Jason
2401960183
Deep Learning - No 1

Import Libraries Needed :

- Numpy
- Pandas
- Tensorflow
- Keras
- Sklearn Metrics

In [1]:

```
import numpy as np
import pandas as pd
from tensorflow import keras
from keras.layers import Dense
from keras.models import Sequential
from sklearn.metrics import classification_report, precision_score, recall_score, f1_s
```

Read data from creditcard.csv using pandas

In [2]:

```
data = pd.read_csv('creditcard.csv')
data.head()
```

Out[2]:

	V1	V2	V3	V4	V5	V6	V7	V8	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.3637
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.2554
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.5146
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.3870
4	NaN	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.8177

Checking the total of rows and columns of the dataset

In [3]:

```
data.shape
```

Out[3]:

```
(284807, 11)
```

Check the info of each columns data type, here it shows that the Class has a int64 datatype because it is the value that determines if it is a fraud or not.

In [4]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 11 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   V1      284802 non-null  float64
 1   V2      284803 non-null  float64
 2   V3      284801 non-null  float64
 3   V4      284800 non-null  float64
 4   V5      284801 non-null  float64
 5   V6      284802 non-null  float64
 6   V7      284802 non-null  float64
 7   V8      284801 non-null  float64
 8   V9      284799 non-null  float64
 9   V10     284802 non-null  float64
 10  Class   284807 non-null  int64  
dtypes: float64(10), int64(1)
memory usage: 23.9 MB
```

1a. Data Preprocesssing

As i analyzes the dataset, there are some missing values across the dataframe. Therefore checking the total null count in each columns first.

In [5]:

```
data.isnull().sum()
```

Out[5]:

```
V1       5
V2       4
V3       6
V4       7
V5       6
V6       5
V7       5
V8       6
V9       8
V10      5
Class    0
dtype: int64
```

Drop the rows that contains the null values and replace the original dataframe

In [6]:

```python
data = data.dropna()
data.isna().sum()
```

Out[6]:

```
V1       0
V2       0
V3       0
V4       0
V5       0
V6       0
V7       0
V8       0
V9       0
V10      0
Class    0
dtype: int64
```

As we can see from the column V1 to V10 that it's input values have different scales which may affect the performance of the BPNN Model. The purpose of normalization is to convert the values of the dataset's numeric columns to a standard scale without losing information or distorting the ranges of values. Some algorithms need normalization in order to properly model the data. Therefore i am using the MinMaxScaler (Normalization) froim sklearn to normalize the data

In [7]:

```python
from sklearn.preprocessing import MinMaxScaler

cols = data.columns

scaler = MinMaxScaler()
data = scaler.fit_transform(data)
```

After normalizing the data the column names are replaced, therefore replacing it with the original columns

In [8]:

```python
data = pd.DataFrame(data, columns = cols.array)
data.head()
```

Out[8]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9
0	0.935192	0.766490	0.881365	0.313023	0.763439	0.267669	0.266815	0.786444	0.475312
1	0.978542	0.770067	0.840298	0.271796	0.766120	0.262192	0.264875	0.786298	0.453981
2	0.935217	0.753118	0.868141	0.268766	0.762329	0.281122	0.270177	0.788042	0.410603
3	0.941878	0.765304	0.868484	0.213661	0.765647	0.275559	0.266803	0.789434	0.414999
4	0.979184	0.768746	0.838200	0.305241	0.767008	0.265762	0.265324	0.786257	0.478797

1b. Splitting Dataset

Splitting the data into two parts first being the test set and residual set with 80% for test and 20% for residuals (consisting of test and validation). After that split the residual set into test and validation by giving the size 0.5 to split by two. Each having a total of 10% of the original dataset

In [9]:

```
from sklearn.model_selection import train_test_split

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


X_train, X_res, y_train, y_res = train_test_split(X,y, train_size=0.8)
X_valid, X_test, y_valid, y_test = train_test_split(X_res,y_res, test_size=0.5)

```

1c. Baseline Architecture of the BPNN Model

First determining the total of nodes in the input layer. Here n is the total of input columns and n_class is the value to be predicted which consists of a binary values (0 and 1).

In [10]:

```
n = len(X.columns) # 10
n_class = len(y.unique()) # 2
```

BPNN Model Building using Keras Sequential Model

- First layer is the input layer that has n nodes (10) with activation function ReLU
- Second layer is the hidden layer that has n*2 nodes (20) with activation function ReLU
- Third layer is the hidden layer that has n*2 nodes (20) with activation function ReLU
- Last layer is the output layer that has n_class nodes (2) with activation function of Softmax

In [11]:

```
model = Sequential()
model.add(Dense(n*2, input_shape=(n,), activation='relu'))
model.add(Dense(n*2, activation='relu'))
model.add(Dense(n_class, activation='softmax'))
```

Display the summary of the model

In [12]:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (None, 20)                220       
                                                                 
 dense_1 (Dense)             (None, 20)                420       
                                                                 
 dense_2 (Dense)             (None, 2)                 42        
                                                                 
=================================================================
Total params: 682
Trainable params: 682
Non-trainable params: 0
_________________________________________________________________
```

Defining the Hyperparameters

- Because the representation of "Class" values is 0 and 1 or (one hot encoding). Therefore in the event that there are two or more label classes, we will use this crossentropy loss function.

In [13]:

```
learning_rate = 0.04
epochs = 10
loss = "categorical_crossentropy"
metrics = ['accuracy']
```

Compile the model using ADAM Optimizer. Adaptive Moment Estimation is an algorithm for optimization technique. When dealing with complex problems involving a lot of data or factors, this optimizer is incredibly effective. It is effective and uses little memory.

In [14]:

```
model.compile(keras.optimizers.Adam(learning_rate=learning_rate), loss = loss ,metrics
```

Since we already have two classes (from y) that are integers, 1 meaning fraud, while 0 denotes not fraud. In order to reformat y into a 2-dimensional vector in terms of [1. 0.] or [0. 1.], one hot encoding will be used. This can be achieved by using keras.utils.to_categorical.

In [15]:

```
vectorized_y_train = keras.utils.to_categorical(y_train,num_classes=None)
vectorized_y_test = keras.utils.to_categorical(y_test,num_classes=None)
vectorized_y_valid =  keras.utils.to_categorical(y_valid,num_classes=None)
```

Fit the train dataset and validation data to the model.

In [16]:

```
model.fit(X_train, vectorized_y_train, validation_data=(X_valid,vectorized_y_valid), ep
```

```
Epoch 1/10
7119/7119 [==============================] - 12s 2ms/step - loss: 0.0132 - a
ccuracy: 0.9982 - val_loss: 0.0052 - val_accuracy: 0.9986
Epoch 2/10
7119/7119 [==============================] - 11s 2ms/step - loss: 0.0072 - a
ccuracy: 0.9987 - val_loss: 0.0045 - val_accuracy: 0.9985
Epoch 3/10
7119/7119 [==============================] - 11s 2ms/step - loss: 0.0067 - a
ccuracy: 0.9988 - val_loss: 0.0059 - val_accuracy: 0.9990
Epoch 4/10
7119/7119 [==============================] - 12s 2ms/step - loss: 0.0062 - a
ccuracy: 0.9989 - val_loss: 0.0046 - val_accuracy: 0.9985
Epoch 5/10
7119/7119 [==============================] - 11s 2ms/step - loss: 0.0060 - a
ccuracy: 0.9989 - val_loss: 0.0043 - val_accuracy: 0.9992
Epoch 6/10
7119/7119 [==============================] - 11s 2ms/step - loss: 0.0058 - a
ccuracy: 0.9989 - val_loss: 0.0037 - val_accuracy: 0.9985
Epoch 7/10
7119/7119 [==============================] - 11s 2ms/step - loss: 0.0057 - a
ccuracy: 0.9990 - val_loss: 0.0046 - val_accuracy: 0.9992
Epoch 8/10
7119/7119 [==============================] - 13s 2ms/step - loss: 0.0058 - a
ccuracy: 0.9990 - val_loss: 0.0042 - val_accuracy: 0.9991
Epoch 9/10
7119/7119 [==============================] - 12s 2ms/step - loss: 0.0056 - a
ccuracy: 0.9990 - val_loss: 0.0040 - val_accuracy: 0.9990
Epoch 10/10
7119/7119 [==============================] - 11s 2ms/step - loss: 0.0057 - a
ccuracy: 0.9990 - val_loss: 0.0081 - val_accuracy: 0.9989
```

Out[16]:

```
<keras.callbacks.History at 0x1c3a14e3970>
```

1e. Analyze the Performance of the Model

Here to evaluate the model by using the sklearn metrics library. Accuracy can be found from the evaluate function provided by keras library to display the accuracy of the current model. Predicted values of the models are therefore evaluated to find the precision, recall, and f1_score

In [17]:

```
def evaluate_model():
    labels=["Normal","Fraud"]

    accuracy = model.evaluate(X_test,vectorized_y_test)[1]

    prediction_control = model.predict(X_test, verbose=1)

    y_pred = np.argmax(prediction_control,axis=1)
    precision = precision_score(y_test,y_pred,labels=labels)
    recall = recall_score(y_test,y_pred,labels=labels)
    f1 = f1_score(y_test,y_pred,labels=labels)

    print("Accuracy: {}\n".format(accuracy))
    print("Fraud Precision:{}".format(precision))
    print("Fraud Recall:{}".format(recall))
    print("Fraud F1-Score:{}\n".format(f1))

    print(classification_report(y_test,y_pred, target_names=labels,digits=8))


evaluate_model()
```

```
890/890 [==============================] - 1s 1ms/step - loss: 0.0083 - accu
racy: 0.9989
890/890 [==============================] - 1s 1ms/step
Accuracy: 0.9989113807678223

Fraud Precision:0.660377358490566
Fraud Recall:0.7291666666666666
Fraud F1-Score:0.693069306930693

              precision    recall  f1-score   support

      Normal  0.99954262 0.99936682 0.99945471     28428
       Fraud  0.66037736 0.72916667 0.69306931        48

    accuracy                      0.99891136     28476
   macro avg  0.82995999 0.86426674 0.84626201     28476
weighted avg  0.99897092 0.99891136 0.99893826     28476
```

1d. Hyperparameters Tuning

Based on the results of each epochs iteration, it shows that the loss values are mostly stagnan meaning that the learning rate is lowly affecting the results in each iteration, therefore here optimizing the learning rate is important.

Tuned hyperparamaters :

- Epochs. A gradient descent hyperparameter that regulates the quantity of full iterations across the training dataset is the number of epochs.
- Batch Size. Gradient descent's batch size hyperparameter determines how many training data must be processed before the model's internal parameters are changed.
- Learning Rate. When the model weights are changed, the learning rate is a hyperparameter that regulates how much to alter the model in response to the estimated error.

In [18]:

```
learning_rate = 0.001
epochs = 10
batch_size = 20
loss = "categorical_crossentropy"
metrics = ['accuracy']
```

Compile the model optimizers once again using AdamOptimizers with hyperparameters that were tuned

In [19]:

```
model.compile(keras.optimizers.Adam(learning_rate=learning_rate), loss = loss ,metrics
```

Fit the train dataset and validation data to the model.

In [20]:

```
model.fit(X_train, vectorized_y_train, validation_data=(X_valid,vectorized_y_valid), ep
```

```
Epoch 1/10
11391/11391 [==============================] - 20s 2ms/step - loss: 0.0041 -
accuracy: 0.9991 - val_loss: 0.0035 - val_accuracy: 0.9992
Epoch 2/10
11391/11391 [==============================] - 19s 2ms/step - loss: 0.0040 -
accuracy: 0.9991 - val_loss: 0.0035 - val_accuracy: 0.9991
Epoch 3/10
11391/11391 [==============================] - 19s 2ms/step - loss: 0.0040 -
accuracy: 0.9991 - val_loss: 0.0036 - val_accuracy: 0.9992
Epoch 4/10
11391/11391 [==============================] - 19s 2ms/step - loss: 0.0040 -
accuracy: 0.9991 - val_loss: 0.0036 - val_accuracy: 0.9992
Epoch 5/10
11391/11391 [==============================] - 18s 2ms/step - loss: 0.0040 -
accuracy: 0.9991 - val_loss: 0.0035 - val_accuracy: 0.9991
Epoch 6/10
11391/11391 [==============================] - 19s 2ms/step - loss: 0.0040 -
accuracy: 0.9991 - val_loss: 0.0038 - val_accuracy: 0.9991
Epoch 7/10
11391/11391 [==============================] - 18s 2ms/step - loss: 0.0040 -
accuracy: 0.9991 - val_loss: 0.0036 - val_accuracy: 0.9991
Epoch 8/10
11391/11391 [==============================] - 19s 2ms/step - loss: 0.0040 -
accuracy: 0.9991 - val_loss: 0.0035 - val_accuracy: 0.9992
Epoch 9/10
11391/11391 [==============================] - 19s 2ms/step - loss: 0.0040 -
accuracy: 0.9991 - val_loss: 0.0035 - val_accuracy: 0.9991
Epoch 10/10
11391/11391 [==============================] - 20s 2ms/step - loss: 0.0040 -
accuracy: 0.9991 - val_loss: 0.0035 - val_accuracy: 0.9991
```

Out[20]:

```
<keras.callbacks.History at 0x1c3a7274b50>
```

Evaluate the hypertuned model using the defined function in earlier stages. Here as we can see from the results below we have a better performance after tuning the hyperparameters on the BPNN Model.

In [21]:

```
evaluate_model()
```

```
890/890 [==============================] - 1s 1ms/step - loss: 0.0043 - accu
racy: 0.9989
890/890 [==============================] - 1s 1ms/step
Accuracy: 0.998946487903595

Fraud Precision:0.75
Fraud Recall:0.5625
Fraud F1-Score:0.6428571428571429

              precision    recall  f1-score   support

      Normal  0.99926160 0.99968341 0.99947246     28428
       Fraud  0.75000000 0.56250000 0.64285714        48

    accuracy                      0.99894648     28476
   macro avg  0.87463080 0.78109171 0.82116480     28476
weighted avg  0.99884144 0.99894648 0.99887134     28476
```