The Perceptron

Objectives of the practical work

The objective is to get hands on experience on the fundamental elements of neural networks:

- perceptron architecture (linear regression)
- loss function
- empirical loss
- gradient descent

For this we will implement from scratch the data-structure and algorithms to train a perceptron. Note that slides related to the perceptron and neural networks in general are available on moodle.

Dataset

The objective of the regression is the prediction of the hydrodynamic performance of sailing yachts from dimensions and velocity. The **inputs** are linked to dimension and hydrodynamics characteristics:

1. Longitudinal position of the center of buoyancy (*flottabilité*), adimensional.
2. Prismatic coefficient, adimensional.
3. Length-displacement ratio, adimensional.
4. Beam -draught ratio ((*tiran d'eau*), adimensional.
5. Length-beam ratio, adimensional.
6. Froude number, adimensional

Target value/predicted value (Output) = Residuary resistance per unit weight of displacement, adimensional

In [3]:
```
# Import some useful libraries and functions

import numpy as np
import pandas
import matplotlib.pyplot as plt
import copy
import time

def print_stats(dataset):
    """Print statistics of a dataset"""
    print(pandas.DataFrame(dataset).describe())
```

In [4]:
```
# Download the data set and place in the current folder (works on linux only)
filename = 'yacht_hydrodynamics.data'

import os.path
import requests

if not os.path.exists(filename):
    print("Downloading dataset...")
    r = requests.get('https://arbimo.github.io/tp-supervised-learning/tp2/' + filename)
    open(filename , 'wb').write(r.content)
```

```
print('Dataset available')
```

```
Dataset available
```

Explore the dataset

- how many examples are there in the dataset?
- how many features for each example?
- what is the ground truth of the 10th example

In [5]:
```
# loads the dataset and slip between inputs (X) and ground truth (Y)
dataset = np.genfromtxt("yacht_hydrodynamics.data", delimiter='')
X = dataset[:, :-1] # examples features
Y = dataset[:, -1]  # ground truth

# Print the first 5 examples
for i in range(0,5):
    print(f"f({X[i]}) = {Y[i]}")
```

```
f([-5.    0.6   4.78  4.24  3.15  0.35]) = 8.62
f([-5.    0.565  4.77   3.99   3.15   0.15 ]) = 0.18
f([-2.3   0.565  4.78   5.35   2.76   0.15 ]) = 0.29
f([-5.    0.6    4.78   4.24   3.15   0.325]) = 6.2
f([0.    0.53  4.78  3.75  3.15  0.175]) = 0.59
```

The following command adds a column to the inputs.

- what is in the value added this column?
- why are we doing this?

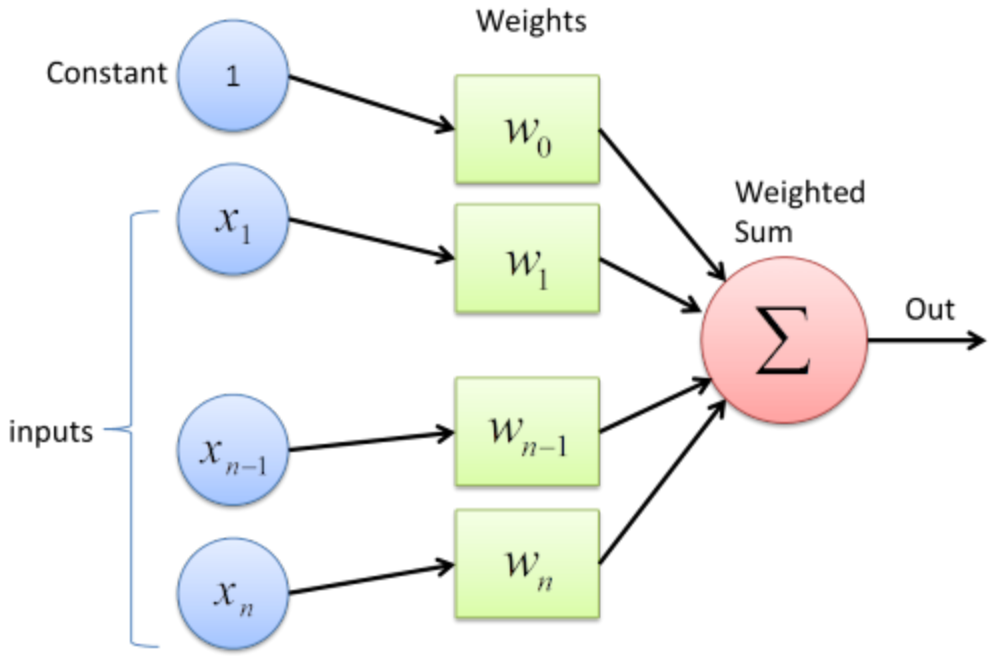
Addition de la constante pour le perceptron

In [6]:
```
X = np.insert(X, 0, np.ones((len(X))), axis= 1)
print_stats(X)
nb_features = len(X[0])
nb_data = len(X)
```

```
           0           1           2           3           4           5  \
count  308.0  308.000000  308.000000  308.000000  308.000000  308.000000
mean     1.0   -2.381818    0.564136    4.788636    3.936818    3.206818
std      0.0    1.513219    0.023290    0.253057    0.548193    0.247998
min      1.0   -5.000000    0.530000    4.340000    2.810000    2.730000
25%      1.0   -2.400000    0.546000    4.770000    3.750000    3.150000
50%      1.0   -2.300000    0.565000    4.780000    3.955000    3.150000
75%      1.0   -2.300000    0.574000    5.100000    4.170000    3.510000
max      1.0    0.000000    0.600000    5.140000    5.350000    3.640000

                6
count  308.000000
mean     0.287500
std      0.100942
min      0.125000
25%      0.200000
50%      0.287500
75%      0.375000
max      0.450000
```

Creating the perceptron



We now want to define a perceptron, that is, a function of the form:

$$h_w(x) = w_0 + w_1 \times x_1 + \cdots + w_n \times x_n$$

- Complete the code snippet below to:
  - create the vector of weight `w`
  - implement the `h` function that evaluate an example based on the vector of weights
  - check if this works on a few examples

In [7]:
```
w = np.ones(nb_features)

def h(w, x):
    return np.dot(x,w)

print(f"ground truth = {Y[0]} \nh_w = {h(w,X[0]):.2f} ")
# print the ground truth and the evaluation of ground truth on the first example
```

```
ground truth = 8.62
h_w = 9.12
```

Loss function

Complete the definiton of the loss function below such that, for a **single** example `x` with ground truth `y`, it returns the L_2 loss of h_w on `x`.

In [8]:
```
def loss(w, x, y):
    return (h(w,x)-y)**2

print(f"The L_2 loss of the first example is {loss(w,X[0],Y[0]):.2f}")
```

```
The L_2 loss of the first example is 0.25
```

Empirical loss

Complete the function below to compute the empirical loss of h_w on a **set** of examples X with associated ground truths Y.

In [9]:
```
def emp_loss(w, X, Y):
    empLoss =0
    for i in range (len(X)):
        empLoss+= loss(w, X[i],Y[i])
    return (empLoss/len(X))

print(f"The empirical loss of the set of examples is  {emp_loss(w,X,Y):.2f}")
```

```
The empirical loss of the set of examples is  229.63
```

Gradient update

A gradient update is of the form: $w \leftarrow w + dw$

- Complete the function below so that it computes the dw term (the 'update') based on a set of examples `(X, Y)` the step (`alpha`)

If you are not sure about the gradient computation, check out the perceptron slides on Moodle (in particular, slide 26). Make sure this computation is clear to you!

In [10]:
```
def compute_update(w, X, Y, alpha):
    nb_features = len(X[0])
    nb_data = len(X)

    w_copy= copy.deepcopy(w)
    for i in range(nb_features):
        dw =0
        for j in range(nb_data):
            dw+= (Y[j]-h(w_copy, X[j]))*X[j][i]
        w[i]+= alpha*dw
```

Gradient descent

Now implement the gradient descent algorithm that will:

- repeatedly apply an update the weights
- stops when a max number of iterations is reached (do not consider early stopping for now)
- returns the final vector of weights

In [11]:
```
def descent(w_init, X, Y, alpha, max_iter):
    w_final =copy.deepcopy(w_init)
    for i in range (max_iter):
        compute_update(w_final, X, Y, alpha)
    return w_final
```

Exploitation

You gradient descent is now complete and you can exploit it to train your perceptron.

- Train your perceptron to get a model.
- Visualize the evolution of the loss on the training set. Has it converged?
- Try training for several choices of `alpha` and `max_iter` . What seem like a reasonable choice?
- What is the loss associated with the final model?
- Is the final model the optimal one for a perceptron?

In [12]:
```
w_init = np.ones((len(X[0])))
w_final = descent(w_init, X, Y, alpha = 10e-8, max_iter=100)
```

Impact of `alpha` and `max_iter`

In [29]:
```
Alphas=np.logspace(-8,-4,20,base=10)
LossAlpha=[]

for alpha in Alphas:
    w_final = descent(w_init, X, Y, alpha = alpha, max_iter=500)
    LossAlpha.append(emp_loss(w_final, X, Y))
```

In [35]:
```
Iters=np.logspace(1,4,20).astype(int)
LossIter=[]
executionTime=[]

for iter in Iters:
    st=time.time()
    w_final = descent(w_init, X, Y, alpha = 10e-5, max_iter=iter)
    et=time.time()-st
    LossIter.append(emp_loss(w_final, X, Y))
    executionTime.append(et)
```

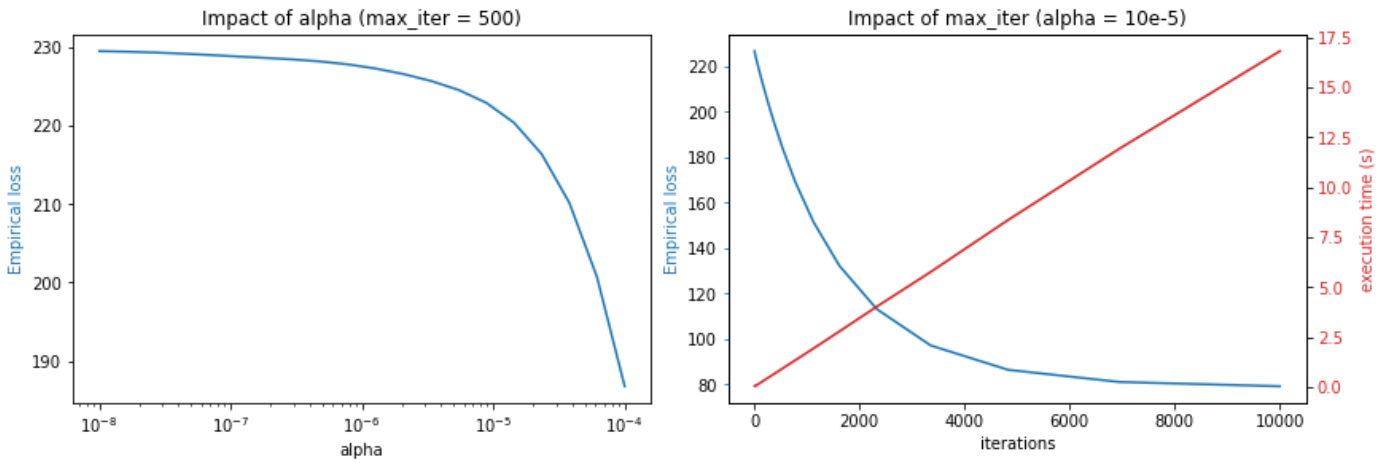
In [45]:
```
fig, axs = plt.subplots(1, 2, constrained_layout=True)

color = 'tab:blue'
axs.flat[0].plot(Alphas,LossAlpha)
axs.flat[0].set_title('Impact of alpha (max_iter = 500)')
axs.flat[0].set_xlabel('alpha')
axs.flat[0].set_xscale('log')
axs.flat[0].set_ylabel('Empirical loss',color='tab:blue')

axs.flat[1].set_title('Impact of max_iter (alpha = 10e-5)')
axs.flat[1].set_xlabel('iterations')
axs.flat[1].set_ylabel('Empirical loss', color='tab:blue')
axs.flat[1].plot(Iters, LossIter, color='tab:blue')
axs.flat[1].tick_params(axis='y', color='tab:blue')

ax2 = axs.flat[1].twinx()
color = 'tab:red'
ax2.set_ylabel('execution time (s)', color='tab:red')
ax2.plot(Iters, executionTime, color='tab:red')
ax2.tick_params(axis='y', labelcolor='tab:red')

plt.show()
```



- `alpha`

  We remark that using a step too small during the decent isn't performant, however if we use a step too big we can't create the model due to overflow.

  We found that using a step of `alpha =10e-5` create the bests results.

- `max_iter`

  Increse the number of iterations allow to reduce the empirical loss, however after 8000 iteration our model seams to have converged.

  After that the decent take more time and don't improve the perfomance much.

  That why we will use `max_iter= 8000` for our final model

Our final model

In [24]:
```
w_init = np.ones((len(X[0])))
w_final = descent(w_init, X, Y, alpha = 10e-5, max_iter=8000)
```

In [28]:
```
print(f"The empirical loss of our first model was {emp_loss(w_init, X, Y):.2f}")
print(f"The empirical loss of our final model is {emp_loss(w_final, X, Y):.2f}")
```

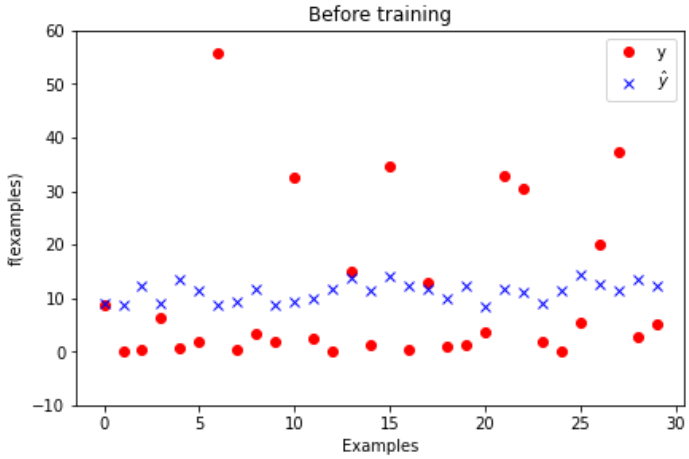
```
The empirical loss of our first model was 229.63
The empirical loss of our final model is 80.08
```

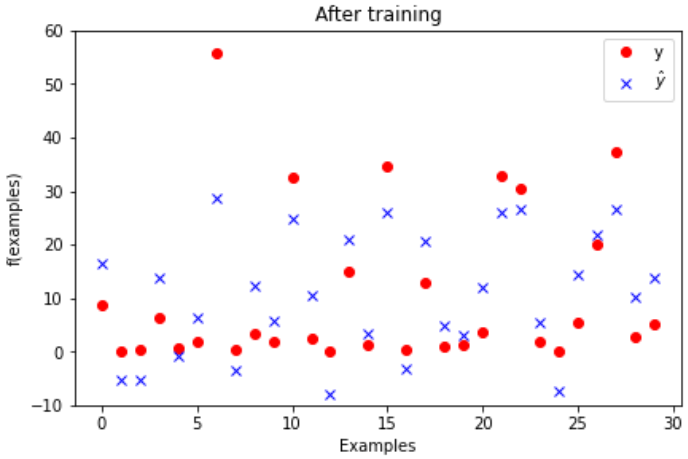
In [26]:
```
num_samples_to_plot = 30

fig, axs = plt.subplots(1, 2, constrained_layout=True)
axs.flat[0].plot(Y[0:num_samples_to_plot], 'ro', label='y')
yw = [h(w_init,x) for x in X]

axs.flat[0].plot(yw[0:num_samples_to_plot], 'bx', label='$\hat{y}$')
axs.flat[0].legend()
axs.flat[0].set_title("Before training")
axs.flat[0].set_xlabel("Examples")
axs.flat[0].set_ylim([-10, 60])
axs.flat[0].set_ylabel("f(examples)")

axs.flat[1].plot(Y[0:num_samples_to_plot], 'ro', label='y')
yw = [h(w_final,x) for x in X]
axs.flat[1].plot(yw[0:num_samples_to_plot], 'bx', label='$\hat{y}$')
axs.flat[1].set_title("After training")
axs.flat[1].legend()
axs.flat[1].set_xlabel("Examples")
axs.flat[1].set_ylim([-10, 60])
axs.flat[1].set_ylabel("f(examples)")
plt.show()
```





Going further

The following are extensions of the work previously done. If attempting them **do not modify** the code you produced above so that it can be evaluated.

Improvements to gradient descent

Consider improving the gradient descent with:

- Stochastic Gradient Descent (SGD), which means selecting a subset of the examples for training
- Detection of convergence to halt the algorithm before the maximum number of iterations

Data normalization

Different input features can have different units, and very different ranges. Within the perceptron computation, these values will be summed together. While gradient descent is normally able to deal with this (by adapting the weights of the perceptron for each input feature), standardizing the input features usually eases the perceptron training, and can sometimes improve accuracy.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler(copy=True)
X_normalized = sc.fit_transform(X)
```

Try applying a standard normalization to the input features (make sure that you keep a feature column that is always equal to 1). Is the convergence faster ? Try to quantify this speed-up. What about accuracy ?