Deep Learning Project: Credit Card Fraud Detection

Objective

The main objective of this analysis is to develop a deep learning model to accurately detect fraudulent credit card transactions. This task focuses on binary classification using supervised learning techniques, with an emphasis on addressing the significant class imbalance in the dataset. Insights from this analysis can help credit card companies reduce financial losses and enhance customer satisfaction by identifying fraudulent transactions efficiently.

Dataset Description

The dataset comes from (https://www.kaggle.com/datasets/joebeachcapital/credit-card-fraud) and contains credit card transactions made by European cardholders in September 2013. It includes a total of 284,807 transactions, of which only 492 (0.172%) are fraudulent, highlighting a significant class imbalance. Features V1 to V28 are Principal Component Analysis (PCA) transformations of the original features, with 'Time' and 'Amount' being untransformed. The 'Class' column is the target variable, where 0 represents non-fraud and 1 represents fraud.

In [65]:

```
# Load the dataset
import pandas as pd
import numpy as np

file_path = r'C:\Users\gcantarella\Desktop\Deep Learning Final Project\creditcar
data = pd.read_csv(file_path)

# Summarize dataset structure
print(data.info())
# Display a preview of the dataset
print(data.head())
# Check for missing values
print(data.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64  
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
None
   Time        V1        V2        V3        V4        V5        V6        V7  \
0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599   
1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803   
2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461   
3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609   
4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941   

         V8        V9  ...       V21       V22       V23       V24       V25  \
0  0.098698  0.363787  ... -0.018307  0.277838 -0.110474  0.066928  0.128539   
1  0.085102 -0.255425  ... -0.225775 -0.638672  0.101288 -0.339846  0.167170   
2  0.247676 -1.514654  ...  0.247998  0.771679  0.909412 -0.689281 -0.327642   
3  0.377436 -1.387024  ... -0.108300  0.005274 -0.190321 -1.175575  0.647376   
4 -0.270533  0.817739  ... -0.009431  0.798278 -0.137458  0.141267 -0.206010   

        V26       V27       V28  Amount  Class  
0 -0.189115  0.133558 -0.021053  149.62      0  
1  0.125895 -0.008983  0.014724    2.69      0  
2 -0.139097 -0.055353 -0.059752  378.66      0  
3 -0.221929  0.062723  0.061458  123.50      0  
4  0.502292  0.219422  0.215153   69.99      0  
```

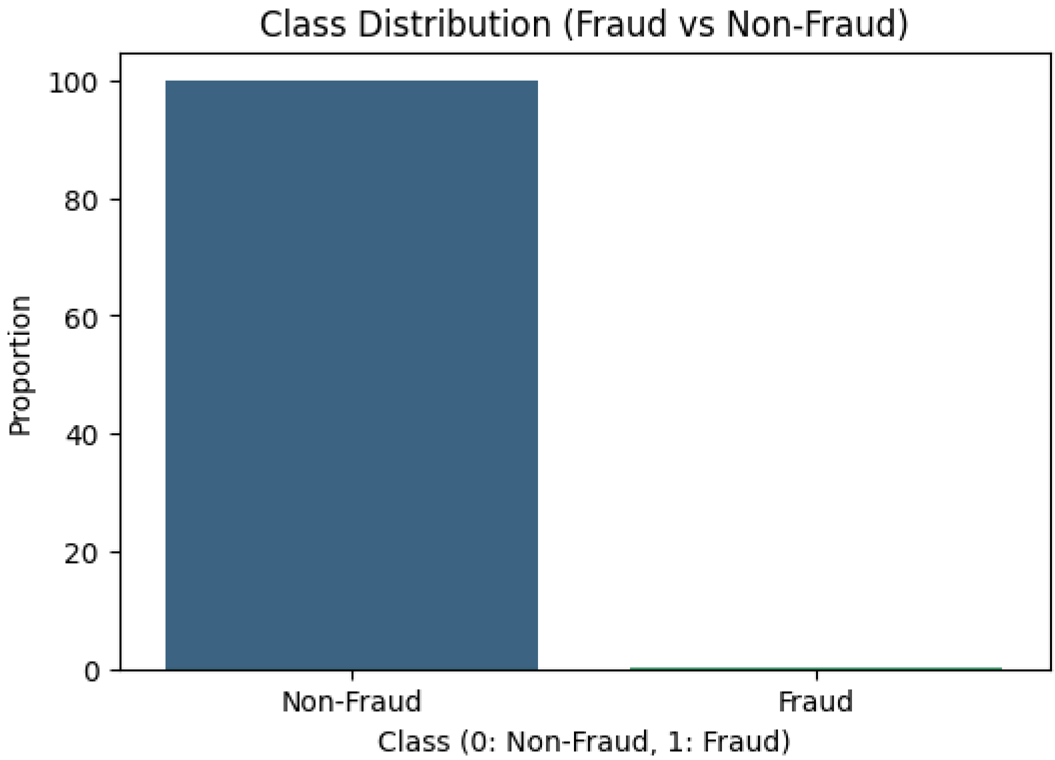
```
[5 rows x 31 columns]
Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

Exploratory Data Analysis (EDA)

In [30]:

```
import matplotlib.pyplot as plt
import seaborn as sns

# Check the class distribution
class_distribution = data['Class'].value_counts(normalize=True) * 100
print(f"Class distribution: \n{class_distribution}")

# Visualize the distribution of the target variable
plt.figure(figsize=(6, 4))
sns.barplot(x=class_distribution.index, y=class_distribution.values, palette="vi
plt.title('Class Distribution (Fraud vs Non-Fraud)')
plt.ylabel('Proportion')
plt.xlabel('Class (0: Non-Fraud, 1: Fraud)')
plt.xticks([0, 1], ['Non-Fraud', 'Fraud'])
plt.legend([],[], frameon=False)  # Removes redundant legend
plt.show()

# Describe the dataset to understand feature statistics
data_description = data.describe()

class_distribution
data_description.iloc[:, :8]
```

```
Class distribution:
Class
0    99.827251
1     0.172749
Name: proportion, dtype: float64
```



Out[30]:

	Time	V1	V2	V3	V4	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.8
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.6
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.3
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.1
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.9
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.4
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.1
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.4

Data Preprocessing

The following steps were undertaken for preprocessing the dataset:

- Normalized 'Time' and 'Amount' using StandardScaler to bring all features onto a similar scale.
- Split the dataset into training (80%) and testing (20%) sets, with stratified sampling to maintain class balance.

In [32]:
```
# Normalize 'Amount' and 'Time' features
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
data[['Time', 'Amount']] = scaler.fit_transform(data[['Time', 'Amount']])

# Split data into features and target
X = data.drop(columns=['Class'])
y = data['Class']

# Train-test split
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

# Confirm shapes of the datasets
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[32]:
```
((227845, 30), (56962, 30), (227845,), (56962,))
```

Model Development

Three deep learning models were developed and evaluated for this task:

1. A simple neural network with two dense layers and a dropout layer.
2. A more complex neural network with additional layers and nodes.
3. A CNN will be implemented.

Simple Neural Network

In [52]:
```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import Sequential, Input
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Preprocessing
# Assuming `data` is a pandas DataFrame containing the dataset
# Separate features and target variable
X = data.drop(columns=['Class'])
y = data['Class']

# Normalize 'Time' and 'Amount' features
scaler = StandardScaler()
X[['Time', 'Amount']] = scaler.fit_transform(X[['Time', 'Amount']])

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Define the simple neural network model
def build_simple_model(input_shape):
```

```
    model = Sequential([
        Input(shape=(input_shape,)),  # Use Input layer to specify the input sha
        Dense(16, activation='relu'),
        Dropout(0.5),
        Dense(8, activation='relu'),
        Dense(1, activation='sigmoid')  # Output layer for binary classification
    ])
    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['AUC']
    )
    return model

# Build and compile the model
simple_model = build_simple_model(X_train.shape[1])

# Early stopping to prevent overfitting
early_stopping = EarlyStopping(
    monitor='val_loss',  # Monitor validation loss
    patience=8,          # Number of epochs with no improvement
    restore_best_weights=True  # Restore the best model weights
)

# Train the model
history = simple_model.fit(
    X_train, y_train,
    validation_split=0.2,  # Use 20% of the training data for validation
    epochs=20,             # Train for a maximum of 20 epochs
    batch_size=512,        # Use a batch size of 512
    callbacks=[early_stopping],  # Early stopping callback
    verbose=1              # Print training progress
)
```

```
Epoch 1/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - AUC: 0.4319 - loss: 0.3550 - val_AUC:
0.8596 - val_loss: 0.0118
Epoch 2/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.8167 - loss: 0.0262 - val_AUC:
0.9150 - val_loss: 0.0053
Epoch 3/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9097 - loss: 0.0129 - val_AUC:
0.9154 - val_loss: 0.0047
Epoch 4/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9097 - loss: 0.0091 - val_AUC:
0.9215 - val_loss: 0.0045
Epoch 5/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9300 - loss: 0.0071 - val_AUC:
0.9274 - val_loss: 0.0043
Epoch 6/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9344 - loss: 0.0063 - val_AUC:
0.9333 - val_loss: 0.0042
Epoch 7/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9244 - loss: 0.0061 - val_AUC:
0.9333 - val_loss: 0.0042
Epoch 8/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9455 - loss: 0.0062 - val_AUC:
0.9334 - val_loss: 0.0042
Epoch 9/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9249 - loss: 0.0059 - val_AUC:
0.9334 - val_loss: 0.0041
Epoch 10/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9428 - loss: 0.0058 - val_AUC:
0.9274 - val_loss: 0.0041
Epoch 11/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - AUC: 0.9389 - loss: 0.0055 - val_AUC:
0.9273 - val_loss: 0.0043
Epoch 12/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9371 - loss: 0.0050 - val_AUC:
0.9273 - val_loss: 0.0041
Epoch 13/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9524 - loss: 0.0045 - val_AUC:
0.9273 - val_loss: 0.0039
Epoch 14/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9525 - loss: 0.0051 - val_AUC:
0.9273 - val_loss: 0.0040
Epoch 15/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9206 - loss: 0.0067 - val_AUC:
0.9333 - val_loss: 0.0041
Epoch 16/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9281 - loss: 0.0050 - val_AUC:
0.9333 - val_loss: 0.0038
Epoch 17/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - AUC: 0.9577 - loss: 0.0044 - val_AUC:
0.9332 - val_loss: 0.0038
Epoch 18/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - AUC: 0.9458 - loss: 0.0051 - val_AUC:
0.9333 - val_loss: 0.0038
Epoch 19/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9582 - loss: 0.0042 - val_AUC:
0.9333 - val_loss: 0.0038
Epoch 20/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9488 - loss: 0.0050 - val_AUC:
0.9334 - val_loss: 0.0041
```

More complex Neural Network

In [41]:
```python
# Define a more complex neural network
def build_complex_model(input_shape):
    model = Sequential([
        Input(shape=(input_shape,)),
        Dense(64, activation='relu'),
        Dropout(0.4),
        Dense(32, activation='relu'),
        Dropout(0.3),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['AUC'])
    return model

# Build and compile the complex model
complex_model = build_complex_model(X_train.shape[1])

# Train the complex model
history_complex = complex_model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=20,
    batch_size=512,
    callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,
    verbose=1
)
```

```
Epoch 1/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - AUC: 0.4348 - loss: 0.1570 - val_AUC: 0.9330 - val_loss: 0.0044
Epoch 2/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.8999 - loss: 0.0064 - val_AUC: 0.9272 - val_loss: 0.0038
Epoch 3/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9315 - loss: 0.0056 - val_AUC: 0.9333 - val_loss: 0.0037
Epoch 4/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9454 - loss: 0.0038 - val_AUC: 0.9333 - val_loss: 0.0035
Epoch 5/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9340 - loss: 0.0041 - val_AUC: 0.9334 - val_loss: 0.0034
Epoch 6/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9557 - loss: 0.0035 - val_AUC: 0.9393 - val_loss: 0.0033
Epoch 7/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9481 - loss: 0.0034 - val_AUC: 0.9334 - val_loss: 0.0033
Epoch 8/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9549 - loss: 0.0035 - val_AUC: 0.9393 - val_loss: 0.0033
Epoch 9/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9475 - loss: 0.0028 - val_AUC: 0.9334 - val_loss: 0.0033
Epoch 10/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - AUC: 0.9462 - loss: 0.0032 - val_AUC: 0.9334 - val_loss: 0.0032
Epoch 11/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9618 - loss: 0.0029 - val_AUC: 0.9452 - val_loss: 0.0031
Epoch 12/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9606 - loss: 0.0030 - val_AUC: 0.9392 - val_loss: 0.0030
Epoch 13/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9451 - loss: 0.0029 - val_AUC: 0.9453 - val_loss: 0.0031
Epoch 14/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9609 - loss: 0.0032 - val_AUC: 0.9393 - val_loss: 0.0032
Epoch 15/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9638 - loss: 0.0031 - val_AUC: 0.9334 - val_loss: 0.0033
Epoch 16/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9615 - loss: 0.0025 - val_AUC: 0.9334 - val_loss: 0.0033
Epoch 17/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9473 - loss: 0.0025 - val_AUC: 0.9394 - val_loss: 0.0033
```

CNN Implementation

We can try using a **Convolutional Neural Network (CNN)** . CNNs are particularly effective at identifying spatial patterns and interactions between features. To apply a CNN to tabular data, we need to reshape the data into a format that mimics an image structure.

Here's how to proceed:

1. Reshape the Input Data
2. Convert the flat feature vectors into a 2D matrix. For example, the 28 PCA features + 2 others could be reshaped into a 6x5 or similar grid.
3. Build a CNN Architecture
4. Add convolutional layers to extract feature interactions.
5. Use pooling layers to reduce dimensions and avoid overfitting.
6. Train and Evaluate. Train the CNN model on the reshaped data.
7. Evaluate using metrics like AUC to compare its performance with the fully connected model.

In [59]:
```
from tensorflow.keras.layers import Conv2D, Flatten, MaxPooling2D, Input
from sklearn.preprocessing import StandardScaler

# Reshape data for CNN
def reshape_for_cnn(X):
    # Assuming 30 features can be reshaped into 5x6 grid
    return X.values.reshape(-1, 5, 6, 1)  # Add channel dimension for grayscale

# Reshape training and testing sets
X_train_cnn = reshape_for_cnn(X_train)
X_test_cnn = reshape_for_cnn(X_test)

# Define a simple CNN model
def build_cnn_model(input_shape):
    model = Sequential([
        Input(shape=input_shape),
        Conv2D(32, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Dropout(0.5),
        Flatten(),
        Dense(64, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['AUC'])
    return model

# Build and compile the CNN model
cnn_model = build_cnn_model((5, 6, 1))

# Train the CNN model
history_cnn = cnn_model.fit(
    X_train_cnn, y_train,
    validation_split=0.2,
    epochs=20,
    batch_size=512,
    callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=8,
    verbose=1
)
```

```
Epoch 1/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - AUC: 0.2867 - loss: 0.1324 - val_AUC:
0.8821 - val_loss: 0.0075
Epoch 2/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - AUC: 0.8214 - loss: 0.0098 - val_AUC:
0.8969 - val_loss: 0.0053
Epoch 3/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - AUC: 0.8788 - loss: 0.0064 - val_AUC:
0.9084 - val_loss: 0.0047
Epoch 4/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - AUC: 0.8971 - loss: 0.0053 - val_AUC:
0.9092 - val_loss: 0.0044
Epoch 5/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - AUC: 0.9271 - loss: 0.0045 - val_AUC:
0.9209 - val_loss: 0.0039
Epoch 6/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - AUC: 0.9135 - loss: 0.0047 - val_AUC:
0.9093 - val_loss: 0.0042
Epoch 7/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - AUC: 0.9038 - loss: 0.0050 - val_AUC:
0.9213 - val_loss: 0.0040
Epoch 8/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - AUC: 0.9061 - loss: 0.0047 - val_AUC:
0.9215 - val_loss: 0.0041
Epoch 9/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - AUC: 0.9394 - loss: 0.0041 - val_AUC:
0.9214 - val_loss: 0.0038
Epoch 10/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - AUC: 0.9256 - loss: 0.0038 - val_AUC:
0.9214 - val_loss: 0.0037
Epoch 11/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - AUC: 0.9226 - loss: 0.0044 - val_AUC:
0.9215 - val_loss: 0.0038
Epoch 12/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - AUC: 0.9177 - loss: 0.0038 - val_AUC:
0.9211 - val_loss: 0.0036
Epoch 13/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - AUC: 0.9124 - loss: 0.0042 - val_AUC:
0.9213 - val_loss: 0.0035
Epoch 14/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9263 - loss: 0.0033 - val_AUC:
0.9215 - val_loss: 0.0036
Epoch 15/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - AUC: 0.9181 - loss: 0.0040 - val_AUC:
0.9214 - val_loss: 0.0036
Epoch 16/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - AUC: 0.9193 - loss: 0.0033 - val_AUC:
0.9215 - val_loss: 0.0036
Epoch 17/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - AUC: 0.9490 - loss: 0.0028 - val_AUC:
0.9272 - val_loss: 0.0035
Epoch 18/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - AUC: 0.9417 - loss: 0.0033 - val_AUC:
0.9271 - val_loss: 0.0034
Epoch 19/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - AUC: 0.9495 - loss: 0.0030 - val_AUC:
0.9214 - val_loss: 0.0034
Epoch 20/20
357/357 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - AUC: 0.9328 - loss: 0.0033 - val_AUC:
0.9275 - val_loss: 0.0034
```

Key Findings and Insights

Model evaluation

In [53]:
```
# Evaluate the simple model
model_eval = simple_model.evaluate(X_test, y_test, verbose=1)
print(f"Test AUC (Simple Model): {model_eval[1]}")
```

```
1781/1781 ━━━━━━━━━━━━━━━━━━━━ 2s 850us/step - AUC: 0.9509 - loss: 0.0026
Test AUC (Simple Model): 0.9484700560569763
```

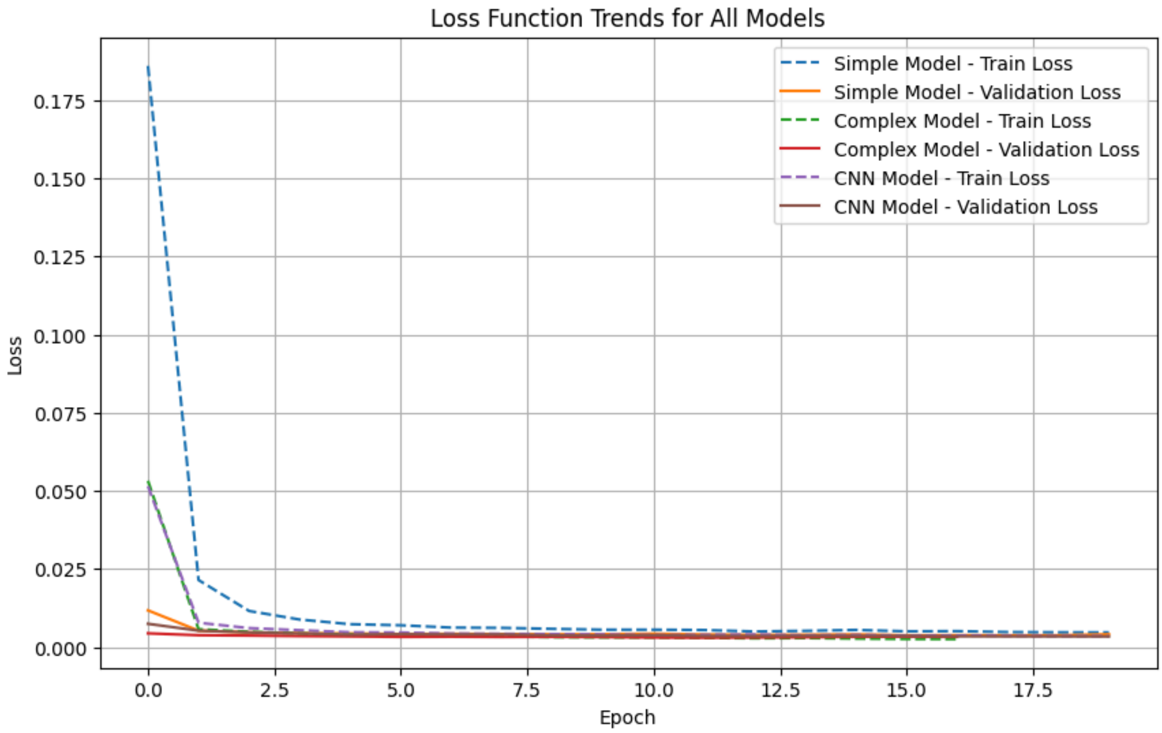
In [56]:
```
# Evaluate the complex model
complex_eval = complex_model.evaluate(X_test, y_test, verbose=1)
print(f"Test AUC (Complex Model): {complex_eval[1]}")
```

```
1781/1781 ━━━━━━━━━━━━━━━━━━━━ 2s 878us/step - AUC: 0.9511 - loss: 0.0022
Test AUC (Complex Model): 0.9536031484603882
```

In [60]:
```
# Evaluate the CNN model
cnn_eval = cnn_model.evaluate(X_test_cnn, y_test, verbose=1)
print(f"Test AUC: {cnn_eval[1]}")
```

```
1781/1781 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - AUC: 0.9507 - loss: 0.0021
Test AUC: 0.9436067342758179
```

In [63]:
```
import matplotlib.pyplot as plt

# Plot loss trends for the three models
plt.figure(figsize=(10, 6))

# Simple Model
plt.plot(history.history['loss'], label='Simple Model - Train Loss', linestyle='
plt.plot(history.history['val_loss'], label='Simple Model - Validation Loss')

# Complex Model
plt.plot(history_complex.history['loss'], label='Complex Model - Train Loss', li
plt.plot(history_complex.history['val_loss'], label='Complex Model - Validation

# CNN Model
plt.plot(history_cnn.history['loss'], label='CNN Model - Train Loss', linestyle=
plt.plot(history_cnn.history['val_loss'], label='CNN Model - Validation Loss')

# Add plot details
plt.title('Loss Function Trends for All Models')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```



Final Comments

The Complex Model (AUC: 0.9536) performed slightly better than the Simple Model (AUC: 0.94847), demonstrating that adding more layers and nodes helped capture more intricate patterns in the data.

The CNN Model (AUC: 0.9436), while slightly lower in performance, still shows competitive results. It is likely effective at capturing feature interactions due to its convolutional layers, but the spatial structure used in reshaping features may not have been optimal.

However, despite the high AUC, the class imbalance remains a significant challenge. The model's performance on metrics like recall for the minority (fraud) class should be closely monitored to ensure fraud detection is effective.

The performance improvement is incremental. Testing advanced techniques like hyperparameter optimization, deeper CNNs, or ensemble methods could provide more gains.

Next Steps

Further improvements could include:

1. **Using SMOTE (Synthetic Minority Oversampling Technique)** or similar methods to address the class imbalance.
2. **Hyperparameter tuning** to further optimize the model's performance.
3. Experimenting with **advanced architectures**, such as **Recurrent Neural Networks (RNNs)** to incorporate time-dependency in the data.

By addressing these areas, the model's ability to detect fraudulent transactions could be further enhanced, potentially improving both recall and precision for the minority class.