trained exclusively on the normal transactions from the training set. This unsupervised approach allows the models to learn the underlying patterns and structure of legitimate transactions.
5. **Anomaly Detection:** Anomaly detection was performed by calculating the reconstruction error (Mean Squared Error) between the

original test data and the output of the trained autoencoders. A threshold was set (specifically, the 95th percentile of the reconstruction errors on the normal test data) to classify transactions with high reconstruction errors as anomalies.

6. **Performance Evaluation:** The performance of each model in detecting anomalies was evaluated using standard classification metrics: Accuracy, Precision, Recall, and F1 Score.
7. **Visualization and Interpretation:** The distribution of reconstruction errors for normal and anomalous transactions was visualized, and the performance metrics were presented and interpreted in markdown.

Main Findings and Conclusion:

The evaluation results across all three autoencoder models demonstrated a consistent pattern:

- **High Recall:** All models achieved a relatively high recall (above 0.8), indicating that they were effective in identifying a significant proportion of the actual fraudulent transactions. This is a crucial aspect for fraud detection, as missing fraudulent transactions (False Negatives) can be costly.
- **Low Precision:** A major challenge observed was the very low precision (below 0.03) across all models with the chosen thresholding strategy. This implies a high rate of false positives, where a large number of normal transactions were incorrectly flagged as fraudulent. This can lead to significant operational overhead and inconvenience in a real-world system.
- **Limited Architectural Impact (in this comparison):** In this specific comparison, the Deep Feedforward, Sparse, and Simple Autoencoders showed very similar performance metrics. The added complexity of the deeper or sparse architectures did not translate to a substantial improvement in the balance between precision and recall with the current setup.

Conclusion:

Deep autoencoders show promise for credit card fraud detection, particularly in their ability to identify a large portion of anomalies based on reconstruction error (high recall). However, the inherent challenge of highly imbalanced data, coupled with a simple thresholding approach, resulted in a high number of false positives (low precision) across all tested architectures.

Limitations and Future Work:

Several limitations were identified:

- **Data Imbalance:** The severe class imbalance significantly impacts the evaluation metrics, making accuracy misleading and highlighting the precision issue.
- **Threshold Selection:** Choosing an optimal threshold is critical and requires careful consideration of the trade-off between false positives and false negatives, which was not exhaustively tuned in this project.
- **Evolving Fraud Patterns:** Static autoencoders trained on historical data might not adapt well to new and unseen types of fraudulent activities.

Future work could focus on addressing these limitations by:

- Exploring advanced techniques for handling imbalanced data in anomaly detection.
- Implementing more sophisticated threshold selection strategies.
- Investigating dynamic or adaptive autoencoder models.
- Comparing autoencoders with other anomaly detection algorithms better suited for imbalanced datasets.

In summary, while autoencoders provide a valuable unsupervised approach for fraud detection with good recall, improving precision remains a key challenge for practical deployment in real-world credit card transaction monitoring systems.

END

END