Predicting Default Payments with Fully-Connected NNs

The dataset contains information on default payments, demographic factors, credit data, history of payment, and bill statements of credit card clients in Taiwan from April 2005 to September 2005.

```
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
```

```
    Mounted at /content/drive
```

Inspecting the data

any comment about data dimensionality/distribution goes here

Librerie necessarie

```
%matplotlib inline
import numpy as np
import pandas as pd

from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn import preprocessing

import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation
```

Importo i dataset precedentemente collegati tramite drive

```
X_train = pd.read_csv('drive/MyDrive/Colab_Notebooks/X_train.csv')
Y_train = pd.read_csv('drive/MyDrive/Colab_Notebooks/y_train.csv')
Test = pd.read_csv('drive/MyDrive/Colab_Notebooks/X_test.csv')
```

Visualizzo i dati e faccio alcune analisi esplorative sul target

```
print (X_train.shape, Y_train.shape)
print (Test.shape)
```

```
    (24000, 24) (24000, 2)
    (6000, 24)
```

```
X_train.info()
X_train.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24000 entries, 0 to 23999
Data columns (total 24 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   ID         24000 non-null  int64  
 1   LIMIT_BAL  24000 non-null  float64
 2   SEX        24000 non-null  int64  
 3   EDUCATION  24000 non-null  int64  
 4   MARRIAGE   24000 non-null  int64  
 5   AGE        24000 non-null  int64  
 6   PAY_0      24000 non-null  int64  
 7   PAY_2      24000 non-null  int64  
 8   PAY_3      24000 non-null  int64  
 9   PAY_4      24000 non-null  int64  
 10  PAY_5      24000 non-null  int64  
 11  PAY_6      24000 non-null  int64  
 12  BILL_AMT1  24000 non-null  float64
 13  BILL_AMT2  24000 non-null  float64
 14  BILL_AMT3  24000 non-null  float64
 15  BILL_AMT4  24000 non-null  float64
 16  BILL_AMT5  24000 non-null  float64
 17  BILL_AMT6  24000 non-null  float64
 18  PAY_AMT1   24000 non-null  float64
 19  PAY_AMT2   24000 non-null  float64
 20  PAY_AMT3   24000 non-null  float64
 21  PAY_AMT4   24000 non-null  float64
 22  PAY_AMT5   24000 non-null  float64
 23  PAY_AMT6   24000 non-null  float64
dtypes: float64(13), int64(11)
memory usage: 4.4 MB
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	
count	24000.000000	24000.000000	24000.000000	24000.000000	24000.000000	24000
mean	15010.821708	167226.653333	1.604917	1.854000	1.551417	35
std	8680.406114	129734.959196	0.488879	0.792176	0.522766	9
min	1.000000	10000.000000	1.000000	0.000000	0.000000	21
25%	7452.500000	50000.000000	1.000000	1.000000	1.000000	28
50%	15061.500000	140000.000000	2.000000	2.000000	2.000000	34
75%	22509.250000	240000.000000	2.000000	2.000000	2.000000	42
max	29999.000000	1000000.000000	2.000000	6.000000	3.000000	79

```
Y_train.info()
Y_train.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24000 entries, 0 to 23999
Data columns (total 2 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   ID                          24000 non-null  int64
 1   default.payment.next.month  24000 non-null  int64
dtypes: int64(2)
memory usage: 375.1 KB
```

	ID	default.payment.next.month
count	24000.000000	24000.000000
mean	15010.821708	0.221792
std	8680.406114	0.415460
min	1.000000	0.000000
25%	7452.500000	0.000000
50%	15061.500000	0.000000
75%	22509.250000	0.000000
max	29999.000000	1.000000

```
Test.info()
Test.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6000 entries, 0 to 5999
Data columns (total 24 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   ID         6000 non-null   int64
 1   LIMIT_BAL  6000 non-null   float64
 2   SEX        6000 non-null   int64
 3   EDUCATION  6000 non-null   int64
 4   MARRIAGE   6000 non-null   int64
 5   AGE        6000 non-null   int64
 6   PAY_0      6000 non-null   int64
 7   PAY_2      6000 non-null   int64
 8   PAY_3      6000 non-null   int64
 9   PAY_4      6000 non-null   int64
 10  PAY_5      6000 non-null   int64
 11  PAY_6      6000 non-null   int64
 12  BILL_AMT1  6000 non-null   float64
 13  BILL_AMT2  6000 non-null   float64
 14  BILL_AMT3  6000 non-null   float64
 15  BILL_AMT4  6000 non-null   float64
 16  BILL_AMT5  6000 non-null   float64
 17  BILL_AMT6  6000 non-null   float64
 18  PAY_AMT1   6000 non-null   float64
 19  PAY_AMT2   6000 non-null   float64
 20  PAY_AMT3   6000 non-null   float64
 21  PAY_AMT4   6000 non-null   float64
 22  PAY_AMT5   6000 non-null   float64
 23  PAY_AMT6   6000 non-null   float64
dtypes: float64(13), int64(11)
memory usage: 1.1 MB
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AG
count	6000.000000	6000.000000	6000.000000	6000.000000	6000.000000	6000.0000
mean	14959.213167	168515.000000	1.599000	1.849667	1.553667	35.4500
std	8580.495129	129804.158748	0.490142	0.783051	0.518811	9.1492
min	7.000000	10000.000000	1.000000	0.000000	0.000000	21.0000
25%	7643.250000	50000.000000	1.000000	1.000000	1.000000	28.0000
50%	14786.500000	140000.000000	2.000000	2.000000	2.000000	34.0000
75%	22437.750000	240000.000000	2.000000	2.000000	2.000000	41.0000
max	30000.000000	800000.000000	2.000000	6.000000	3.000000	72.0000

Noto che i valori non sono tutti dello stesso tipo, sarà quindi necessario portarli tutti in float64 per utilizzarli al meglio.

Inoltre, in accordo con il Data Dictionary fornito si può notare che nelle colonne **EDUCATION** e **MARRIAGE** il valore minimo presente è zero anche se il valore minimo che tale colonna può assumere è uno: per questo motivo, nella fase di preparazione dei dati, porteremo tutti gli 0 per **EDUCATION** a 6 (unknown) e per **MARRIAGE** a 3 (other).

```
print(X_train['EDUCATION'].value_counts())
```

```
2    11186
1     8481
3     3959
5      224
4       97
6       43
0       10
Name: EDUCATION, dtype: int64
```

```
print(X_train['MARRIAGE'].value_counts())
```

```
2    12747
1    10942
3      266
0       45
Name: MARRIAGE, dtype: int64
```

Controllo quindi se il target sia sbilanciato.

```
Y_train['default.payment.next.month'].value_counts()
```

```
0    18677
1     5323
Name: default.payment.next.month, dtype: int64
```

Noto che il dataset è profondamente sbilanciata, troviamo che la classe di maggioranza è **default.payment.next.month = 0** sarà quindi necessario bilanciarla.

Preparing the data

describe the choice made during the preprocessing operations, also taking into account the previous considerations during the data inspection.

Per prepare in modo più efficiente i dati riunisco **X_train** e **Y_train** in un unico dataset **df** e su esso eseguo il bilanciamento del target.

```
df = pd.merge(X_train, Y_train, on='ID')
```

```
df.describe()
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	
count	24000.000000	24000.000000	24000.000000	24000.000000	24000.000000	24000
mean	15010.821708	167226.653333	1.604917	1.854000	1.551417	35
std	8680.406114	129734.959196	0.488879	0.792176	0.522766	9
min	1.000000	10000.000000	1.000000	0.000000	0.000000	21
25%	7452.500000	50000.000000	1.000000	1.000000	1.000000	28
50%	15061.500000	140000.000000	2.000000	2.000000	2.000000	34
75%	22509.250000	240000.000000	2.000000	2.000000	2.000000	42
max	29999.000000	1000000.000000	2.000000	6.000000	3.000000	79

Come precedentemente citato correggo i valori di **EDUCATION** e **MARRIAGE**

```
fil = (df.EDUCATION == 0)
df.loc[fil, 'EDUCATION'] = 6
df.EDUCATION.value_counts()
```

```
2    11186
1     8481
3     3959
5      224
4       97
6       53
Name: EDUCATION, dtype: int64
```

```
df.loc[df.MARRIAGE == 0, 'MARRIAGE'] = 3
df.MARRIAGE.value_counts()
```

```
2    12747
1    10942
3      311
Name: MARRIAGE, dtype: int64
```

Correggo quindi il bilanciamento del target effettuando un downsampling della classe maggioritaria

```
df = df.sort_values(by=['default.payment.next.month'])


df = df.iloc[10000:]


#Downsampling
df = df.sample(frac=1)


df['default.payment.next.month'].value_counts()
```

```
0    8677
1    5323
Name: default.payment.next.month, dtype: int64
```

Ricostruisco quindi la divisione tra il target e le altre variabili.

```
X_train_temp = df
Y_train_temp = df[['ID','default.payment.next.month']]
Y_train_temp['default.payment.next.month'].value_counts()
Y_train_temp
```

	ID	default.payment.next.month
1795	9323	0
636	13264	0
695	29915	0
7121	21290	0
11844	19252	1
...	...	...
2401	11306	1
225	23993	0
13354	4839	1
22988	10358	0
23870	29825	1

14000 rows × 2 columns

```
X_train_temp.drop(['ID','SEX','default.payment.next.month'],axis=1,inplace=True)
Y_train_temp.drop(['ID'],axis=1,inplace=True)
Y_train_temp
```

```
/usr/local/lib/python3.7/dist-packages/pandas/core/frame.py:4174: SettingWi
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs
  errors=errors,
```

	default.payment.next.month
1795	0
636	0
695	0
7121	0
11844	1
...	...
2401	1
225	0
13354	1
22988	0
23870	1

14000 rows × 1 columns

Divido i dati per l'addestramento e la validazione.

```
X_train, X_val, Y_train, Y_val = train_test_split(X_train_temp, Y_train_temp, te
```

```
print(X_train.shape, Y_train.shape, X_val.shape,Y_val.shape, X_train_temp.shape)
```

```
(11200, 22) (11200, 1) (2800, 22) (2800, 1) (14000, 22)
```

Per completare la preparazione dei dati li converto interamente in **float32** ed effettuo la normalizzazione.

```
X_train = X_train.astype('float32')
X_val = X_val.astype('float32')
```

```
scaler = preprocessing.MinMaxScaler((0,1))
scaler.fit(X_train_temp)

XX_train = scaler.transform(X_train.values)
XX_val  = scaler.transform(X_val.values)

YY_train = Y_train.values
YY_val  = Y_val.values


print (XX_train.shape, YY_train.shape, XX_val.shape, YY_val.shape)
```

```
    (11200, 22) (11200, 1) (2800, 22) (2800, 1)
```

Building the network

Definisco le metriche che mi serviranno per valutare il modello.

```
from keras import backend as K

def recall_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
    recall = true_positives / (possible_positives + K.epsilon())
    return recall

def precision_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
    precision = true_positives / (predicted_positives + K.epsilon())
    return precision

def f1_m(y_true, y_pred):
    precision = precision_m(y_true, y_pred)
    recall = recall_m(y_true, y_pred)
    return 2*((precision*recall)/(precision+recall+K.epsilon()))
```

Per la costruzione del modello ho deciso di creare la mia rete neurale con tre Layer nascosti rispettivamente da 256, 128, 64 che dopo numerose prove è risulato essere il miglior numero di nodi e layer da impiegare.

L'ottimizzatore scelto è SGD con un learning rate di *0.005*.

La funzione di attivazione scelta è una di quelle standard ovvero **relu** e come funzione di attivazione del layer di output **sigmoid** perchè è più efficente dal punto di vista computazionale rispetto a **softmax**.

```
nb_classes = 1

initializer = keras.initializers.GlorotUniform(seed=1234)

# Modello
model = Sequential()
model.add(Dense(256, input_shape=(X_train.shape[1],), activation = "relu", kerne
model.add(Dense(128, activation = "relu", kernel_initializer=initializer))
model.add(Dense(64, activation = "relu", kernel_initializer=initializer))
model.add(Dense(nb_classes, activation='sigmoid',kernel_initializer=initializer)

from tensorflow.keras.optimizers import SGD
model.compile(optimizer=SGD(learning_rate=0.005), loss='binary_crossentropy',met
```

Dopo una serie di prove ho trovato che la grandezza migliore per il **batch_size** è 128.

```
n_epochs = 300
batch_size = 128

history = model.fit(XX_train, YY_train, epochs=n_epochs, batch_size=batch_size,
```

```
Epoch 1/300
88/88 [==============================] - 2s 9ms/step - loss: 0.6929 - acc
Epoch 2/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6793 - acc
Epoch 3/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6718 - acc
Epoch 4/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6670 - acc
Epoch 5/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6635 - acc
Epoch 6/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6607 - acc
Epoch 7/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6584 - acc
Epoch 8/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6564 - acc
Epoch 9/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6547 - acc
Epoch 10/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6530 - acc
Epoch 11/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6514 - acc
Epoch 12/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6499 - acc
Epoch 13/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6483 - acc
```

```
88/88 [==============================] - 0s 5ms/step - loss: 0.6485 - acc
Epoch 14/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6468 - acc
Epoch 15/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6451 - acc
Epoch 16/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6433 - acc
Epoch 17/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6415 - acc
Epoch 18/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6398 - acc
Epoch 19/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6380 - acc
Epoch 20/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6362 - acc
Epoch 21/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6344 - acc
Epoch 22/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6326 - acc
Epoch 23/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6307 - acc
Epoch 24/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6289 - acc
Epoch 25/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6270 - acc
Epoch 26/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6251 - acc
Epoch 27/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6233 - acc
Epoch 28/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6214 - acc
Epoch 29/300
88/88 [==============================] - 0s 5ms/step - loss: 0.6195 - acc
Epoch 30/300
```

Analyze and comment the training results

here goes any comment/visualization of the training history and any initial consideration on the training results

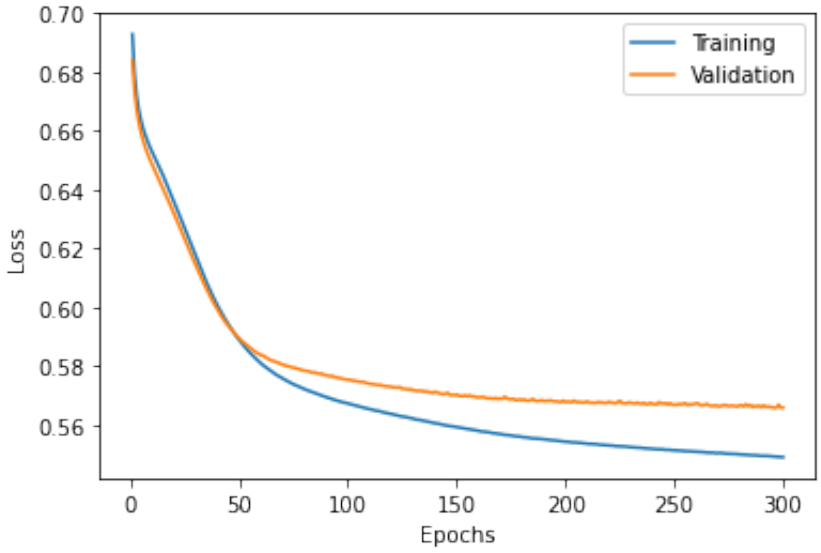
Visualizzo ora il training history e il grafico che riassume la loss e l'accuracy del training e della validazione.

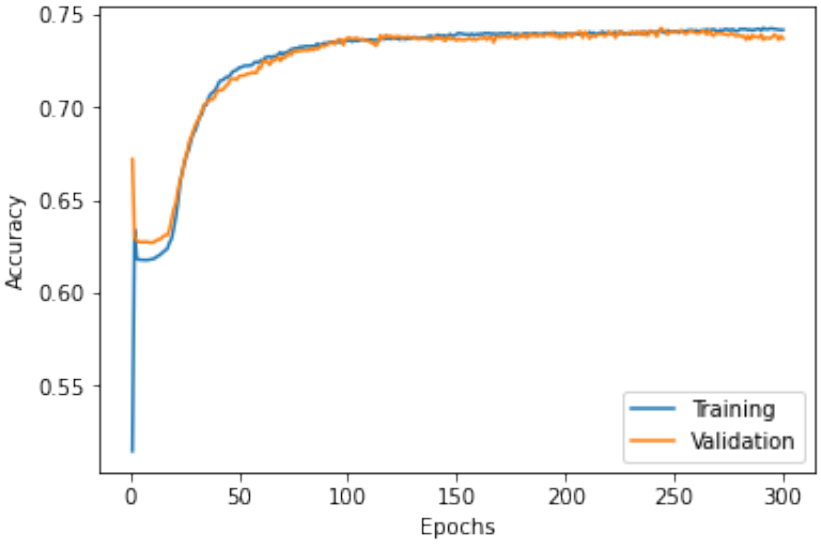
```
print('history dict:', history.history)
```

```
history dict: {'loss': [0.6928974986076355, 0.679283857345581, 0.6718062758
```

```
#plot training history

from matplotlib import pyplot as plt
```

```
x_plot = list(range(1,n_epochs+1))

def plot_history(network_history):
    plt.figure()
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    #plt.ylim(0.6,0.8)#just for better viz
    plt.plot(x_plot, network_history.history['loss'])
    plt.plot(x_plot, network_history.history['val_loss'])
    plt.legend(['Training', 'Validation'])

    plt.figure()
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.plot(x_plot, network_history.history['accuracy'])
    plt.plot(x_plot, network_history.history['val_accuracy'])
    plt.legend(['Training', 'Validation'], loc='lower right')
    plt.show()


plot_history(history)
```





Dal grafico possiamo vedere come la trainingloss tenda a diminuire e questo è una cosa positiva perchè l'obbiettivo del problema di ottimizzazione è minimizzarla. La validation loss vediamo che tende a rimanere discostata e iniziare ad alzarsi, segno che con l'aumentare delle epoche avremo un overfitt ovvero che avremmo memorizzato alla perfezione il trainig set e il modello fallirà sul validation data. Questo andamento è segno che non bisogna aumentare le epoche.

Per l'accuracy vediamo che si attesta sul 75% e possiamo osservare come la training e la validation accuracy siano simili che ci da un indicazione sul fatto che non stiamo andando in completo overfitting, ma possiamo già intuire che se proseguiamo con le epoche il training tenderà ad avvicinarsi ad uno e la validation a crollare come già sembra fare intorno alle 260 epoche trovandoci quindi in un caso di overfitting.

Validate the model and comment the results

La valutazione del modello si basa su loss,accuracy, f1, precision e recall per avere un idea il più completa possibile del modello.

```
loss, accuracy, f1_score, precision, recall = model.evaluate(XX_val, YY_val, bat
```

```
    22/22 [==============================] - 0s 2ms/step - loss: 0.5659 - accur
```

```
print("\n%s: %.2f" % ("loss", loss))
print("\n%s: %.2f" % ("accuracy", accuracy))
print("\n%s: %.2f" % ("f1_score", f1_score))
print("\n%s: %.2f" % ("precision", precision))
print("\n%s: %.2f" % ("recall", recall))
```

```
    loss: 0.57

    accuracy: 0.74

    f1_score: 0.57

    precision: 0.73

    recall: 0.47
```

Il modello in particolare presenta un buon livello di f1_score ma non altissimo da assicurare una buona capacità del modello, si può anche notare come l'accuracy e la precision non siano molto alta se pure buone.

Make predictions (on the provided test set)

Date le misure di performance precedentemnte illustrate penso che il modello potrebbe non performare al meglio su un dei dai che non sono mai stati visti.

```
# preprocessing per fare le predizioni
Test.drop(['ID','SEX'], axis=1, inplace=True)
test = Test.astype('float32')
test = scaler.transform(test.values)
```

```
predictions = model.predict(test)
print('predictions shape:', predictions.shape)
predictions[:10]
```

```
predictions shape: (6000, 1)
array([[0.2647308 ],
       [0.22536147],
       [0.2619921 ],
       [0.2816325 ],
       [0.31144792],
       [0.22998843],
       [0.44146666],
       [0.52903944],
       [0.20752895],
       [0.33353367]], dtype=float32)
```

OPTIONAL -- Export the predictions in the format indicated in the assignment release page.

```
y_classes = (predictions > 0.5).astype(np.uint8)
unique, counts = np.unique(y_classes, return_counts=True)
dict(zip(unique, counts))
```

```
{0: 4919, 1: 1081}
```

```
np.savetxt("Artemisia_Sarteschi_829677_score2.txt",y_classes, fmt='%-d')
```

OPTIONAL -- Implement some regularization methods of your choice and make a comparison between (training/validation) performances of regularized models (also compare with the case of no regularization)

Attempts in this section will be taken into account, if well-enough done, to (at least partially) compensate for potential incorrectessness in the mandatory sections. On the other hand, any incorrectessness in *this* section won't be taken into account in the final score.