TD/TP 1 Deep Learning : MLP

Predicting the cars' consumption with a MLP

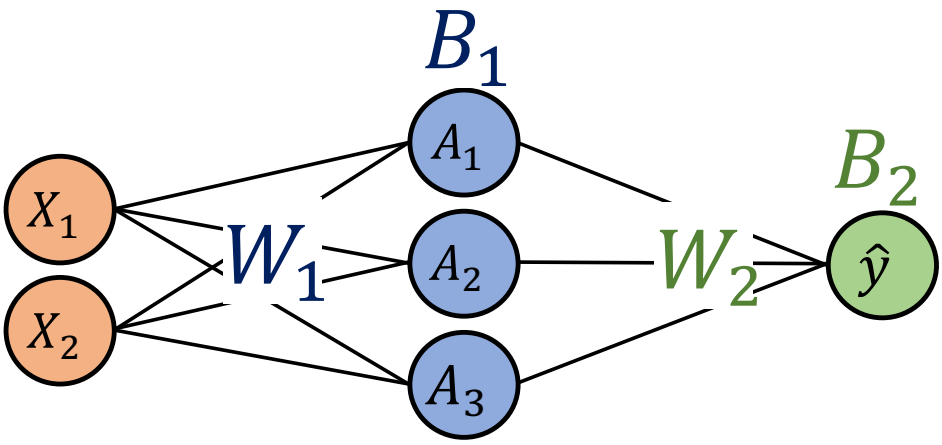
Goal

In this practical work, the goal will be first to do some calculus on multi-layers perceptron. In a second time, we will see how to build a MLP model for a regression problem in order to predict the consumption of several types of cars.

---

1 Some calculus

Let a multi-layers perceptron be the following :



At $t = 0$, the network is initialized with the following parameters :

$$W_1 = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} = \begin{bmatrix} 0.1 & 0.5 & 0.3 \\ -0.2 & 0 & 0.5 \end{bmatrix}$$

$$B_1 = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -0.1 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ -0.1 \\ 0.2 \end{bmatrix}$$

$$B_2 = 0.3$$

For each neuron, the activation function is the identity.

1- Inputs $X_1 = 0.2$ et $X_2 = 0.5$ are presented to the network. Compute the prediction (output) $\hat{y}$.

2- The cost function used in this network is the `Mean Square Error (MSE)` and the learning rate is 0.1. For the pair of inputs X_1 and X_2, the output should be 0.992. Apply the backpropagation in order to update all parameters of the network (W et B).

2 Consumption prediction

We will address a problem of predicting continuous values. It is therefor a regression problem which is different from a classification problem.

2.1 Auto MPG dataset

We will use the AutoMPG dataset for predicting the consumption of cars from the 70s and 80s. The dataset includes 398 car samples with several attributes like weight, number of cylinders, power, etc. For each sample, a consumption in MPG (miles per gallon) is associated. The dataset and its description can be found here : https://archive.ics.uci.edu/ml/datasets/auto+mpg

2.2 Libraries

Start by importing useful libraries. We would for instance need the visualization library seaborn that can be installed like the following :

```
$ !pip install -q seaborn
```

Import :

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

2.3 Data preparation

2.3.1 Get the data and visualization

Load data with pandas :

```
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data'
column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',
'Acceleration', 'Model Year', 'Origin']
raw_dataset = pd.read_csv(url, names=column_names, na_values='?',
comment='\t', sep=' ', skipinitialspace=True)
```

Display some lignes with for instance the methods `head()` or `tail()` :

```
dataset = raw_dataset.copy()
dataset.head()
```

You should see some lignes of the dataset.

2.3.2 Data cleaning

The dataset contains some 'nan' values, especially for the Horsepower attribute. For simplicity, we will remove samples where there are missing values (entire lines) :

```
dataset = dataset.dropna()
dataset.shape
```

The dataset now contains 392 instances.

The Origin attribute corresponds to a categorical value (country of origin) and not a numerical value. To facilitate the training procedure, we will transform this attribute as a one-hot vector. We can for instance use the method `get_dummies()` from pandas :

```
dataset['Origin'] = dataset['Origin'].map({1: 'USA', 2: 'Europe', 3: 'Japan'})
dataset = pd.get_dummies(dataset, columns=['Origin'], prefix='', prefix_sep='')
dataset.tail()
```

2.3.3 Training set and Test set

We can split the dataset into a training set and a test set :

```
train_dataset = dataset.sample(frac=0.8, random_state=0)
test_dataset = dataset.drop(train_dataset.index)
print(train_dataset.shape, test_dataset.shape)
```

2.3.4 Looking at the data

An important step before training a model is to have a deep look at the available data. For instance, it can be interesting to analyze the joint distribution of pairs of attributes. The advantage of using the seaborn library is that it becomes very simple to do that with the `pairplot()` function :

```
sns.pairplot(train_dataset[['MPG', 'Cylinders', 'Displacement', 'Weight']], diag_kind='kde')
```

It can also be interesting to get some general statistics on attributes. We can for instance simply observe that each feature (attribute) has different scale and range :

```
train_dataset.describe().transpose()
```

2.3.5 Splitting inputs and outputs

We then need to split the output to be predicted from the input attributes :

```
train_features = train_dataset.copy()
test_features = test_dataset.copy()

train_labels = train_features.pop('MPG')
test_labels = test_features.pop('MPG')
```

2.3.6 Normalization

As we have seen while analyzing statistics, each attribute has a different scale and range. It is then very important to normalize the data to make each attribute has the same impact. This is a very good practice to remind, it facilitates the training phase and makes it more stable.

In Keras, a specific layer allows to do this normalization : `tf.keras.layers.Normalization`. It is a simple way to add this normalization step within a network as it is done in a similar way as adding a layer.

We then need to create a layer and then apply on the data with the `adapt()` method :

```
normalizer = tf.keras.layers.Normalization(axis=-1)
normalizer.adapt(np.array(train_features, dtype='float32'))
```

For instance, in a networlk, when the "adapt" layer is called on a sample, it is transformed to make each attribute independently normalized :

```
first = np.array(train_features[:1], dtype='float32')
with np.printoptions(precision=2, suppress=True):
  print('First example:', first)
  print('Normalized:', normalizer(first).numpy())
```

2.4 Linear Regression

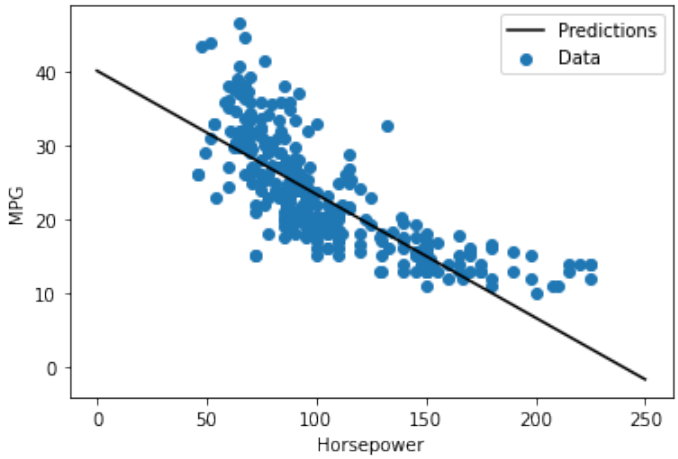
Before building a multi-layers perceptron, we will first observe how a simple linear regression is able to achieve the task.

2.4.1 Linear regression with a single attribute

1- Use what we already done in the Colab exercises during lectures for building a model with Keras in order to do a linear regression with the Horsepower attribute only. **Do not forget to add the normalization layer**, between the input layer and the Dense layer.

2- COnfigure your model with `Adam` optimization and a learning rate of 0.1. Train your model from the training set on 100 epochs. Choose a loss adapted to the problem (for example `Mean Absolute Error` or `Mean Square Error`) and take 20% of the training set for the validation. Visualize the loss curves for the training set and the validation set.

3- Evaluate your model on the test set and save the performance in a dictionary for doing comparisons later. Visualize the predictions with respect to the inputs :



2.4.2 Linear regression with multiple attributes

1- Use the same idea for this time building a linear regression model taking as input all available attributes.

2- Train and evaluate your model on the test set for saving its performance in the dictionary previously created.

2.5 Regression with a MLP

We can now build a deep neural network by adding hidden layers entre les entrées et les sorties in order to add some non-linearity in the prediction. As previously, we will analyze performances of such a model when a single attribute is considered or when all attributes are used.

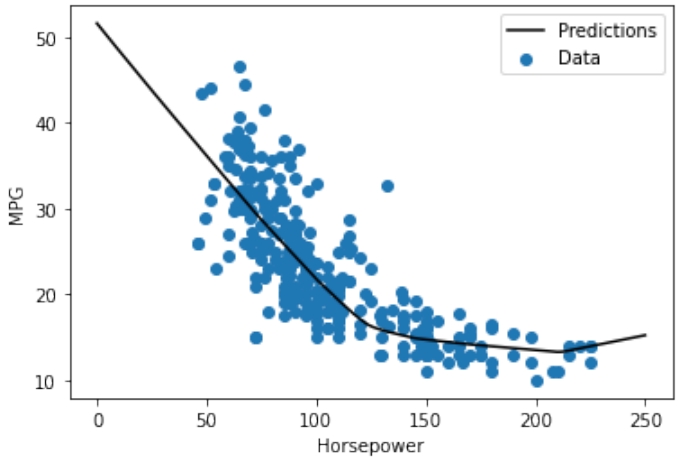
1- For both cases, create a deep model with two hidden layers of 64 neurons and `ReLU` as activation function. For the optimization, choose `Adam` with a learning rate of 0.001. In order to avoid doing same steps twice, it is advised to use a python function for creating the model. This function will take as parameter the input size and the normalization layer adapted to data according to cases.

2- Train the first model during 100 epochs from a single attribute and plot the learning curves.

3- Evaluate the model on the test set and then save the performance in the dictionary. As a single attribute is used, it is quite easy to visualize the prediction with respect to the input for comparing with the linear regression :

4- Train this time a similar model on all the available attributes during 100 epochs as well.

5- Evaluate the latter model and save the performance in the dictionary.



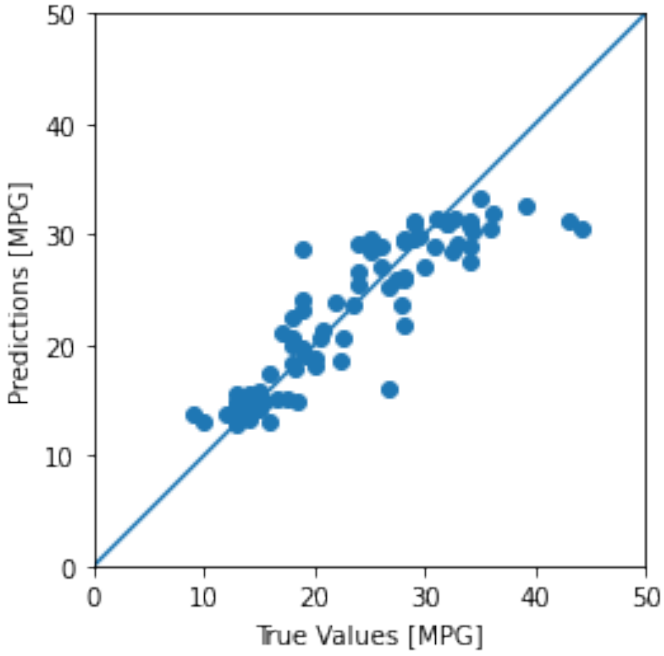
2.6 Performance

As all models have been trained, we can now compare their performance on the test set. The MLP model using all attributes seems to be the best performing.

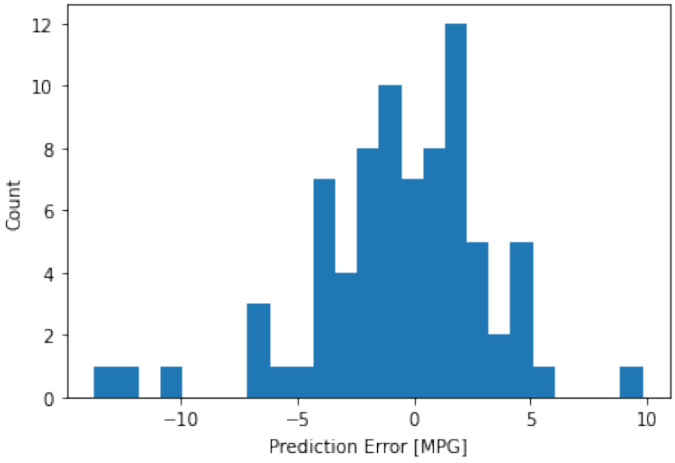
1- Show your dictionary for comparing the performance of the four models. For instance, here is a results of one training of four models :

	Mean absolute error [MPG]
horsepower_model	3.645887
linear_model	3.150480
dnn_horsepower_model	2.915265
dnn_model	2.799380

2- Use the best model for doing predictions and compare them to true values :



3- Compute the prediction errors and plot their distribution using a historgram (`plt.hist()`) :

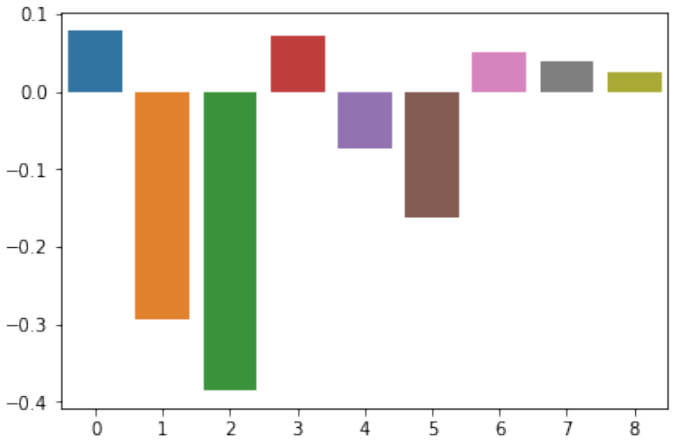


Bonus

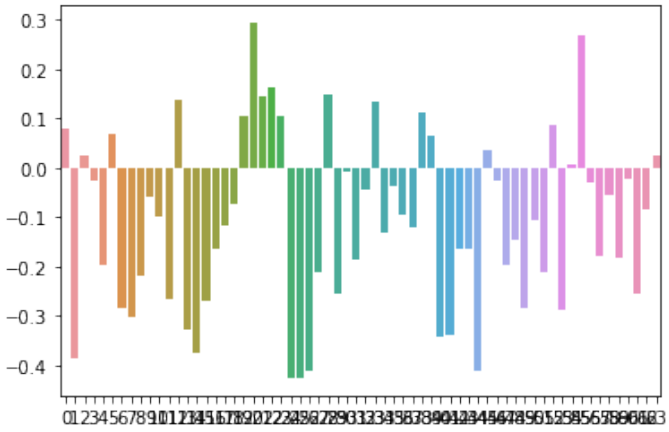
Once a model has been trained, it is possible to analyze the trained parameters in order to observe which information goes through the network and how. Each layer contains a list of parameters, usually of size 2 corresponding to weights w and bias b.

1- Get the parameters w and b from the first layer of the model and check matrices' sizes.

2- Visualize, for instance using a histogram with seaborn (`sns.barplot`), values of w linking all attributes (inputs) to the first neuron of the hidden layer.



2- Visualize, for instance using a histogram with seaborn (`sns.barplot`), values of w linking the first attribute (input) to all neurons of the hidden layer.



What is the impact of negative values of w?