Detecting Credit Card Fraud Deep Learning Project

Introduction to Deep Learning Final (CU-Boulder MSDS)

by: Kevin Boyle, December 5th 2023

Problem Description

Hello! For this project, I am going to read in a dataset with anonymous credit card information. I got this dataset from the following Kaggle link/competition: https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud/data

With this dataset, I am going to see if I can use deep learning modeling and techniques to determine whether a transaction is fraudulent or not.

I am going to perform exploratory data analysis (EDA) and visualizations and do any necessary cleaning of the data, and then create a deep learning model which will analyze that cleaned data. From there, I will also do a comparison across a few other sequential deep learning models, and see which of the deep learning models is much improved for this type of analysis.

I will conclude whether or not utilizing deep learning techniques for this purpose would be appropriate for the problem at hand.

Thanks!

EDA Procedure

To begin, I will import all the necessary libraries and packages, as well as load in the player information dataset.

In [656…

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import re
import itertools
import keras
import sklearn.metrics as metrics
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

In [621…

```
ccinfo = pd.read_csv("creditcard.csv")
```

In [622…

```
ccinfo.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
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64  
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

In [623…

```
ccinfo.head(10)
```

Out[623]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.3637
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.2554
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.5146
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.3870
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.8177
5	2.0	-0.425966	0.960523	1.141109	-0.168252	0.420987	-0.029728	0.476201	0.260314	-0.5686
6	4.0	1.229658	0.141004	0.045371	1.202613	0.191881	0.272708	-0.005159	0.081213	0.4649
7	7.0	-0.644269	1.417964	1.074380	-0.492199	0.948934	0.428118	1.120631	-3.807864	0.6153
8	7.0	-0.894286	0.286157	-0.113192	-0.271526	2.669599	3.721818	0.370145	0.851084	-0.3920
9	9.0	-0.338262	1.119593	1.044367	-0.222187	0.499361	-0.246761	0.651583	0.069539	-0.7367

10 rows × 31 columns

In [624…

```
ccinfo.describe()
```

Out[624]:

	Time	V1	V2	V3	V4	V5	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.8480
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	1.487
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.3322
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.68
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.9856
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.3301

8 rows × 31 columns

In [625…

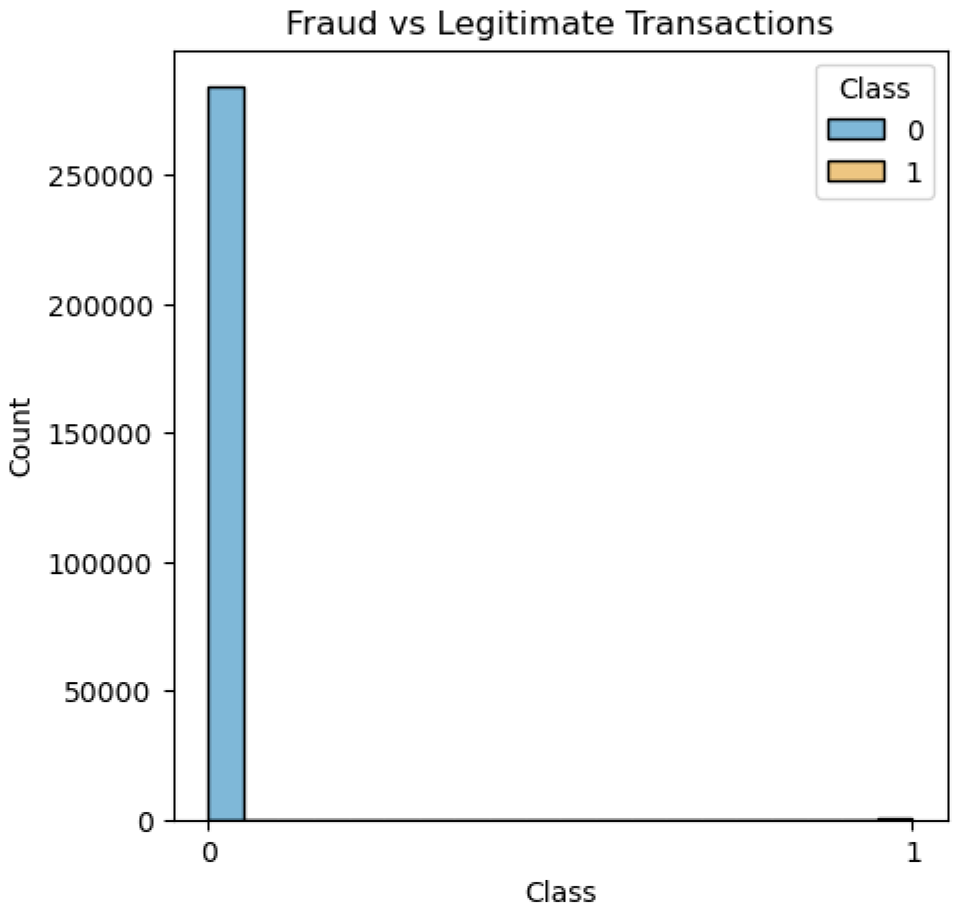
```
ccinfo['Class'].value_counts()
```

Out[625]:

```
Class
0    284315
1       492
Name: count, dtype: int64
```

In [626…

```
fig, ax = plt.subplots(figsize=(5, 5))
sns.histplot(
    data = ccinfo,
    x = 'Class',
    hue = 'Class',
    palette = 'colorblind', legend = True,
    ).set(
        title = 'Fraud vs Legitimate Transactions');
ax.locator_params(axis='x', integer=True)
```



In [627…
```
ccinfo['Class'].value_counts(1)
```

Out[627]:
```
Class
0    0.998273
1    0.001727
Name: proportion, dtype: float64
```

From these few commands and some additional information on the Kaggle website, I can determine some very important information.

- I can see there are 284,807 total transactions in the database.
- There are no null values.
- It contains only numerical input variables which are the result of a PCA transformation. Features V1, V2, ... V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning.
- The 'Class' column determines what is fraud and what is not. There are 492 (0.0017%) transactions that are fraud and 284315 (99.8273%) that are not. Therefore, this data is highly unbalanced to being not fraud.

Model Building and Analysis

I will now split the data into testing and training datasets. For my X, I will take in all the data except for the Class, and the y, I will use the Class column.

In [628…
```
X = ccinfo.drop(['Class'], axis = 1)
y = ccinfo['Class']
```

Next, I will split it into testing and training data.

In [629…
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=4
```

I will create my first sequential model to determine whether these transactions are fraudulent.

My initial model will have the following characteristics:

- 3 Dense layers, including 1 output layer with 2 outputs (1/0, for Fraud or Legit)
- Kernel initializer of normal.
- Initial input dimension of 30, since that's how many columns are in the X data.
- Relu activations throughout.
- The fit will have a batch_size of 25 and 20 epochs.

In [630…
```
model = Sequential()
model.add(Dense(X_train.shape[1], input_dim=30, kernel_initializer='normal', activation=
model.add(Dense(32, kernel_initializer='normal', activation='relu'))
model.add(Dense(2, kernel_initializer='normal', activation='relu'))
```

In [631…
```
model.compile(optimizer = keras.optimizers.legacy.Adam(learning_rate=0.01),
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_20"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_84 (Dense)            (None, 30)                930

 dense_85 (Dense)            (None, 32)                992

 dense_86 (Dense)            (None, 2)                 66

=================================================================
Total params: 1988 (7.77 KB)
Trainable params: 1988 (7.77 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [633…
```
model.fit(X_train, y_train, validation_split=0.2, batch_size=25, epochs=20, shuffle=True
```

```
Epoch 1/20
7292/7292 [==============================] - 51s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 2/20
7292/7292 [==============================] - 51s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 3/20
7292/7292 [==============================] - 52s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 4/20
7292/7292 [==============================] - 52s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 5/20
7292/7292 [==============================] - 52s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 6/20
7292/7292 [==============================] - 52s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 7/20
7292/7292 [==============================] - 51s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 8/20
7292/7292 [==============================] - 51s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 9/20
7292/7292 [==============================] - 51s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 10/20
7292/7292 [==============================] - 52s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 11/20
7292/7292 [==============================] - 52s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 12/20
7292/7292 [==============================] - 52s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 13/20
7292/7292 [==============================] - 53s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 14/20
7292/7292 [==============================] - 53s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 15/20
7292/7292 [==============================] - 53s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 16/20
7292/7292 [==============================] - 53s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 17/20
7292/7292 [==============================] - 51s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 18/20
7292/7292 [==============================] - 53s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 19/20
7292/7292 [==============================] - 53s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
Epoch 20/20
7292/7292 [==============================] - 53s 7ms/step - loss: 0.0261 - accuracy: 0.9
983 - val_loss: 0.0288 - val_accuracy: 0.9981
```

```
Out[633]: <keras.src.callbacks.History at 0x37982c460>
```

After fitting the model, I am going to run the predictions on the test data based on the model. Since it is binary with two outputs, I need to use the argmax function to get the max function between two outputs.

In [634…
```
predictions = np.argmax(model.predict(X_test),axis=1)
```

```
1781/1781 [==============================] - 3s 1ms/step
```

In [635…
```
y_arr = y_test.values
```

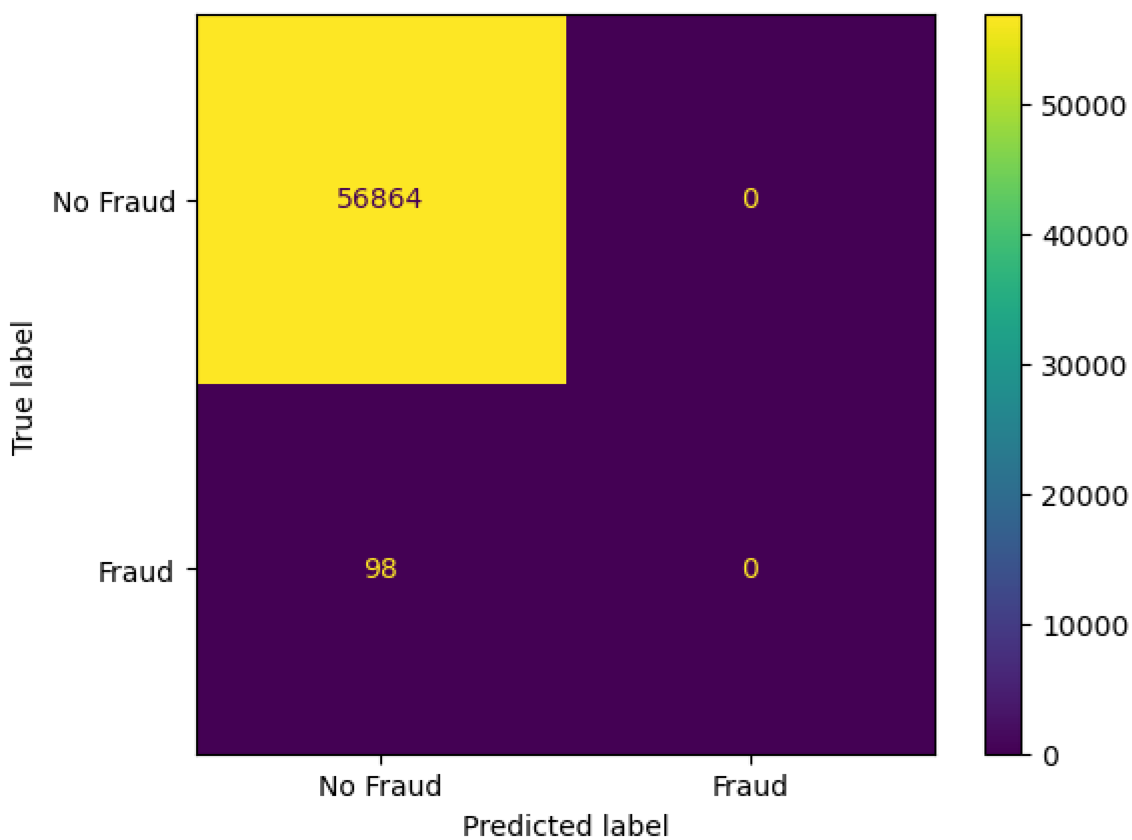
In [636…
```
len(predictions)
```

Out[636]:
```
56962
```

In [637…
```
num = 0
denom = len(predictions)

for x in range(len(predictions)):
    if predictions[x] == y_arr[x]:
        num += 1

print("The accuracy of the model is: " + str(num/denom))
```

```
The accuracy of the model is: 0.9982795547909132
```

In [638…
```
cm = confusion_matrix(y_test, predictions)
labels = ['No Fraud', 'Fraud']

disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                               display_labels=labels)
disp.plot()
plt.show()
```



The result ends up being 99.83% accurate between the y_test values and the prediction values. However, it appears as though the model predicted "No Fraud" for every line, so it may just be coincidence. I am going to tweak it to see if I can do better but also have the model not just say everything is "legitimate".

Now I will try to optimize parameters a bit more. For my next attempt attempt, I will modify the hyperparameters a bit more. I will use the softmax activation function at the end for mutual exclusivity between the two outputs, which are FRAUD and LEGITIMATE. In addition, I will use sparse_categorical_crossentropy loss. My batch size on fit will be 300 as opposed to 25, and epochs will remain at 20. I am hoping these slight tweaks will create an even better result!

In [647…
```
model2 = Sequential()
model2.add(Dense(X_train.shape[1], input_dim=30, kernel_initializer='normal', activation
model2.add(Dense(32, kernel_initializer='normal', activation='relu'))
model2.add(Dense(2, kernel_initializer='normal', activation='softmax'))
```

In [648…
```
model2.compile(optimizer = keras.optimizers.legacy.Adam(learning_rate=0.01),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model2.summary()
```

```
Model: "sequential_23"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_93 (Dense)            (None, 30)                930

 dense_94 (Dense)            (None, 32)                992

 dense_95 (Dense)            (None, 2)                 66

=================================================================
Total params: 1988 (7.77 KB)
Trainable params: 1988 (7.77 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [649…
```
model2.fit(X_train, y_train, validation_split=0.2, batch_size=300, epochs=20, shuffle=Tr
```

```
Epoch 1/20
608/608 [==============================] - 5s 7ms/step - loss: 42.2366 - accuracy: 0.991
7 - val_loss: 39.4052 - val_accuracy: 0.9980
Epoch 2/20
608/608 [==============================] - 4s 7ms/step - loss: 45.9404 - accuracy: 0.994
0 - val_loss: 98.3991 - val_accuracy: 0.9981
Epoch 3/20
608/608 [==============================] - 4s 7ms/step - loss: 121.9919 - accuracy: 0.99
43 - val_loss: 157.3178 - val_accuracy: 0.9981
Epoch 4/20
608/608 [==============================] - 4s 7ms/step - loss: 107.1712 - accuracy: 0.99
45 - val_loss: 241.9851 - val_accuracy: 0.9981
Epoch 5/20
608/608 [==============================] - 4s 7ms/step - loss: 239.1114 - accuracy: 0.99
32 - val_loss: 328.8581 - val_accuracy: 0.9981
Epoch 6/20
608/608 [==============================] - 4s 7ms/step - loss: 297.2598 - accuracy: 0.99
66 - val_loss: 172.0857 - val_accuracy: 0.9981
Epoch 7/20
608/608 [==============================] - 4s 7ms/step - loss: 258.8644 - accuracy: 0.99
66 - val_loss: 70.7450 - val_accuracy: 0.9974
Epoch 8/20
608/608 [==============================] - 5s 8ms/step - loss: 183.8157 - accuracy: 0.99
65 - val_loss: 221.7480 - val_accuracy: 0.9981
Epoch 9/20
608/608 [==============================] - 4s 7ms/step - loss: 77.1080 - accuracy: 0.997
5 - val_loss: 24.4290 - val_accuracy: 0.9980
Epoch 10/20
608/608 [==============================] - 4s 7ms/step - loss: 546.4597 - accuracy: 0.99
23 - val_loss: 874.1191 - val_accuracy: 0.9981
Epoch 11/20
608/608 [==============================] - 4s 7ms/step - loss: 459.5817 - accuracy: 0.99
67 - val_loss: 1051.6115 - val_accuracy: 0.9981
Epoch 12/20
608/608 [==============================] - 5s 7ms/step - loss: 621.8335 - accuracy: 0.99
50 - val_loss: 793.6611 - val_accuracy: 0.9981
Epoch 13/20
608/608 [==============================] - 4s 7ms/step - loss: 507.9813 - accuracy: 0.99
83 - val_loss: 48.2257 - val_accuracy: 0.9981
Epoch 14/20
608/608 [==============================] - 4s 7ms/step - loss: 480.0686 - accuracy: 0.99
67 - val_loss: 218.2258 - val_accuracy: 0.9981
Epoch 15/20
608/608 [==============================] - 4s 7ms/step - loss: 388.3762 - accuracy: 0.99
69 - val_loss: 130.3745 - val_accuracy: 0.9981
Epoch 16/20
608/608 [==============================] - 4s 7ms/step - loss: 1093.1620 - accuracy: 0.9
936 - val_loss: 1306.1116 - val_accuracy: 0.9981
Epoch 17/20
608/608 [==============================] - 4s 7ms/step - loss: 1061.7357 - accuracy: 0.9
961 - val_loss: 3038.8206 - val_accuracy: 0.9981
Epoch 18/20
608/608 [==============================] - 4s 7ms/step - loss: 1856.2539 - accuracy: 0.9
970 - val_loss: 3150.7549 - val_accuracy: 0.9981
Epoch 19/20
608/608 [==============================] - 4s 7ms/step - loss: 2559.7476 - accuracy: 0.9
967 - val_loss: 1869.5924 - val_accuracy: 0.9981
Epoch 20/20
608/608 [==============================] - 4s 7ms/step - loss: 1749.4158 - accuracy: 0.9
967 - val_loss: 2004.5817 - val_accuracy: 0.9981
```

Out[649]:
```
<keras.src.callbacks.History at 0x375016110>
```

In [651…
```
predictions = np.argmax(model2.predict(X_test),axis=1)
```

```
1781/1781 [==============================] - 3s 1ms/step
```

In [652…
```
y_arr = y_test.values
```

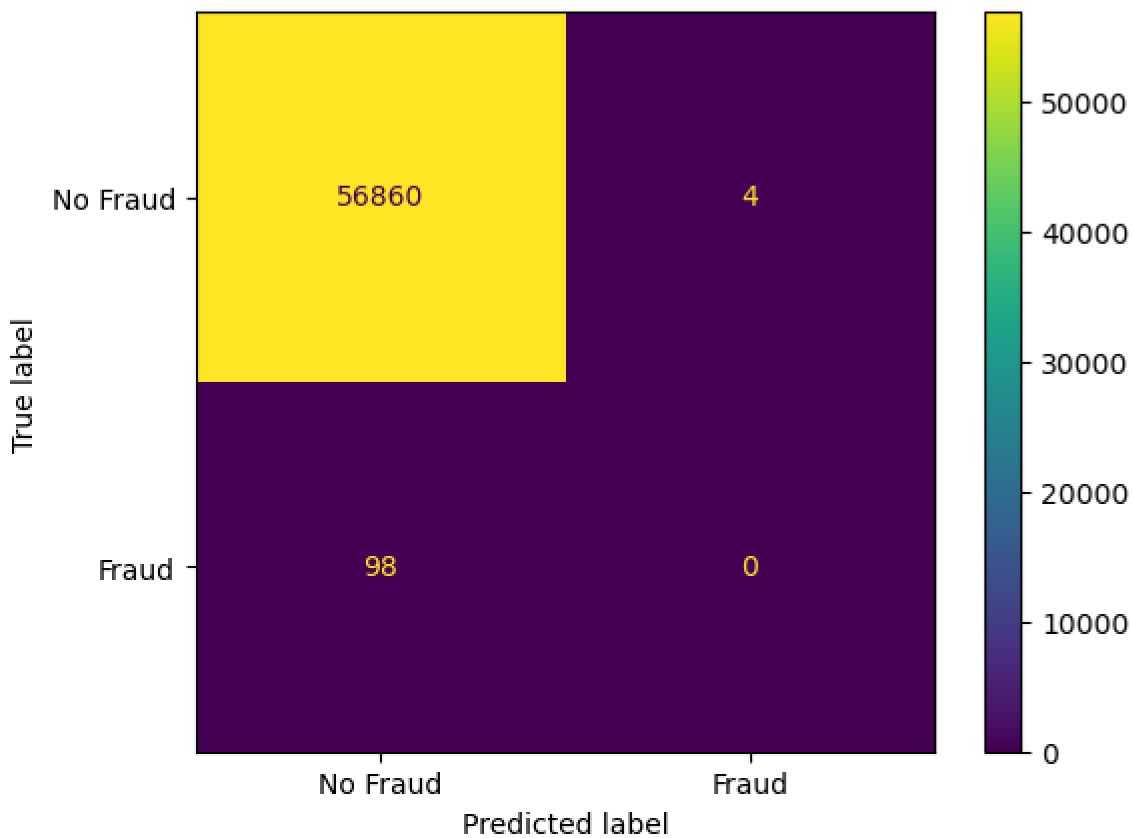
In [653…
```
len(predictions)
```

Out[653]:
```
56962
```

In [654…
```
num = 0
denom = len(predictions)

for x in range(len(predictions)):
    if predictions[x] == y_arr[x]:
        num += 1

print("The accuracy of the model is: " + str(num/denom))
```

```
The accuracy of the model is: 0.9982093325374811
```

In [655…
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(y_test, predictions)
labels = ['No Fraud', 'Fraud']

disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                               display_labels=labels)
disp.plot()
plt.show()
```



The second model with some "tweaks" to the hyperparameters ended up being slightly less accurate, at about 99.82%. However, this time, the model did think a few (4) of the lines were fraud, so it was no longer just saying "no fraud" for every single line item.

Results / Conclusion

In conclusion, I took in a dataset of credit card transactions that had undergone PCA. Some of these transactions were fraudulent, so I created some deep learning models to attempt if I could correctly predict/determine if a transaction was fraud or if it was a legitimate transaction.

Overall, I would say that my two models were successful, although it is a bit hard to determine just how successful based on how lopsided the data is. I got 99.8% accuracy for both models, although the first one would have been just as accurate if it just guessed "0" for each entry. At least the second one did try to guess "1" as fraud every once in a while.

I did learn a ton about deep learning and how to create sequential models using Keras, but here are a few things I would do differently if I was going to start over with this project:

- Pick a dataset that had more fraud, or at least fraud in higher percentages.
- Continue to optimize hyperparameters. Change up loss functions or optimizers, for example.
- Maybe accuracy is not the best metric here, and should be some other metric to track the success of this model.

Thank you for taking the time to read my project!