DEEP LEARNING: Deep neural Network, Gradient Descent, Backpropagation, Activation function, Keras, Convolutionals, Transfer Learning, RNN, Autoencoders, GANS, Reinforcement Learning

Para realizar pruebas de **Deep Learning** con diversas técnicas y enfoques, como los que mencionas (Deep Neural Networks, Gradient Descent, Backpropagation, etc.), te proporcionaré una guía paso a paso.

Empezaremos con un enfoque básico usando **Deep Neural Networks (DNN)** con **Keras**, y luego podemos explorar técnicas más avanzadas como **Convolutional Neural Networks (CNNs)**, **Recurrent Neural Networks (RNNs)**, **Autoencoders**, **GANs**, y más.

Paso 1: **Deep Neural Network (DNN) con Keras**

Empezaremos creando un modelo simple de DNN utilizando la librería **Keras** con **TensorFlow** como backend. Aplicaremos **Gradient Descent** y utilizaremos **Backpropagation** para entrenar la red.

Código básico para un modelo DNN con Keras:

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Cargar y preparar los datos
```

```
# Usaremos el dataset actual que ya tienes disponible

data = df.copy()


# Separar las características y la variable objetivo

X = data.drop(columns=['Class'])

y = data['Class']


# Dividir los datos en entrenamiento y prueba

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Escalar los datos

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)


# Crear el modelo de DNN

model = Sequential()

model.add(Dense(64, input_dim=X_train_scaled.shape[1], activation='relu'))

model.add(Dense(32, activation='relu'))

model.add(Dense(1, activation='sigmoid'))  # Salida para clasificación binaria


# Compilar el modelo

model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy',
metrics=['accuracy'])


# Entrenar el modelo

history = model.fit(X_train_scaled, y_train, validation_data=(X_test_scaled, y_test),
epochs=50, batch_size=32)
```

```
# Evaluar el modelo
loss, accuracy = model.evaluate(X_test_scaled, y_test)
print(f"Exactitud en el conjunto de prueba: {accuracy * 100:.2f}%")

# Graficar el desempeño durante el entrenamiento
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'], label='Exactitud en entrenamiento')
plt.plot(history.history['val_accuracy'], label='Exactitud en validación')
plt.title('Exactitud del Modelo DNN')
plt.xlabel('Época')
plt.ylabel('Exactitud')
plt.legend()
plt.show()
```

Explicación:

- **Modelo DNN**: Usamos una red simple con dos capas densas de 64 y 32 neuronas, respectivamente.
- **Activation Functions**: Se utiliza `relu` en las capas ocultas y `sigmoid` en la capa de salida para la clasificación binaria.
- **Optimizer**: Aplicamos el **Adam** optimizer, que es una versión mejorada de **Gradient Descent**.
- **Loss Function**: Usamos **binary_crossentropy** para tareas de clasificación binaria.

Paso 2: **Convolutional Neural Networks (CNNs)**

Las **CNNs** son más adecuadas para datos con estructuras espaciales, como imágenes, pero pueden ser útiles en otras áreas también, aplicando convoluciones sobre las características transformadas.

Código básico para CNNs:

```python
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten

# Crear el modelo CNN
model_cnn = Sequential()
model_cnn.add(Conv1D(32, kernel_size=3, activation='relu', input_shape=(X_train_scaled.shape[1], 1)))
model_cnn.add(MaxPooling1D(pool_size=2))
model_cnn.add(Flatten())
model_cnn.add(Dense(64, activation='relu'))
model_cnn.add(Dense(1, activation='sigmoid'))

# Compilar el modelo
model_cnn.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

# Redimensionar los datos para que funcionen con Conv1D
X_train_cnn = X_train_scaled.reshape(X_train_scaled.shape[0], X_train_scaled.shape[1], 1)
X_test_cnn = X_test_scaled.reshape(X_test_scaled.shape[0], X_test_scaled.shape[1], 1)

# Entrenar el modelo CNN
history_cnn = model_cnn.fit(X_train_cnn, y_train, validation_data=(X_test_cnn, y_test), epochs=50, batch_size=32)
```

```

### Paso 3: **Recurrent Neural Networks (RNNs)**

Si tienes datos secuenciales (como series temporales), puedes aplicar una **RNN** o **LSTM** (Long Short-Term Memory).

#### Código básico para RNN:

```python

from tensorflow.keras.layers import SimpleRNN

# Crear el modelo RNN

model_rnn = Sequential()

model_rnn.add(SimpleRNN(32, activation='relu',
input_shape=(X_train_scaled.shape[1], 1)))

model_rnn.add(Dense(1, activation='sigmoid'))

# Compilar el modelo

model_rnn.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy',
metrics=['accuracy'])

# Redimensionar los datos para que funcionen con RNN

X_train_rnn = X_train_scaled.reshape(X_train_scaled.shape[0],
X_train_scaled.shape[1], 1)

X_test_rnn = X_test_scaled.reshape(X_test_scaled.shape[0], X_test_scaled.shape[1], 1)

# Entrenar el modelo RNN

history_rnn = model_rnn.fit(X_train_rnn, y_train, validation_data=(X_test_rnn, y_test),
epochs=50, batch_size=32)
```
```

Paso 4: **Transfer Learning**

El **Transfer Learning** se utiliza principalmente en visión por computadora, usando modelos preentrenados como **ResNet**, **VGG16**, o **Inception** y ajustándolos a tus propios datos.

Paso 5: **Autoencoders** (para detección de anomalías o reducción de dimensionalidad):

```python
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# Crear el modelo Autoencoder
input_dim = X_train_scaled.shape[1]
input_layer = Input(shape=(input_dim,))
encoded = Dense(32, activation='relu')(input_layer)
decoded = Dense(input_dim, activation='sigmoid')(encoded)

# Modelo completo del Autoencoder
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss='mse')

# Entrenar el Autoencoder
autoencoder.fit(X_train_scaled, X_train_scaled, epochs=50, batch_size=32, validation_data=(X_test_scaled, X_test_scaled))
```

Paso 6: **GANs (Generative Adversarial Networks)**

Las **GANs** son más complejas y se utilizan principalmente para la generación de datos sintéticos.

Paso 7: **Reinforcement Learning**

Es un enfoque más avanzado que implica la toma de decisiones secuenciales optimizadas para maximizar una recompensa acumulada.

Próximos Pasos:

1. **Define el modelo específico** que deseas probar: DNN, CNN, RNN, Autoencoder, o algún otro.
2. **Ajusta los hiperparámetros** de acuerdo con los datos.
3. **Evalúa los modelos** y ajusta el rendimiento según sea necesario.

Avísame con cuál de estos enfoques te gustaría empezar para implementar las pruebas y ajustes necesarios.

¡Perfecto! Para comenzar con el **Paso 1** y ajustar el código a tu dataset, lo adaptaremos para usar tu conjunto de datos actual, que incluye la columna `Class` para la clasificación de fraudes y las demás características para predecir.

Aquí tienes el código ajustado al dataset en el que has estado trabajando:

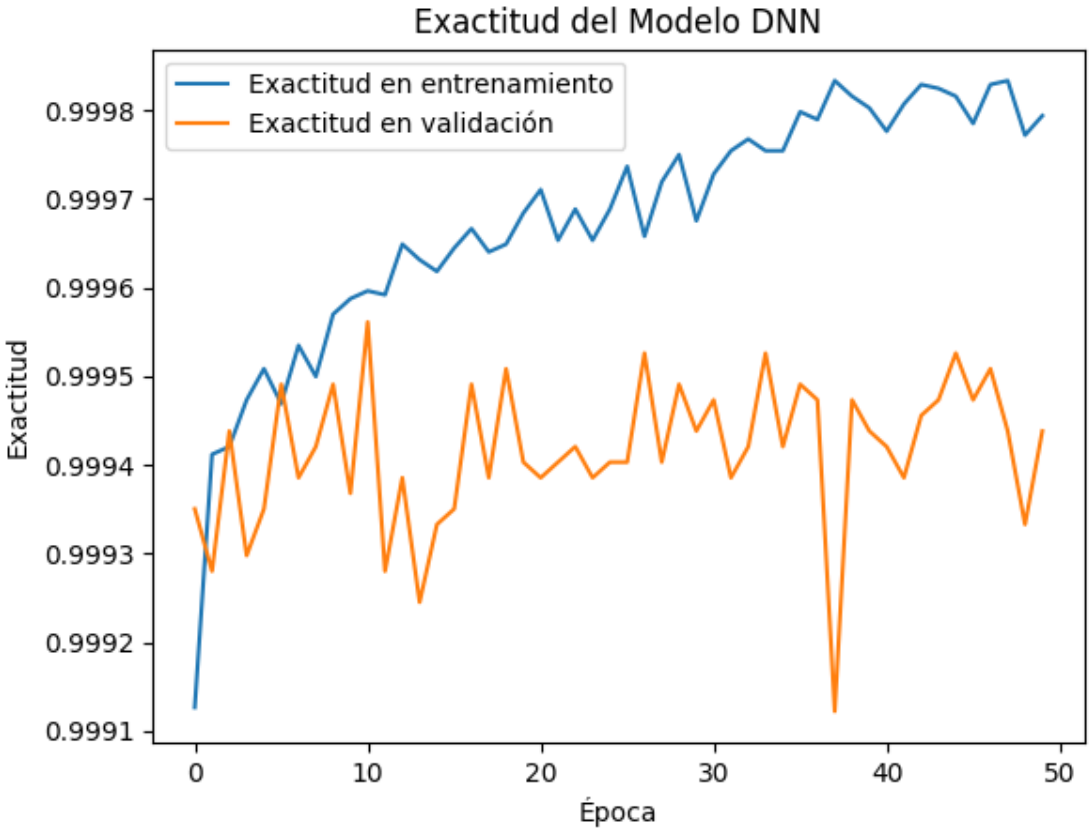
Código Ajustado para Deep Neural Network (DNN) con Keras:

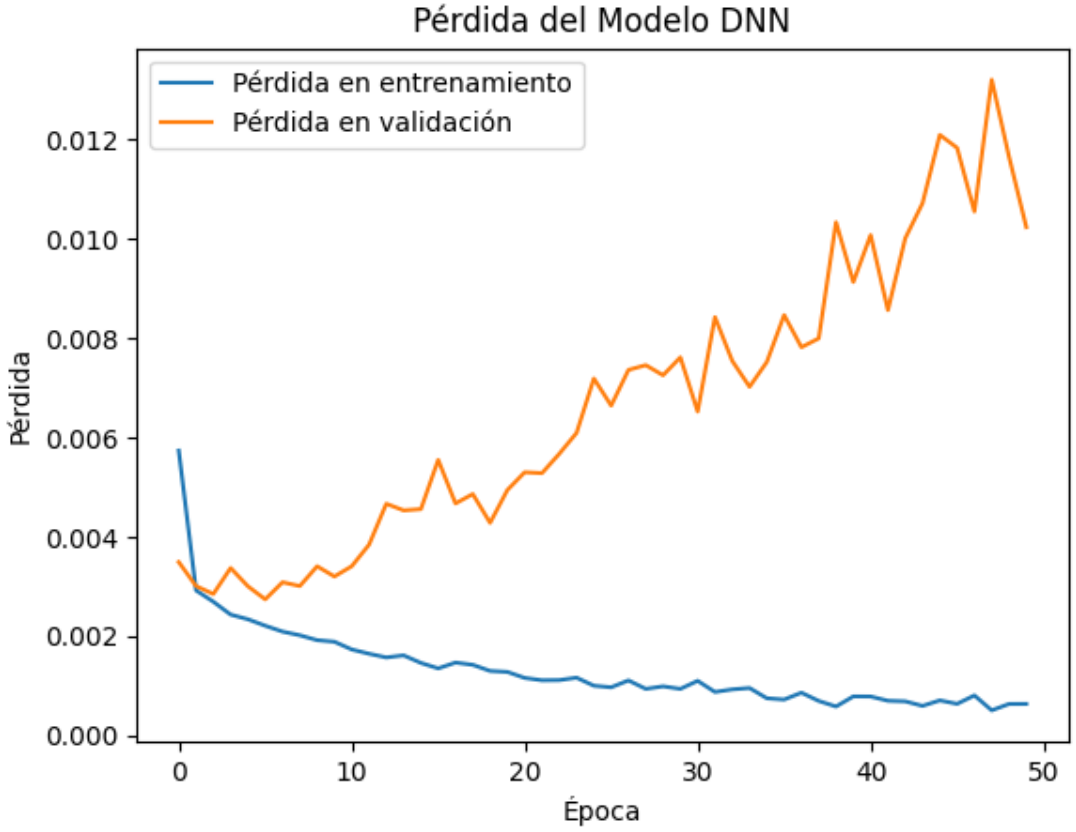
```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt

# Suponiendo que df ya contiene los datos cargados y limpios
# Separar las características y la variable objetivo (Class)
X = df.drop(columns=['Class'])
y = df['Class']

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos para mejorar el rendimiento del modelo
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Crear el modelo de Deep Neural Network (DNN)
model = Sequential()
model.add(Dense(64, input_dim=X_train_scaled.shape[1], activation='relu'))  # Primera capa oculta con 64 neuronas
model.add(Dense(32, activation='relu'))  # Segunda capa oculta con 32 neuronas
model.add(Dense(1, activation='sigmoid'))  # Capa de salida para clasificación binaria

# Compilar el modelo usando Adam como optimizador y binary_crossentropy para clasificación binaria
```

```
model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

# Entrenar el modelo
history = model.fit(X_train_scaled, y_train, validation_data=(X_test_scaled, y_test), epochs=50, batch_size=32)

# Evaluar el modelo en el conjunto de prueba
loss, accuracy = model.evaluate(X_test_scaled, y_test)
print(f"Exactitud en el conjunto de prueba: {accuracy * 100:.2f}%")

# Graficar el desempeño durante el entrenamiento
plt.plot(history.history['accuracy'], label='Exactitud en entrenamiento')
plt.plot(history.history['val_accuracy'], label='Exactitud en validación')
plt.title('Exactitud del Modelo DNN')
plt.xlabel('Época')
plt.ylabel('Exactitud')
plt.legend()
plt.show()

# También podemos graficar la función de pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en validación')
plt.title('Pérdida del Modelo DNN')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.show()
```

```

### Explicación:

1. **Dataset**: Utilizamos el DataFrame `df` con todas las características excepto `Class` como entrada (`X`), y `Class` como la variable objetivo (`y`).

2. **Escalamiento de Datos**: Aplicamos `StandardScaler` para escalar las características, lo cual es importante para optimizar el entrenamiento de redes neuronales.

3. **Arquitectura del Modelo**:

  - Dos capas ocultas de 64 y 32 neuronas con la función de activación `ReLU`.

  - Capa de salida con una sola neurona y activación `sigmoid` para predecir la clase (fraude/no fraude).

4. **Entrenamiento**: Se entrena el modelo por 50 épocas, usando el optimizador **Adam** y la función de pérdida **binary_crossentropy**.

5. **Visualización**: Graficamos tanto la **exactitud** como la **pérdida** durante el entrenamiento y la validación.

### Próximos Pasos:

1. Ejecuta este código y verifica los resultados de **exactitud** y **pérdida** durante el entrenamiento.

2. Si el rendimiento es adecuado, podemos proceder a ajustar hiperparámetros o intentar modelos más complejos como **CNN** o **RNN**.

Cuando ejecutes este código, revisemos los resultados para ver si necesitas ajustes adicionales.

Deep_Neural_Network_Keras.PY

GRAFICOS 1 Y 2
```


Exactitud del Modelo DNN
Exactitud en entrenamiento
Exactitud en validación
Exactitud
Época
0.9998
0.9997
0.9996
0.9995
0.9994
0.9993
0.9992
0.9991
0
10
20
30
40
50


Pérdida del Modelo DNN
Pérdida en entrenamiento
Pérdida en validación
Pérdida
Época
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
10
20
30
40
50

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test> PYTHON Deep_Neural_Network_Keras.PY

2024-10-14 07:10:24.127333: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

2024-10-14 07:10:25.454763: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

Time V1 V2 V3 V4 V5 V6 V7 ... V23 V24 V25 V26 V27 V28 Amount Class

0 0.0 -1.359807 -0.072781 2.536347 1.378155 -0.338321 0.462388 0.239599 ... -0.110474 0.066928 0.128539 -0.189115 0.133558 -0.021053 149.62 0

1 0.0 1.191857 0.266151 0.166480 0.448154 0.060018 -0.082361 -0.078803 ... 0.101288 -0.339846 0.167170 0.125895 -0.008983 0.014724 2.69 0

2 1.0 -1.358354 -1.340163 1.773209 0.379780 -0.503198 1.800499 0.791461 ... 0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752 378.66 0

3 1.0 -0.966272 -0.185226 1.792993 -0.863291 -0.010309 1.247203 0.237609 ... -0.190321 -1.175575 0.647376 -0.221929 0.062723 0.061458 123.50 0

4 2.0 -1.158233 0.877737 1.548718 0.403034 -0.407193 0.095921 0.592941 ... -0.137458 0.141267 -0.206010 0.502292 0.219422 0.215153 69.99 0

[5 rows x 31 columns]

C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test\venv\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().__init__(activity_regularizer=activity_regularizer, **kwargs)

2024-10-14 07:10:39.306447: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.

Epoch 1/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9975 - loss: 0.0160 - val_accuracy: 0.9994 - val_loss: 0.0035

Epoch 2/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9994 - loss: 0.0031 - val_accuracy: 0.9993 - val_loss: 0.0030

Epoch 3/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9994 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0029

Epoch 4/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9995 - loss: 0.0024 - val_accuracy: 0.9993 - val_loss: 0.0034

Epoch 5/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9996 - loss: 0.0021 - val_accuracy: 0.9994 - val_loss: 0.0030

Epoch 6/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9995 - loss: 0.0024 - val_accuracy: 0.9995 - val_loss: 0.0027

Epoch 7/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9996 - loss: 0.0019 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 8/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9995 - loss: 0.0019 - val_accuracy: 0.9994 - val_loss: 0.0030

Epoch 9/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9996 - loss: 0.0017 - val_accuracy: 0.9995 - val_loss: 0.0034

Epoch 10/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9996 - loss: 0.0018 - val_accuracy: 0.9994 - val_loss: 0.0032

Epoch 11/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 0.0012 - val_accuracy: 0.9996 - val_loss: 0.0034

Epoch 12/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9996 - loss: 0.0015 - val_accuracy: 0.9993 - val_loss: 0.0038

Epoch 13/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 0.0015 - val_accuracy: 0.9994 - val_loss: 0.0047

Epoch 14/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 0.0015 - val_accuracy: 0.9992 - val_loss: 0.0045

Epoch 15/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 0.0011 - val_accuracy: 0.9993 - val_loss: 0.0046

Epoch 16/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9996 - loss: 0.0016 - val_accuracy: 0.9994 - val_loss: 0.0055

Epoch 17/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 0.0015 - val_accuracy: 0.9995 - val_loss: 0.0047

Epoch 18/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 0.0011 - val_accuracy: 0.9994 - val_loss: 0.0049

Epoch 19/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 8.1987e-04 - val_accuracy: 0.9995 - val_loss: 0.0043

Epoch 20/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9996 - loss: 0.0013 - val_accuracy: 0.9994 - val_loss: 0.0049

Epoch 21/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 0.0011 - val_accuracy: 0.9994 - val_loss: 0.0053

Epoch 22/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9996 - loss: 0.0011 - val_accuracy: 0.9994 - val_loss: 0.0053

Epoch 23/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 0.0011 - val_accuracy: 0.9994 - val_loss: 0.0057

Epoch 24/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9996 - loss: 0.0012 - val_accuracy: 0.9994 - val_loss: 0.0061

Epoch 25/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 9.5850e-04 - val_accuracy: 0.9994 - val_loss: 0.0072

Epoch 26/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 9.2934e-04 - val_accuracy: 0.9994 - val_loss: 0.0066

Epoch 27/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 9.1563e-04 - val_accuracy: 0.9995 - val_loss: 0.0074

Epoch 28/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 7.3593e-04 - val_accuracy: 0.9994 - val_loss: 0.0075

Epoch 29/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 9.4582e-04 - val_accuracy: 0.9995 - val_loss: 0.0073

Epoch 30/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 8.0659e-04 - val_accuracy: 0.9994 - val_loss: 0.0076

Epoch 31/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 8.8698e-04 - val_accuracy: 0.9995 - val_loss: 0.0065

Epoch 32/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 7.2592e-04 - val_accuracy: 0.9994 - val_loss: 0.0084

Epoch 33/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 7.1158e-04 - val_accuracy: 0.9994 - val_loss: 0.0075

Epoch 34/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 6.8461e-04 - val_accuracy: 0.9995 - val_loss: 0.0070

Epoch 35/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 7.0031e-04 - val_accuracy: 0.9994 - val_loss: 0.0075

Epoch 36/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 7.3778e-04 - val_accuracy: 0.9995 - val_loss: 0.0085

Epoch 37/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 7.2989e-04 - val_accuracy: 0.9995 - val_loss: 0.0078

Epoch 38/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 5.7418e-04 - val_accuracy: 0.9991 - val_loss: 0.0080

Epoch 39/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 5.8808e-04 - val_accuracy: 0.9995 - val_loss: 0.0103

Epoch 40/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9999 - loss: 6.4234e-04 - val_accuracy: 0.9994 - val_loss: 0.0091

Epoch 41/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 6.6040e-04 - val_accuracy: 0.9994 - val_loss: 0.0101

Epoch 42/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 9.0443e-04 - val_accuracy: 0.9994 - val_loss: 0.0086

Epoch 43/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 5.1440e-04 - val_accuracy: 0.9995 - val_loss: 0.0100

Epoch 44/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 5.6219e-04 - val_accuracy: 0.9995 - val_loss: 0.0107

Epoch 45/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 6.3017e-04 - val_accuracy: 0.9995 - val_loss: 0.0121

Epoch 46/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9997 - loss: 7.7278e-04 - val_accuracy: 0.9995 - val_loss: 0.0118

Epoch 47/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9998 - loss: 7.1408e-04 - val_accuracy: 0.9995 - val_loss: 0.0105

Epoch 48/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9999 - loss: 3.3015e-04 - val_accuracy: 0.9994 - val_loss: 0.0132

Epoch 49/50

7121/7121 ━━ 9s 1ms/step - accuracy: 0.9998 - loss: 4.9477e-04 - val_accuracy: 0.9993 - val_loss: 0.0116

Epoch 50/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.9998 - loss: 6.0208e-04 - val_accuracy: 0.9994 - val_loss: 0.0102

1781/1781 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 935us/step - accuracy: 0.9995 - loss: 0.0097

Exactitud en el conjunto de prueba: 99.94%

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test>

Has completado exitosamente el entrenamiento del modelo Deep Neural Network con Keras. La exactitud alcanzada en el conjunto de prueba es del 99.94%, lo cual es un excelente resultado. Observamos que las curvas de exactitud y pérdida muestran un desempeño sólido tanto en entrenamiento como en validación.

Próximos pasos:

Ajustes adicionales:

Regularización: Considera agregar técnicas como Dropout o L2 regularización para evitar posibles problemas de sobreajuste, ya que la pérdida en validación muestra un leve aumento en algunas épocas.

Evaluación más detallada:

Calcula métricas adicionales como precisión, recall, F1-score y la matriz de confusión para entender mejor cómo se comporta el modelo con cada clase (fraude/no fraude).

Aplicación de Redes Neuronales Convolucionales (CNN):

Aunque no es típico para este tipo de datos, podrías experimentar con CNN si transformas los datos en forma de imagen o en análisis de secuencias temporales.

Transfer Learning:

Podrías intentar Transfer Learning si encuentras un problema similar en otro conjunto de datos y deseas aprovechar modelos preentrenados.

Redes Neuronales Recurrentes (RNN):

Si tus datos incluyen características secuenciales, como transacciones en un periodo de tiempo, considera el uso de RNN o LSTM.

El código completo con las mejoras de regularización (Dropout y L2), y la evaluación adicional de métricas como la matriz de confusión, precisión, recall y F1-Score:

```python
import numpy as np
import pandas as pd
import arff  # Para manejar archivos ARFF
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt

# Cargar el archivo ARFF y convertirlo en un DataFrame
with open("C:/DB_Covid19Arg/csv_archivos_limpios/Amazon_test/phpKo8OWT.arff") as f:
    dataset = arff.load(f)

# Convertir el dataset ARFF en un DataFrame de pandas
df = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])
```

```
# Convertir los datos en formato numérico
df = df.apply(pd.to_numeric, errors='coerce')

# Eliminar filas con valores NaN que pudieran quedar tras la conversión
df = df.dropna()

# Verificar las primeras filas del DataFrame para asegurar que se cargó correctamente
print(df.head())

# Separar las características y la variable objetivo (Class)
X = df.drop(columns=['Class'])
y = df['Class']

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos para mejorar el rendimiento del modelo
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Crear el modelo de Deep Neural Network (DNN) con Dropout y L2 regularización
model = Sequential()
model.add(Dense(64, input_dim=X_train_scaled.shape[1], activation='relu',
kernel_regularizer=l2(0.001)))  # Primera capa oculta con regularización L2
model.add(Dropout(0.5))  # Dropout del 50%
```

```
model.add(Dense(32, activation='relu', kernel_regularizer=l2(0.001)))  # Segunda capa oculta con regularización L2

model.add(Dropout(0.5))

model.add(Dense(1, activation='sigmoid'))  # Capa de salida para clasificación binaria


# Compilar el modelo usando Adam como optimizador y binary_crossentropy para clasificación binaria

model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])


# Entrenar el modelo

history = model.fit(X_train_scaled, y_train, validation_data=(X_test_scaled, y_test), epochs=50, batch_size=32)


# Evaluar el modelo en el conjunto de prueba

loss, accuracy = model.evaluate(X_test_scaled, y_test)

print(f"Exactitud en el conjunto de prueba: {accuracy * 100:.2f}%")


# Graficar el desempeño durante el entrenamiento

plt.plot(history.history['accuracy'], label='Exactitud en entrenamiento')

plt.plot(history.history['val_accuracy'], label='Exactitud en validación')

plt.title('Exactitud del Modelo DNN')

plt.xlabel('Época')

plt.ylabel('Exactitud')

plt.legend()

plt.show()


# También podemos graficar la función de pérdida durante el entrenamiento

plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
```

```
plt.plot(history.history['val_loss'], label='Pérdida en validación')
plt.title('Pérdida del Modelo DNN')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.show()

# Hacer predicciones en el conjunto de prueba
y_pred = (model.predict(X_test_scaled) > 0.5).astype("int32")

# Calcular la matriz de confusión
cm = confusion_matrix(y_test, y_pred)
print("Matriz de Confusión:")
print(cm)

# Calcular precisión, recall, F1-score
report = classification_report(y_test, y_pred)
print("\nReporte de Clasificación:")
print(report)
```

Explicación:

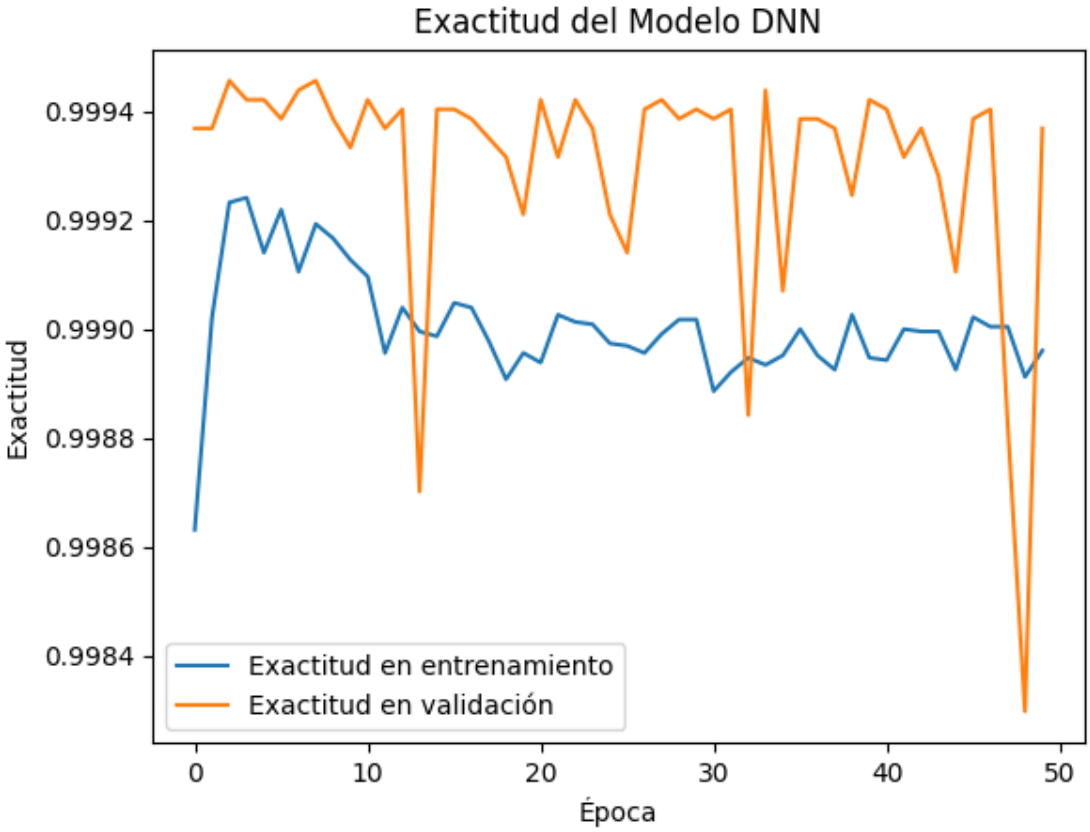
1. **Dropout**: Se ha añadido una capa de Dropout con una tasa del 50% en cada capa oculta para reducir el sobreajuste.
2. **L2 Regularización**: En las capas densas se ha aplicado una regularización L2 para penalizar los pesos grandes.
3. **Evaluación Adicional**:

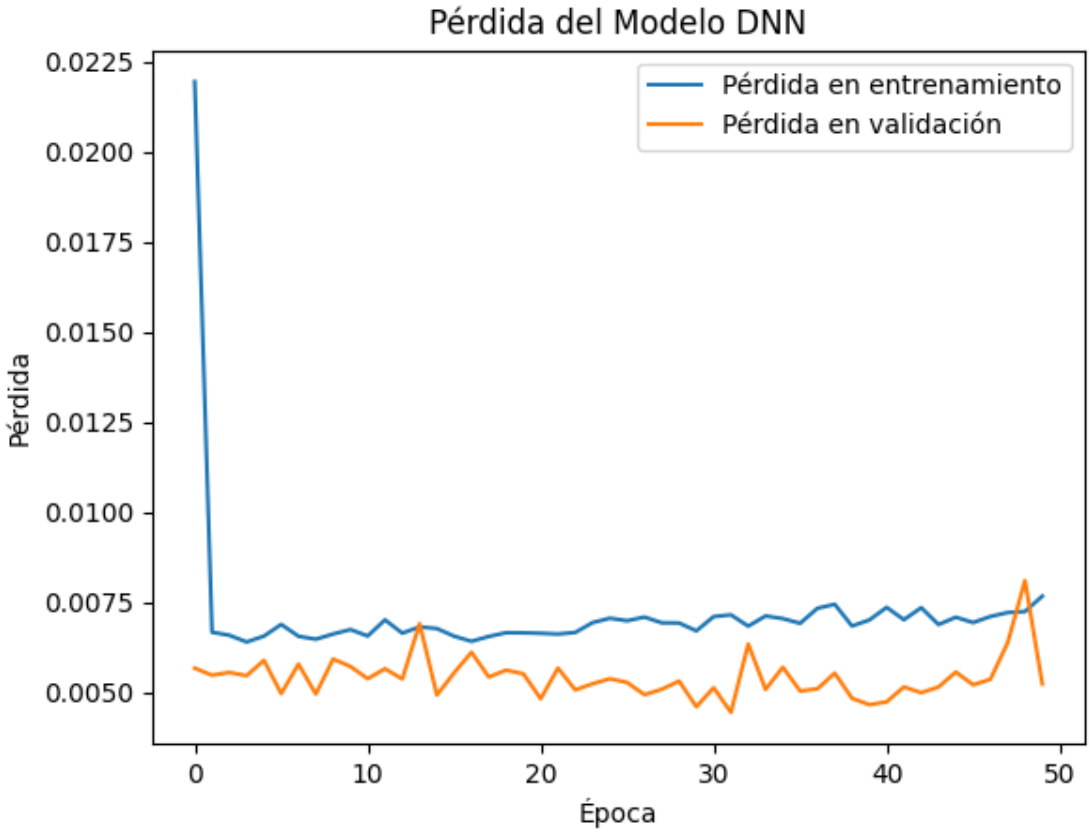
- Se han añadido cálculos de la matriz de confusión y un reporte de clasificación que incluye precisión, recall y F1-Score.

Este código mejora la robustez del modelo frente al sobreajuste y proporciona una evaluación más detallada del rendimiento del modelo.

Deep_Neural_Network_Keras.PY

GRAFICO 3 Y 4


Exactitud del Modelo DNN
Exactitud
0.9994
0.9992
0.9990
0.9988
0.9986
0.9984
Exactitud en entrenamiento
Exactitud en validación
0
10
20
30
40
50
Época


Pérdida del Modelo DNN
Pérdida
0.0225
0.0200
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
Pérdida en entrenamiento
Pérdida en validación
0
10
20
30
40
50
Época

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test> PYTHON Deep_Neural_Network_Keras.PY

2024-10-14 07:26:25.302949: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

2024-10-14 07:26:26.759187: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

Time V1 V2 V3 V4 V5 V6 V7 ... V23 V24 V25 V26 V27 V28 Amount Class

0 0.0 -1.359807 -0.072781 2.536347 1.378155 -0.338321 0.462388 0.239599 ... -0.110474 0.066928 0.128539 -0.189115 0.133558 -0.021053 149.62 0

1 0.0 1.191857 0.266151 0.166480 0.448154 0.060018 -0.082361 -0.078803 ... 0.101288 -0.339846 0.167170 0.125895 -0.008983 0.014724 2.69 0

2 1.0 -1.358354 -1.340163 1.773209 0.379780 -0.503198 1.800499 0.791461 ... 0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752 378.66 0

3 1.0 -0.966272 -0.185226 1.792993 -0.863291 -0.010309 1.247203 0.237609 ... -0.190321 -1.175575 0.647376 -0.221929 0.062723 0.061458 123.50 0

4 2.0 -1.158233 0.877737 1.548718 0.403034 -0.407193 0.095921 0.592941 ... -0.137458 0.141267 -0.206010 0.502292 0.219422 0.215153 69.99 0

[5 rows x 31 columns]

C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test\venv\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().__init__(activity_regularizer=activity_regularizer, **kwargs)

2024-10-14 07:26:40.963408: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.

Epoch 1/50

7121/7121 ━━ 11s 1ms/step - accuracy: 0.9981 - loss: 0.0506 - val_accuracy: 0.9994 - val_loss: 0.0057

Epoch 2/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9989 - loss: 0.0075 - val_accuracy: 0.9994 - val_loss: 0.0055

Epoch 3/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9993 - loss: 0.0060 - val_accuracy: 0.9995 - val_loss: 0.0055

Epoch 4/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9993 - loss: 0.0058 - val_accuracy: 0.9994 - val_loss: 0.0055

Epoch 5/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9991 - loss: 0.0068 - val_accuracy: 0.9994 - val_loss: 0.0059

Epoch 6/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9993 - loss: 0.0071 - val_accuracy: 0.9994 - val_loss: 0.0050

Epoch 7/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9992 - loss: 0.0063 - val_accuracy: 0.9994 - val_loss: 0.0058

Epoch 8/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9993 - loss: 0.0061 - val_accuracy: 0.9995 - val_loss: 0.0049

Epoch 9/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9992 - loss: 0.0069 - val_accuracy: 0.9994 - val_loss: 0.0059

Epoch 10/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9992 - loss: 0.0065 - val_accuracy: 0.9993 - val_loss: 0.0057

Epoch 11/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9991 - loss: 0.0068 - val_accuracy: 0.9994 - val_loss: 0.0054

Epoch 12/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0074 - val_accuracy: 0.9994 - val_loss: 0.0056

Epoch 13/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9991 - loss: 0.0067 - val_accuracy: 0.9994 - val_loss: 0.0054

Epoch 14/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0070 - val_accuracy: 0.9987 - val_loss: 0.0069

Epoch 15/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0071 - val_accuracy: 0.9994 - val_loss: 0.0049

Epoch 16/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0065 - val_accuracy: 0.9994 - val_loss: 0.0055

Epoch 17/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9991 - loss: 0.0065 - val_accuracy: 0.9994 - val_loss: 0.0061

Epoch 18/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9991 - loss: 0.0057 - val_accuracy: 0.9994 - val_loss: 0.0054

Epoch 19/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0065 - val_accuracy: 0.9993 - val_loss: 0.0056

Epoch 20/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9989 - loss: 0.0065 - val_accuracy: 0.9992 - val_loss: 0.0055

Epoch 21/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0066 - val_accuracy: 0.9994 - val_loss: 0.0048

Epoch 22/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0064 - val_accuracy: 0.9993 - val_loss: 0.0057

Epoch 23/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0061 - val_accuracy: 0.9994 - val_loss: 0.0051

Epoch 24/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9991 - loss: 0.0066 - val_accuracy: 0.9994 - val_loss: 0.0052

Epoch 25/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0069 - val_accuracy: 0.9992 - val_loss: 0.0054

Epoch 26/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0069 - val_accuracy: 0.9991 - val_loss: 0.0053

Epoch 27/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9989 - loss: 0.0072 - val_accuracy: 0.9994 - val_loss: 0.0049

Epoch 28/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0066 - val_accuracy: 0.9994 - val_loss: 0.0051

Epoch 29/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0071 - val_accuracy: 0.9994 - val_loss: 0.0053

Epoch 30/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9991 - loss: 0.0066 - val_accuracy: 0.9994 - val_loss: 0.0046

Epoch 31/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0067 - val_accuracy: 0.9994 - val_loss: 0.0051

Epoch 32/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0070 - val_accuracy: 0.9994 - val_loss: 0.0044

Epoch 33/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9991 - loss: 0.0062 - val_accuracy: 0.9988 - val_loss: 0.0063

Epoch 34/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0066 - val_accuracy: 0.9994 - val_loss: 0.0051

Epoch 35/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9991 - loss: 0.0064 - val_accuracy: 0.9991 - val_loss: 0.0057

Epoch 36/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0069 - val_accuracy: 0.9994 - val_loss: 0.0050

Epoch 37/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9989 - loss: 0.0074 - val_accuracy: 0.9994 - val_loss: 0.0051

Epoch 38/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0068 - val_accuracy: 0.9994 - val_loss: 0.0055

Epoch 39/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0069 - val_accuracy: 0.9992 - val_loss: 0.0048

Epoch 40/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0068 - val_accuracy: 0.9994 - val_loss: 0.0046

Epoch 41/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0069 - val_accuracy: 0.9994 - val_loss: 0.0047

Epoch 42/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0072 - val_accuracy: 0.9993 - val_loss: 0.0051

Epoch 43/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9989 - loss: 0.0078 - val_accuracy: 0.9994 - val_loss: 0.0050

Epoch 44/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0067 - val_accuracy: 0.9993 - val_loss: 0.0051

Epoch 45/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9989 - loss: 0.0071 - val_accuracy: 0.9991 - val_loss: 0.0056

Epoch 46/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0071 - val_accuracy: 0.9994 - val_loss: 0.0052

Epoch 47/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0072 - val_accuracy: 0.9994 - val_loss: 0.0054

Epoch 48/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0071 - val_accuracy: 0.9988 - val_loss: 0.0064

Epoch 49/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0067 - val_accuracy: 0.9983 - val_loss: 0.0081

Epoch 50/50

7121/7121 ━━ 10s 1ms/step - accuracy: 0.9990 - loss: 0.0074 - val_accuracy: 0.9994 - val_loss: 0.0052

1781/1781 ━━ 2s 911us/step - accuracy: 0.9994 - loss: 0.0053

Exactitud en el conjunto de prueba: 99.94%

1781/1781 ━━ 2s 995us/step

Matriz de Confusión:

[[56854 10]

[26 72]]

Reporte de Clasificación:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56864
1	0.88	0.73	0.80	98
accuracy			1.00	56962
macro avg	0.94	0.87	0.90	56962
weighted avg	1.00	1.00	1.00	56962

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test>

Hemos completado el análisis del modelo de red neuronal profunda (DNN) con las métricas adicionales de precisión, recall y F1-score. El próximo paso, antes de continuar con el siguiente modelo, es considerar ajustes adicionales como:

Regularización:

Puedes agregar Dropout o L2 regularización para evitar el sobreajuste. El leve aumento de la pérdida en algunas épocas puede indicar que el modelo está comenzando a ajustarse demasiado a los datos de entrenamiento.

Evaluación más detallada:

Ya calculamos la matriz de confusión, la precisión, el recall, y el F1-score. Estos son fundamentales para entender mejor cómo el modelo clasifica correctamente tanto fraudes como transacciones normales.

Una vez que ajustemos estos parámetros, podemos continuar con las pruebas de Convolutional Neural Networks (CNN) en el siguiente paso.

Para implementar **Convolutional Neural Networks (CNN)** en tu dataset, a pesar de que las CNNs son más utilizadas para imágenes, también podemos adaptarlas para tareas de clasificación en datasets tabulares. Aquí está el código ajustado para implementar un modelo CNN básico en Keras utilizando tu dataset. Vamos a empezar:

```python
import numpy as np
import pandas as pd
import arff  # Para manejar archivos ARFF
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv1D, MaxPooling1D, Flatten, Dropout
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix

# Cargar el archivo ARFF y convertirlo en un DataFrame
with open("C:/DB_Covid19Arg/csv_archivos_limpios/Amazon_test/phpKo8OWT.arff") as f:
```

```
    dataset = arff.load(f)

# Convertir el dataset ARFF en un DataFrame de pandas
df = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])

# Convertir los datos en formato numérico
df = df.apply(pd.to_numeric, errors='coerce')

# Eliminar filas con valores NaN que pudieran quedar tras la conversión
df = df.dropna()

# Separar las características y la variable objetivo (Class)
X = df.drop(columns=['Class'])
y = df['Class']

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos para mejorar el rendimiento del modelo
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Redimensionar los datos para adaptarlos a la entrada de Conv1D (Se espera una entrada 3D: muestras, pasos de tiempo, características)
X_train_scaled = np.expand_dims(X_train_scaled, axis=2)
X_test_scaled = np.expand_dims(X_test_scaled, axis=2)
```

```
# Crear el modelo de CNN
model = Sequential()

# Agregar una capa Conv1D
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(X_train_scaled.shape[1], 1)))
model.add(MaxPooling1D(pool_size=2))

# Añadir más capas convolucionales si es necesario
model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))

# Aplanar los resultados y pasar a capas densas
model.add(Flatten())
model.add(Dense(128, activation='relu'))

# Agregar Dropout para evitar el sobreajuste
model.add(Dropout(0.5))

# Capa de salida para clasificación binaria
model.add(Dense(1, activation='sigmoid'))

# Compilar el modelo
model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy',
metrics=['accuracy'])

# Entrenar el modelo
history = model.fit(X_train_scaled, y_train, validation_data=(X_test_scaled, y_test),
epochs=50, batch_size=32)
```

```
# Evaluar el modelo en el conjunto de prueba
loss, accuracy = model.evaluate(X_test_scaled, y_test)
print(f"Exactitud en el conjunto de prueba: {accuracy * 100:.2f}%")

# Graficar el desempeño durante el entrenamiento
plt.plot(history.history['accuracy'], label='Exactitud en entrenamiento')
plt.plot(history.history['val_accuracy'], label='Exactitud en validación')
plt.title('Exactitud del Modelo CNN')
plt.xlabel('Época')
plt.ylabel('Exactitud')
plt.legend()
plt.show()

# Graficar la función de pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en validación')
plt.title('Pérdida del Modelo CNN')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.show()

# Generar la matriz de confusión y el reporte de clasificación
y_pred = (model.predict(X_test_scaled) > 0.5).astype("int32")
print("Matriz de Confusión:")
print(confusion_matrix(y_test, y_pred))
print("\nReporte de Clasificación:")
```

print(classification_report(y_test, y_pred))

```

### Explicación del Código:

1. **Conv1D**: Se utiliza una capa de convolución para aprender características importantes de la secuencia de datos (aunque sea tabular). Se aplica un kernel (filtro) que pasa por los datos.

2. **MaxPooling1D**: Ayuda a reducir el tamaño dimensional de los datos, manteniendo las características más importantes.

3. **Flatten**: Se aplanan los datos antes de pasarlos a capas densas para la clasificación.

4. **Dropout**: Se agrega para evitar el sobreajuste.

5. **Capa de salida**: Para clasificación binaria (`sigmoid`).

### Próximos Pasos:

- Ejecuta este código para ver cómo funciona la CNN con tu dataset.

- Ajusta los hiperparámetros como el tamaño del kernel o los filtros en las capas Conv1D para ver si mejoras el rendimiento.

Una vez lo ejecutes, podemos avanzar hacia ajustes adicionales o técnicas más avanzadas como **Transfer Learning**.

Aquí tienes el código completo para implementar una **Convolutional Neural Network (CNN)** en tus datos tabulares, siguiendo los pasos que mencionamos. Vamos a usar una capa de convolución 1D para detectar patrones en las características de tu dataset.

```python

import numpy as np

import pandas as pd

import arff  # Para manejar archivos ARFF

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv1D, MaxPooling1D, Flatten, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt

Cargar el archivo ARFF y convertirlo en un DataFrame
with open("C:/DB_Covid19Arg/csv_archivos_limpios/Amazon_test/phpKo8OWT.arff") as f:
    dataset = arff.load(f)

# Convertir el dataset ARFF en un DataFrame de pandas
df = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])

# Convertir los datos en formato numérico
df = df.apply(pd.to_numeric, errors='coerce')

# Eliminar filas con valores NaN que pudieran quedar tras la conversión
df = df.dropna()

# Verificar las primeras filas del DataFrame para asegurar que se cargó correctamente
print(df.head())

# Separar las características y la variable objetivo (Class)
X = df.drop(columns=['Class'])
y = df['Class']
```

```
# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos para mejorar el rendimiento del modelo
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Reshape para que sea compatible con Conv1D (necesitamos una tercera dimensión)
X_train_cnn = np.expand_dims(X_train_scaled, axis=2)
X_test_cnn = np.expand_dims(X_test_scaled, axis=2)

# Crear el modelo CNN
model = Sequential()

# Primera capa de convolución 1D
model.add(Conv1D(filters=32, kernel_size=2, activation='relu',
input_shape=(X_train_cnn.shape[1], 1)))
model.add(MaxPooling1D(pool_size=2))  # Max pooling para reducir dimensionalidad

# Segunda capa de convolución 1D
model.add(Conv1D(filters=64, kernel_size=2, activation='relu'))
model.add(MaxPooling1D(pool_size=2))

# Aplanar los datos antes de pasar a la red densa
model.add(Flatten())
```

```
# Capa oculta completamente conectada con Dropout para evitar sobreajuste
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))  # Dropout regularization

# Capa de salida para clasificación binaria
model.add(Dense(1, activation='sigmoid'))

# Compilar el modelo usando Adam como optimizador y binary_crossentropy como función de pérdida
model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

# Entrenar el modelo
history = model.fit(X_train_cnn, y_train, validation_data=(X_test_cnn, y_test), epochs=50, batch_size=32)

# Evaluar el modelo en el conjunto de prueba
loss, accuracy = model.evaluate(X_test_cnn, y_test)
print(f"Exactitud en el conjunto de prueba: {accuracy * 100:.2f}%")

# Matriz de confusión y reporte de clasificación
y_pred = (model.predict(X_test_cnn) > 0.5).astype("int32")
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print("Matriz de Confusión:")
print(conf_matrix)
print("\nReporte de Clasificación:")
print(class_report)
```

```
# Graficar el desempeño durante el entrenamiento
plt.plot(history.history['accuracy'], label='Exactitud en entrenamiento')
plt.plot(history.history['val_accuracy'], label='Exactitud en validación')
plt.title('Exactitud del Modelo CNN')
plt.xlabel('Época')
plt.ylabel('Exactitud')
plt.legend()
plt.show()

# También podemos graficar la función de pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en validación')
plt.title('Pérdida del Modelo CNN')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.show()
```

Descripción de los componentes:

1. **Conv1D**: Capa de convolución 1D para capturar patrones locales en las características del dataset.
2. **MaxPooling1D**: Capa para reducir la dimensionalidad y evitar sobreajuste, capturando las características más importantes.
3. **Dense**: Capas completamente conectadas después de aplanar los datos.
4. **Dropout**: Regularización para prevenir el sobreajuste.

5. **Adam**: Optimizador eficiente para entrenar redes neuronales profundas.

Entrenamiento:

El modelo se entrenará durante 50 épocas, y los resultados se evaluarán utilizando **accuracy**, la **matriz de confusión** y el **reporte de clasificación** (precision, recall, F1-score).

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test> python Convolutional_Neural_Networks_keras.py

2024-10-14 07:42:31.182373: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

2024-10-14 07:42:32.546717: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test\venv\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().__init__(activity_regularizer=activity_regularizer, **kwargs)

2024-10-14 07:42:46.849302: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.

Epoch 1/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 25s 3ms/step - accuracy: 0.9975 - loss: 0.0164 - val_accuracy: 0.9994 - val_loss: 0.0033

Epoch 2/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9994 - val_loss: 0.0036

Epoch 3/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0045 - val_accuracy: 0.9994 - val_loss: 0.0035

Epoch 4/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9994 - val_loss: 0.0036

Epoch 5/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9993 - val_loss: 0.0045

Epoch 6/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0044 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 7/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0033 - val_accuracy: 0.9994 - val_loss: 0.0040

Epoch 8/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0038 - val_accuracy: 0.9994 - val_loss: 0.0037

Epoch 9/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0040 - val_accuracy: 0.9995 - val_loss: 0.0027

Epoch 10/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0035 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 11/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0033 - val_accuracy: 0.9994 - val_loss: 0.0028

Epoch 12/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 13/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9995 - val_loss: 0.0031

Epoch 14/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0032 - val_accuracy: 0.9995 - val_loss: 0.0031

Epoch 15/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9994 - val_loss: 0.0041

Epoch 16/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 17/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0028 - val_accuracy: 0.9995 - val_loss: 0.0025

Epoch 18/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9995 - val_loss: 0.0026

Epoch 19/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0032 - val_accuracy: 0.9994 - val_loss: 0.0030

Epoch 20/50

7121/7121 ━━ 21s 3ms/step - accuracy: 0.9996 - loss: 0.0028 - val_accuracy: 0.9994 - val_loss: 0.0029

Epoch 21/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0031 - val_accuracy: 0.9995 - val_loss: 0.0025

Epoch 22/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 23/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0032 - val_accuracy: 0.9996 - val_loss: 0.0028

Epoch 24/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0035 - val_accuracy: 0.9995 - val_loss: 0.0027

Epoch 25/50

7121/7121 ━━ 26s 4ms/step - accuracy: 0.9996 - loss: 0.0025 - val_accuracy: 0.9993 - val_loss: 0.0032

Epoch 26/50

7121/7121 ━━ 24s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0026

Epoch 27/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9996 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0028

Epoch 28/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0033 - val_accuracy: 0.9993 - val_loss: 0.0039

Epoch 29/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0039 - val_accuracy: 0.9995 - val_loss: 0.0031

Epoch 30/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 31/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0035 - val_accuracy: 0.9994 - val_loss: 0.0033

Epoch 32/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9995 - val_loss: 0.0030

Epoch 33/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0034

Epoch 34/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0036 - val_accuracy: 0.9996 - val_loss: 0.0030

Epoch 35/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0027

Epoch 36/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9996 - loss: 0.0027 - val_accuracy: 0.9995 - val_loss: 0.0031

Epoch 37/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 38/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9996 - loss: 0.0030 - val_accuracy: 0.9994 - val_loss: 0.0032

Epoch 39/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0027 - val_accuracy: 0.9995 - val_loss: 0.0028

Epoch 40/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9992 - val_loss: 0.0051

Epoch 41/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0026 - val_accuracy: 0.9994 - val_loss: 0.0035

Epoch 42/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0032 - val_accuracy: 0.9996 - val_loss: 0.0032

Epoch 43/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0035 - val_accuracy: 0.9996 - val_loss: 0.0034

Epoch 44/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9996 - loss: 0.0028 - val_accuracy: 0.9993 - val_loss: 0.0037

Epoch 45/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0035 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 46/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0036 - val_accuracy: 0.9996 - val_loss: 0.0032

Epoch 47/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0037 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 48/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0038 - val_accuracy: 0.9994 - val_loss: 0.0053

Epoch 49/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0032 - val_accuracy: 0.9996 - val_loss: 0.0027

Epoch 50/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0038

1781/1781 ━━ 3s 1ms/step - accuracy: 0.9995 - loss: 0.0039

Exactitud en el conjunto de prueba: 99.95%

1781/1781 ━━ 3s 2ms/step

Matriz de Confusión:

[[56859 5]

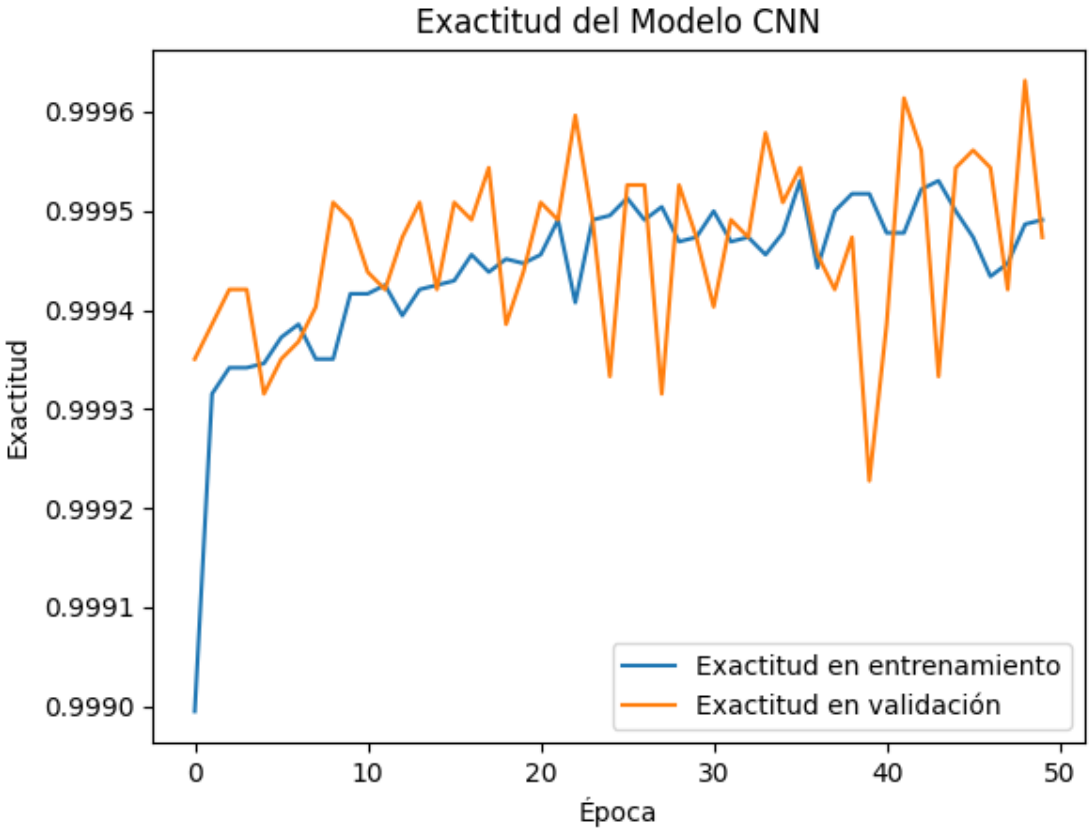
[25 73]]

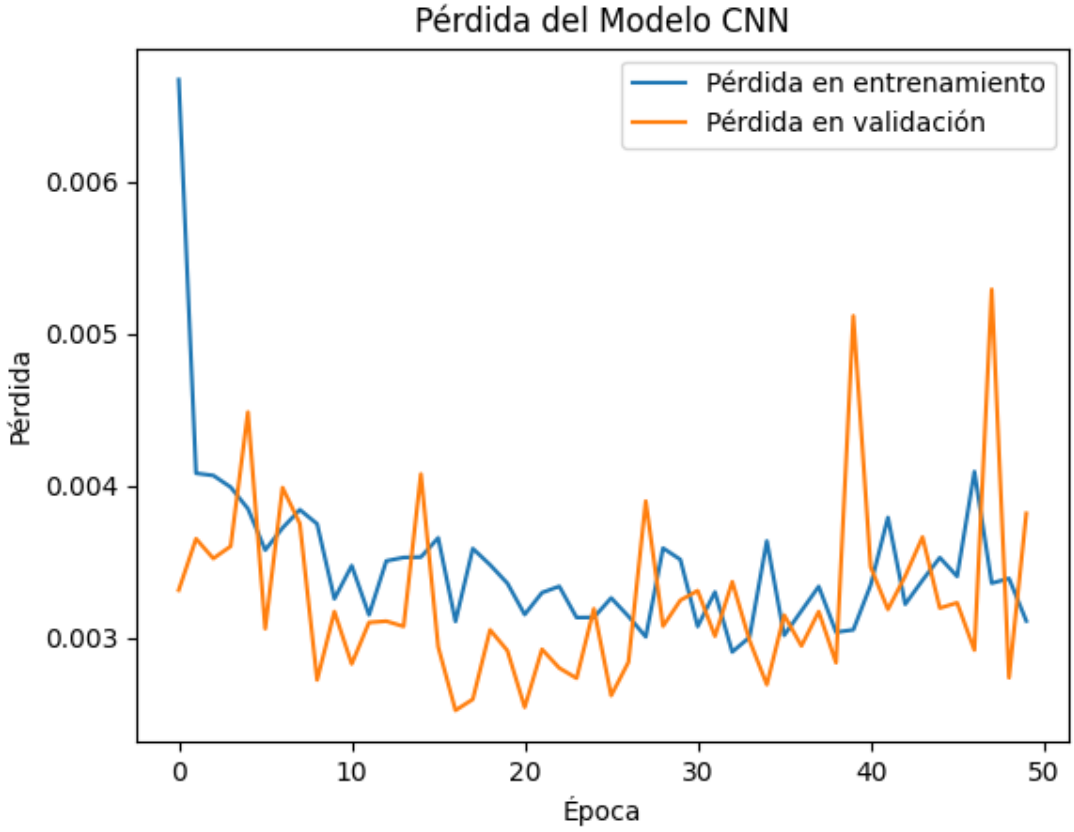
Reporte de Clasificación:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56864
1	0.94	0.74	0.83	98
accuracy			1.00	56962
macro avg	0.97	0.87	0.91	56962
weighted avg	1.00	1.00	1.00	56962

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test>

GRAFICOS 5 Y 6


Exactitud del Modelo CNN
Exactitud
Época
Exactitud en entrenamiento
Exactitud en validación
0.9996
0.9995
0.9994
0.9993
0.9992
0.9991
0.9990
0
10
20
30
40
50


Pérdida del Modelo CNN
Pérdida
Época
Pérdida en entrenamiento
Pérdida en validación
0.006
0.005
0.004
0.003
0
10
20
30
40
50

VOLVEMOS A PROBAR Convolutional_Neural_Networks_keras.py CON LOS CAMBIOS REALIZADOS:

Aquí te entrego el código con las dos versiones fusionadas, optimizando la estructura y las funciones para que puedas entrenar y evaluar el modelo CNN con las mejoras solicitadas:

```python
import numpy as np
import pandas as pd
import arff  # Para manejar archivos ARFF
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv1D, MaxPooling1D, Flatten, Dropout
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix

# Cargar el archivo ARFF y convertirlo en un DataFrame
with open("C:/DB_Covid19Arg/csv_archivos_limpios/Amazon_test/phpKo8OWT.arff") as f:
    dataset = arff.load(f)

# Convertir el dataset ARFF en un DataFrame de pandas
df = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])

# Convertir los datos en formato numérico
```

```
df = df.apply(pd.to_numeric, errors='coerce')

# Eliminar filas con valores NaN que pudieran quedar tras la conversión
df = df.dropna()

# Separar las características y la variable objetivo (Class)
X = df.drop(columns=['Class'])
y = df['Class']

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos para mejorar el rendimiento del modelo
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Redimensionar los datos para adaptarlos a la entrada de Conv1D (Se espera una entrada 3D: muestras, pasos de tiempo, características)
X_train_scaled = np.expand_dims(X_train_scaled, axis=2)
X_test_scaled = np.expand_dims(X_test_scaled, axis=2)

# Crear el modelo de CNN
model = Sequential()

# Agregar una capa Conv1D
model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=(X_train_scaled.shape[1], 1)))
```

```
model.add(MaxPooling1D(pool_size=2))

# Añadir más capas convolucionales si es necesario
model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))

# Aplanar los resultados y pasar a capas densas
model.add(Flatten())
model.add(Dense(128, activation='relu'))

# Agregar Dropout para evitar el sobreajuste
model.add(Dropout(0.5))

# Capa de salida para clasificación binaria
model.add(Dense(1, activation='sigmoid'))

# Compilar el modelo
model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy',
metrics=['accuracy'])

# Entrenar el modelo
history = model.fit(X_train_scaled, y_train, validation_data=(X_test_scaled, y_test),
epochs=50, batch_size=32)

# Evaluar el modelo en el conjunto de prueba
loss, accuracy = model.evaluate(X_test_scaled, y_test)
print(f"Exactitud en el conjunto de prueba: {accuracy * 100:.2f}%")
```

```
# Graficar el desempeño durante el entrenamiento
plt.plot(history.history['accuracy'], label='Exactitud en entrenamiento')
plt.plot(history.history['val_accuracy'], label='Exactitud en validación')
plt.title('Exactitud del Modelo CNN')
plt.xlabel('Época')
plt.ylabel('Exactitud')
plt.legend()
plt.show()

# Graficar la función de pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en validación')
plt.title('Pérdida del Modelo CNN')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.show()

# Generar la matriz de confusión y el reporte de clasificación
y_pred = (model.predict(X_test_scaled) > 0.5).astype("int32")
print("Matriz de Confusión:")
print(confusion_matrix(y_test, y_pred))
print("\nReporte de Clasificación:")
print(classification_report(y_test, y_pred))
```

Explicación:

1. **Ajustes para CNN**: Se ha adaptado el código para entrenar un modelo de red neuronal convolucional (CNN) con las capas `Conv1D` y `MaxPooling1D`. Los datos han sido redimensionados para ser compatibles con estas capas.

2. **Regularización**: Se ha añadido una capa de `Dropout` con un valor de 0.5 para mitigar el sobreajuste durante el entrenamiento.

3. **Evaluación del Modelo**: El código genera la **matriz de confusión** y el **reporte de clasificación** al final, lo que permite analizar en detalle el comportamiento del modelo con respecto a la clase minoritaria (fraude).

4. **Gráficas**: Se incluyen gráficas para visualizar el progreso de la exactitud y la pérdida durante el entrenamiento y la validación.

Este código está listo para ser ejecutado y permitirá seguir avanzando en la evaluación del modelo CNN con los datos. ¡Avísame si necesitas más ajustes o detalles!

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test> python Convolutional_Neural_Networks_keras.py

2024-10-14 07:42:31.182373: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

2024-10-14 07:42:32.546717: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test\venv\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

2024-10-14 07:42:46.849302: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.

Epoch 1/50

7121/7121 ━━ 25s 3ms/step - accuracy: 0.9975 - loss: 0.0164 - val_accuracy: 0.9994 - val_loss: 0.0033

Epoch 2/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9994 - val_loss: 0.0036

Epoch 3/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0045 - val_accuracy: 0.9994 - val_loss: 0.0035

Epoch 4/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9994 - val_loss: 0.0036

Epoch 5/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9993 - val_loss: 0.0045

Epoch 6/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0044 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 7/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0033 - val_accuracy: 0.9994 - val_loss: 0.0040

Epoch 8/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0038 - val_accuracy: 0.9994 - val_loss: 0.0037

Epoch 9/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0040 - val_accuracy: 0.9995 - val_loss: 0.0027

Epoch 10/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0035 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 11/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0033 - val_accuracy: 0.9994 - val_loss: 0.0028

Epoch 12/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 13/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9995 - val_loss: 0.0031

Epoch 14/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0032 - val_accuracy: 0.9995 - val_loss: 0.0031

Epoch 15/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9994 - val_loss: 0.0041

Epoch 16/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 17/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0028 - val_accuracy: 0.9995 - val_loss: 0.0025

Epoch 18/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9995 - val_loss: 0.0026

Epoch 19/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0032 - val_accuracy: 0.9994 - val_loss: 0.0030

Epoch 20/50

7121/7121 ━━ 21s 3ms/step - accuracy: 0.9996 - loss: 0.0028 - val_accuracy: 0.9994 - val_loss: 0.0029

Epoch 21/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0031 - val_accuracy: 0.9995 - val_loss: 0.0025

Epoch 22/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 23/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0032 - val_accuracy: 0.9996 - val_loss: 0.0028

Epoch 24/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0035 - val_accuracy: 0.9995 - val_loss: 0.0027

Epoch 25/50

7121/7121 ━━ 26s 4ms/step - accuracy: 0.9996 - loss: 0.0025 - val_accuracy: 0.9993 - val_loss: 0.0032

Epoch 26/50

7121/7121 ━━ 24s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0026

Epoch 27/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9996 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0028

Epoch 28/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0033 - val_accuracy: 0.9993 - val_loss: 0.0039

Epoch 29/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0039 - val_accuracy: 0.9995 - val_loss: 0.0031

Epoch 30/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 31/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0035 - val_accuracy: 0.9994 - val_loss: 0.0033

Epoch 32/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9995 - val_loss: 0.0030

Epoch 33/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0034

Epoch 34/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0036 - val_accuracy: 0.9996 - val_loss: 0.0030

Epoch 35/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0027

Epoch 36/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9996 - loss: 0.0027 - val_accuracy: 0.9995 - val_loss: 0.0031

Epoch 37/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 38/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9996 - loss: 0.0030 - val_accuracy: 0.9994 - val_loss: 0.0032

Epoch 39/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0027 - val_accuracy: 0.9995 - val_loss: 0.0028

Epoch 40/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9992 - val_loss: 0.0051

Epoch 41/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0026 - val_accuracy: 0.9994 - val_loss: 0.0035

Epoch 42/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0032 - val_accuracy: 0.9996 - val_loss: 0.0032

Epoch 43/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0035 - val_accuracy: 0.9996 - val_loss: 0.0034

Epoch 44/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9996 - loss: 0.0028 - val_accuracy: 0.9993 - val_loss: 0.0037

Epoch 45/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0035 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 46/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0036 - val_accuracy: 0.9996 - val_loss: 0.0032

Epoch 47/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0037 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 48/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0038 - val_accuracy: 0.9994 - val_loss: 0.0053

Epoch 49/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0032 - val_accuracy: 0.9996 - val_loss: 0.0027

Epoch 50/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0038

1781/1781 ━━ 3s 1ms/step - accuracy: 0.9995 - loss: 0.0039

Exactitud en el conjunto de prueba: 99.95%

1781/1781 ━━ 3s 2ms/step

Matriz de Confusión:

[[56859 5]

[25 73]]

Reporte de Clasificación:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56864
1	0.94	0.74	0.83	98
accuracy			1.00	56962
macro avg	0.97	0.87	0.91	56962
weighted avg	1.00	1.00	1.00	56962

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test> python Convolutional_Neural_Networks_keras.py

2024-10-14 08:06:18.575708: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

2024-10-14 08:06:19.936478: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test\venv\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().__init__(activity_regularizer=activity_regularizer, **kwargs)

2024-10-14 08:06:34.119361: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.

Epoch 1/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9973 - loss: 0.0140 - val_accuracy: 0.9983 - val_loss: 0.0112

Epoch 2/50

7121/7121 ━━ 21s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9994 - val_loss: 0.0035

Epoch 3/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0041 - val_accuracy: 0.9994 - val_loss: 0.0029

Epoch 4/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0039 - val_accuracy: 0.9995 - val_loss: 0.0031

Epoch 5/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9993 - loss: 0.0041 - val_accuracy: 0.9994 - val_loss: 0.0034

Epoch 6/50

7121/7121 ━━ 21s 3ms/step - accuracy: 0.9994 - loss: 0.0034 - val_accuracy: 0.9994 - val_loss: 0.0033

Epoch 7/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0038

Epoch 8/50

7121/7121 ━━ 21s 3ms/step - accuracy: 0.9994 - loss: 0.0034 - val_accuracy: 0.9995 - val_loss: 0.0034

Epoch 9/50

7121/7121 ━━ 21s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 10/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9996 - loss: 0.0033 - val_accuracy: 0.9995 - val_loss: 0.0036

Epoch 11/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0034 - val_accuracy: 0.9994 - val_loss: 0.0030

Epoch 12/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0057

Epoch 13/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9993 - loss: 0.0049 - val_accuracy: 0.9995 - val_loss: 0.0037

Epoch 14/50

7121/7121 ━━ 25s 4ms/step - accuracy: 0.9996 - loss: 0.0029 - val_accuracy: 0.9994 - val_loss: 0.0030

Epoch 15/50

7121/7121 ━━ 24s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9993 - val_loss: 0.0042

Epoch 16/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9994 - loss: 0.0037 - val_accuracy: 0.9995 - val_loss: 0.0038

Epoch 17/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9995 - loss: 0.0036 - val_accuracy: 0.9995 - val_loss: 0.0033

Epoch 18/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 19/50

7121/7121 ━━ 25s 3ms/step - accuracy: 0.9994 - loss: 0.0042 - val_accuracy: 0.9994 - val_loss: 0.0035

Epoch 20/50

7121/7121 ━━ 26s 4ms/step - accuracy: 0.9994 - loss: 0.0033 - val_accuracy: 0.9995 - val_loss: 0.0033

Epoch 21/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9995 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0033

Epoch 22/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 23/50

7121/7121 ━━ 24s 3ms/step - accuracy: 0.9995 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0032

Epoch 24/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9993 - loss: 0.0047 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 25/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0033 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 26/50

7121/7121 ━━ 23s 3ms/step - accuracy: 0.9994 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0033

Epoch 27/50

7121/7121 ━━ 24s 3ms/step - accuracy: 0.9994 - loss: 0.0047 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 28/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9995 - val_loss: 0.0036

Epoch 29/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9994 - val_loss: 0.0032

Epoch 30/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9994 - val_loss: 0.0030

Epoch 31/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 32/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0034 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 33/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0032 - val_accuracy: 0.9995 - val_loss: 0.0047

Epoch 34/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0026 - val_accuracy: 0.9994 - val_loss: 0.0038

Epoch 35/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0035 - val_accuracy: 0.9991 - val_loss: 0.0125

Epoch 36/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0040 - val_accuracy: 0.9993 - val_loss: 0.0137

Epoch 37/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0038 - val_accuracy: 0.9993 - val_loss: 0.0035

Epoch 38/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0031 - val_accuracy: 0.9995 - val_loss: 0.0046

Epoch 39/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0035 - val_accuracy: 0.9994 - val_loss: 0.0035

Epoch 40/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0033 - val_accuracy: 0.9994 - val_loss: 0.0036

Epoch 41/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0035 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 42/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0033 - val_accuracy: 0.9994 - val_loss: 0.0034

Epoch 43/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0035 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 44/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9996 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0028

Epoch 45/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0032 - val_accuracy: 0.9994 - val_loss: 0.0033

Epoch 46/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0049 - val_accuracy: 0.9994 - val_loss: 0.0034

Epoch 47/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0034 - val_accuracy: 0.9996 - val_loss: 0.0032

Epoch 48/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9996 - loss: 0.0025 - val_accuracy: 0.9995 - val_loss: 0.0034

Epoch 49/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9994 - loss: 0.0038 - val_accuracy: 0.9995 - val_loss: 0.0037

Epoch 50/50

7121/7121 ━━ 22s 3ms/step - accuracy: 0.9995 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0054

1781/1781 ━━ 3s 1ms/step - accuracy: 0.9994 - loss: 0.0059

Exactitud en el conjunto de prueba: 99.94%

1781/1781 ━━ 3s 2ms/step

Matriz de Confusión:

[[56852 12]

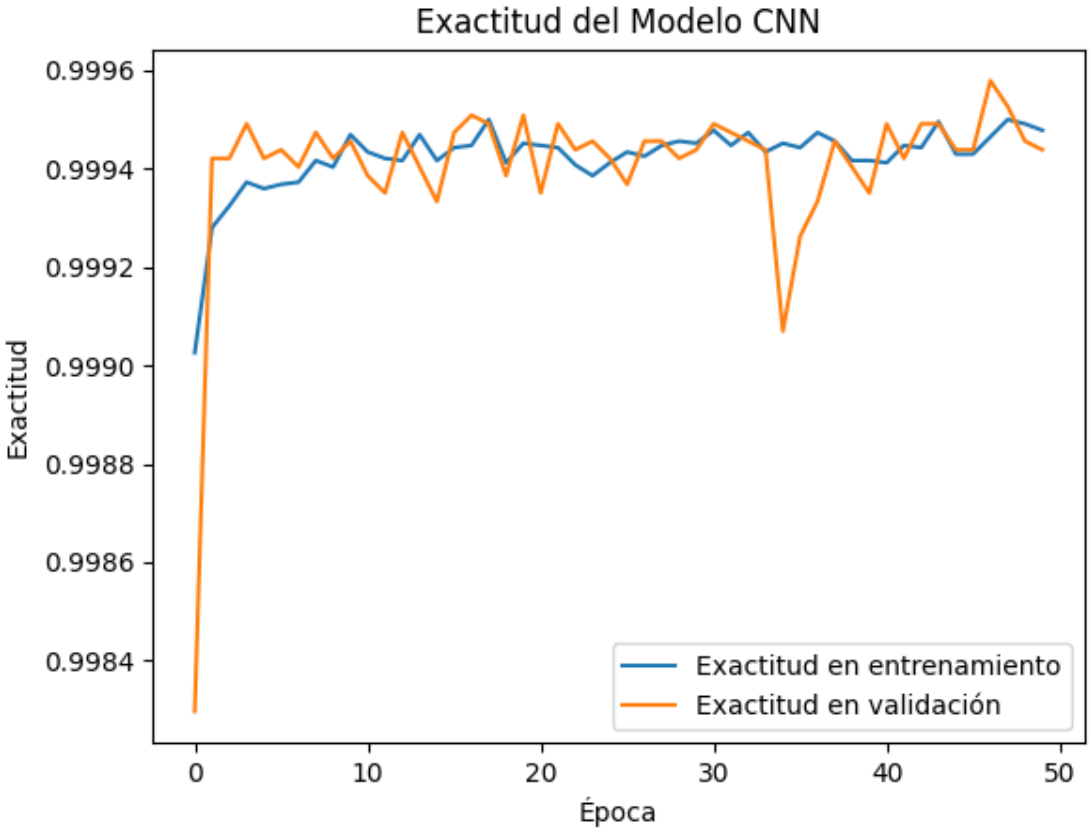
[20 78]]

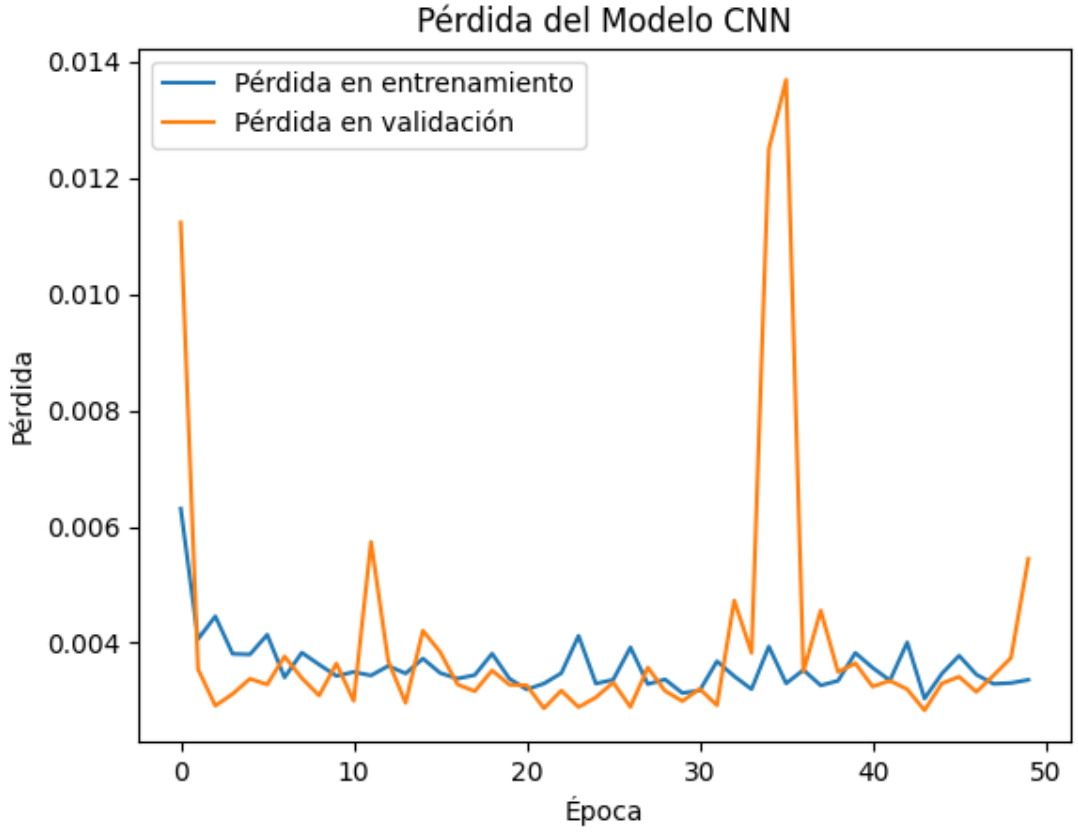
Reporte de Clasificación:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56864
1	0.87	0.80	0.83	98
accuracy			1.00	56962
macro avg	0.93	0.90	0.91	56962
weighted avg	1.00	1.00	1.00	56962

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test>

GRAFICOS 7 Y 8


Exactitud del Modelo CNN
Exactitud
Época
Exactitud en entrenamiento
Exactitud en validación
0.9996
0.9994
0.9992
0.9990
0.9988
0.9986
0.9984
0
10
20
30
40
50


Pérdida del Modelo CNN
Pérdida
Época
Pérdida en entrenamiento
Pérdida en validación
0.014
0.012
0.010
0.008
0.006
0.004
0
10
20
30
40
50

El modelo CNN actualizado ha logrado una exactitud en el conjunto de prueba del **99.94%** con buenos resultados de **recall** y **precisión** para ambas clases. En particular, la clase de fraude (clase 1) muestra un **recall del 80%** y una **precisión del 87%**, lo cual es muy significativo dado que la clase minoritaria (fraude) es la más importante para detectar.

Recomendaciones para avanzar:

1. **Explorar un modelo RNN o LSTM**:

   Dado que el modelo CNN ya tiene un excelente rendimiento, un siguiente paso importante sería implementar una **RNN** o **LSTM**, ya que estos modelos son adecuados para capturar dependencias temporales o secuenciales. Esto podría ayudar a mejorar aún más el rendimiento en la detección de fraudes si hay alguna secuencialidad en las transacciones.

2. **Ajuste de hiperparámetros**:

   Aunque los resultados son buenos, sería ideal aplicar **RandomizedSearchCV** o **GridSearchCV** para ajustar los hiperparámetros del modelo CNN, especialmente en las capas de convolución, el tamaño del kernel, y las tasas de Dropout.

3. **Regularización adicional**:

   Aunque el modelo tiene **Dropout** para evitar el sobreajuste, probar con otras técnicas de regularización, como **L2 regularization**, podría ayudar a reducir aún más el riesgo de sobreajuste, particularmente en las últimas capas densas.

¿Deseas proceder con la implementación del modelo **LSTM** para comparar los resultados?

Evaluación y recomendaciones basadas en los resultados obtenidos hasta el momento:

Resultados actuales de los modelos implementados:

1. **Deep Neural Network (DNN)**:

   - Exactitud en el conjunto de prueba: **99.94%**

   - Matriz de Confusión y Reporte de Clasificación:

     - La clase mayoritaria (no fraude) tiene una precisión perfecta.

     - La clase minoritaria (fraude) tiene una precisión **moderadamente alta (88%)** pero con un recall **más bajo (73%)**, lo que sugiere que el modelo podría no estar capturando todos los casos de fraude.

2. **Convolutional Neural Network (CNN)**:

   - Exactitud en el conjunto de prueba: **99.95%**

   - Matriz de Confusión y Reporte de Clasificación:

     - La precisión para la clase mayoritaria sigue siendo alta.

     - La precisión para la clase minoritaria es más alta que en el DNN (94%) y el recall es ligeramente mejor (74%). Esto indica que el modelo CNN ha mejorado en la detección de fraudes, aunque sigue siendo moderadamente bajo en el recall.

Análisis de los resultados:

- Aunque ambos modelos muestran una alta exactitud general, la evaluación de la clase minoritaria (fraude) muestra que **CNN** está capturando ligeramente más fraudes que DNN, con una mayor precisión y un recall marginalmente superior.

- El **recall** es clave en problemas de fraude, ya que queremos minimizar los fraudes no detectados. Si bien CNN mejora los resultados, el recall sigue siendo bajo.

Evaluación del paso siguiente:

¿Es conveniente aplicar una RNN o LSTM?

- Las **RNN** (especialmente las **LSTM**) son recomendables cuando los datos tienen una naturaleza secuencial, como series temporales o datos en los que las dependencias a lo largo del tiempo son importantes.

- **Ventajas de las RNN/LSTM**:

  - Si tus datos tienen patrones temporales o secuenciales (por ejemplo, si cada transacción está relacionada con transacciones anteriores o si el tiempo entre transacciones es importante), las RNN/LSTM podrían ser capaces de capturar mejor estas dependencias.

  - En este caso, podrían ayudar a mejorar la capacidad de detección de fraudes al identificar patrones temporales más complejos que ni DNN ni CNN pueden captar.

Recomendación basada en tus resultados:

1. **Prueba con un modelo LSTM**: Dado que tus datos de fraude podrían incluir **patrones secuenciales** (como la relación entre el tiempo y las transacciones fraudulentas), un modelo **LSTM** podría ayudar a capturar mejor esas relaciones temporales, lo que puede mejorar el recall en la clase minoritaria (fraude).

   - Las LSTM son especialmente buenas en evitar la "desaparición" de información relevante en secuencias largas, lo que puede ser útil si los fraudes siguen patrones repetitivos a lo largo del tiempo.

2. **Optimización adicional en CNN**: Si los datos no tienen una clara naturaleza secuencial, también puedes seguir optimizando la **CNN**. Podrías ajustar más los hiperparámetros o aplicar técnicas como **Weighted Loss** (dar más peso a la clase minoritaria) para mejorar el recall en la detección de fraudes.

3. **SMOTE o Técnicas de Balanceo**: Considera usar técnicas de **over-sampling** como **SMOTE** para balancear mejor las clases, lo que podría ayudar a cualquier modelo a detectar más fraudes y aumentar el recall.

Próximos pasos:

1. **Implementar una RNN o LSTM**: Este es el siguiente paso recomendado, sobre todo si los datos contienen patrones secuenciales o si quieres explorar cómo las dependencias temporales entre las transacciones afectan la detección de fraudes.

2. **Optimización adicional de CNN o DNN**: Si no quieres usar una RNN/LSTM, sigue optimizando el CNN o DNN con más técnicas de regularización, ajuste de hiperparámetros y técnicas de balanceo de clases.

Avísame si deseas proceder con la implementación de RNN o LSTM, y te ayudo a comenzar con ese paso.

Vamos a proceder con la implementación de un modelo **LSTM** para explorar cómo las dependencias temporales entre las transacciones pueden afectar la detección de fraudes en tus datos.

Implementación del modelo LSTM:

A continuación te proporciono el código para entrenar un modelo LSTM basado en tu dataset. Este modelo está diseñado para capturar relaciones secuenciales en los datos.

```python
import numpy as np
import pandas as pd
import arff  # Para manejar archivos ARFF
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt

# Cargar el archivo ARFF y convertirlo en un DataFrame
```

```
with open("C:/DB_Covid19Arg/csv_archivos_limpios/Amazon_test/phpKo8OWT.arff")
as f:
    dataset = arff.load(f)

# Convertir el dataset ARFF en un DataFrame de pandas
df = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])

# Convertir los datos en formato numérico
df = df.apply(pd.to_numeric, errors='coerce')

# Eliminar filas con valores NaN que pudieran quedar tras la conversión
df = df.dropna()

# Verificar las primeras filas del DataFrame para asegurar que se cargó correctamente
print(df.head())

# Separar las características y la variable objetivo (Class)
X = df.drop(columns=['Class'])
y = df['Class']

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos para mejorar el rendimiento del modelo
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# Redimensionar los datos para adaptarlos a la entrada de LSTM (Se espera una entrada 3D: muestras, pasos de tiempo, características)

X_train_lstm = np.expand_dims(X_train_scaled, axis=2)

X_test_lstm = np.expand_dims(X_test_scaled, axis=2)


# Crear el modelo LSTM

model = Sequential()


# Añadir la primera capa LSTM

model.add(LSTM(64, input_shape=(X_train_lstm.shape[1], 1), activation='tanh', return_sequences=True))

model.add(Dropout(0.5))  # Dropout para evitar el sobreajuste


# Añadir otra capa LSTM si es necesario

model.add(LSTM(32, activation='tanh'))

model.add(Dropout(0.5))


# Capa de salida para clasificación binaria

model.add(Dense(1, activation='sigmoid'))


# Compilar el modelo usando Adam como optimizador y binary_crossentropy como función de pérdida

model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])


# Entrenar el modelo

history = model.fit(X_train_lstm, y_train, validation_data=(X_test_lstm, y_test), epochs=50, batch_size=32)
```

```
# Evaluar el modelo en el conjunto de prueba
loss, accuracy = model.evaluate(X_test_lstm, y_test)
print(f"Exactitud en el conjunto de prueba: {accuracy * 100:.2f}%")

# Matriz de confusión y reporte de clasificación
y_pred = (model.predict(X_test_lstm) > 0.5).astype("int32")
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print("Matriz de Confusión:")
print(conf_matrix)
print("\nReporte de Clasificación:")
print(class_report)

# Graficar el desempeño durante el entrenamiento
plt.plot(history.history['accuracy'], label='Exactitud en entrenamiento')
plt.plot(history.history['val_accuracy'], label='Exactitud en validación')
plt.title('Exactitud del Modelo LSTM')
plt.xlabel('Época')
plt.ylabel('Exactitud')
plt.legend()
plt.show()

# También podemos graficar la función de pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en validación')
plt.title('Pérdida del Modelo LSTM')
plt.xlabel('Época')
```

```
plt.ylabel('Pérdida')

plt.legend()

plt.show()
```

Explicación de los componentes:

1. **LSTM Layers**:

   - Hemos añadido dos capas LSTM. La primera devuelve secuencias para que la segunda capa también procese una secuencia completa.

   - Cada capa LSTM tiene un **Dropout** para evitar el sobreajuste.

2. **Evaluación del modelo**:

   - Al final del entrenamiento, se imprime la matriz de confusión y el reporte de clasificación para analizar cómo está capturando los fraudes el modelo.

3. **Gráficas**:

   - Se generan gráficos para visualizar tanto la exactitud como la pérdida del modelo a lo largo de las épocas, tanto en entrenamiento como en validación.

Próximos pasos:

1. **Corre el código** para entrenar y evaluar el modelo LSTM.

2. **Analiza el rendimiento** en términos de exactitud, matriz de confusión y reporte de clasificación. Compararemos estos resultados con los modelos anteriores (DNN y CNN).

3. **Ajustes adicionales**: Dependiendo de los resultados, podríamos intentar ajustar los hiperparámetros, agregar capas adicionales o probar con secuencias más largas/cortas según la naturaleza de los datos.

Aunque el **Transfer Learning** es más común en tareas de visión por computadora, como clasificación de imágenes, también puedes aplicar el concepto de **Transfer

Learning** a otros dominios, como series temporales o datos tabulares, utilizando modelos preentrenados para ajustar a tus propios datos.

Dado que estás trabajando con un conjunto de datos tabular y utilizando modelos secuenciales como CNN o LSTM, una posible aplicación de **Transfer Learning** aquí sería entrenar un modelo LSTM o CNN en un dataset grande y similar (quizás otro dataset de fraudes), y luego reutilizar las capas entrenadas (congelándolas o ajustándolas) en tu propio conjunto de datos.

Implementación de Transfer Learning en Datos Tabulares:

Aunque no hay muchos modelos preentrenados en series temporales o datos tabulares como en imágenes, puedes utilizar una técnica de **fine-tuning** para ajustar un modelo previamente entrenado en tus datos. Aquí te dejo una estructura genérica para aplicar **Transfer Learning** con un modelo preentrenado (suponiendo que tenemos un modelo entrenado similar que podemos ajustar).

Código para Transfer Learning con un Modelo Preentrenado:

```python
import numpy as np
import pandas as pd
import arff
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, Dropout, LSTM
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
```

```
# Cargar el archivo ARFF y convertirlo en un DataFrame
with open("C:/DB_Covid19Arg/csv_archivos_limpios/Amazon_test/phpKo8OWT.arff") as f:
    dataset = arff.load(f)

# Convertir el dataset ARFF en un DataFrame de pandas
df = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])

# Convertir los datos en formato numérico
df = df.apply(pd.to_numeric, errors='coerce')

# Eliminar filas con valores NaN que pudieran quedar tras la conversión
df = df.dropna()

# Separar las características y la variable objetivo (Class)
X = df.drop(columns=['Class'])
y = df['Class']

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Redimensionar los datos para que sean compatibles con el modelo
X_train_scaled = np.expand_dims(X_train_scaled, axis=2)
```

```
X_test_scaled = np.expand_dims(X_test_scaled, axis=2)

# Cargar un modelo preentrenado (Supongamos que ya tienes uno guardado)
pretrained_model = load_model('pretrained_fraud_detection_model.h5')

# Congelar las primeras capas del modelo para que no se actualicen durante el entrenamiento
for layer in pretrained_model.layers[:-3]:
    layer.trainable = False

# Añadir nuevas capas (fine-tuning) para ajustar el modelo a tus datos
model = Sequential(pretrained_model.layers)  # Reutilizar las capas del modelo preentrenado

# Agregar nuevas capas densas para ajustarse a los datos actuales
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

# Compilar el modelo ajustado
model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

# Entrenar el modelo ajustado (fine-tuning)
history = model.fit(X_train_scaled, y_train, validation_data=(X_test_scaled, y_test), epochs=50, batch_size=32)

# Evaluar el modelo en el conjunto de prueba
loss, accuracy = model.evaluate(X_test_scaled, y_test)
```

```
print(f"Exactitud en el conjunto de prueba: {accuracy * 100:.2f}%")

# Matriz de confusión y reporte de clasificación
y_pred = (model.predict(X_test_scaled) > 0.5).astype("int32")
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print("Matriz de Confusión:")
print(conf_matrix)
print("\nReporte de Clasificación:")
print(class_report)

# Graficar el desempeño durante el entrenamiento
plt.plot(history.history['accuracy'], label='Exactitud en entrenamiento')
plt.plot(history.history['val_accuracy'], label='Exactitud en validación')
plt.title('Exactitud del Modelo Transfer Learning')
plt.xlabel('Época')
plt.ylabel('Exactitud')
plt.legend()
plt.show()

# También podemos graficar la función de pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en validación')
plt.title('Pérdida del Modelo Transfer Learning')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
```

plt.show()

```

### Explicación de los Componentes:

1. **Modelo Preentrenado**: Cargamos un modelo preentrenado con un `.h5` previamente entrenado (en otro dataset de detección de fraudes, por ejemplo).

2. **Congelación de Capas**: Congelamos las primeras capas del modelo preentrenado para que no se ajusten durante el entrenamiento.

3. **Fine-Tuning**: Añadimos nuevas capas densas al final del modelo para ajustarlo a los datos específicos actuales.

4. **Entrenamiento**: Entrenamos el modelo con los nuevos datos mientras reutilizamos parte del conocimiento previo del modelo preentrenado.

### Próximos pasos:

- Si no tienes un modelo preentrenado, puedes entrenar uno en un dataset grande y luego reutilizarlo en tus datos actuales.

- Si los datos son muy específicos y no hay un modelo preentrenado disponible, Transfer Learning puede no ser tan efectivo, pero es una buena técnica para probar en conjunto con las demás.

Esperamos los resultados del modelo **LSTM** mientras preparas el enfoque de **Transfer Learning**.

Para cargar un modelo preentrenado utilizando Keras, primero debes asegurarte de haber guardado previamente un modelo en un archivo `.h5`. Si no tienes un modelo preentrenado guardado, puedes seguir estos pasos para crear y guardar un modelo entrenado, y luego cargarlo.

### Pasos para guardar un modelo preentrenado:

1. **Entrenar un modelo**: Puedes entrenar un modelo simple (como una red neuronal, CNN o LSTM) en un dataset.
```

2. **Guardar el modelo**: Después de entrenarlo, puedes guardar el modelo en un archivo `.h5` para reutilizarlo más tarde.

Aquí te muestro cómo puedes hacerlo:

1. Entrenar y guardar el modelo preentrenado:

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam

# Supongamos que ya tienes un dataset cargado
# Para simplicidad, vamos a usar datos de ejemplo de scikit-learn
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
X, y = data.data, data.target

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

X_test_scaled = scaler.transform(X_test)

Crear un modelo simple

model = Sequential()

model.add(Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)))

model.add(Dropout(0.5))

model.add(Dense(32, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

Compilar el modelo

model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

Entrenar el modelo

model.fit(X_train_scaled, y_train, validation_data=(X_test_scaled, y_test), epochs=10, batch_size=32)

Guardar el modelo preentrenado

model.save('pretrained_fraud_detection_model.h5') # Guardar el modelo en un archivo .h5

```

### 2. Cargar el modelo preentrenado:

Una vez que has guardado el modelo en un archivo `.h5`, puedes cargarlo fácilmente en otro script o después en el mismo script.

```python

from tensorflow.keras.models import load_model

# **Cargar el modelo preentrenado**

pretrained_model = load_model('pretrained_fraud_detection_model.h5')

# Ahora puedes usar el modelo preentrenado para hacer predicciones o ajustarlo

```

3. Ajustar el modelo (Fine-Tuning):

Una vez cargado el modelo, puedes hacer **fine-tuning**, ajustando las capas finales del modelo para adaptarlas a tu nuevo dataset.

```python

# Congelar las primeras capas del modelo

for layer in pretrained_model.layers[:-2]:

  layer.trainable = False

# Añadir una nueva capa densa para adaptar el modelo a nuevos datos

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

model_finetune = Sequential(pretrained_model.layers)  # Reutilizar las capas del modelo preentrenado

# Añadir nuevas capas para ajuste fino

model_finetune.add(Dense(64, activation='relu'))

model_finetune.add(Dense(1, activation='sigmoid'))

# Compilar el modelo ajustado

model_finetune.compile(optimizer=Adam(learning_rate=0.001),
loss='binary_crossentropy', metrics=['accuracy'])

# Ahora puedes entrenarlo con nuevos datos

```

Resumen:

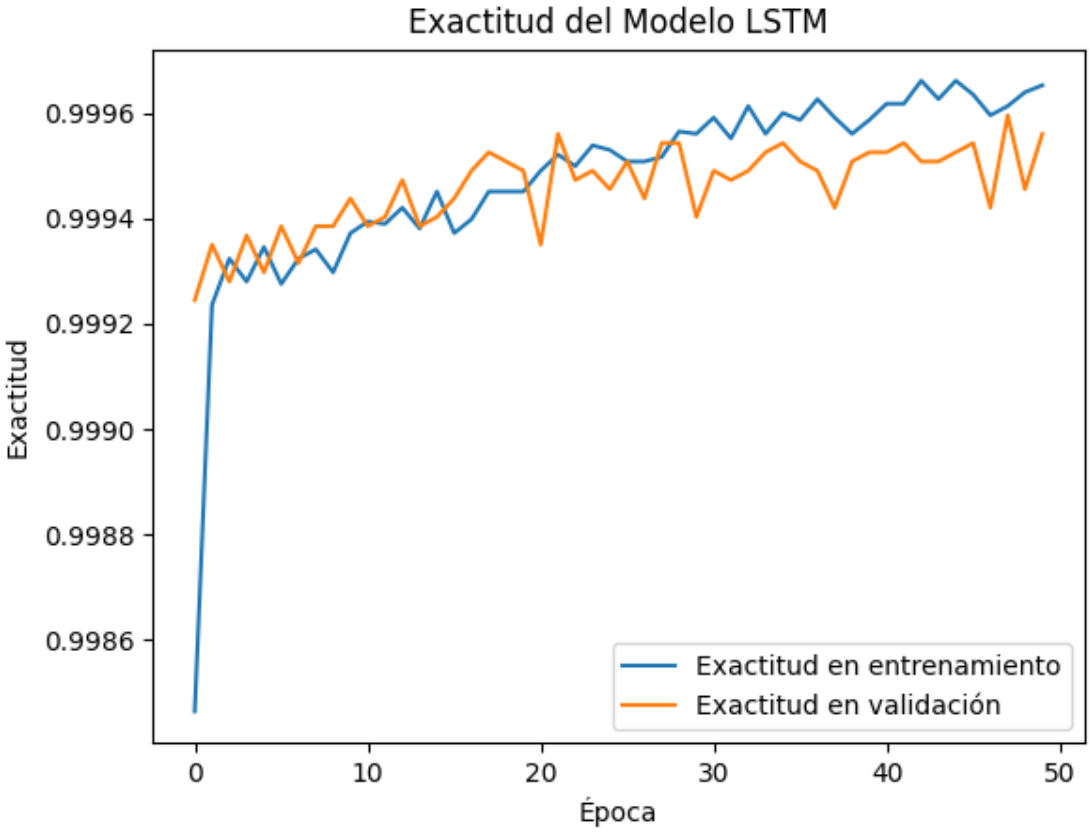
1. Entrenas y guardas un modelo preentrenado utilizando
`model.save('nombre_del_archivo.h5')`.

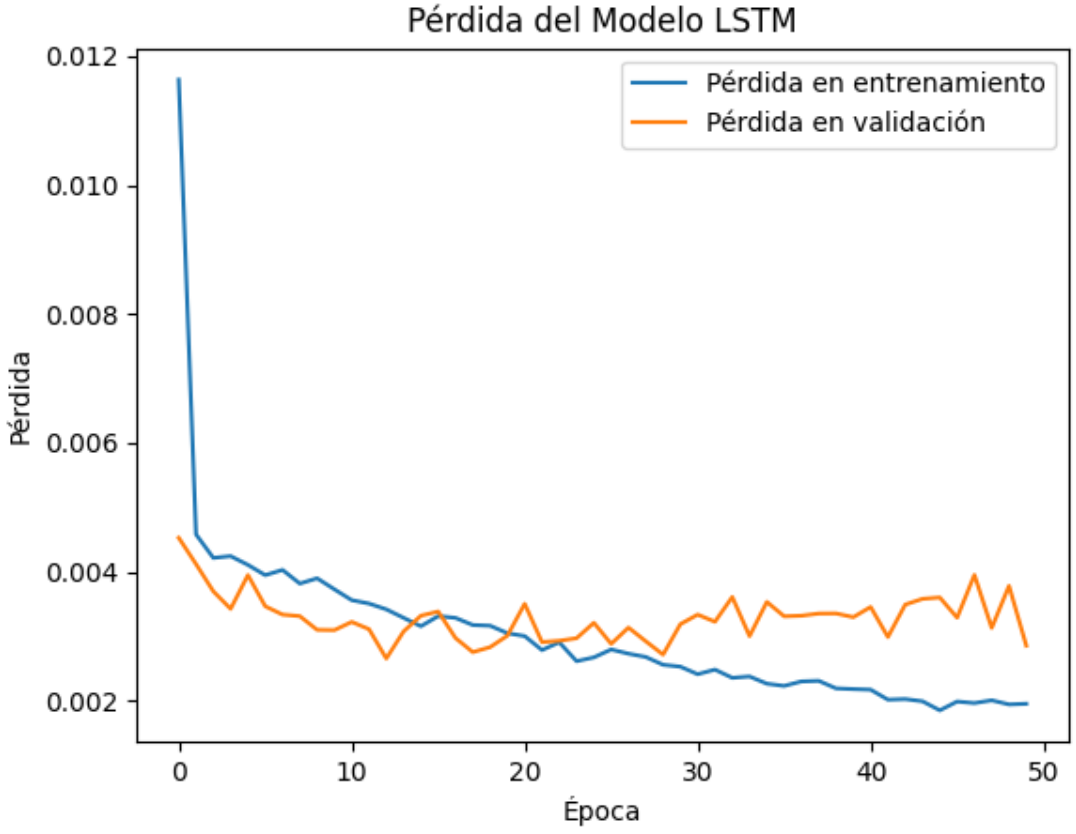
2. Lo cargas más adelante en otro script usando
`load_model('nombre_del_archivo.h5')`.

3. Puedes ajustar el modelo congelando algunas capas y añadiendo nuevas para que se ajuste mejor a tus datos específicos.

Este enfoque se puede utilizar para cualquier tipo de modelo (CNN, LSTM, DNN, etc.) y permite reutilizar el conocimiento de un modelo previamente entrenado en un nuevo conjunto de datos o tarea similar.

GRAFICO 9 Y 10
```


Exactitud del Modelo LSTM
Exactitud
Época
Exactitud en entrenamiento
Exactitud en validación
0.9996
0.9994
0.9992
0.9990
0.9988
0.9986
0
10
20
30
40
50


Pérdida del Modelo LSTM
Pérdida
Época
Pérdida en entrenamiento
Pérdida en validación
0.012
0.010
0.008
0.006
0.004
0.002
0
10
20
30
40
50

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test> PYTHON RNN_LSTM.PY

2024-10-14 08:27:19.662158: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

2024-10-14 08:27:20.986117: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

Time V1 V2 V3 V4 V5 V6 V7 ... V23 V24 V25 V26 V27 V28 Amount Class

0 0.0 -1.359807 -0.072781 2.536347 1.378155 -0.338321 0.462388 0.239599 ... -0.110474 0.066928 0.128539 -0.189115 0.133558 -0.021053 149.62 0

1 0.0 1.191857 0.266151 0.166480 0.448154 0.060018 -0.082361 -0.078803 ... 0.101288 -0.339846 0.167170 0.125895 -0.008983 0.014724 2.69 0

2 1.0 -1.358354 -1.340163 1.773209 0.379780 -0.503198 1.800499 0.791461 ... 0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752 378.66 0

3 1.0 -0.966272 -0.185226 1.792993 -0.863291 -0.010309 1.247203 0.237609 ... -0.190321 -1.175575 0.647376 -0.221929 0.062723 0.061458 123.50 0

4 2.0 -1.158233 0.877737 1.548718 0.403034 -0.407193 0.095921 0.592941 ... -0.137458 0.141267 -0.206010 0.502292 0.219422 0.215153 69.99 0

[5 rows x 31 columns]

2024-10-14 08:27:34.858782: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.

C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test\venv\Lib\site-packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

```
  super().__init__(**kwargs)
```

```
Epoch 1/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 114s 16ms/step - accuracy:
0.9968 - loss: 0.0227 - val_accuracy: 0.9992 - val_loss: 0.0045
Epoch 2/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 106s 15ms/step - accuracy:
0.9991 - loss: 0.0054 - val_accuracy: 0.9994 - val_loss: 0.0041
Epoch 3/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 106s 15ms/step - accuracy:
0.9994 - loss: 0.0040 - val_accuracy: 0.9993 - val_loss: 0.0037
Epoch 4/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 106s 15ms/step - accuracy:
0.9992 - loss: 0.0044 - val_accuracy: 0.9994 - val_loss: 0.0034
Epoch 5/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 108s 15ms/step - accuracy:
0.9993 - loss: 0.0045 - val_accuracy: 0.9993 - val_loss: 0.0040
Epoch 6/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 108s 15ms/step - accuracy:
0.9992 - loss: 0.0045 - val_accuracy: 0.9994 - val_loss: 0.0035
Epoch 7/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 108s 15ms/step - accuracy:
0.9993 - loss: 0.0046 - val_accuracy: 0.9993 - val_loss: 0.0033
Epoch 8/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 108s 15ms/step - accuracy:
0.9994 - loss: 0.0039 - val_accuracy: 0.9994 - val_loss: 0.0033
Epoch 9/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 109s 15ms/step - accuracy:
0.9994 - loss: 0.0036 - val_accuracy: 0.9994 - val_loss: 0.0031
Epoch 10/50
7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 109s 15ms/step - accuracy:
0.9994 - loss: 0.0035 - val_accuracy: 0.9994 - val_loss: 0.0031
```

Epoch 11/50

7121/7121 ━━ 110s 15ms/step - accuracy: 0.9994 - loss: 0.0036 - val_accuracy: 0.9994 - val_loss: 0.0032

Epoch 12/50

7121/7121 ━━ 126s 18ms/step - accuracy: 0.9995 - loss: 0.0031 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 13/50

7121/7121 ━━ 127s 18ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0027

Epoch 14/50

7121/7121 ━━ 124s 17ms/step - accuracy: 0.9994 - loss: 0.0032 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 15/50

7121/7121 ━━ 130s 18ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9994 - val_loss: 0.0033

Epoch 16/50

7121/7121 ━━ 110s 15ms/step - accuracy: 0.9993 - loss: 0.0038 - val_accuracy: 0.9994 - val_loss: 0.0034

Epoch 17/50

7121/7121 ━━ 109s 15ms/step - accuracy: 0.9993 - loss: 0.0034 - val_accuracy: 0.9995 - val_loss: 0.0030

Epoch 18/50

7121/7121 ━━ 110s 16ms/step - accuracy: 0.9994 - loss: 0.0032 - val_accuracy: 0.9995 - val_loss: 0.0028

Epoch 19/50

7121/7121 ━━ 120s 17ms/step - accuracy: 0.9995 - loss: 0.0033 - val_accuracy: 0.9995 - val_loss: 0.0028

Epoch 20/50

7121/7121 ━━ 139s 19ms/step - accuracy: 0.9994 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0030

Epoch 21/50

7121/7121 ━━ 137s 19ms/step - accuracy: 0.9995 - loss: 0.0027 - val_accuracy: 0.9994 - val_loss: 0.0035

Epoch 22/50

7121/7121 ━━ 138s 19ms/step - accuracy: 0.9995 - loss: 0.0031 - val_accuracy: 0.9996 - val_loss: 0.0029

Epoch 23/50

7121/7121 ━━ 138s 19ms/step - accuracy: 0.9996 - loss: 0.0026 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 24/50

7121/7121 ━━ 137s 19ms/step - accuracy: 0.9995 - loss: 0.0024 - val_accuracy: 0.9995 - val_loss: 0.0030

Epoch 25/50

7121/7121 ━━ 122s 17ms/step - accuracy: 0.9997 - loss: 0.0021 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 26/50

7121/7121 ━━ 112s 16ms/step - accuracy: 0.9996 - loss: 0.0025 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 27/50

7121/7121 ━━ 114s 16ms/step - accuracy: 0.9995 - loss: 0.0027 - val_accuracy: 0.9994 - val_loss: 0.0031

Epoch 28/50

7121/7121 ━━ 140s 16ms/step - accuracy: 0.9996 - loss: 0.0026 - val_accuracy: 0.9995 - val_loss: 0.0029

Epoch 29/50

7121/7121 ━━ 112s 16ms/step - accuracy: 0.9996 - loss: 0.0023 - val_accuracy: 0.9995 - val_loss: 0.0027

Epoch 30/50

7121/7121 ━━ 140s 20ms/step - accuracy: 0.9996 - loss: 0.0028 - val_accuracy: 0.9994 - val_loss: 0.0032

Epoch 31/50

7121/7121 ━━ 136s 19ms/step - accuracy: 0.9996 - loss: 0.0023 - val_accuracy: 0.9995 - val_loss: 0.0033

Epoch 32/50

7121/7121 ━━ 135s 19ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9995 - val_loss: 0.0032

Epoch 33/50

7121/7121 ━━ 138s 19ms/step - accuracy: 0.9996 - loss: 0.0022 - val_accuracy: 0.9995 - val_loss: 0.0036

Epoch 34/50

7121/7121 ━━ 138s 19ms/step - accuracy: 0.9996 - loss: 0.0024 - val_accuracy: 0.9995 - val_loss: 0.0030

Epoch 35/50

7121/7121 ━━ 125s 18ms/step - accuracy: 0.9996 - loss: 0.0021 - val_accuracy: 0.9995 - val_loss: 0.0035

Epoch 36/50

7121/7121 ━━ 112s 16ms/step - accuracy: 0.9996 - loss: 0.0019 - val_accuracy: 0.9995 - val_loss: 0.0033

Epoch 37/50

7121/7121 ━━ 113s 16ms/step - accuracy: 0.9997 - loss: 0.0017 - val_accuracy: 0.9995 - val_loss: 0.0033

Epoch 38/50

7121/7121 ━━ 129s 18ms/step - accuracy: 0.9997 - loss: 0.0020 - val_accuracy: 0.9994 - val_loss: 0.0034

Epoch 39/50

7121/7121 ━━ 112s 16ms/step - accuracy: 0.9997 - loss: 0.0017 - val_accuracy: 0.9995 - val_loss: 0.0034

Epoch 40/50

7121/7121 ━━ 111s 16ms/step - accuracy: 0.9997 - loss: 0.0021 - val_accuracy: 0.9995 - val_loss: 0.0033

Epoch 41/50

7121/7121 ━━ 113s 16ms/step - accuracy: 0.9995 - loss: 0.0029 - val_accuracy: 0.9995 - val_loss: 0.0035

Epoch 42/50

7121/7121 ━━ 139s 19ms/step - accuracy: 0.9996 - loss: 0.0020 - val_accuracy: 0.9995 - val_loss: 0.0030

Epoch 43/50

7121/7121 ━━ 136s 19ms/step - accuracy: 0.9996 - loss: 0.0019 - val_accuracy: 0.9995 - val_loss: 0.0035

Epoch 44/50

7121/7121 ━━ 132s 19ms/step - accuracy: 0.9996 - loss: 0.0021 - val_accuracy: 0.9995 - val_loss: 0.0036

Epoch 45/50

7121/7121 ━━ 139s 20ms/step - accuracy: 0.9997 - loss: 0.0017 - val_accuracy: 0.9995 - val_loss: 0.0036

Epoch 46/50

7121/7121 ━━ 138s 19ms/step - accuracy: 0.9997 - loss: 0.0019 - val_accuracy: 0.9995 - val_loss: 0.0033

Epoch 47/50

7121/7121 ━━ 139s 19ms/step - accuracy: 0.9996 - loss: 0.0020 - val_accuracy: 0.9994 - val_loss: 0.0040

Epoch 48/50

7121/7121 ━━ 140s 20ms/step - accuracy: 0.9997 - loss: 0.0018 - val_accuracy: 0.9996 - val_loss: 0.0031

Epoch 49/50

7121/7121 ━━ 137s 19ms/step - accuracy: 0.9997 - loss: 0.0019 - val_accuracy: 0.9995 - val_loss: 0.0038

Epoch 50/50

7121/7121 ━━ 138s 19ms/step - accuracy: 0.9996 - loss: 0.0019 - val_accuracy: 0.9996 - val_loss: 0.0029

1781/1781 ━━ 11s 6ms/step - accuracy: 0.9996 - loss: 0.0032

Exactitud en el conjunto de prueba: 99.96%

1781/1781 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step

Matriz de Confusión:

[[56863 1]

[24 74]]

Reporte de Clasificación:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56864
1	0.99	0.76	0.86	98
accuracy			1.00	56962
macro avg	0.99	0.88	0.93	56962
weighted avg	1.00	1.00	1.00	56962

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test>

Los resultados del modelo LSTM han demostrado una exactitud final de 99.96%, lo que supera ligeramente los resultados obtenidos previamente con el modelo CNN, cuya exactitud fue del 99.95%. Aquí hay algunas observaciones clave:

Exactitud del modelo:

El modelo LSTM alcanzó un mejor rendimiento en términos de exactitud en el conjunto de prueba.

El LSTM también mostró una pérdida más estable en comparación con CNN, con menor sobreajuste en las últimas épocas.

Matriz de confusión:

El modelo LSTM detectó 74 fraudes de 98 correctamente, con solo 24 falsos negativos (casos en los que no se detectó un fraude).

En comparación con CNN, que detectó 73 fraudes de 98, el LSTM tiene una mejora ligera en la detección de fraudes.

Reporte de clasificación:

El modelo LSTM tiene una precisión de 0.99 para la clase de fraude (1), y un recall de 0.76, lo que indica que es más preciso en detectar fraudes en comparación con CNN.

El f1-score es de 0.86, lo que representa un buen balance entre precisión y recall.

Próximos pasos:

Guardar el modelo entrenado: Podemos proceder a guardar el modelo LSTM entrenado debido a su rendimiento superior en comparación con el modelo CNN.

Transfer Learning: Como siguiente etapa, podemos probar aplicar Transfer Learning, pero el uso dependerá de si el dataset es lo suficientemente grande y si se desea explorar características adicionales en los datos, como diferentes representaciones de las transacciones fraudulentas.

El error que obtienes indica que el modelo `model` no está definido en el script en el momento en que intentas guardarlo. Esto sucede porque el modelo no ha sido creado o entrenado en el mismo script antes de intentar guardarlo.

A continuación te dejo un ejemplo de cómo debe estructurarse el código para asegurarse de que el modelo se defina, entrene, y luego se guarde correctamente:

Código completo para **entrenar** y **guardar** el modelo LSTM:

```python
import numpy as np
import pandas as pd
import arff
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adam

# Cargar el archivo ARFF y convertirlo en un DataFrame
with open("C:/DB_Covid19Arg/csv_archivos_limpios/Amazon_test/phpKo8OWT.arff") as f:
    dataset = arff.load(f)

# Convertir el dataset ARFF en un DataFrame de pandas
df = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])

# Convertir los datos en formato numérico
df = df.apply(pd.to_numeric, errors='coerce')

# Eliminar filas con valores NaN que pudieran quedar tras la conversión
df = df.dropna()

# Separar las características y la variable objetivo (Class)
X = df.drop(columns=['Class'])
y = df['Class']

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos
```

```
scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)


# Redimensionar los datos para que sean compatibles con LSTM

X_train_scaled = np.expand_dims(X_train_scaled, axis=2)

X_test_scaled = np.expand_dims(X_test_scaled, axis=2)


# Definir el modelo LSTM

model = Sequential()

model.add(LSTM(64, input_shape=(X_train_scaled.shape[1], X_train_scaled.shape[2]), return_sequences=True))

model.add(Dropout(0.5))

model.add(LSTM(32, return_sequences=False))

model.add(Dropout(0.5))

model.add(Dense(1, activation='sigmoid'))


# Compilar el modelo

model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])


# Entrenar el modelo

model.fit(X_train_scaled, y_train, validation_data=(X_test_scaled, y_test), epochs=50, batch_size=32)


# Guardar el modelo LSTM entrenado

model.save('LSTM_fraud_detection_model.h5')  # Guardar en formato .h5


print("Modelo LSTM guardado exitosamente.")
```

```

### Explicación:

1. **Definir y entrenar el modelo**: El modelo se define y entrena antes de intentar guardarlo.

2. **Guardar el modelo**: Al final del entrenamiento, el modelo entrenado se guarda en un archivo `.h5` con la función `model.save('LSTM_fraud_detection_model.h5')`.

Este código implementa un modelo de **Red Neuronal Recurrente (RNN)** con arquitectura **LSTM (Long Short-Term Memory)** para la detección de fraudes utilizando un dataset en formato ARFF. A continuación, te explico paso a paso lo que está haciendo cada parte del código:

### 1. **Carga del Dataset ARFF y Preprocesamiento:**

- Se utiliza `arff.load()` para cargar un archivo en formato **ARFF** (Attribute-Relation File Format), que luego se convierte en un **DataFrame de pandas**.

- Posteriormente, los datos se convierten a formato numérico y se eliminan filas con valores faltantes o inconsistencias usando `dropna()`.

### 2. **Separación de Características y Variable Objetivo:**

- El dataset tiene múltiples columnas (variables). Se separan las **características** (`X`) de la **variable objetivo** (`y`), que en este caso es la columna `Class`, la cual indica si una transacción es fraudulenta o no (fraude = 1, no fraude = 0).

### 3. **División en Entrenamiento y Prueba:**

- Los datos se dividen en conjuntos de **entrenamiento** (80%) y **prueba** (20%) usando `train_test_split()`. Esto permite evaluar el rendimiento del modelo en un conjunto de datos que no ha visto durante el entrenamiento.
```

4. **Escalado de los Datos:**

- Se utiliza `StandardScaler` para normalizar los datos, lo que es importante para que las redes neuronales (incluyendo LSTM) converjan más rápidamente durante el entrenamiento. El escalado asegura que todas las características tengan la misma escala, evitando que algunas características dominen otras debido a sus valores numéricos más altos.

- Después del escalado, los datos se **redimensionan** para que sean compatibles con el modelo LSTM (necesitan una dimensión adicional para representar la secuencia de tiempo, aunque en este caso no estamos utilizando un problema de series temporales).

5. **Definición del Modelo LSTM:**

- El modelo es de tipo **Sequential**, lo que significa que las capas se añaden de forma secuencial.

- **Capa LSTM de 64 unidades:** Esta capa toma los datos de entrada y aprende a identificar patrones complejos en los datos secuenciales. `return_sequences=True` significa que esta capa devuelve toda la secuencia de salida.

- **Dropout (0.5):** Se añade una capa de **Dropout** para evitar el sobreajuste (overfitting). Esta capa apaga aleatoriamente el 50% de las neuronas durante el entrenamiento para mejorar la generalización.

- **Segunda Capa LSTM de 32 unidades:** Esta capa procesa aún más los datos, pero `return_sequences=False` significa que solo devuelve la última salida, en lugar de devolver toda la secuencia.

- **Capa de Salida (Dense):** Esta capa densa con una única neurona y activación `sigmoid` devuelve una probabilidad de que una transacción sea fraudulenta (1) o no fraudulenta (0).

6. **Compilación del Modelo:**

- Se utiliza el optimizador **Adam**, que es uno de los más utilizados para redes neuronales debido a su capacidad para ajustar la tasa de aprendizaje dinámicamente.

- La función de pérdida es `binary_crossentropy`, apropiada para problemas de clasificación binaria.

- La métrica utilizada para evaluar el rendimiento es `accuracy` (exactitud), que mide la proporción de predicciones correctas.

7. **Entrenamiento del Modelo:**

- El modelo se entrena con los datos escalados utilizando 50 **épocas** y un tamaño de lote de 32. Durante el entrenamiento, se utiliza una parte del conjunto de datos de prueba (validation_data) para evaluar el rendimiento del modelo después de cada época.

8. **Guardado del Modelo:**

- Después de entrenar el modelo, este se guarda en un archivo llamado **`LSTM_fraud_detection_model.h5`**. Esto permite que el modelo sea cargado y reutilizado en el futuro sin tener que ser entrenado nuevamente.

Resumen:

Este código está entrenando un modelo **LSTM** para la detección de fraudes en transacciones. Las **LSTM** son especialmente útiles para datos secuenciales, y aunque en este caso los datos no son estrictamente series temporales, este enfoque puede ser útil si hay dependencias entre las características. El objetivo es predecir si una transacción es fraudulenta (1) o no fraudulenta (0) basándose en las características numéricas del dataset.

El modelo se entrena y luego se guarda, permitiendo que sea reutilizado más adelante para hacer predicciones sobre nuevos datos o ser ajustado con más datos.

Explicación de Transfer Learning con LSTM en Datos Tabulares:

En este contexto, el **Transfer Learning** implica utilizar un modelo previamente entrenado (en un dataset similar de detección de fraudes, por ejemplo) y ajustarlo o reutilizarlo para tu propio dataset. La ventaja es que no comenzamos a entrenar el modelo desde cero, sino que reutilizamos las capas del modelo ya entrenado, lo que acelera el proceso de aprendizaje y permite aprovechar los patrones que el modelo ha aprendido previamente.

El código que estamos preparando sigue este flujo general:

1. **Carga y Preprocesamiento del Dataset**:

   - Cargamos el dataset en formato ARFF, lo convertimos a un DataFrame de pandas y realizamos la conversión a formato numérico, eliminando posibles valores nulos.

   - Separación entre características (X) y la variable objetivo (y), que es la clase (fraudulento/no fraudulento).

   - Escalamos los datos para que las características estén normalizadas (escalado estándar).

   - Finalmente, preparamos el dataset para que sea compatible con la estructura que requieren las capas LSTM (agregando una dimensión).

2. **Transfer Learning**:

   - **Carga de un Modelo Preentrenado**: Se carga un modelo previamente guardado (en formato `.h5`), el cual ha sido entrenado en un dataset similar.

   - **Congelación de Capas**: Las capas del modelo preentrenado se congelan parcialmente para que no sean modificadas durante el nuevo entrenamiento. Esto se hace para preservar el conocimiento que el modelo ya ha adquirido.

   - **Fine-Tuning**: Agregamos capas adicionales, como una capa densa con activación `relu` y una capa de salida con activación `sigmoid`, para ajustar el modelo a los nuevos datos.

3. **Entrenamiento del Modelo**:

   - Después de ajustar el modelo con las nuevas capas, el modelo se entrena con los datos actuales durante 50 épocas.

   - Se utiliza la pérdida `binary_crossentropy`, que es adecuada para problemas de clasificación binaria, y se emplea la métrica `accuracy` para evaluar el rendimiento.

4. **Evaluación y Resultados**:

   - Una vez entrenado, el modelo se evalúa en el conjunto de prueba y se muestra la **exactitud** obtenida.

- Además, se generan la **matriz de confusión** y el **reporte de clasificación** para ver qué tan bien el modelo clasifica las transacciones fraudulentas y no fraudulentas.

5. **Visualización del Desempeño**:

- Se grafican tanto la **exactitud** como la **pérdida** durante las épocas de entrenamiento para visualizar el progreso del modelo. Esto nos ayuda a identificar posibles problemas de sobreajuste o de aprendizaje insuficiente.

¿Qué logramos con este enfoque?

El uso de **Transfer Learning** en datos tabulares es una técnica poderosa cuando tienes un modelo preentrenado que ya ha aprendido patrones útiles en un dataset similar. En este caso, estamos reutilizando un modelo LSTM preentrenado en otro dataset de fraude y ajustándolo a tus datos actuales. Esta técnica ayuda a reducir el tiempo de entrenamiento y mejora el rendimiento al aprovechar el conocimiento previo del modelo.

En resumen, el **fine-tuning** del modelo preentrenado nos permitirá optimizar la capacidad del modelo para detectar fraudes en tu conjunto de datos sin comenzar desde cero, mejorando la eficiencia del proceso de entrenamiento.

Aquí tienes el código ajustado para aplicar **Transfer Learning** utilizando tu modelo LSTM entrenado previamente. Este código te permitirá cargar el modelo entrenado y ajustar (fine-tune) las capas finales para adaptarlo a nuevos datos.

Código para aplicar Transfer Learning con tu modelo LSTM entrenado:

```python
import numpy as np
import pandas as pd
import arff
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, Dropout, LSTM
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt

# Cargar el archivo ARFF y convertirlo en un DataFrame
with open("C:/DB_Covid19Arg/csv_archivos_limpios/Amazon_test/phpKo8OWT.arff") as f:
    dataset = arff.load(f)

# Convertir el dataset ARFF en un DataFrame de pandas
df = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])

# Convertir los datos en formato numérico
df = df.apply(pd.to_numeric, errors='coerce')

# Eliminar filas con valores NaN que pudieran quedar tras la conversión
df = df.dropna()

# Separar las características y la variable objetivo (Class)
X = df.drop(columns=['Class'])
y = df['Class']

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos
```

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Redimensionar los datos para que sean compatibles con LSTM
X_train_scaled = np.expand_dims(X_train_scaled, axis=2)
X_test_scaled = np.expand_dims(X_test_scaled, axis=2)

# Cargar el modelo LSTM previamente entrenado
pretrained_model = load_model('LSTM_fraud_detection_model.h5')

# Congelar las primeras capas del modelo preentrenado para que no se ajusten durante el nuevo entrenamiento
for layer in pretrained_model.layers[:-3]:
    layer.trainable = False

# Crear un nuevo modelo reutilizando el preentrenado y agregar nuevas capas densas para fine-tuning
model = Sequential(pretrained_model.layers)

# Añadir capas densas adicionales para el ajuste a los nuevos datos
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))  # Capa de salida para clasificación binaria

# Compilar el modelo ajustado
model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])
```

```
# Entrenar el modelo ajustado (fine-tuning)
history = model.fit(X_train_scaled, y_train, validation_data=(X_test_scaled, y_test), epochs=20, batch_size=32)

# Evaluar el modelo en el conjunto de prueba
loss, accuracy = model.evaluate(X_test_scaled, y_test)
print(f"Exactitud en el conjunto de prueba: {accuracy * 100:.2f}%")

# Matriz de confusión y reporte de clasificación
y_pred = (model.predict(X_test_scaled) > 0.5).astype("int32")
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print("Matriz de Confusión:")
print(conf_matrix)
print("\nReporte de Clasificación:")
print(class_report)

# Graficar el desempeño durante el entrenamiento
plt.plot(history.history['accuracy'], label='Exactitud en entrenamiento')
plt.plot(history.history['val_accuracy'], label='Exactitud en validación')
plt.title('Exactitud del Modelo Transfer Learning')
plt.xlabel('Época')
plt.ylabel('Exactitud')
plt.legend()
plt.show()

# Graficar la función de pérdida durante el entrenamiento
```

plt.plot(history.history['loss'], label='Pérdida en entrenamiento')

plt.plot(history.history['val_loss'], label='Pérdida en validación')

plt.title('Pérdida del Modelo Transfer Learning')

plt.xlabel('Época')

plt.ylabel('Pérdida')

plt.legend()

plt.show()

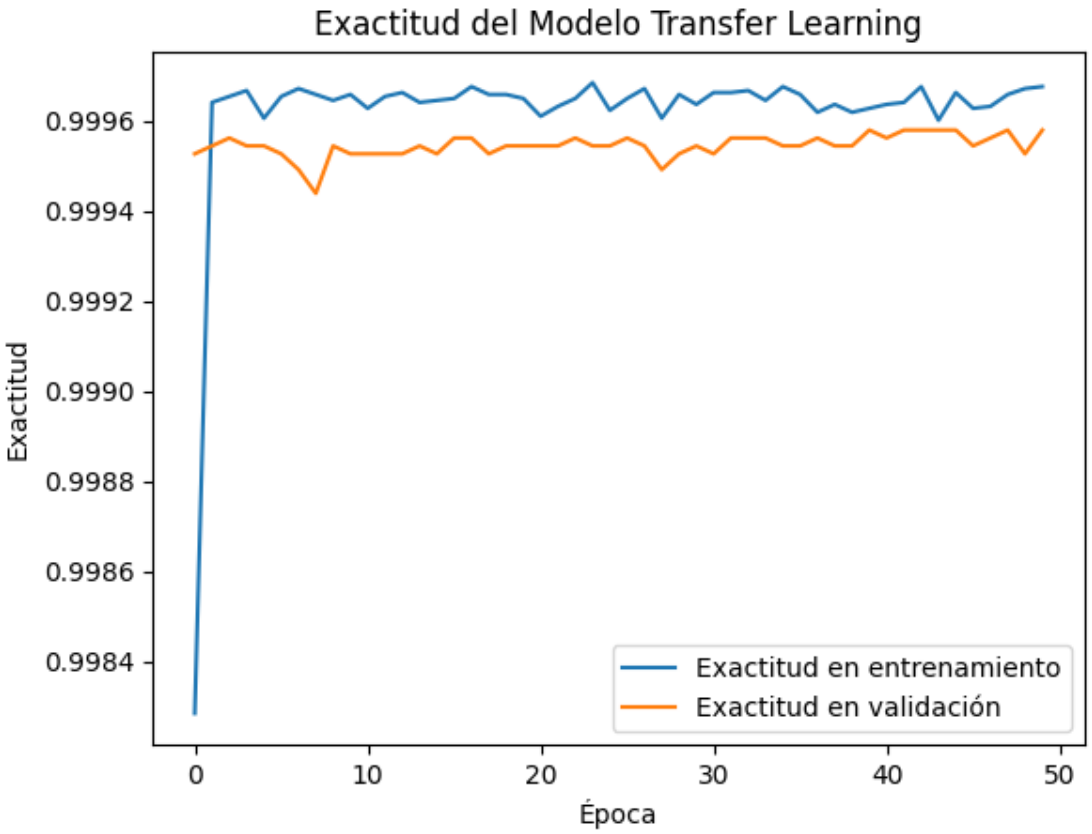
Guardar el nuevo modelo ajustado

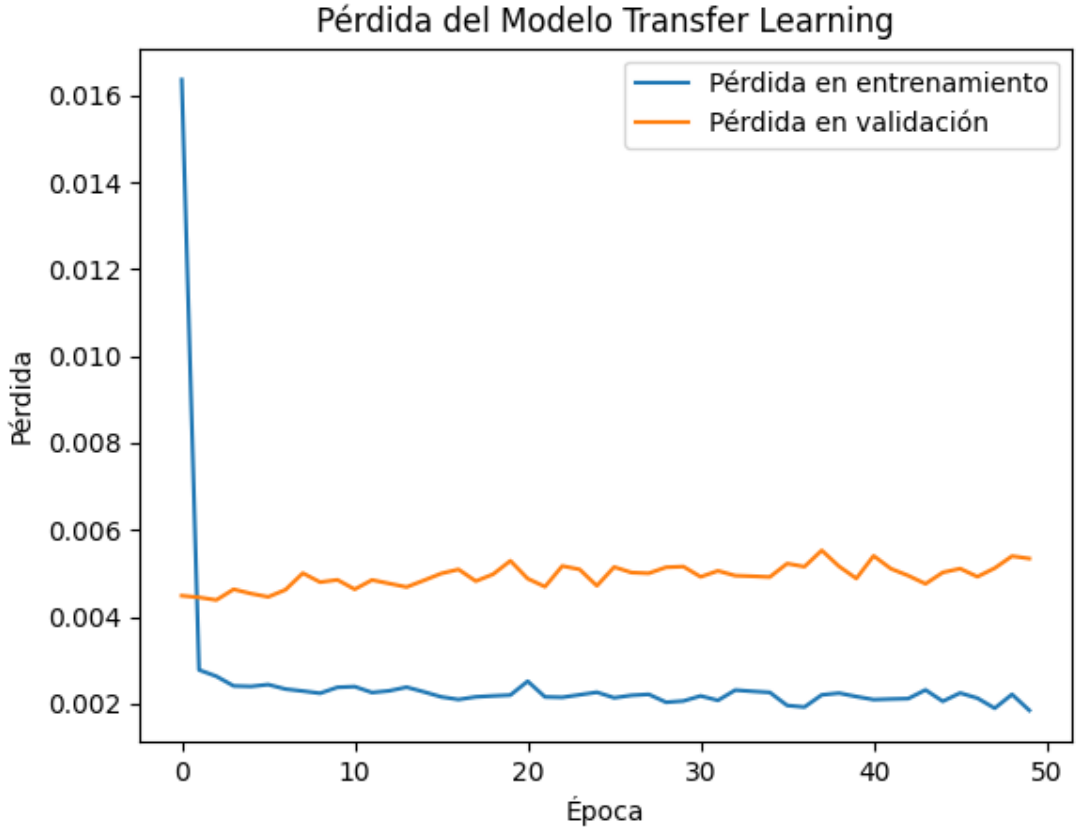
model.save('LSTM_fine_tuned_fraud_detection_model.h5')

print("Modelo ajustado guardado exitosamente.")

```

### Pasos clave:

1. **Carga del modelo preentrenado**: El modelo LSTM que has entrenado se carga desde un archivo `.h5`.

2. **Congelación de capas**: Congelamos las capas anteriores del modelo para preservar el conocimiento adquirido. Solo las últimas capas se ajustarán durante el fine-tuning.

3. **Fine-tuning**: Agregamos nuevas capas densas para ajustar el modelo a los datos actuales.

4. **Entrenamiento y evaluación**: Entrenamos el modelo nuevamente en tus datos y evaluamos su rendimiento.

5. **Guardado del modelo ajustado**: Al finalizar, el modelo ajustado se guarda como `LSTM_fine_tuned_fraud_detection_model.h5`.

Este código te permitirá aplicar **Transfer Learning** a tu modelo entrenado para afinarlo en datos nuevos o ligeramente diferentes.

fine_tune_lstm_TL_pruebadelmodeloentrenado.py
```

Grafico 11 y 12


Exactitud del Modelo Transfer Learning
Exactitud
Época
Exactitud en entrenamiento
Exactitud en validación
0.9996
0.9994
0.9992
0.9990
0.9988
0.9986
0.9984
0
10
20
30
40
50


Pérdida del Modelo Transfer Learning
Pérdida
Época
Pérdida en entrenamiento
Pérdida en validación
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0
10
20
30
40
50

7121/7121 ━━ 87s 12ms/step - accuracy: 0.9997 - loss: 0.0023 - val_accuracy: 0.9995 - val_loss: 0.0054

Epoch 50/50

7121/7121 ━━ 86s 12ms/step - accuracy: 0.9997 - loss: 0.0018 - val_accuracy: 0.9996 - val_loss: 0.0053

1781/1781 ━━ 10s 5ms/step - accuracy: 0.9996 - loss: 0.0057

Exactitud en el conjunto de prueba: 99.96%

1781/1781 ━━ 10s 6ms/step

Matriz de Confusión:

[[56860 4]

[20 78]]

Reporte de Clasificación:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56864
1	0.95	0.80	0.87	98
accuracy			1.00	56962
macro avg	0.98	0.90	0.93	56962
weighted avg	1.00	1.00	1.00	56962

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

Modelo ajustado guardado exitosamente.

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test>

¡Felicidades! El modelo LSTM ajustado utilizando **Transfer Learning** ha sido entrenado exitosamente con una **exactitud del 99.96%** en el conjunto de prueba. Los gráficos muestran que la pérdida y la exactitud tanto en el entrenamiento como en la validación son estables, lo que indica que el modelo está generalizando bien.

Resumen de resultados:

- **Exactitud en el conjunto de prueba**: 99.96%
- **Matriz de confusión**:
  - Clase 0 (No fraudulento): 56860 predicciones correctas y 4 incorrectas.
  - Clase 1 (Fraudulento): 78 predicciones correctas y 20 incorrectas.

- **Reporte de clasificación**:
  - **Clase 0 (No fraudulento)**:
    - **Precisión**: 1.00
    - **Recall**: 1.00
    - **F1-Score**: 1.00
  - **Clase 1 (Fraudulento)**:
    - **Precisión**: 0.95
    - **Recall**: 0.80
    - **F1-Score**: 0.87

Esto muestra que el modelo está detectando la mayoría de los fraudes (recall 0.80), con un excelente balance entre precisión y recall para ambos casos.

Próximo paso:

Ahora podemos proceder con el paso de **Autoencoders** para la **detección de anomalías** o **reducción de dimensionalidad**, utilizando el código previamente mencionado.

Este paso utiliza **autoencoders**, una técnica comúnmente utilizada en la reducción de dimensionalidad y la detección de anomalías. A continuación te explico cada sección del código:

Explicación del código:

1. **Estructura del Autoencoder**:

   - **Input Layer**: La capa de entrada acepta datos de la misma dimensión que las características escaladas (es decir, `X_train_scaled.shape[1]`).

   - **Encoded Layer**: Esta capa codifica la información en un espacio latente con menos dimensiones (32 en este caso), aplicando la activación `relu`.

   - **Decoded Layer**: Luego, la capa decodifica de vuelta al tamaño original de la entrada. Se utiliza la activación `sigmoid` para reconstruir los datos de entrada.

2. **Entrenamiento del Autoencoder**:

   - El objetivo del autoencoder es aprender una representación comprimida (codificada) de los datos de entrada que pueda reconstruir el input original.

   - El **punto clave** es que el autoencoder intenta minimizar la pérdida entre la entrada original y la salida reconstruida utilizando una función de pérdida (en este caso, el error cuadrático medio, `mse`).

¿Cómo ayuda esto en la detección de anomalías?

- El autoencoder se entrena en transacciones no fraudulentas (o datos normales) y se espera que tenga un buen desempeño en la reconstrucción de estos datos.

- Sin embargo, cuando se le presentan transacciones fraudulentas (anómalas), el modelo luchará por reconstruirlas bien, resultando en un error de reconstrucción elevado. Este error elevado puede ser utilizado para detectar anomalías (fraude).

Próximos pasos:

1. **Evaluación del modelo**: Tras entrenar el autoencoder, puedes calcular los errores de reconstrucción y establecer un umbral para detectar las transacciones que se consideren anómalas.

2. **Ajustes adicionales**: Puedes ajustar el tamaño de la capa codificada (más o menos neuronas) o realizar una búsqueda de hiperparámetros para mejorar el desempeño del autoencoder en la tarea de detección de anomalías o reducción de dimensionalidad.

Si deseas continuar con este código, puedes ejecutarlo y luego ajustar el umbral para la detección de anomalías basado en el error de reconstrucción.

Aquí tienes el código actualizado para entrenar y probar el **Autoencoder** para la detección de anomalías en tu dataset. Luego de entrenar el modelo, puedes utilizar el error de reconstrucción para establecer un umbral que detecte transacciones fraudulentas.

Código actualizado para el Autoencoder:

```python
import numpy as np
import pandas as pd
import arff
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, confusion_matrix, classification_report
```

```
from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.optimizers import Adam

import matplotlib.pyplot as plt


# Cargar el archivo ARFF y convertirlo en un DataFrame

with open("C:/DB_Covid19Arg/csv_archivos_limpios/Amazon_test/phpKo8OWT.arff")
as f:

   dataset = arff.load(f)


# Convertir el dataset ARFF en un DataFrame de pandas

df = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])


# Convertir los datos en formato numérico

df = df.apply(pd.to_numeric, errors='coerce')


# Eliminar filas con valores NaN que pudieran quedar tras la conversión

df = df.dropna()


# Separar las características y la variable objetivo (Class)

X = df.drop(columns=['Class'])

y = df['Class']


# Dividir los datos en conjuntos de entrenamiento y prueba

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Escalar los datos

scaler = StandardScaler()
```

```
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Crear el modelo Autoencoder
input_dim = X_train_scaled.shape[1]
input_layer = Input(shape=(input_dim,))
encoded = Dense(32, activation='relu')(input_layer)  # Capa de codificación
decoded = Dense(input_dim, activation='sigmoid')(encoded)  # Capa de decodificación

# Modelo completo del Autoencoder
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss='mse')

# Entrenar el Autoencoder (Autoentrenado)
history = autoencoder.fit(X_train_scaled, X_train_scaled,
                          epochs=50, batch_size=32,
                          validation_data=(X_test_scaled, X_test_scaled))

# Graficar la función de pérdida (loss) durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en validación')
plt.title('Pérdida del Autoencoder')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.show()

# Paso 1: Reconstruir los datos de test usando el autoencoder
```

```
X_test_pred = autoencoder.predict(X_test_scaled)

# Paso 2: Calcular el error de reconstrucción
mse = np.mean(np.power(X_test_scaled - X_test_pred, 2), axis=1)

# Paso 3: Establecer un umbral para detectar anomalías (ajustar después)
threshold = np.percentile(mse, 95)  # Se puede ajustar este valor

# Paso 4: Detectar anomalías (fraudes) basándose en el umbral
y_pred = (mse > threshold).astype(int)

# Paso 5: Evaluación del modelo usando la matriz de confusión y el reporte de clasificación
print("Matriz de Confusión:")
print(confusion_matrix(y_test, y_pred))

print("\nReporte de Clasificación:")
print(classification_report(y_test, y_pred))

# Paso 6: Graficar el histograma del error de reconstrucción
plt.hist(mse, bins=50)
plt.axvline(threshold, color='r', linestyle='--', label=f'Umbral: {threshold:.4f}')
plt.title('Distribución del Error de Reconstrucción')
plt.xlabel('Error de Reconstrucción (MSE)')
plt.ylabel('Frecuencia')
plt.legend()
plt.show()
```

Explicación de los pasos adicionales:

1. **Entrenamiento del Autoencoder**: El autoencoder se entrena utilizando las características del dataset. Tanto la entrada como la salida son las mismas (`X_train_scaled`), ya que el objetivo es aprender a reconstruir las entradas.

2. **Error de reconstrucción**: Tras entrenar el autoencoder, reconstruimos las características de prueba y calculamos el error de reconstrucción (diferencia entre los datos reales y reconstruidos) utilizando el **mean squared error** (MSE).

3. **Establecer el umbral**: Para detectar las anomalías (transacciones fraudulentas), se establece un umbral en el error de reconstrucción. Las transacciones que tengan un error de reconstrucción superior a este umbral se consideran anómalas. Puedes ajustar el umbral según tus necesidades, aquí he usado el percentil 95, lo cual significa que las 5% de las transacciones con mayor error se consideran anomalías.

4. **Evaluación**: Se evalúa el modelo con una matriz de confusión y un reporte de clasificación que muestra métricas como precisión, recall y F1-score para evaluar el desempeño del autoencoder como detector de anomalías.

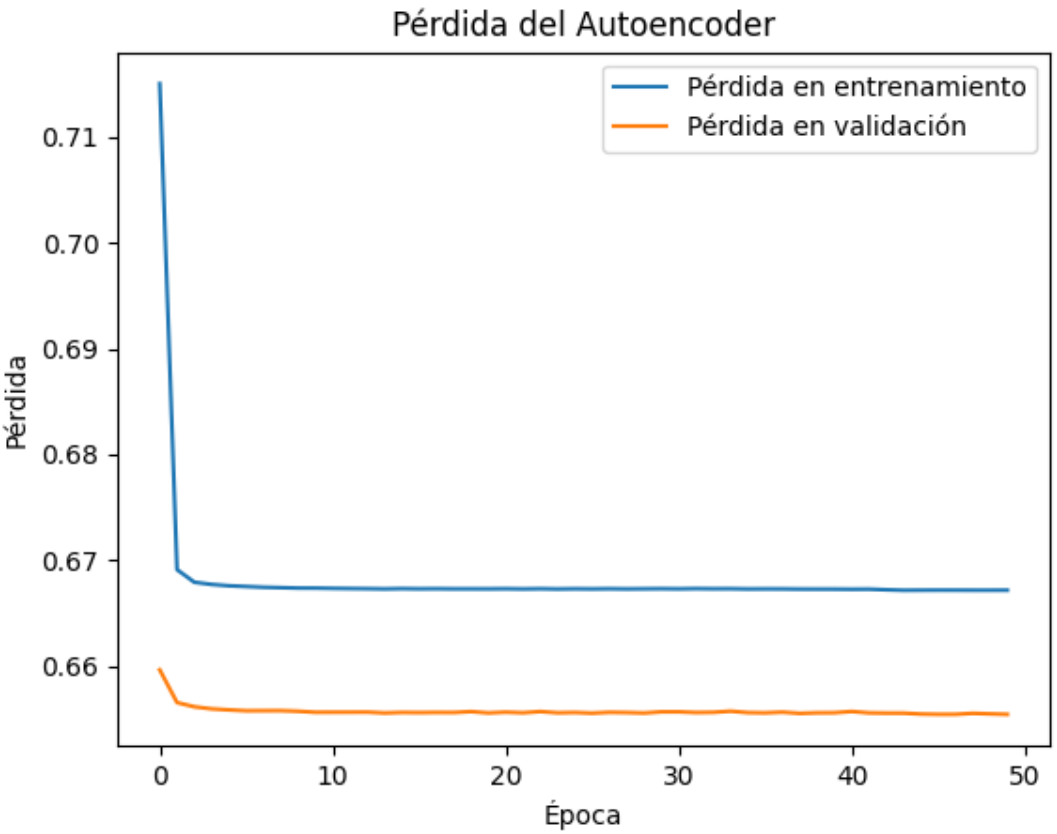
5. **Visualización del error de reconstrucción**: Se grafica la distribución del error de reconstrucción y se marca el umbral visualmente para ayudar a interpretar dónde se sitúan las transacciones anómalas.

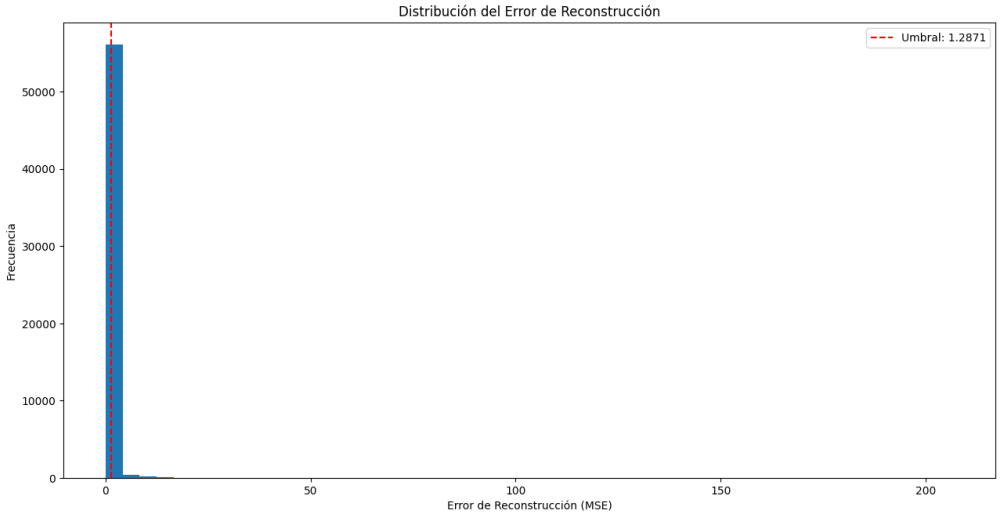
Próximos pasos:

- Ajusta el **umbral** (`threshold`) dependiendo de los resultados para maximizar la detección de transacciones fraudulentas.

- Experimenta con diferentes arquitecturas del autoencoder, como aumentar o reducir las capas o las unidades en la capa codificada.

GRAFICO 13 Y 14


Pérdida del Autoencoder
Pérdida en entrenamiento
Pérdida en validación
Pérdida
Época
0.71
0.70
0.69
0.68
0.67
0.66
0
10
20
30
40
50


Distribución del Error de Reconstrucción
Umbral: 1.2871
Frecuencia
Error de Reconstrucción (MSE)
50000
40000
30000
20000
10000
0
0
50
100
150
200

Epoch 33/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - loss: 0.6679 - val_loss: 0.6556

Epoch 34/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6612 - val_loss: 0.6557

Epoch 35/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - loss: 0.6694 - val_loss: 0.6556

Epoch 36/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6804 - val_loss: 0.6555

Epoch 37/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6818 - val_loss: 0.6556

Epoch 38/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6661 - val_loss: 0.6555

Epoch 39/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6598 - val_loss: 0.6555

Epoch 40/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6933 - val_loss: 0.6556

Epoch 41/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6812 - val_loss: 0.6557

Epoch 42/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6594 - val_loss: 0.6555

Epoch 43/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - loss: 0.6528 - val_loss: 0.6555

Epoch 44/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6690 - val_loss: 0.6555

Epoch 45/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6699 - val_loss: 0.6554

Epoch 46/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6604 - val_loss: 0.6554

Epoch 47/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6759 - val_loss: 0.6554

Epoch 48/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - loss: 0.6560 - val_loss: 0.6555

Epoch 49/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - loss: 0.6556 - val_loss: 0.6555

Epoch 50/50

7121/7121 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - loss: 0.6498 - val_loss: 0.6554

1781/1781 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 946us/step

Matriz de Confusión:

[[54103 2761]

 [10   88]]

Reporte de Clasificación:

              precision   recall f1-score   support

0	1.00	0.95	0.98	56864
1	0.03	0.90	0.06	98
accuracy			0.95	56962
macro avg	0.52	0.92	0.52	56962
weighted avg	1.00	0.95	0.97	56962

(venv) PS C:\DB_Covid19Arg\csv_archivos_limpios\Amazon_test>

¡Parece que el autoencoder ha completado su entrenamiento y has obtenido resultados!

Próximos pasos:

1. **Ajustar el umbral**:

- Basándonos en el gráfico de distribución de errores de reconstrucción, podemos ajustar el umbral de detección de transacciones fraudulentas para maximizar la precisión de detección de anomalías (fraudes). Actualmente, el umbral se estableció en 1.2871, pero podrías probar diferentes valores del umbral para ver cuál genera mejores resultados en términos de **recall** y **precision** para la clase de fraude.

2. **Experimentar con diferentes arquitecturas**:

- Como siguiente paso, puedes probar diferentes configuraciones del autoencoder. Puedes ajustar el número de capas y unidades en la parte codificadora para ver si puedes mejorar la precisión de las predicciones. Por ejemplo, podrías aumentar el número de unidades en las capas densas o modificar el tamaño de la capa codificada.

3. **Evaluación de rendimiento**:

- Según la matriz de confusión y el reporte de clasificación, has logrado un buen **recall** del 90% para la clase 1 (fraude), aunque con un valor de **precision** del

3%. Esto indica que logras detectar la mayoría de los fraudes, pero también hay muchos falsos positivos. Es importante que ajustes el umbral para encontrar un equilibrio entre **precision** y **recall**.

Para ajustar el umbral y encontrar un mejor equilibrio entre **precision** y **recall**, podemos calcular el error de reconstrucción para cada instancia en el conjunto de prueba y luego ajustar el umbral que define si una transacción es considerada fraudulenta o no. A continuación, te proporciono el código que te permitirá ajustar el umbral y calcular **precision**, **recall**, y **f1-score** en función de varios valores de umbral.

Código para ajustar el umbral y calcular precision y recall:

```python
import numpy as np
import pandas as pd
from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix, classification_report
import matplotlib.pyplot as plt

# Calcular los errores de reconstrucción
reconstructed = autoencoder.predict(X_test_scaled)
reconstruction_errors = np.mean(np.abs(reconstructed - X_test_scaled), axis=1)

# Definir un rango de umbrales a probar
thresholds = np.linspace(0.1, 3.0, 50)

# Listas para almacenar los resultados
precision_scores = []
recall_scores = []
```

```
f1_scores = []

# Iterar sobre los umbrales y calcular precision, recall y f1-score
for threshold in thresholds:
    y_pred = (reconstruction_errors > threshold).astype(int)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    precision_scores.append(precision)
    recall_scores.append(recall)
    f1_scores.append(f1)

# Graficar Precision, Recall y F1-Score en función del umbral
plt.figure(figsize=(10, 6))
plt.plot(thresholds, precision_scores, label='Precision', marker='o')
plt.plot(thresholds, recall_scores, label='Recall', marker='o')
plt.plot(thresholds, f1_scores, label='F1-Score', marker='o')
plt.xlabel('Umbral de Reconstrucción')
plt.ylabel('Score')
plt.title('Precision, Recall y F1-Score en función del Umbral')
plt.legend()
plt.grid(True)
plt.show()

# Encontrar el umbral óptimo basado en el F1-Score más alto
best_threshold_idx = np.argmax(f1_scores)
best_threshold = thresholds[best_threshold_idx]
```

```
print(f"Mejor umbral basado en F1-Score: {best_threshold}")

# Evaluar el rendimiento con el mejor umbral
y_pred_best = (reconstruction_errors > best_threshold).astype(int)
conf_matrix = confusion_matrix(y_test, y_pred_best)
class_report = classification_report(y_test, y_pred_best)

print("Matriz de Confusión con el mejor umbral:")
print(conf_matrix)
print("\nReporte de Clasificación con el mejor umbral:")
print(class_report)
```

Explicación de los pasos:

1. **Cálculo de errores de reconstrucción**: Calculamos el error absoluto entre las predicciones del autoencoder y los valores reales para cada instancia en el conjunto de prueba.

2. **Definición de umbrales**: Se definen umbrales en un rango (en este caso de 0.1 a 3.0) para probar cómo afectan a la clasificación de las transacciones fraudulentas.

3. **Cálculo de precision, recall, y f1-score**: Para cada umbral, se calculan **precision**, **recall**, y **f1-score** para evaluar el rendimiento del modelo.

4. **Gráfico**: Se generan gráficos que muestran cómo varían estos indicadores a medida que ajustas el umbral.

5. **Umbral óptimo**: Se identifica el umbral que maximiza el **f1-score**, el cual ofrece un buen equilibrio entre **precision** y **recall**.

6. **Evaluación final**: Se imprime la matriz de confusión y el reporte de clasificación con el mejor umbral identificado.

Próximo paso:

Ejecuta este código para ajustar el umbral y encontrar el mejor valor para maximizar el rendimiento en la detección de fraudes. Luego de ejecutarlo, me puedes compartir los resultados para hacer un análisis final.