perceptron

May 7, 2022

1 Lab2 : Perceptron

1.1 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.

1.2 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

[22]:
```python
# Import some useful libraries and functions

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

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

[23]:
```python
# 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
```

1.2.1 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

[24]:
```
# 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]}")

print("\nThere are "+str(len(X)) +" examples in this dataset.")
print("We have " + str(len(X[0]))+" features for each example.")
print("The ground truth of the 10th example is "+ str(Y[9])+".")
```

```
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

There are 308 examples in this dataset.
We have 6 features for each example.
The ground truth of the 10th example is 1.83.
```

The following command adds a column to the inputs.

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

[25]:
```
X = np.insert(X, 0, np.ones((len(X))), axis= 1)
#print_stats(X)
```

```
#We added a column with all ones so that each line (each x) has now a new 1 as␣
↪the first element
#Now when we computes, we can simply put h_w = W * X
#instead of h_w = w0 + W[1::]*X.
```

1.3 Creating the perceptron

Perceptron for regression

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`, initialize to arbitrary values (we suggest 0)
  - implement the `h` function that evaluate an example based on the vector of weights
  - check if this works on a few examples

[5]:
```
w = np.ones(len(X[0]))

def h(w, x):
    return np.sum(w*x)

# print the ground truth and the evaluation of h_w on the first example
print("h_w : " + str(h(w,X[0])) + "\nGround Truth : " + str(Y[0]))
```

```
h_w : 9.120000000000001
Ground Truth : 8.62
```

1.4 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`.

[6]:
```
def loss(w, x, y):
    return (y-h(w, x))**2
```

1.5 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.

[7]:
```
def emp_loss(w, X, Y):
    l = 0
    for x,y in zip(X,Y):
        l += loss(w,x,y)
    return l/len(X)
```

1.6 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 25 to 27).

```
[8]: def compute_update(w, X, Y, alpha):
         dw = np.zeros(len(w))

         for i in range(len(dw)):
             s = 0
             for x,y in zip(X,Y):
                 s += (y-h(w,x))*x[i]
             dw[i] = alpha*s
         return dw

     compute_update(w, X, Y, alpha = 10e-7)
```

```
[8]: array([-2.79274000e-04,  9.82236000e-05, -1.61811839e-04, -1.38917757e-03,
            -1.22565631e-03, -9.12266110e-04,  2.97171975e-04])
```

1.7 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

```
[9]: def descent(w_init, X, Y, alpha, max_iter):
         w = w_init
         for i in range(max_iter):
             w+=compute_update(w,X,Y,alpha)
         return w
```

1.8 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?

[10]:
```
#Initialization
w = np.ones(len(X[0]))
init_loss = emp_loss(w,X,Y)

#Training
w = descent(w, X, Y, 10e-5,1000)
final_loss = emp_loss(w,X,Y)
print("Initial Loss : "+ str(init_loss)+"\nLoss after a gradient descent :␣
 ↪"+str(final_loss))
```

```
Initial Loss : 229.62868491558436
Loss after a gradient descent : 157.62324245730682
```

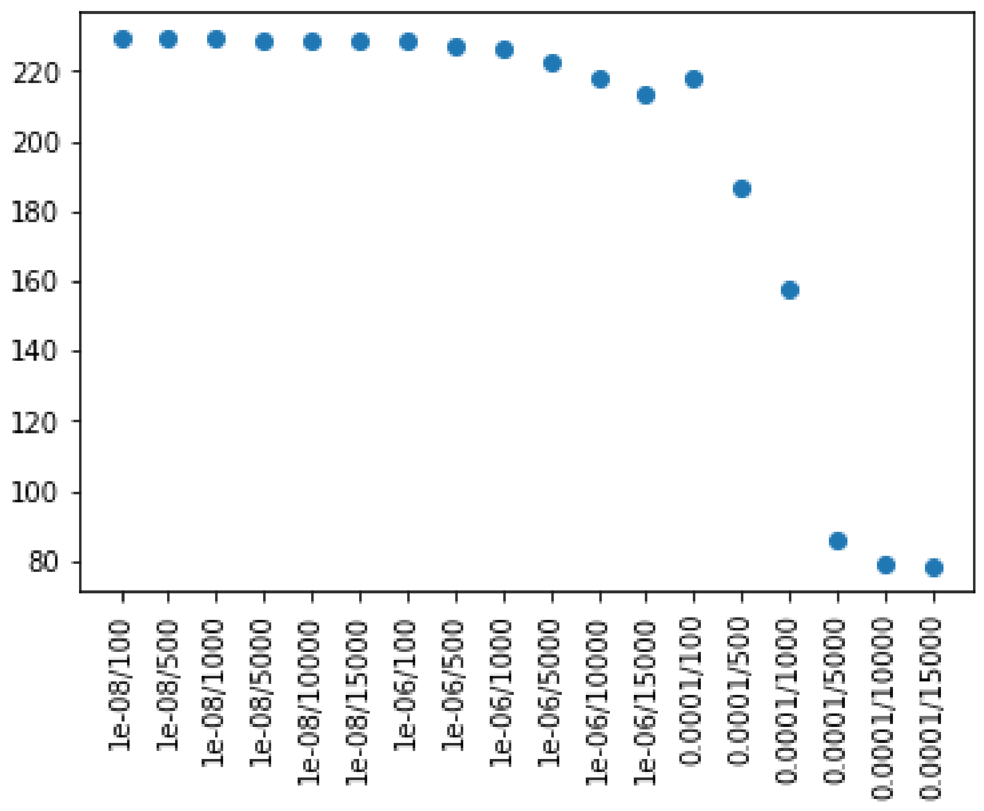
We can see that our gradient descent converged to a better empirical loss, therefore a better solution.

[11]:
```
tab_alpha= [10e-9,10e-7,10e-5] #Differents alphas
tab_max_iter = [100,500,1000,5000,10000, 15000] #Differents maximum iterations


tab_w = [] #This is will be a matrix of the differents final w and their␣
 ↪empirical losses
            #with a line corresponding to a number of maximum iterations
            #and  a column corresponding to an alpha
for alpha in tab_alpha:
    L = []
    for max_iter in tab_max_iter:
        w = np.ones(len(X[0]))
        w = descent(w,X,Y,alpha,max_iter)
        emp_lost = emp_loss(w,X,Y)
        L.append((w,emp_lost))
    tab_w.append(L)
```

[12]:
```
ordo = [ x[1] for i in tab_w for x in i]
ab = [ str(i)+"/"+ str(n) for i in tab_alpha for n in tab_max_iter]

import matplotlib.pyplot as plt
plt.scatter(ab, ordo)
plt.xticks(rotation = 90)
plt.show()
```



We can see that with our differents caracteristics, we have the best solution an alpha of 10e-5. We have an overflow with higher alphas so this will be our best. Considering the number of iterations, we have a better result with 15000 but considering the computation time either 10000 or 15000 will be good as the differences in the empirical loss is not that high. An the computation time is very long for higher numbers as it is our only stopping criteria.

[15]:
```
print("The loss associated is 15000 maximum iteration and an alpha of 10e-5 is␣
↪"+str(tab_w[2][5][1])+".")
```

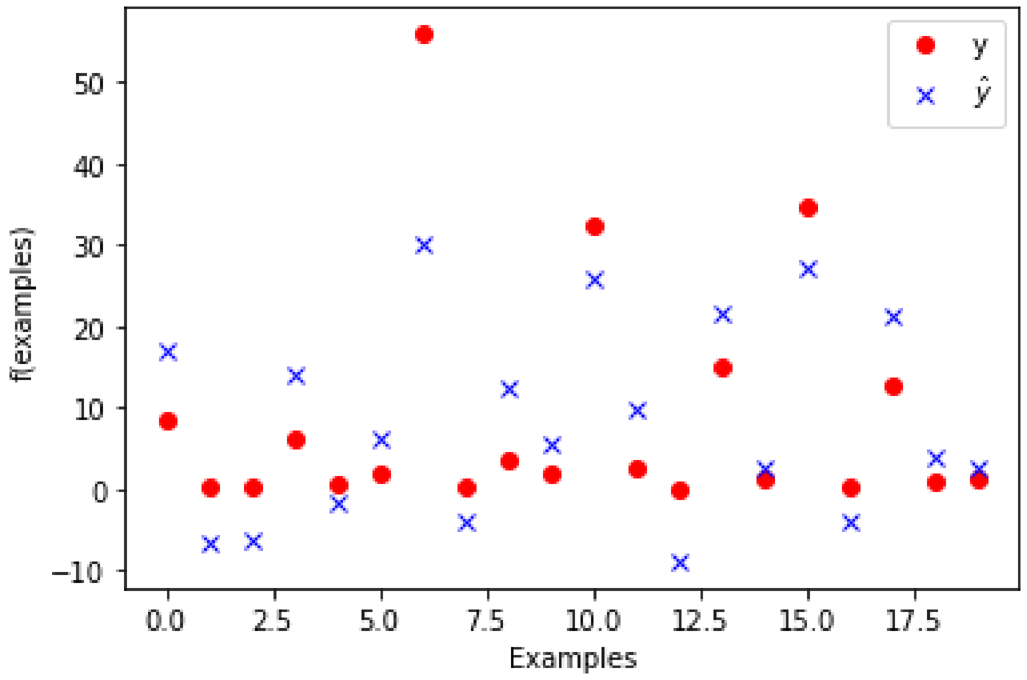
```
The loss associated is 15000 maximum iteration and an alpha of 10e-5 is
78.63602657156517.
```

[16]:
```
w = tab_w[2][5][0] #This is our best W

# Code sample that can be used to visualize the difference between the ground␣
↪truth and our best prediction

num_samples_to_plot = 20
plt.plot(Y[0:num_samples_to_plot], 'ro', label='y')
yw = [h(w,x) for x in X]
plt.plot(yw[0:num_samples_to_plot], 'bx', label='$\hat{y}$')
plt.legend()
plt.xlabel("Examples")
plt.ylabel("f(examples)")
```

```
[16]: Text(0, 0.5, 'f(examples)')
```



Our final model is not optimal because increasing the number of maximum iterations can give us a better w. However considering the computation time, we can say that it's a pretty good model compared to the initial one. Our algorithm can be improved with a better stopping criteria because some iterations can be useless if w or the empirical loss doesn't improve.