Introduction to Deep Learning

2022 - Aurelien COULOUMY

1. Introduction to Neural networks

➤ Exercice 1.1 - Basic perceptron ★☆☆☆☆

1.1.1 Import relevant libraries such as Numpy

```
import numpy as np
import ipywidgets as widgets
from ipywidgets import interact, interact_manual
```

1.1.2 Define an activation function, for instance logistic one.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

1.1.3 Create a class that compute feedforward perceptron process.

```
class Neuron:

  def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias

  def feedforward(self, inputs):
    total = np.dot(self.weights, inputs) + self.bias
    return sigmoid(total)
```

1.1.4 Test your work by selecting randomly inputs, related weights and bias.

```
from random import randrange
```

```
x1 = randrange(10)
x2 = randrange(10)

w1 = randrange(10)
w2 = randrange(10)

b = randrange(10)

print(x1, x2, w1, w2, b)
```

```
x = np.array([x1, x2])
weights = np.array([w1, w2])

n = Neuron(weights, b)
n.feedforward(x)
```

```
@interact
def inputs(x1=(-10,10,1), x2=(-10,10,1), w1=(-10,10,1), w2=(-10,10,1), b=(-10,10,1)):
    x = np.array([x1, x2])
    weights = np.array([w1, w2])
    n = Neuron(weights, b)
    return round(n.feedforward(x),5)
```

1.1.5 Experiment feedforward process by changing activation function and dimension of inputs. Eventually develop a new widget based on this.

```
#TO DO - Computre the neuron class for the different activation functions
```

```
def relu(z):
    return max(0,z)

def tanh(z):
    return (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z))

def elu(z,alpha):
    return z if z >= 0 else alpha*(e^z -1)
```

➤ Exercice 1.2 - Multi layer perceptron (MLP) ★★☆☆☆

1.2.1 Import all the libraries you need (Numpy).

```
import numpy as np
```

1.2.2 Create a class that could mimic a MLP with 2 inputs, 1 hidden layer of 2 neurons and 1 output layer of 1 neuron.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


class Neuron:

  def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias

  def feedforward(self, inputs):
    total = np.dot(self.weights, inputs) + self.bias
    return sigmoid(total)


class OurNeuralNetwork:

  def __init__(self):
    weights = np.array([0, 1])
    bias = 0
    self.h1 = Neuron(weights, bias)
    self.h2 = Neuron(weights, bias)
    self.o1 = Neuron(weights, bias)

  def feedforward(self, x):
    out_h1 = self.h1.feedforward(x)
    out_h2 = self.h2.feedforward(x)
    out_o1 = self.o1.feedforward(np.array([out_h1, out_h2]))

    return out_o1
```

1.2.3 Test your MLP feedforward process by selecting randomly several pairs of x inputs and observe results.

```
from random import randrange


x1 = randrange(10)
x2 = randrange(10)
x = np.array([x1, x2])
network = OurNeuralNetwork()
network.feedforward(x)
```

1.2.4 Run several time your work and check manually (excel ?) your results

```
#to do
```

➤ Exercice 1.3 - Loss function ★☆☆☆☆

1.3.1 Import all the libraries you need (Numpy).

```
import numpy as np
```

1.3.2 Define a cost function for instance the MSE (manually)

```
def mse_loss(y_true, y_pred):
  return ((y_true - y_pred) ** 2).mean()
```

1.3.3 Experiment your function on two arbitrary observed and predicted vectors

```
y_obs = np.array([1, 1, 0.5, 1])
y_pred = np.array([0, 0, 0, 0])
mse_loss(y_obs, y_pred)
```

1.3.4 Run the calculation on other vectors and explain results

```
y_obs = np.array([0, 0, 0.5, 0])
y_pred = np.array([0, 0, 0, 0])
mse_loss(y_obs, y_pred)

# MSE has decreased which is normal considering observed and predicted vectors are closer than the previous ex

# TO DO :  Create MAE function and another loss function and run similar results
```

➤ Exercice 1.4 - Test activation functions ★☆☆☆☆

1.4.1 Import math and matplotlib libraries for your work

```
from matplotlib import pyplot
from math import exp
```

1.4.2 Define and print relu function. Discuss the shape

```
def rectified(x):
    return max(0.0, x)

inputs = [x for x in range(-10, 10)]
outputs = [rectified(x) for x in inputs]
pyplot.plot(inputs, outputs)
pyplot.show()
```

1.4.3 Define and print sigmoid function. Discuss the shape

```
def sigmoid(x):
    return 1.0 / (1.0 + exp(-x))

inputs = [x for x in range(-10, 10)]
outputs = [sigmoid(x) for x in inputs]
pyplot.plot(inputs, outputs)
pyplot.show()
```

1.4.4 Define and print tanh function. Discuss the shape

```
def tanh(x):
    return (exp(x) - exp(-x)) / (exp(x) + exp(-x))


inputs = [x for x in range(-10, 10)]
outputs = [tanh(x) for x in inputs]
pyplot.plot(inputs, outputs)
pyplot.show()
```

1.4.5 Explore softmax function : define function, test it and print it.

```
from numpy import exp


def softmax(vector):
    e = exp(vector)
    return e / e.sum()


inputs = [1, 3, 2]
result = softmax(inputs)
result


@interact
def inputs(cl1=(0,10,1), cl2=(0,10,1), cl3=(0,10,1)):
    classes = ['cl1', 'cl2', 'cl3']
    inputs = [cl1, cl2, cl3]
    result = softmax(inputs)
    pyplot.bar(classes, result)
    return pyplot.show()
```

➤ Exercice 1.5 - Naive backpropagation ★★☆☆☆

1.5.1 Import libraries we will use during the exercice

```
import pandas as pd
import numpy as np

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
```

1.5.2 Load Iris dataset and split it to get train and test set

```
df = load_iris()
df.data[0:5]


X=df.data
y=df.target
X.shape



y = pd.get_dummies(y).values
y[:3]


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=123)


X_train.shape
```

1.5.3 Initialize main hypothesis and parameters so that we could have:

- a 1 hidden layer MLP
- with inputs and 3 outputs

```
learning_rate = 0.05
iterations = 10000
```

```
N = y_train.size
input_size = 4
hidden_size = 2
output_size = 3
results = pd.DataFrame(columns=["mse", "accuracy"])
np.random.seed(10)
W1 = np.random.normal(scale=0.5, size=(input_size, hidden_size))
W2 = np.random.normal(scale=0.5, size=(hidden_size , output_size))
W2
```

1.5.4 Define Activation and loss function we may want to use (sigmoid, Mse, accuracy)

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def mean_squared_error(y_pred, y_true):
    return ((y_pred - y_true)**2).sum() / (2*y_pred.size)

def accuracy(y_pred, y_true):
    acc = y_pred.argmax(axis=1) == y_true.argmax(axis=1)
    return acc.mean()
```

1.5.5 Implement feedforward and backpropagation process for the MLP

```
for itr in range(iterations):

    # feedforward
    Z1 = np.dot(X_train, W1)
    A1 = sigmoid(Z1)
    Z2 = np.dot(A1, W2)
    A2 = sigmoid(Z2)

    # Error
    mse = mean_squared_error(A2, y_train)
    acc = accuracy(A2, y_train)
    results = results.append({"mse":mse, "accuracy":acc},ignore_index=True )

    # backpropagation
    E1 = A2 - y_train
    dW1 = E1 * A2 * (1 - A2)
    E2 = np.dot(dW1, W2.T)
    dW2 = E2 * A1 * (1 - A1)

    # Gradient weight updates
    W2_update = np.dot(A1.T, dW1) / N
    W1_update = np.dot(X_train.T, dW2) / N
    W2 = W2 - learning_rate * W2_update
    W1 = W1 - learning_rate * W1_update
```

1.5.6 Provide plots that represent MSE and Accuracy for train set

```
results.mse.plot(title="Mean Squared Error")
```

```
results.accuracy.plot(title="Accuracy")
```

1.5.6 Use model weights to infere on test dataset and provide final accuracy

```
Z1 = np.dot(X_test, W1)
A1 = sigmoid(Z1)
Z2 = np.dot(A1, W2)
A2 = sigmoid(Z2)

acc = accuracy(A2, y_test)
print(acc)
```

```
#TO DO improve accuracy value on test set by changing initial parameters
```

➤ Exercice 1.6 - Introductive churn Sklearn case study ★★★☆☆

1.6.1 Import the data we will need for th exercice

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import accuracy_score
```

1.6.2 Load the data set and define a dataframe. Explore this dataset

```
df=pd.read_csv('/content/drive/MyDrive/Tech/202209_intro_dl/data_course_1_intro_DL/churn_hr_v3.csv', sep = ";'
df.head()
```

1.6.3 Get more info about column properties: type, statistics, missing values, etc.

```
df.info()
```

```
df.describe()
```

1.6.4 Explore dataset: run some univariate or multivariate analysis to understand variable interactions

```
features=['number_project','time_spend_company','Work_accident','left', 'promotion_last_5years', 'departments'
```

```
fig=plt.subplots(figsize=(20,15))
for i, j in enumerate(features):
     plt.subplot(3, 3, i+1)
     plt.subplots_adjust(hspace = 0.3)
     sns.countplot(x=j,data = df)
     plt.title("No. of employee")
```

```
sns.heatmap(df.corr(), annot=True)
```

1.6.5 Propse a basic label encoding approach to get numerical values instead of categorial ones

```
df['salary'].head()
```

```
le = preprocessing.LabelEncoder()
df['salary']=le.fit_transform(df['salary'])
df['salary'].head()
```

```
df['departments']=le.fit_transform(df['departments'])
df['departments'].head()
```

1.6.6 Split your data into a train / test part using sklearn

```
X=df[['satisfaction_level', 'last_evaluation', 'number_project', 'average_montly_hours', 'time_spend_company',
y=df['left']
print(X.shape)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, test_size=0.3, random_state=123)
```

```
print(X_train.shape)
X_train.head()
```

```
y_train.describe()
```

1.6.7 Let us develop and train a first MLP classifier using sklearn. The ANN should use 2 hidden layers with 8 and 4 units.

```
import warnings
warnings.filterwarnings("ignore")

mlp_clf = MLPClassifier(hidden_layer_sizes=(8,4),
                    random_state=123,
                    max_iter=500,
                    #verbose=True
                    )
mlp_clf.fit(X_train,y_train)
```

1.6.8 Print the loss function for the train set all along epoch iteration and discuss results and compute accuracy result on test set.

```
plt.title("Evolution of error")
plt.xlabel("Iterations (epochs)")
plt.ylabel("Error")
plt.plot(mlp_clf.loss_curve_)
plt.show()

ypred = mlp_clf.predict(X_test)
clf_acc = accuracy_score(y_test,ypred)
print(clf_acc)

plot_confusion_matrix(mlp_clf, X_test, y_test)
plt.show()
```

1.6.9 Retrain a new MLP classifier by upgrading parameters such as the solver, activation function, the learning rate, regularization techniques, etc.

```
mlp_clf_tanh = MLPClassifier(hidden_layer_sizes=(8,4),
                    random_state=123,
                    activation = 'tanh',
                    max_iter=500,
                    )
mlp_clf_tanh.fit(X_train,y_train)

mlp_clf_sgd = MLPClassifier(hidden_layer_sizes=(8,4),
                    random_state=123,
                    learning_rate = 'adaptive',
                    max_iter=500,
                    batch_size = 100,
                    solver = 'sgd')
mlp_clf_sgd.fit(X_train,y_train)

mlp_clf_earlstp = MLPClassifier(hidden_layer_sizes=(8,4),
                    random_state=123,
                    max_iter=500,
                    early_stopping = True)
mlp_clf_earlstp.fit(X_train,y_train)

mlp_clf_l2 = MLPClassifier(hidden_layer_sizes=(8,4),
                    random_state=123,
                    max_iter=500,
                   alpha = 0.001)
mlp_clf_l2.fit(X_train,y_train)

mlp_clf_optima = MLPClassifier(hidden_layer_sizes=(8,4),
                    random_state=123,
                    activation = 'tanh',
                    batch_size  = 100,
                    shuffle = True,
                    max_iter=1000,
                    alpha = 0.05,)
mlp_clf_optima.fit(X_train,y_train)
```

1.6.10 Print losses and confusion matrix on best model to conclude.

```
fig=plt.subplots(figsize=(10,8))
plt.title("Evolution of error")
plt.xlabel("Iterations (epochs)")
plt.ylabel("Error")
plt.plot(mlp_clf.loss_curve_, label ='baseline')
plt.plot(mlp_clf_tanh.loss_curve_ , label ='tanh')
plt.plot(mlp_clf_sgd.loss_curve_ ,label = 'sgd')
plt.plot(mlp_clf_earlstp.loss_curve_ , label = 'early stopping')
plt.plot(mlp_clf_l2.loss_curve_, label = 'L2 regul')
plt.plot(mlp_clf_optima.loss_curve_, label = 'Optimal')

plt.legend()
plt.show()
```

```
ypred = mlp_clf_optima.predict(X_test)
clf_optima_acc = accuracy_score(y_test,ypred)
print(clf_optima_acc)
```

```
plot_confusion_matrix(mlp_clf_optima, X_test, y_test)
plt.show()
```

➤ Exercice 1.7 - Introductive diabetes Keras case study ★★★☆☆

1.7.1 Import set of libraries you plan to use

```
import pandas as pd

from sklearn import preprocessing
from sklearn.model_selection import train_test_split

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

1.7.2 Load your dataset and analyze it

```
df = pd.read_csv('/content/drive/MyDrive/Tech/202209_intro_dl/data_course_1_intro_DL/diabetes_v2.csv', sep = '
df.head()
```

```
df.describe()
```

1.7.3 Split your dataset into train and test set using sklearn process or any other approach.

```
X=df[['nb_pregn','plasma_gluc','blood_press','triceps_thick','serun_insulin','bmi','diabetes_ped','age']]
y=df['class']
print(X.shape)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, test_size=0.2, random_state=123)
```

1.7.4 Let us try to set a first MLP with:

- a first hidden layer that expects 8 inputs (columns), has 6 nodes and uses relu activation function
- a second hidden layer that has 4 nodes and uses relu activation function
- an output layer with one node and uses the sigmoid function

```
k_mlp_clf = Sequential()
k_mlp_clf .add(Dense(6, input_shape=(8,), activation='relu'))
k_mlp_clf .add(Dense(4, activation='relu'))
k_mlp_clf .add(Dense(1, activation='sigmoid'))

k_mlp_clf.summary()
```

1.7.5 Define a loss, an optimize and metrics you may want to study and fit your model

```
k_mlp_clf.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
results = k_mlp_clf.fit(X_train, y_train,
                        epochs=50,
                        validation_data = (X_test, y_test),
                        verbose = 0)
```

1.7.6 Print metrics and plot training loss curve per epoch

```
plt.plot(results.epoch, results.history["loss"], 'g', label='Training loss')
plt.title('Training loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
loss, accuracy = k_mlp_clf.evaluate(X_test, y_test)
print('Loss:' + str(loss))
print('Accuracy:' + str(accuracy))
```

1.7.8 Look for other option and optimize you MLP

```
#Strategy 1
k_mlp_clf_strt1 = Sequential()
k_mlp_clf_strt1 .add(Dense(6, input_shape=(8,), activation='relu'))
k_mlp_clf_strt1.add(Dropout(0.3))
k_mlp_clf_strt1 .add(Dense(4, activation='relu'))
k_mlp_clf_strt1.add(Dropout(0.2))
k_mlp_clf_strt1 .add(Dense(1, activation='sigmoid'))
```

```
k_mlp_clf_strt1.summary()
```

```
k_mlp_clf_strt1.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
results_strt1 = k_mlp_clf_strt1.fit(X_train, y_train,
                                    epochs=200,
                                    verbose = 1,
                                    validation_data = (X_test, y_test)
                                    )
```

```
plt.plot(results_strt1.epoch, results_strt1.history["loss"], 'b', label='Training loss')
plt.plot(results_strt1.epoch, results_strt1.history["val_loss"], 'r', label='Validation loss')
plt.title('Train & validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

➤ Exercice 1.8 - Regularization effects with Tensorflow ★★★★☆

1.8.1 Import all the libraries you will need for your work

```
import matplotlib.pyplot as plt

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

import tensorflow as tf

from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Dense
```

1.8.2 Load Iris dataset, prepare and split your data

```
df = load_iris()
X = df.data
y = df.target
X[0:10]
```

```
y = to_categorical(y)
ss = StandardScaler()
X = ss.fit_transform(X)
X[0:10]
```

```
X_train, X_test, y_train, y_test = train_test_split(X,y)
```

1.8.3 Propose a first model baseline with the architecture that you want. Just define softmax as output activation.

```
dl_baseline = Sequential([
    Dense(512, activation='tanh', input_shape = X_train[0].shape),
    Dense(256, activation='tanh'),
    Dense(128, activation='tanh'),
    Dense(64, activation='tanh'),
    Dense(32, activation='relu'),
    Dense(3, activation='softmax')
    ],
    )

print(dl_baseline.summary())
```

1.8.4 Fix the number of epochs and the batch size you would aim at working one and train a first MLP using stochastic gradient optimizer and the adapted loss function.

```
n_epochs = 200
n_batch_size = 150
```

```
dl_baseline.compile(optimizer='sgd',loss='categorical_crossentropy', metrics=['acc', 'mse'])
hist = dl_baseline.fit(X_train, y_train, epochs=n_epochs, batch_size=n_batch_size, validation_data=(X_test,y_t
```

1.8.5 Print loss, accuracy and mse using a reusable function

```
def result_print (model,X_test, y_test):
    loss, acc, mse = model.evaluate(X_test, y_test)
    print(f"For {model.name}: \nLoss is {loss},\nAccuracy is {acc*100},\nMSE is {mse}")

result_print (dl_baseline,X_test, y_test)
```

1.8.6 Define Loss values for train and validation set. Use a function to be able to reuse it later.

```
def plot_loss(historical):
    fig=plt.subplots(figsize=(10,8))
    plt.plot(historical.history['loss'], label = 'loss')
    plt.plot(historical.history['val_loss'], label='val loss')
    plt.title("Loss vs Val_Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()

plot_loss(hist)
```

1.8.7 Run the same task for the accuracy values

```
def plot_acc(historical):
    fig=plt.subplots(figsize=(10,8))
    plt.plot(hist.history['acc'], label = 'acc')
    plt.plot(hist.history['val_acc'], label='val acc')
    plt.title("acc vs Val_acc")
    plt.xlabel("Epochs")
    plt.ylabel("acc")
```

```
    plt.legend()
    plt.show()


plot_acc(hist)
```

1.8.8 Create a new MLP that uses L1 regularization at least on one layer and print error measures.

```
dl_l1 = Sequential([
    Dense(512, activation='tanh', input_shape = X_train[0].shape, kernel_regularizer='l1'),
    Dense(256, activation='tanh', kernel_regularizer='l1'),
    Dense(128, activation='tanh', kernel_regularizer='l1'),
    Dense(64, activation='tanh'),
    Dense(32, activation='relu'),
    Dense(3, activation='softmax')
])

dl_l1 .compile(optimizer='sgd',loss='categorical_crossentropy', metrics=['acc', 'mse'])
hist2 = dl_l1 .fit(X_train, y_train,  epochs=n_epochs, batch_size=n_batch_size, validation_data=(X_test,y_test

result_print (dl_l1, X_test, y_test)
```

1.8.9 Create a new MLP that uses L2 regularization at least on one layer and print error measures.

```
dl_l2 = Sequential([
    Dense(512, activation='tanh', input_shape = X_train[0].shape, kernel_regularizer='l2'),
    Dense(512//2, activation='tanh', kernel_regularizer='l2'),
    Dense(512//4, activation='tanh', kernel_regularizer='l2'),
    Dense(512//8, activation='tanh', kernel_regularizer='l2'),
    Dense(32, activation='relu', kernel_regularizer='l2'),
    Dense(3, activation='softmax')
])

dl_l2 .compile(optimizer='sgd',loss='categorical_crossentropy', metrics=['acc', 'mse'])
hist3 = dl_l2 .fit(X_train, y_train,  epochs=n_epochs, batch_size=n_batch_size, validation_data=(X_test,y_test

result_print (dl_l2, X_test, y_test)
```

1.8.10 Create a new MLP that uses Dropout regularization at least on one layer and print error measures.

```
dl_dropout = Sequential([
    Dense(512, activation='tanh', input_shape = X_train[0].shape),
    tf.keras.layers.Dropout(0.5),
    Dense(512//2, activation='tanh'),
    tf.keras.layers.Dropout(0.5),
    Dense(512//4, activation='tanh'),
    tf.keras.layers.Dropout(0.5),
    Dense(512//8, activation='tanh'),
    tf.keras.layers.Dropout(0.3),
    Dense(32, activation='relu'),
    Dense(3, activation='softmax')
])

dl_dropout.compile(optimizer='sgd',loss='categorical_crossentropy', metrics=['acc', 'mse'])
hist4 = dl_dropout.fit(X_train, y_train,  epochs=n_epochs, batch_size=n_batch_size, validation_data=(X_test,y_

result_print (dl_dropout, X_test, y_test)
```

1.8.11 Propose a chart that allows to compare all the accuracies of validation set for the different MLPs developed previously.

```
fig=plt.subplots(figsize=(10,8))
plt.plot(hist.history['val_acc'], label='val acc baseline')
plt.plot(hist2.history['val_acc'], label='val acc L1')
plt.plot(hist3.history['val_acc'], label='val acc L2')
plt.plot(hist4.history['val_acc'], label='val acc dropout')
plt.title("Val_acc comparison")
plt.xlabel("Epochs")
```

```
plt.ylabel("acc")
plt.legend()
plt.show()
```

➤ Exercice 1.9 - Regression with PyTorch ★★★★★

1.9.1 Import all the libraries we will need for the exercice

```
import numpy as np
import pandas as pd
import seaborn as sns

from tqdm.notebook import tqdm
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
```

1.9.2 Load the wine dataset and check its content

```
df = pd.read_csv("/content/drive/MyDrive/Tech/202209_intro_dl/data_course_1_intro_DL/winequality_v2.csv")
df.head()
```

```
df.describe()
```

```
df.shape
```

1.9.3 Split the dataset into a train, validate and test set according the proportion of your choice

```
X = df.iloc[:, 0:-1]
y = df.iloc[:, -1]
```

```
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=123)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.2, stratify=y_trainval,
```

1.9.4 Transform your dataset using a min max scaler (or any other normalization you may juge relevant). Define also your datasets as array for futur calculations.

```
scaler = MinMaxScaler()

X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

X_train, y_train = np.array(X_train), np.array(y_train)
X_val, y_val = np.array(X_val), np.array(y_val)
X_test, y_test = np.array(X_test), np.array(y_test)
```

```
y_train, y_test, y_val = y_train.astype(float), y_test.astype(float), y_val.astype(float)
```

1.9.5 Initialize your datasets as torch objects

```
class RegressionDataset(Dataset):

    def __init__(self, X_data, y_data):
        self.X_data = X_data
        self.y_data = y_data

    def __getitem__(self, index):
```

```
        return self.X_data[index], self.y_data[index]

    def __len__ (self):
        return len(self.X_data)

train_dataset = RegressionDataset(torch.from_numpy(X_train).float(), torch.from_numpy(y_train).float())
val_dataset = RegressionDataset(torch.from_numpy(X_val).float(), torch.from_numpy(y_val).float())
test_dataset = RegressionDataset(torch.from_numpy(X_test).float(), torch.from_numpy(y_test).float())
```

1.9.6 Choose epochs, batch size, learning rate for your MLP

```
EPOCHS = 120
BATCH_SIZE = 200
LEARNING_RATE = 0.001
NUM_FEATURES = len(X.columns)
```

1.9.7 Load your data using DataLoader

```
train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=1)
test_loader = DataLoader(dataset=test_dataset, batch_size=1)
```

1.9.8 Define MLP regression archiecture with at least 3 hidden layers and relu activation function.

```
class MultipleRegression(nn.Module):

    def __init__(self, num_features):
        super(MultipleRegression, self).__init__()
        self.layer_1 = nn.Linear(num_features, 8)
        self.layer_2 = nn.Linear(8, 16)
        self.layer_3 = nn.Linear(16, 8)
        self.layer_out = nn.Linear(8, 1)
        self.relu = nn.ReLU()

    def forward(self, inputs):
            x = self.relu(self.layer_1(inputs))
            x = self.relu(self.layer_2(x))
            x = self.relu(self.layer_3(x))
            x = self.layer_out(x)
            return (x)

    def predict(self, test_inputs):
            x = self.relu(self.layer_1(test_inputs))
            x = self.relu(self.layer_2(x))
            x = self.relu(self.layer_3(x))
            x = self.layer_out(x)
            return (x)
```

1.9.9 Force your model to work on colab GPU (if possible) using torch.device

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

1.9.10 Compile last parameters of your MLP: device, loss function, optimizer, etc.

```
model = MultipleRegression(NUM_FEATURES)
model.to(device)
print(model)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

1.9.11 Define a dictionnary to store your loss results through epochs

```
loss_stats = {
    'train': [],
    "val": []
}
```

1.9.12 Run training and validation of your model and print error mesures (from val and train set) trough epochs.

```
for e in tqdm(range(1, EPOCHS+1)):

    # TRAINING
    train_epoch_loss = 0
    model.train()
    for X_train_batch, y_train_batch in train_loader:
        X_train_batch, y_train_batch = X_train_batch.to(device), y_train_batch.to(device)
        optimizer.zero_grad()

        y_train_pred = model(X_train_batch)

        train_loss = criterion(y_train_pred, y_train_batch.unsqueeze(1))
        train_loss.backward()
        optimizer.step()

        train_epoch_loss += train_loss.item()


    # VALIDATION
    with torch.no_grad():
        val_epoch_loss = 0
        model.eval()
        for X_val_batch, y_val_batch in val_loader:
            X_val_batch, y_val_batch = X_val_batch.to(device), y_val_batch.to(device)
            y_val_pred = model(X_val_batch)

            val_loss = criterion(y_val_pred, y_val_batch.unsqueeze(1))
            val_epoch_loss += val_loss.item()

    loss_stats['train'].append(train_epoch_loss/len(train_loader))
    loss_stats['val'].append(val_epoch_loss/len(val_loader))

    print(f'Epoch {e+0:03}: | Train Loss: {train_epoch_loss/len(train_loader):.5f} | Val Loss: {val_epoch_loss
```

1.9.13 Plot the loss curve per epoch for train and validation set

```
train_val_loss_df = pd.DataFrame.from_dict(loss_stats).reset_index().melt(id_vars=['index']).rename(columns={'
plt.figure(figsize=(15,8))
sns.lineplot(data=train_val_loss_df, x = "epochs", y="value", hue="variable").set_title('Train-Val Loss/Epoch'
```

1.9.14 Let us infere on test set and provide MSE and R2 metrics

```
y_pred_list = []
with torch.no_grad():
    model.eval()
    for X_batch, _ in test_loader:
        X_batch = X_batch.to(device)
        y_test_pred = model(X_batch)
        y_pred_list.append(y_test_pred.cpu().numpy())
y_pred_list = [a.squeeze().tolist() for a in y_pred_list]


mse = mean_squared_error(y_test, y_pred_list)
r_square = r2_score(y_test, y_pred_list)
print("Mean Squared Error :",mse)
print("R^2 :",r_square)
```

1.9.15 Provide also a qqplot to analyze results

```
plt.scatter(y_test, y_pred_list, c = (np.subtract(y_test, y_pred_list)**2),cmap='viridis')
plt.colorbar()
plt.xlim(2, 9)
plt.ylim(2, 9)
```

1.9.16 Provide any further analysis to explore model results

#