Proyecto #2
Redes Neuronales

Badillo L. Antonio[#1], Bello M. Daniel[*2], Di Pietro Z. José[#3]

[#]*Ingeniería Informática, Universidad Católica Andrés Bello, Montalbán, Caracas, Venezuela*

[1]aabadillo.18@est.ucab.edu.ve, [2]dabello.18@est.ucab.edu.ve,
[3]jadipietro.19@est.ucab.edu.ve

INTRODUCCIÓN

Este informe detalla el proceso del desarrollo de un sistema de Redes Neuronales, donde analizamos, estudiamos, clasificamos y limpiamos, según la necesidad existente, los datos de entrada para dar respuesta a un (1) problema propuesto.

Un red neuronal es un modelo simplificado que emula el modo en que el cerebro humano procesa la información: Funciona simultaneando un número elevado de unidades de procesamiento interconectadas. Las unidades de procesamiento se organizan en capas y se conectan con fuerzas de conexión variables (o ponderaciones) [1]

- **Capa de entrada**: Los nodos de entrada procesan los datos, los analizan o los clasifican y los pasan a la siguiente capa.
- **Capas ocultas**: El valor de cada unidad oculta es alguna función de los predictores; la forma exacta de la función depende en parte del tipo de red. Cada capa oculta analiza la salida de la capa anterior, la procesa aún más y la pasa a la siguiente capa.
- **Capa de salida**: Proporciona el resultado final de todo el procesamiento de datos que realiza la red neuronal artificial.[2]

Para el desarrollo de este proyecto, se decidió trabajar con el problema número 1, el cual describe:

> *'La empresa VISA lo ha contratado para construir un modelo predictivo de fraudes en transacciones de tarjetas de crédito. Para ello usted cuenta con un dataset de transacciones de tarjetas de crédito. Por razones de confidencialidad, las características V1, V2, ..., V28 son anónimas y no se conoce el significado original de la característica. Además se cuenta con el instante de tiempo de cada transacción (medido en segundos desde la transacción más temprana del dataset) y el monto de la transacción. La característica de salida es si la transacción es un fraude o no (1 si es un fraude, 0 en caso contrario).'*

Para llegar al modelo de Redes Neuronales que nos sea de mayor utilidad debemos realizar ciertos estudios a la data de prueba que tenemos para llegar al modelo indicado que resuelva nuestra problemática.

Se utilizó Python 3X y librerías para el uso de disposición y análisis de datos, en conjunto de Keras, una librería para implementar redes neuronales. Además, mediante la herramienta Google Colab, se pudo desarrollar el proyecto de manera colaborativa.

DESARROLLO

Primeramente, se importan las librerías a utilizar:

- Tensor Flow
- Numpy
- Matplotlib
- Pandas
- zipfile
- BytesIO
- Keras (se importa luego)

Importando dependencias

```
[ ] import tensorflow as tf
    import numpy as np
    import matplotlib.pyplot as plt
    import pandas as pd
    import requests, zipfile
    from io import BytesIO
```

Luego, se importa el dataset del proporcionado por la empresa: creditcard.csv. Además se verifica que no existan columnas incompletas. Este proceso se realiza debido a que se debe entrenar a los modelos con datos completos.

Importando el dataset y explorando los datos

```
[ ] url = 'https://www.googleapis.com/drive/v3/files/18B9aQ8YdG3nyuKzh5a0Q_t_8ZnFHqfAP?alt=media&key=AIzaSyBpOtCklEeXCT6Vh7y3mgqO7q7veo8wE24'
    file_name = 'creditcard.csv'

    r = requests.get(url)

    zip = zipfile.ZipFile(BytesIO(r.content))
    creditcards = pd.read_csv(zip.open(file_name))
    # creditcards = creditcards[:50000]
    creditcards.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
```

```
] creditcards.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	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	0.098698	0.363787	...	-0.018307	0.277838	-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	0.085102	-0.255425	...	-0.225775	-0.638672	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	0.247676	-1.514654	...	0.247998	0.771679	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	0.377436	-1.387024	...	-0.108300	0.005274	-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	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	0

5 rows × 31 columns

A continuación se puede observar que todos los atributos son numéricos. No existen atributos tipo object o string formateado que se deba limpiar

```
[ ] creditcards.describe()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V2
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	...	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+0
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	1.487313e-15	-5.556467e-16	1.213481e-16	-2.406331e-15	...	1.654067e-16	-3.568593e-16	2.578648e-16	4.473266e-15	5.340915e-16	1.683437e-1
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e+00	1.098632e+00	...	7.345240e-01	7.257016e-01	6.244603e-01	6.056471e-01	5.212781e-01	4.822270e-0
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e+01	-1.343407e+01	...	-3.483038e+01	-1.093314e+01	-4.480774e+01	-2.836627e+00	-1.029540e+01	-2.604551e+0
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e-01	-6.430976e-01	...	-2.283949e-01	-5.423504e-01	-1.618463e-01	-3.545861e-01	-3.171451e-01	-3.269839e-0
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e-02	-5.142873e-02	...	-2.945017e-02	6.781943e-03	-1.119293e-02	4.097606e-02	1.659350e-02	-5.213911e-0
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e-01	5.971390e-01	...	1.863772e-01	5.285536e-01	1.476421e-01	4.395266e-01	3.507156e-01	2.409522e-0
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e+01	1.559499e+01	...	2.720284e+01	1.050309e+01	2.252841e+01	4.584549e+00	7.519589e+00	3.517346e+0

8 rows × 31 columns

Se puede observar que todos los atributos son numéricos. No existen atributos tipo object o string formateado que se deba limpiar. Además, se puede observar que no existen columnas incompletas, incluyendo la columna correspondiente a la clase.

Igualmente, se verifica que el dataset no posea filas con datos incompletos.

```
[ ] null_data = creditcards[creditcards.isnull().any(axis=1)]
    display(null_data)
```

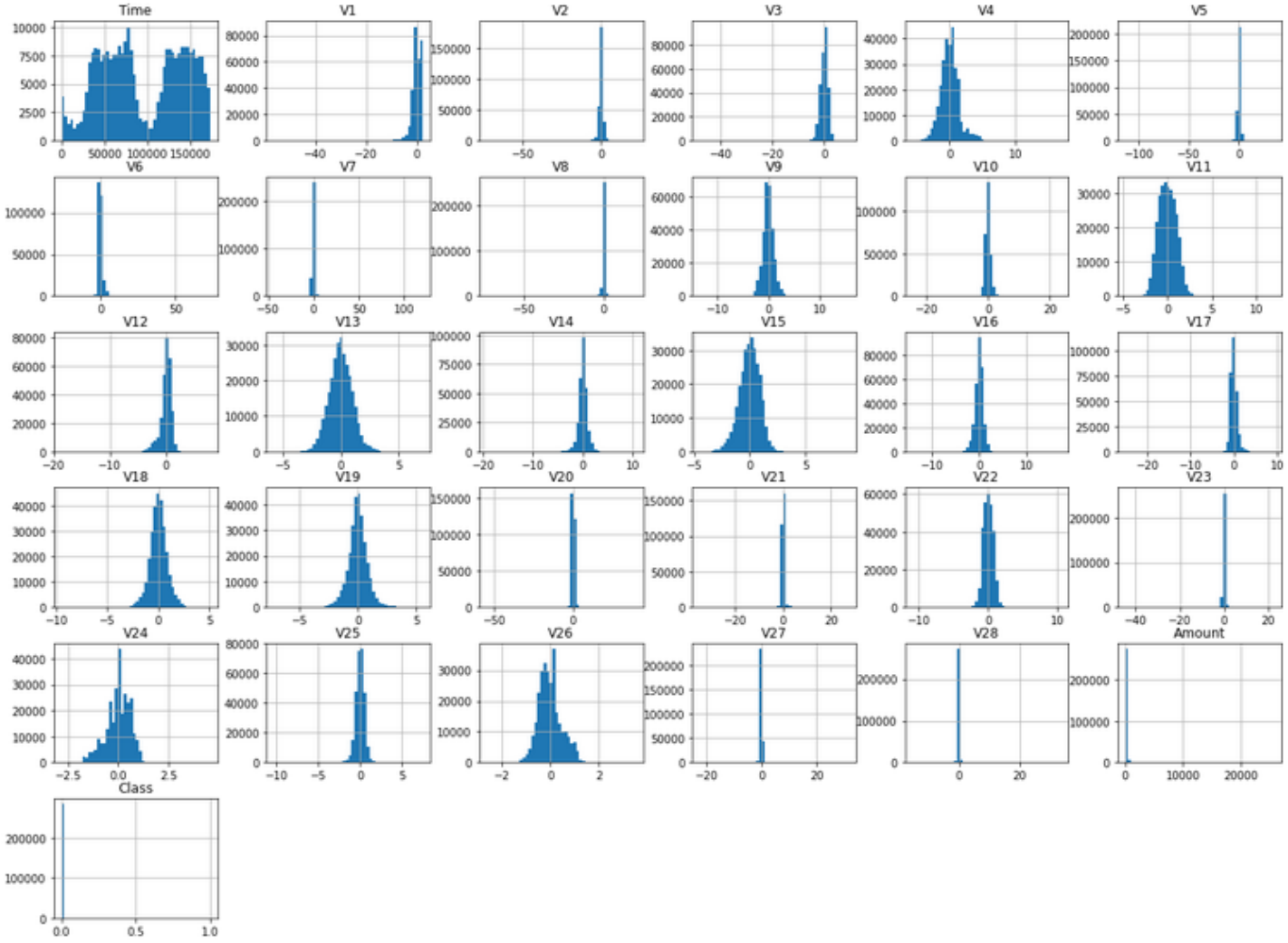
Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class

0 rows × 31 columns

Es de suma importancia verificar la completitud del dataset.

Si existieran datos incompletos, se debería emplear una técnica para rellenar los datos de aquellos elementos, pero este no es el caso.

Para poder observar y analizar bien la información, se grafican todos los atributos con un histograma.



Es de notar que la columna *Class*, que corresponde a si la transacción en cuestión es fraudulenta o no, tiene un gran índice de valores 0 comparado con la cantidad de valores 1. Esto será de importancia luego, donde se corregirá esta discrepancia.

De igual manera, se puede analizar la semejanza de los histogramas de las características anónimas con la distribución normal, lo indica la confiabilidad de dataset empleado.

Seguidamente, en el proceso de análisis de datos, se procede a calcular la correlación de los datos con respecto a la columna Class.

```
corr = creditcards.corr()
corr["Class"].sort_values(ascending = False)
```

```
Class     1.000000
V11       0.154876
V4        0.133447
V2        0.091289
V21       0.040413
V19       0.034783
V20       0.020090
V8        0.019875
V27       0.017580
V28       0.009536
Amount    0.005632
V26       0.004455
V25       0.003308
V22       0.000805
V23      -0.002685
V15      -0.004223
V13      -0.004570
V24      -0.007221
Time     -0.012323
V6       -0.043643
V5       -0.094974
V9       -0.097733
V1       -0.101347
V18      -0.111485
V7       -0.187257
V3       -0.192961
V16      -0.196539
V10      -0.216883
V12      -0.260593
V14      -0.302544
V17      -0.326481
Name: Class, dtype: float64
```

En la matriz de correlación podemos observar que las columnas con más correlación con Class son: V17, V14, V12, V10, V16, V7, V11, V4, V18 y V1. Igualmente, se puede eliminar la columna "Time" puesto que no es relevante para el modelo, debido a su baja correlación.

Luego de aislar la columna Class, observamos que la cantidad de casos que resultan Fraude no se encuentra equilibrada con respecto a los datos de No Fraude. Esto puede llevar a errores en nuestro modelo.

```
print("Cantidad de datos No Fraude", len(x[y == 0]))
print("Cantidad de datos Fraude", len(x[y == 1]))
```

```
Cantidad de datos No Fraude 284315
Cantidad de datos Fraude 492
```

La cantidad de casos de No Fraude, supera a la cantidad de caso de Fraude en casi un 57900%. Para solucionar este caso debemos hacer uso de alguna técnica de Oversampling, la cual consiste en aumentar la cantidad de ejemplos de la clase minoritaria creando nuevos datos en base a los ya existentes en el dataset. En este caso se utilizará la técnica de SMOTE de Oversampling en el conjunto de datos de entrenamiento.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
# Tomamos 20% de los datos para el conjunto de pruebas
```

```
[ ] from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X_train, y_train)
print("Cantidad de datos No Fraude", len(X_res[y_res == 0]))
print("Cantidad de datos Fraude", len(X_res[y_res == 1]))
```

```
Cantidad de datos No Fraude 227451
Cantidad de datos Fraude 227451
```

Aquí podemos ver como utilizando la técnica SMOTE aumentamos el número de datos para la clase de Fraude en el conjunto de entrenamiento, por lo que ahora tenemos ambas clases balanceadas.

Finalmente, luego de realizar la limpieza de datos, se procede a crear el conjunto de validación a partir del conjunto de entrenamiento actual.

```
[ ] X_train, X_valid, y_train, y_valid = train_test_split(X_res, y_res, test_size=0.2, random_state=42)

    from sklearn.preprocessing import StandardScaler # Escalamos los datos

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_valid = scaler.transform(X_valid)
    X_test = scaler.transform(X_test)
```

En la siguiente sección, se procede a crear y probar varias arquitecturas de redes neuronales usando los conjuntos ya creados.

Arquitectura 1

La primera arquitectura implementada fué una red neuronal de 1 capa intermedia con 20 neuronas.

Es importante señalar que se utilizaron las funciones de activación relu para la capa intermedia y sigmoid para la de salida.

La cantidad de neuronas de la capa de entrada corresponde a la cantidad de datos de entrada, por eso se define en función de la cantidad de columnas del dataset de X_train.

La cantidad de neuronas para la capa de salida es 1, ya que este problema es de clasificación binaria.

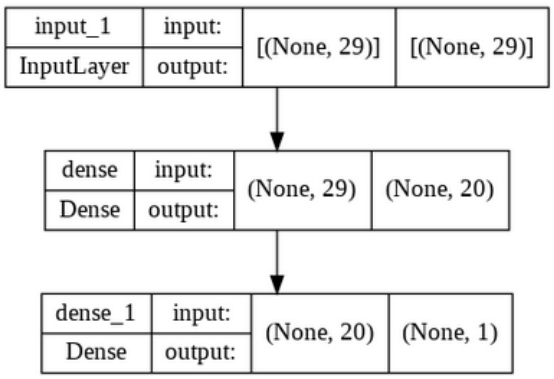
A continuación se muestra la implementación de la red neuronal,

```
[ ] arquitectura1 = keras.models.Sequential([
        keras.layers.InputLayer(input_shape=X_train.shape[1:]),
        keras.layers.Dense(20, activation='relu'),
        keras.layers.Dense(1, activation='sigmoid')
    ])
    arquitectura1.compile(
        loss='binary_crossentropy',
        optimizer=keras.optimizers.SGD(learning_rate=1e-2),
        metrics=[
            keras.metrics.Precision(name='precision'),
            keras.metrics.Recall(name='recall'),
            keras.metrics.Accuracy(name='accuracy')
          ])

    arquitectura1.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                600

 dense_1 (Dense)             (None, 1)                 21

=================================================================
Total params: 621
Trainable params: 621
Non-trainable params: 0
_________________________________________________________________
```

Keras permite diagramar el modelo especificado de la red neuronal. Para esta primera arquitectura, se puede ver de forma gráfica lo explicado anteriormente.

```
[ ] keras.utils.plot_model(arquitectura1, to_file='aquitectura1.png', show_shapes=True, show_layer_names=True)
```



Se entrena el modelo con el método fit.

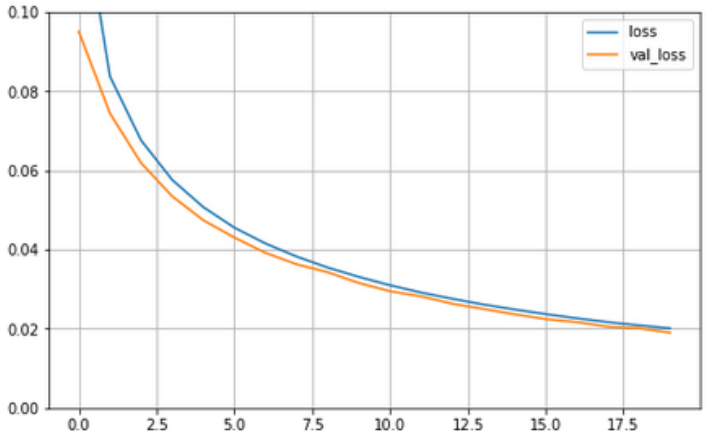
```
history = arquitectura1.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))
print(history)

Epoch 1/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.1330 - precision: 0.9816 - recall: 0.9135 - accuracy: 0.0287 - val_loss: 0.0951 - val_precision: 0.9787 - val_recall: 0.9445 - val_accuracy: 0.0503
Epoch 2/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0838 - precision: 0.9819 - recall: 0.9516 - accuracy: 0.0646 - val_loss: 0.0744 - val_precision: 0.9839 - val_recall: 0.9586 - val_accuracy: 0.0794
Epoch 3/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0676 - precision: 0.9846 - recall: 0.9643 - accuracy: 0.0932 - val_loss: 0.0619 - val_precision: 0.9842 - val_recall: 0.9692 - val_accuracy: 0.1080
Epoch 4/20
11373/11373 [==============================] - 36s 3ms/step - loss: 0.0576 - precision: 0.9861 - recall: 0.9709 - accuracy: 0.1220 - val_loss: 0.0535 - val_precision: 0.9867 - val_recall: 0.9728 - val_accuracy: 0.1325
Epoch 5/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0507 - precision: 0.9868 - recall: 0.9754 - accuracy: 0.1398 - val_loss: 0.0474 - val_precision: 0.9885 - val_recall: 0.9744 - val_accuracy: 0.1462
Epoch 6/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0455 - precision: 0.9877 - recall: 0.9786 - accuracy: 0.1516 - val_loss: 0.0431 - val_precision: 0.9867 - val_recall: 0.9816 - val_accuracy: 0.1574
Epoch 7/20
11373/11373 [==============================] - 22s 2ms/step - loss: 0.0415 - precision: 0.9884 - recall: 0.9814 - accuracy: 0.1615 - val_loss: 0.0392 - val_precision: 0.9892 - val_recall: 0.9819 - val_accuracy: 0.1660
Epoch 8/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0382 - precision: 0.9893 - recall: 0.9837 - accuracy: 0.1680 - val_loss: 0.0363 - val_precision: 0.9893 - val_recall: 0.9854 - val_accuracy: 0.1681
Epoch 9/20
11373/11373 [==============================] - 22s 2ms/step - loss: 0.0355 - precision: 0.9897 - recall: 0.9860 - accuracy: 0.1709 - val_loss: 0.0343 - val_precision: 0.9904 - val_recall: 0.9852 - val_accuracy: 0.1731
Epoch 10/20
11373/11373 [==============================] - 22s 2ms/step - loss: 0.0331 - precision: 0.9903 - recall: 0.9876 - accuracy: 0.1726 - val_loss: 0.0316 - val_precision: 0.9899 - val_recall: 0.9888 - val_accuracy: 0.1793
Epoch 11/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0310 - precision: 0.9905 - recall: 0.9891 - accuracy: 0.1777 - val_loss: 0.0295 - val_precision: 0.9904 - val_recall: 0.9897 - val_accuracy: 0.1821
Epoch 12/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0292 - precision: 0.9908 - recall: 0.9906 - accuracy: 0.1822 - val_loss: 0.0282 - val_precision: 0.9904 - val_recall: 0.9920 - val_accuracy: 0.1829
Epoch 13/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0276 - precision: 0.9912 - recall: 0.9918 - accuracy: 0.1800 - val_loss: 0.0263 - val_precision: 0.9901 - val_recall: 0.9936 - val_accuracy: 0.1857
Epoch 14/20
11373/11373 [==============================] - 22s 2ms/step - loss: 0.0261 - precision: 0.9914 - recall: 0.9928 - accuracy: 0.1842 - val_loss: 0.0250 - val_precision: 0.9902 - val_recall: 0.9945 - val_accuracy: 0.1868
Epoch 15/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0249 - precision: 0.9917 - recall: 0.9938 - accuracy: 0.1857 - val_loss: 0.0237 - val_precision: 0.9919 - val_recall: 0.9943 - val_accuracy: 0.1891
Epoch 16/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0237 - precision: 0.9919 - recall: 0.9947 - accuracy: 0.1885 - val_loss: 0.0225 - val_precision: 0.9921 - val_recall: 0.9951 - val_accuracy: 0.1904
Epoch 17/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0227 - precision: 0.9923 - recall: 0.9956 - accuracy: 0.1891 - val_loss: 0.0217 - val_precision: 0.9919 - val_recall: 0.9962 - val_accuracy: 0.1931
Epoch 18/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0217 - precision: 0.9926 - recall: 0.9962 - accuracy: 0.1917 - val_loss: 0.0205 - val_precision: 0.9926 - val_recall: 0.9969 - val_accuracy: 0.1950
Epoch 19/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0209 - precision: 0.9929 - recall: 0.9967 - accuracy: 0.1963 - val_loss: 0.0201 - val_precision: 0.9929 - val_recall: 0.9968 - val_accuracy: 0.2002
Epoch 20/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0201 - precision: 0.9930 - recall: 0.9972 - accuracy: 0.1987 - val_loss: 0.0190 - val_precision: 0.9928 - val_recall: 0.9980 - val_accuracy: 0.2042
<keras.callbacks.History object at 0x7f668b783b50>
```

Una vez entrenado el modelo 1, se procede a predecir los datos del conjunto de prueba.

Se diagraman los indicadores loss y val_loss.

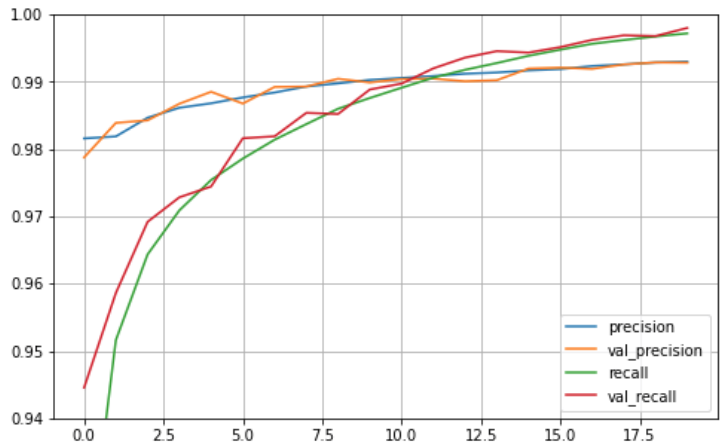
```
[ ]  predicted = arquitectura1.predict(X_test)

     from sklearn.metrics import precision_recall_curve, average_precision_score

     # Calculate average precision and the PR curve
     average_precision = average_precision_score(y_test, predicted)
     print(average_precision)
     # Obtain precision and recall

     loss_history = {
         'loss': history.history['loss'],
         'val_loss': history.history['val_loss']
     }
     pd.DataFrame(loss_history).plot(figsize=(8, 5))
     plt.grid(True)
     plt.gca().set_ylim(0, 0.10)
     plt.show()
```

```
0.7420283966921536
```



Se diagraman los indicadores precision, val_precision, recall y val_recall.

```
] prec_rec_history = {
       'precision': history.history['precision'],
       'val_precision': history.history['val_precision'],
       'recall': history.history['recall'],
       'val_recall': history.history['val_recall']
   }
   pd.DataFrame(prec_rec_history).plot(figsize=(8, 5))
   plt.grid(True)
   plt.gca().set_ylim(0.94, 1)
   plt.show()
```



```
predictions = arquitectura1.predict(X_test)
y_pred = (predictions > 0.4)
print(predictions)
print(y_pred)
```

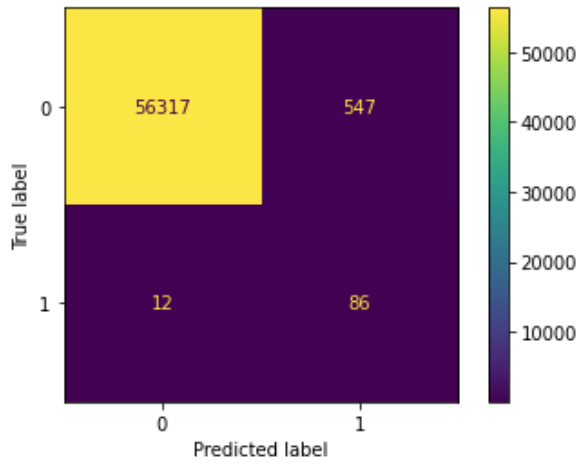
```
[[1.0000000e+00]
 [3.7392676e-03]
 [2.1295965e-02]
 ...
 [6.6905909e-10]
 [1.5886289e-06]
 [1.6817803e-05]]
[[ True]
 [False]
 [False]
 ...
 [False]
 [False]
 [False]]
```

Se predice el conjunto X de los datos de prueba. Se utiliza un trade-off entre precisión y recall 60-40, favoreciendo las predicciones positivas. Esto hace que disminuya la cantidad de falsos negativos y aumente la cantidad de falsos positivos.

Se decidió usar este trade-off debido a las especificaciones de los requerimientos del problema. Es mejor disminuir la cantidad de falsos negativos (es decir, que se cuelen menos transacciones fraudulentas), compensando con el aumento de falsos positivos, ya que, por lo visto anteriormente, los positivos son casos atípicos y los cuales se pueden verificar todos mediante un factor humano.

A continuación, se procedió a ilustrar la matriz de confusión del modelo 1 usando los datos de prueba. Se puede observar la baja cantidad de falsos negativos y su desproporción con los falsos positivos (por lo explicado anteriormente).

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

conf_matrix = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix)
disp.plot()
plt.show()
```



Finalmente, se evaluó el modelo 1.

```
arquitectura1.evaluate(X_test, y_test)
```

```
1781/1781 [==============================] - 3s 2ms/step - loss: 0.0267 - precision: 0.1717 - recall: 0.8673 - accuracy: 0.0011
[0.026682548224925995,
 0.17171716690063477,
 0.8673469424247742,
 0.0010708894114941359]
```

Arquitectura 2

La siguiente arquitectura implementada consta de 10 capas intermedias, de 1 neurona cada una.

Es importante señalar que se utilizaron las funciones de activación relu para la capa intermedia y sigmoid para la de salida.

La cantidad de neuronas de la capa de entrada corresponde a la cantidad de datos de entrada, por eso se define en función de la cantidad de columnas del dataset de X_train.

La cantidad de neuronas para la capa de salida es 1, ya que este problema es de clasificación binaria.

A continuación se muestra la implementación de la red neuronal.

Arquitectura 2: 10 capas ocultas

Capa 1-10: 1 neurona - relu

Capa de salida: 1 neurona - sigmoid

```
arquitectura2 = keras.models.Sequential([
    keras.layers.InputLayer(input_shape=X_train.shape[1:]),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='relu'),
    keras.layers.Dense(1, activation='sigmoid')
])
arquitectura2.compile(
    loss='binary_crossentropy',
    optimizer=keras.optimizers.SGD(learning_rate=1e-2),
    metrics=[
        keras.metrics.Precision(name='precision'),
        keras.metrics.Recall(name='recall'),
        keras.metrics.Accuracy(name='accuracy')
      ])

arquitectura2.summary()
```

Se define la red neuronal según las especificaciones previas.

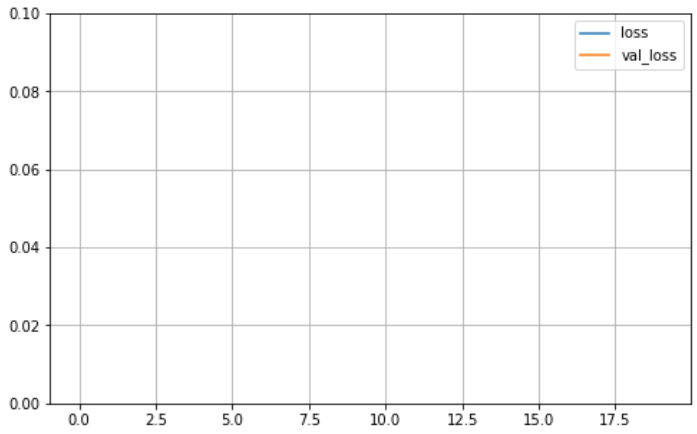
```
history = arquitectura2.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))
print(history)
```

```
Epoch 1/20
11373/11373 [==============================] - 28s 2ms/step - loss: 0.6932 - precision: 0.4996 - recall: 0.3958 - accuracy: 0.0000e+00 -
Epoch 2/20
11373/11373 [==============================] - 28s 2ms/step - loss: 0.6932 - precision: 0.5008 - recall: 0.5066 - accuracy: 0.0000e+00 -
Epoch 3/20
11373/11373 [==============================] - 28s 2ms/step - loss: 0.6932 - precision: 0.4983 - recall: 0.4806 - accuracy: 0.0000e+00 -
Epoch 4/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.6932 - precision: 0.4985 - recall: 0.3955 - accuracy: 0.0000e+00 -
Epoch 5/20
11373/11373 [==============================] - 28s 2ms/step - loss: 0.6932 - precision: 0.4990 - recall: 0.4564 - accuracy: 0.0000e+00 -
Epoch 6/20
11373/11373 [==============================] - 27s 2ms/step - loss: 0.6932 - precision: 0.4993 - recall: 0.4618 - accuracy: 0.0000e+00 -
Epoch 7/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.6932 - precision: 0.5003 - recall: 0.4438 - accuracy: 0.0000e+00 -
Epoch 8/20
11373/11373 [==============================] - 27s 2ms/step - loss: 0.6932 - precision: 0.4999 - recall: 0.4686 - accuracy: 0.0000e+00 -
Epoch 9/20
11373/11373 [==============================] - 27s 2ms/step - loss: 0.6932 - precision: 0.5003 - recall: 0.4245 - accuracy: 0.0000e+00 -
Epoch 10/20
11373/11373 [==============================] - 27s 2ms/step - loss: 0.6932 - precision: 0.4974 - recall: 0.4868 - accuracy: 0.0000e+00 -
Epoch 11/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.6932 - precision: 0.5005 - recall: 0.4550 - accuracy: 0.0000e+00 -
Epoch 12/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.6932 - precision: 0.4998 - recall: 0.4746 - accuracy: 0.0000e+00 -
Epoch 13/20
11373/11373 [==============================] - 28s 2ms/step - loss: 0.6932 - precision: 0.4991 - recall: 0.5042 - accuracy: 0.0000e+00 -
Epoch 14/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.6932 - precision: 0.4979 - recall: 0.4829 - accuracy: 0.0000e+00 -
Epoch 15/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.6932 - precision: 0.4993 - recall: 0.4321 - accuracy: 0.0000e+00 -
Epoch 16/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.6932 - precision: 0.5008 - recall: 0.4753 - accuracy: 0.0000e+00 -
Epoch 17/20
11373/11373 [==============================] - 27s 2ms/step - loss: 0.6932 - precision: 0.5012 - recall: 0.4528 - accuracy: 0.0000e+00 -
Epoch 18/20
11373/11373 [==============================] - 27s 2ms/step - loss: 0.6932 - precision: 0.4994 - recall: 0.4750 - accuracy: 0.0000e+00 -
Epoch 19/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.6932 - precision: 0.4992 - recall: 0.4616 - accuracy: 0.0000e+00 -
Epoch 20/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.6932 - precision: 0.5009 - recall: 0.4522 - accuracy: 0.0000e+00 -
<keras.callbacks.History object at 0x7f6605f78e50>
```

Se entrena el modelo mediante el método fit.

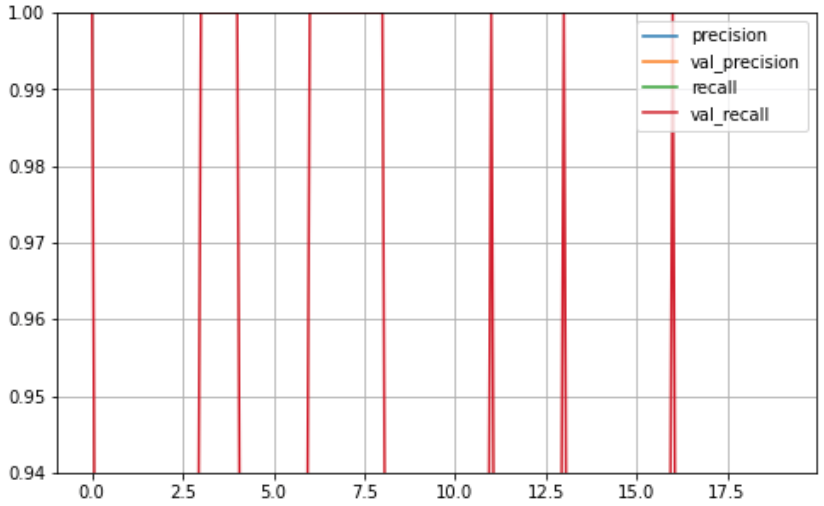
```
predicted = arquitectura2.predict(X_test)

# Calculate average precision and the PR curve
average_precision = average_precision_score(y_test, predicted)
print(average_precision)
# Obtain precision and recall

loss_history = {
    'loss': history.history['loss'],
    'val_loss': history.history['val_loss']
}
pd.DataFrame(loss_history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 0.10)
plt.show()
```

```
0.0017204452090867595
```



Se grafica la pérdida. Se puede observar la gran deficiencia de este modelo.

```
prec_rec_history = {
    'precision': history.history['precision'],
    'val_precision': history.history['val_precision'],
    'recall': history.history['recall'],
    'val_recall': history.history['val_recall']
}
pd.DataFrame(prec_rec_history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0.94, 1)
plt.show()
```

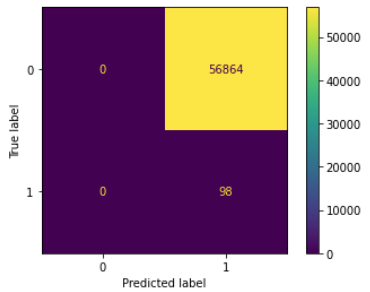


Se grafica las precisión, val_precision, recall y val_recall.

Se puede observar a simple vista la deficiencia de esta arquitectura.

```
predictions = arquitectura2.predict(X_test)
y_pred = (predictions > 0.4)
```

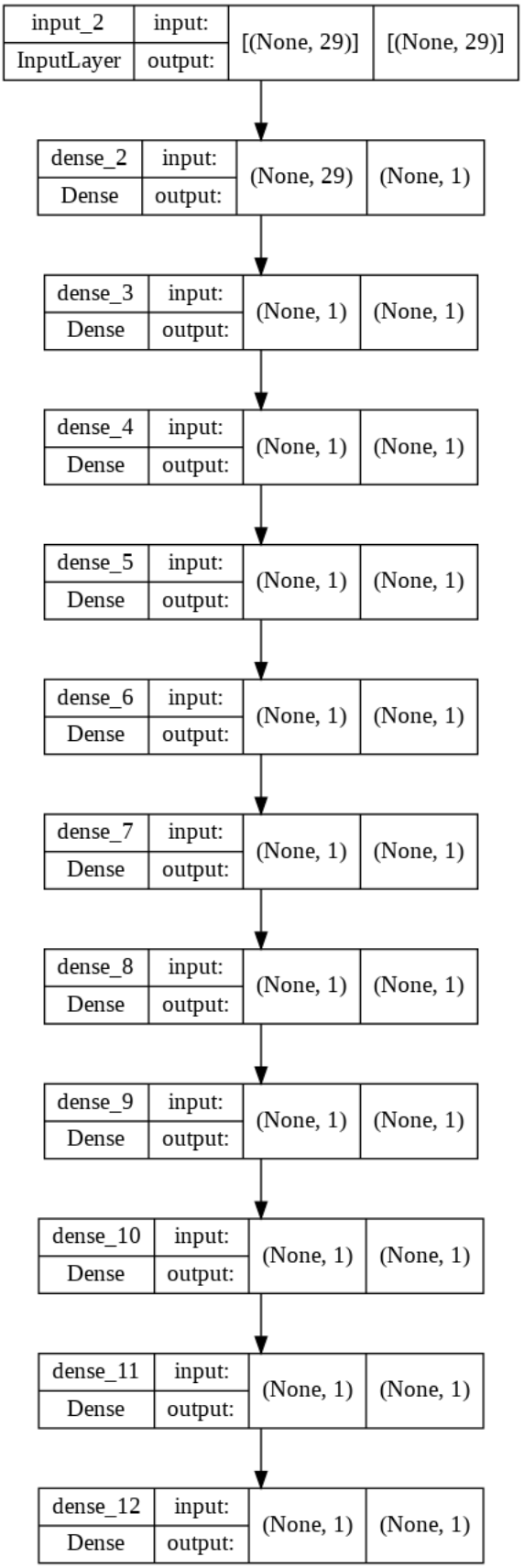
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

conf_matrix = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix)
disp.plot()
plt.show()
```




```
arquitectura2.evaluate(X_test, y_test)
```

```
1781/1781 [==============================] - 3s 2ms/step - loss: 0.6908 - precision: 0.0000e+00 - recall: 0.0000e+00 - accuracy: 0.0000e+00
[0.6908077597618103, 0.0, 0.0, 0.0]
```

Al igual que con la arquitectura 1, se estableció un trade-off 60/40 y se estableció la matriz de confusión. Se puede observar que la gran mayoría de las predicciones son falsos positivos.



Por último, se expresa de forma gráfica la arquitectura empleada.

Arquitectura 3

La última arquitectura implementada consta de 4 capas intermedias, de 5 neuronas cada una.

Al igual que las arquitecturas anteriores se utilizaron las funciones de activación relu para la capa intermedia y sigmoid para la de salida.

La cantidad de neuronas de la capa de entrada corresponde a la cantidad de datos de entrada, por eso se define en función de la cantidad de columnas del dataset de X_train.

La cantidad de neuronas para la capa de salida es 1, ya que este problema es de clasificación binaria.

A continuación se muestra la implementación de la red neuronal.

Arquitectura 3: 4 Capas ocultas

Capas 1-4: 5 neuronas relu

Capa de salida: 1 neurona sigmoid

```
[34] arquitectura3 = keras.models.Sequential([
         keras.layers.InputLayer(input_shape=X_train.shape[1:]),
         keras.layers.Dense(5, activation='relu'),
         keras.layers.Dense(5, activation='relu'),
         keras.layers.Dense(5, activation='relu'),
         keras.layers.Dense(5, activation='relu'),
         keras.layers.Dense(1, activation='sigmoid')
     ])
     arquitectura3.compile(
         loss='binary_crossentropy',
         optimizer=keras.optimizers.SGD(learning_rate=1e-2),
         metrics=[
             keras.metrics.Precision(name='precision'),
             keras.metrics.Recall(name='recall'),
             keras.metrics.Accuracy(name='accuracy')
           ])

     arquitectura3.summary()
```

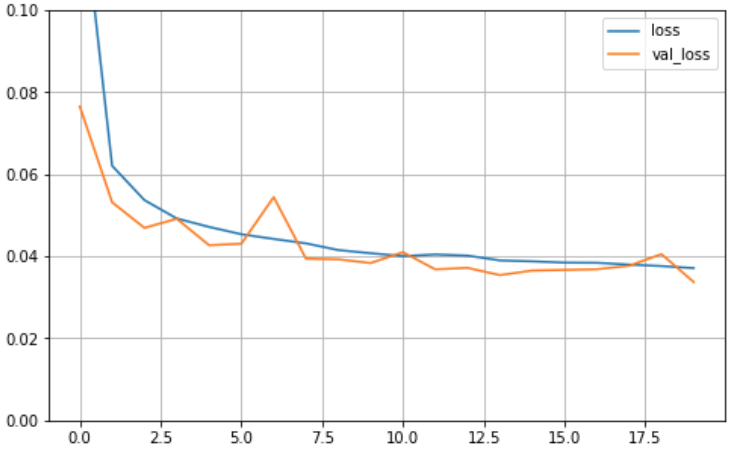
```
history = arquitectura3.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))
print(history)
```

```
Epoch 1/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.1324 - precision: 0.9699 - recall: 0.9192 - accuracy: 0.0996 -
Epoch 2/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.0621 - precision: 0.9802 - recall: 0.9704 - accuracy: 0.1115 -
Epoch 3/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0537 - precision: 0.9812 - recall: 0.9773 - accuracy: 0.0323 -
Epoch 4/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0492 - precision: 0.9816 - recall: 0.9794 - accuracy: 7.3368e-
Epoch 5/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0471 - precision: 0.9816 - recall: 0.9810 - accuracy: 1.6487e-
Epoch 6/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0454 - precision: 0.9816 - recall: 0.9824 - accuracy: 1.2091e-
Epoch 7/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0442 - precision: 0.9817 - recall: 0.9833 - accuracy: 2.5005e-
Epoch 8/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0431 - precision: 0.9821 - recall: 0.9842 - accuracy: 4.4790e-
Epoch 9/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0415 - precision: 0.9823 - recall: 0.9858 - accuracy: 6.8147e-
Epoch 10/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.0407 - precision: 0.9826 - recall: 0.9863 - accuracy: 0.0012 -
Epoch 11/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0400 - precision: 0.9829 - recall: 0.9871 - accuracy: 0.0011 -
Epoch 12/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.0404 - precision: 0.9827 - recall: 0.9869 - accuracy: 0.0340 -
Epoch 13/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0402 - precision: 0.9827 - recall: 0.9872 - accuracy: 0.0203 -
Epoch 14/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0390 - precision: 0.9828 - recall: 0.9875 - accuracy: 0.0015 -
Epoch 15/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0388 - precision: 0.9830 - recall: 0.9879 - accuracy: 0.0017 -
Epoch 16/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0385 - precision: 0.9834 - recall: 0.9878 - accuracy: 0.0018 -
Epoch 17/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0384 - precision: 0.9834 - recall: 0.9883 - accuracy: 0.0018 -
Epoch 18/20
11373/11373 [==============================] - 27s 2ms/step - loss: 0.0380 - precision: 0.9828 - recall: 0.9884 - accuracy: 0.0019 -
Epoch 19/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0376 - precision: 0.9829 - recall: 0.9888 - accuracy: 0.0020 -
Epoch 20/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0371 - precision: 0.9827 - recall: 0.9895 - accuracy: 0.0022 -
```

Ejecutamos el método fit para entrenar el modelo.

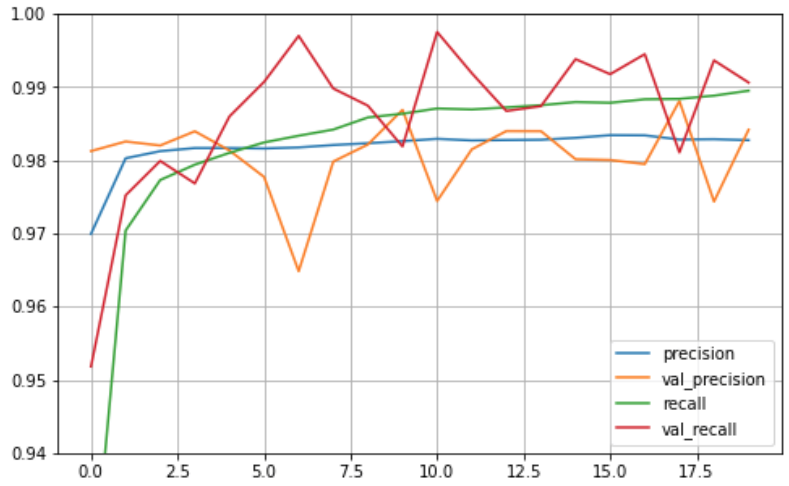
```
predicted = arquitectura3.predict(X_test)

# Calculate average precision and the PR curve
average_precision = average_precision_score(y_test, predicted)
print(average_precision)
# Obtain precision and recall

loss_history = {
    'loss': history.history['loss'],
    'val_loss': history.history['val_loss']
}
pd.DataFrame(loss_history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 0.10)
plt.show()
```

```
0.7354091897796429
```



Aquí podemos observar la gráfica de pérdida de este modelo, observamos que en cierto punto se mantiene estable.

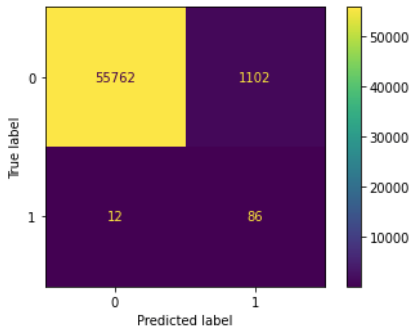
```
prec_rec_history = {
    'precision': history.history['precision'],
    'val_precision': history.history['val_precision'],
    'recall': history.history['recall'],
    'val_recall': history.history['val_recall']
}
pd.DataFrame(prec_rec_history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0.94, 1)
plt.show()
```



Ahora observamos la gráfica de precision, val_precision, recall y val_recall.

```
predictions = arquitectura3.predict(X_test)
y_pred = (predictions > 0.4)
```

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

conf_matrix = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix)
disp.plot()
plt.show()
```




```
arquitectura3.evaluate(X_test, y_test)
```

```
1781/1781 [==============================] - 3s 2ms/step - loss: 0.0410 - precision: 0.0856 - recall: 0.8776 - accuracy: 0.0047
[0.040973000228405,
 0.08557213842868805,
 0.8775510191917419,
 0.0046522244811058044]
```

Para finalizar realizamos la matriz de confusión con los datos de prueba, podemos observar que se han clasificado bien los No Fraude, y por lo ya explicado en la arquitectura 1 sobre el trade-off utilizado para esta problemática podemos ver muy pocos casos de falsos negativos.

Hyperparameter Tunning

Para esta sección decidimos utilizar el modelo con la arquitectura 3.

Hyperparameter Tunning

```
[41] def build_model(learning_rate=3e-3):
         model = keras.models.Sequential()

         # Capas del modelo, Arquitectura 3
         model.add(keras.layers.InputLayer(input_shape=X_train.shape[1:]))
         keras.layers.Dense(5, activation='relu'),
         keras.layers.Dense(5, activation='relu'),
         keras.layers.Dense(5, activation='relu'),
         keras.layers.Dense(5, activation='relu'),
         keras.layers.Dense(1, activation='sigmoid')


         optimizer = keras.optimizers.SGD(learning_rate=learning_rate)
         model.compile(loss="binary_crossentropy", optimizer=optimizer)
         return model
```

```
[42] keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)

     keras_reg.fit(X_train, y_train, epochs=50,
                   validation_data=(X_valid, y_valid),
                   callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

Ejecutamos el método fit para entrenar el modelo.

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "learning_rate": reciprocal(3e-4, 3e-2).rvs(1000),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs, n_iter=10, cv=3, verbose=2)
rnd_search_cv.fit(X_train, y_train, epochs=20,
                  validation_data=(X_valid, y_valid))
```

Luego con la función RandomizedSearchCV de Sickit Learn buscaremos los Hyperparameters que pueden mejorar este modelo, esta sección es la más pesada con respecto al tiempo de ejecución.

```
rnd_search_cv.best_params_
```

```
{'learning_rate': 0.0032582942126651027}
```

```
rnd_search_cv.best_score_
```

```
-4.950124263763428
```

```
rnd_search_cv.best_estimator_
```

```
<keras.wrappers.scikit_learn.KerasRegressor at 0x7fd926a45e90>
```

```
rnd_search_cv.score(X_test, y_test)
```

```
1781/1781 [==============================] - 2s 1ms/step - loss: 1.3850
-1.3850030899047852
```

```
model = rnd_search_cv.best_estimator_.model
model
```

```
<keras.engine.sequential.Sequential at 0x7fd9201c44d0>
```

```
model.evaluate(X_test, y_test)
```

```
1781/1781 [==============================] - 2s 1ms/step - loss: 1.3850
1.3850030899047852
```

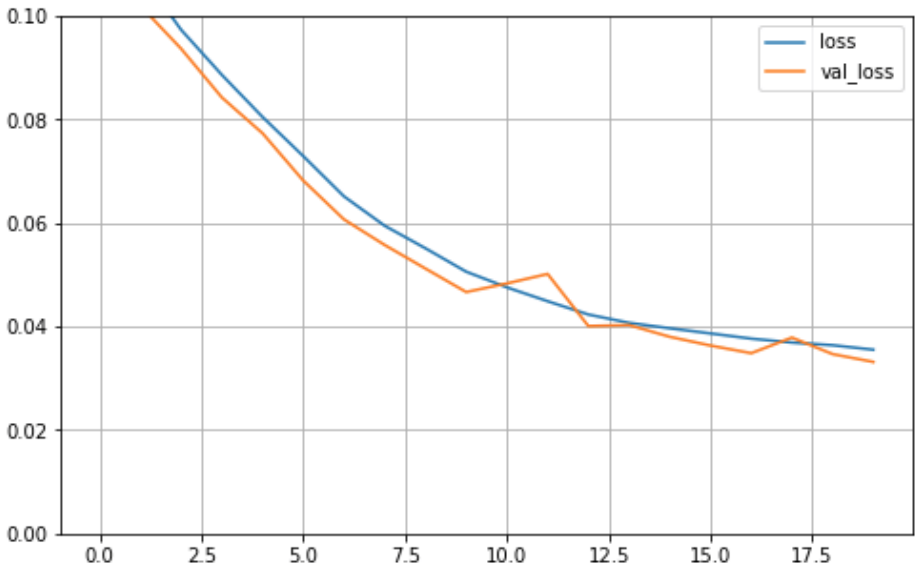
Ya con los resultados podemos ver que recomienda un learning_rate: **0.0032582942126651027**

```
arquitectura3_op = keras.models.Sequential([
    keras.layers.InputLayer(input_shape=X_train.shape[1:]),
    keras.layers.Dense(5, activation='relu'),
    keras.layers.Dense(5, activation='relu'),
    keras.layers.Dense(5, activation='relu'),
    keras.layers.Dense(5, activation='relu'),
    keras.layers.Dense(1, activation='sigmoid')
])
arquitectura3_op.compile(
    loss='binary_crossentropy',
    optimizer=keras.optimizers.SGD(learning_rate=0.0032582942126651027),
    metrics=[
        keras.metrics.Precision(name='precision'),
        keras.metrics.Recall(name='recall'),
        keras.metrics.Accuracy(name='accuracy')
      ])

history = arquitectura3_op.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))
```

Volvemos a crear el modelo con la arquitectura 3 pero en este caso con el learning_rate recomendado anteriormente

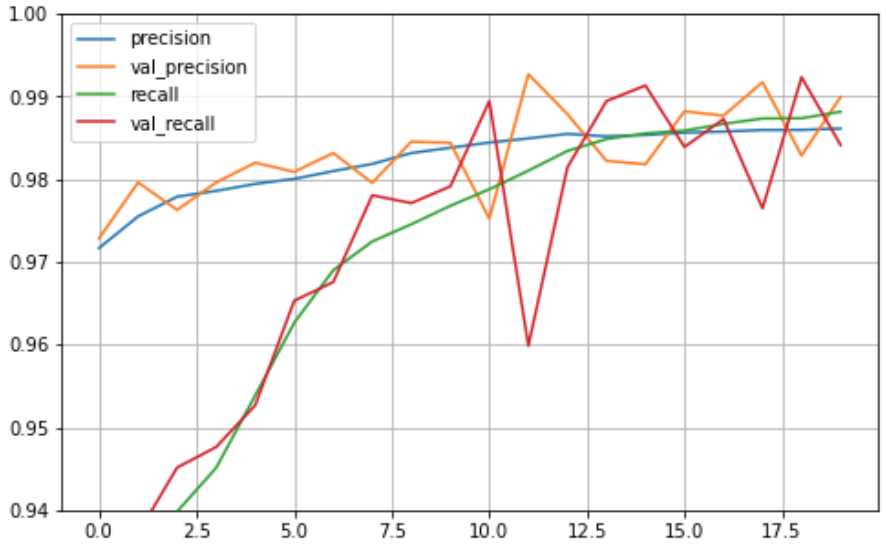
```
Epoch 1/20
11373/11373 [==============================] - 27s 2ms/step - loss: 0.2303 - precision: 0.9717 - recall: 0.8713 - accuracy: 0.0000e+00 -
Epoch 2/20
11373/11373 [==============================] - 38s 3ms/step - loss: 0.1082 - precision: 0.9755 - recall: 0.9352 - accuracy: 0.0000e+00 -
Epoch 3/20
11373/11373 [==============================] - 39s 3ms/step - loss: 0.0973 - precision: 0.9779 - recall: 0.9398 - accuracy: 0.0000e+00 -
Epoch 4/20
11373/11373 [==============================] - 38s 3ms/step - loss: 0.0886 - precision: 0.9786 - recall: 0.9451 - accuracy: 0.0000e+00 -
Epoch 5/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0805 - precision: 0.9794 - recall: 0.9538 - accuracy: 0.0000e+00 -
Epoch 6/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0729 - precision: 0.9800 - recall: 0.9626 - accuracy: 2.7478e-06 -
Epoch 7/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0651 - precision: 0.9810 - recall: 0.9690 - accuracy: 8.2435e-06 -
Epoch 8/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0595 - precision: 0.9818 - recall: 0.9725 - accuracy: 3.2974e-05 -
Epoch 9/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0551 - precision: 0.9831 - recall: 0.9745 - accuracy: 4.3966e-05 -
Epoch 10/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.0506 - precision: 0.9838 - recall: 0.9768 - accuracy: 4.3966e-05 -
Epoch 11/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0476 - precision: 0.9844 - recall: 0.9788 - accuracy: 5.2209e-05 -
Epoch 12/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0449 - precision: 0.9849 - recall: 0.9810 - accuracy: 6.0453e-05 -
Epoch 13/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.0423 - precision: 0.9855 - recall: 0.9834 - accuracy: 7.6940e-05 -
Epoch 14/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0407 - precision: 0.9852 - recall: 0.9848 - accuracy: 9.6175e-05 -
Epoch 15/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0397 - precision: 0.9853 - recall: 0.9855 - accuracy: 1.2915e-04 -
Epoch 16/20
11373/11373 [==============================] - 24s 2ms/step - loss: 0.0387 - precision: 0.9856 - recall: 0.9859 - accuracy: 1.4014e-04 -
Epoch 17/20
11373/11373 [==============================] - 26s 2ms/step - loss: 0.0377 - precision: 0.9857 - recall: 0.9867 - accuracy: 1.7311e-04 -
Epoch 18/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0369 - precision: 0.9859 - recall: 0.9873 - accuracy: 2.1158e-04 -
Epoch 19/20
11373/11373 [==============================] - 23s 2ms/step - loss: 0.0364 - precision: 0.9859 - recall: 0.9873 - accuracy: 2.3082e-04 -
Epoch 20/20
11373/11373 [==============================] - 25s 2ms/step - loss: 0.0355 - precision: 0.9861 - recall: 0.9881 - accuracy: 2.3082e-04 -
```

```python
loss_history = {
    'loss': history.history['loss'],
    'val_loss': history.history['val_loss']
}
pd.DataFrame(loss_history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 0.10)
plt.show()
```



Graficamos la curva de loss y val_loss, podemos ver que mejoro notablemente con respecto al modelo sin optimizar.

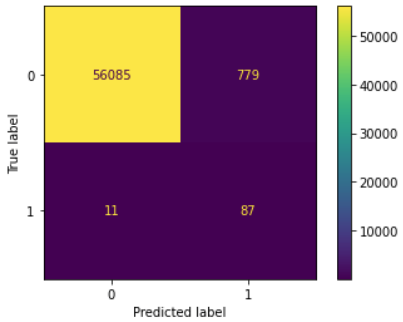
```
prec_rec_history = {
    'precision': history.history['precision'],
    'val_precision': history.history['val_precision'],
    'recall': history.history['recall'],
    'val_recall': history.history['val_recall']
}
pd.DataFrame(prec_rec_history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0.94, 1)
plt.show()
```



Al igual que la gráfica de loss podemos ver mejoría con respecto al modelo sin optimizar

```
predictions = arquitectura3_op.predict(X_test)
y_pred = (predictions > 0.4)
```

```
conf_matrix = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix)
disp.plot()
plt.show()
```




```
arquitectura3_op.evaluate(X_test, y_test)
```

```
1781/1781 [==============================] - 3s 2ms/step - loss: 0.0354 - precision: 0.1231 - recall: 0.8878 - accuracy: 4.7400e-04
[0.035427920520305634,
 0.12305516004562378,
 0.8877550959587097,
 0.00047400020412169397]
```

Para finalizar podemos ver en la matriz de confusión y la función evaluate que se ajustó mejor el modelo con respecto a la precisión y en este caso vemos menos clasificados en falsos positivos.

Conclusiones

El proceso de limpieza y análisis de datos fue crucial para el modelado del sistema de aprendizaje automático, ya que sin este paso, los resultados pudieran no reflejar la situación actual del problema, creando así en un modelo de predicción incorrecto e impreciso.

La selección del modelo de predicción de aprendizaje automático depende mucho de la naturaleza del problema, así como de la dimensión del conjunto de datos. En este caso, una red neuronal se usó para reconocer y clasificar transacciones fraudulentas de una entidad bancaria.

Para llegar a un buen modelo predictivo usando una red neuronal fue de alta importancia la selección y entonación de los hiperparámetros del modelo. Por esto, primero se probaron varios parámetros, específicamente variando el número de capas ocultas, el número de neuronas, y el tipo de función de activación. Luego se procedió a comparar y escoger el mejor modelo guiado por la menor tasa de error.

Sin embargo, este proceso manual no es el más óptimo para llegar a un modelo predictivo preciso, por lo que se tuvieron que aplicar otras técnicas para mejorar aún más el modelo. El proceso usado es denominado Hyperparameter Tunning. Usando Randomized Search, el proceso consiste en probar automáticamente múltiples combinaciones de parámetros, comparar los resultados y escoger el mejor modelo. También se usó una técnica denominada early-stopping, la cual disminuye los recursos de procesamiento, ya que descarta los modelos que no cumplan con un alto gradiente de cambio y mejora.

Referencias

[1] IBM. “El modelo de redes neuronales.” https://www.ibm.com/docs/es/spss-modeler/SaaS?topic=networks-neural-model.
[2] AWS. “¿Qué es una red neuronal?”. https://aws.amazon.com/es/what-is/neural-network/
[3] Wilmer R., Bertha M.. ”Redes neuronales artificiales aplicadas al reconocimiento de patrones”. Universidad Técnica de Machala. 2018. ISBN: 978-9942-24-100-9.