Compte rendu Modélisation non linéaire

Julie Courgibet

24 mars 2021

1 Exercice 1 : Perceptron

Il s'agit du code perceptron.py. L'exercice a été codé en Python, en utilisant uniquement la librairie numpy (manipulation de matrices). Après l'import de cette librairie, on crée les deux variables X (entrée du perceptron) et Y (sortie, ou variables à prédire.

```
1  import numpy as np
2
3
4  X_1 = [i/20 for i in range(1, 21)]
5  X_2 = [2*(i-2)/20 if i%2==0 else 2*(i+2)/20 for i in range(1, 21)]
6  X = np.array([X_1, X_2])
7  Y = np.array([0 if i%2==0 else 1 for i in range(1, 21)])
```

Pour reproduire les résultats, on peut fixer les nombres aléatoires (ligne 8). On génère ainsi les poids W et biais (soit 3 variables) de façon aléatoire, en suivant la loi normale. On fixe le pas, ou le taux d'apprentissage à 0.1. 100 passages - variables epochs - dans le perceptron se sont révélés suffisants pour prédire à 100% Y.

```
8  np.random.seed(42)
9
10 W = np.random.randn(1, 2)
11 biais = np.random.randn(1, 1)
12
13 epoch = 100
14 lr = 0.1
```

On définit ensuite deux fonctions qui sont la fonction d'activation utilisée pour le réseau de neurones (c'est ce qui rend les équations non linéaires), et la fonction de calcul d'erreur (pour ajuster les poids du neurones).

```
15 def activation(W, X, B):
16     Y_predit = np.dot(W, X) - B
17     return [1 if y_i > 0 else 0 for y_i in Y_predit[0]]
18
19
20 def erreur(Y_predit, Y_vrai):
21     return abs(Y_predit-Y_vrai).sum()/len(Y_predit)
```

Cette dernière partie est l'algorithme d'apprentissage du réseau de neurones. La condition d'arrêt est le nombre maximal d'epochs autorisés (100) ou une erreur de prédiction inférieure à 1%, ce qui dans notre cas (20 données), signifie un taux d'erreur nul.

```
erreur_predition = 1
iteration = 0
while erreur_predition > 0.01 and iteration < epoch : # Taux d'erreur de 10%
    iteration += 1
    for individu in range(X.shape[1]):
        Y_predit = activation(W, X, biais)
        for i in range(X.shape[0]):
            W[0][i] += lr * (Y[individu] - Y_predit[individu]) * X[i][individu]
        biais += -lr * (Y[individu] - Y_predit[individu])
    Y_predit = activation(W, X, biais)
    erreur_predition = erreur(Y_predit, Y)
    if iteration%10==0:
        print("iteration ", str(iteration), " erreur : ", str(erreur_predition))

print("Poids et biais :")
print(W, biais)
print("Prédiction : ")
print(activation(W, X, biais))
print("Itération : ", str(iteration))
```

Les résultats suivants font partie des solutions : w1 = -0.468 w2 = 0.252 b = 0.048

En changeant les poids de départ (W et b), on obtient d'autres solutions (qui fonctionnent avec 2 itérations) : w1 = -0.53 w2 = 0.25 b = 0.002

2 Exercice 2 : Prédiction de Y à partir de données X

2.1 Méthode du gradient : modélisation avec une équation

L'équation nous est donnée et est :

$$\phi(a,b,c) = \sum_{i=1}^{n=10} (a\ln(p[i,1]) + b\ln(p[i,2]) + c - \ln(d[i]))^2 \tag{1}$$

Cela correspond à estimer Y (ici d), comme la fonction de X (p1 et p2) :

$$d[i] = \exp(a\ln(p[i,1]) + b\ln(p[i,2]) + c) \tag{2}$$

On utilise donc la méthode du gradient constant pour connaitre a, b et c les paramètres de cette équation.

Cette partie correspond au fichier descente_gradient.py. La librairie pandas a été utilisée pour importer les données contenues dans les fichiers csv, ainsi que les fonctions, logarithme népérien (log), exponentielle et puissance de la librairie math.

```
from math import log, pow, exp
import pandas as pd

global pas
pas = 0.01

global x, y
x = pd.read_csv("p.csv", header=None)
y = pd.read_csv("d.csv", header=None)
```

Les variables ont aussi été fixées : le pas (ou taux d'apprentissage) ainsi que les variables x et y qui correspondent respectivement à p et d.

En posant :

$$f(a,b,c) = (a\ln(p[i,1]) + b\ln(p[i,2]) + c - \ln(d[i]))^2 \tag{3}$$

On obtient les différentes dérivées :

$$\frac{\partial f(a,b,c)}{\partial a} = 2 * (a\ln(p[i,1]) + b\ln(p[i,2]) + c - \ln(d[i])) * \ln(p[i,1]) \tag{4}$$

$$\frac{\partial f(a,b,c)}{\partial b} = 2 * (a\ln(p[i,1]) + b\ln(p[i,2]) + c - \ln(d[i])) * \ln(p[i,2]) \tag{5}$$

$$\frac{\partial f(a,b,c)}{\partial c} = 2 * (a\ln(p[i,1]) + b\ln(p[i,2]) + c - \ln(d[i])) \tag{6}$$

On code donc ces quatre fonctions pour les appeler dans l'algorithme du gradient constant. Dans le code Python, les variables a, b et c sont appelées alpha, beta et gamma. En effet, il est toujours mieux d'avoir des noms de variables peu équivoques et a, b ou c me semblait dangereux (facile à écraser, à confondre etc).

```
10 def f_x_y(alpha, beta, gamma, i):
11     f = alpha*log(x[0][i]) + beta*log(x[1][i]) + gamma - log(y[i][0])
12     return pow(f, 2)
13
14 def f_x_y_derivative_alpha(alpha, beta, gamma, i):
15     f = alpha*log(x[0][i]) + beta*log(x[1][i]) + gamma - log(y[i][0])
16     return 2 * f * log(x[0][i])
17
18 def f_x_y_derivative_beta(alpha, beta, gamma, i):
19     f = alpha*log(x[0][i]) + beta*log(x[1][i]) + gamma - log(y[i][0])
20     return 2 * f * log(x[1][i])
21
22 def f_x_y_derivative_gamma(alpha, beta, gamma, i):
23     f = alpha*log(x[0][i]) + beta*log(x[1][i]) + gamma - log(y[i][0])
24     return 2 * f
```

On peut aussi coder la fonction Phi, qui calcule la fonction $\phi(a,b,c)$, et son gradient :

```
25 def Phi(alpha, beta, gamma):
26     somme = 0
27     for i in range(x.shape[0]):
28         somme += f_x_y(alpha, beta, gamma, i)
29     return somme
30
31
32 def gradient_Phi(alpha, beta, gamma):
33     # Alpha
34     somme_alpha1 = 0
35     for i in range(x.shape[0]):
36         somme_alpha1 += f_x_y_derivative_alpha(alpha, beta, gamma, i)
37     alpha_1 = alpha - pas*somme_alpha1
38     # Beta
39     somme_beta1 = 0
40     for i in range(x.shape[0]):
41         somme_beta1 += f_x_y_derivative_beta(alpha, beta, gamma, i)
42     beta_1 = beta - pas*somme_beta1
43     # Gamma
44     somme_gamma1 = 0
45     for i in range(x.shape[0]):
46         somme_gamma1 += f_x_y_derivative_gamma(alpha, beta, gamma, i)
```

```
    gamma_1 = gamma - pas*somme_gamma1
    return alpha_1, beta_1, gamma_1
```

Enfin, on code la méthode à gradient constant : on calcule sur toutes les données.

```
def main(alpha, beta, gamma):
    Phi0 = Phi(alpha, beta, gamma)
    alpha_1, beta_1, gamma_1 = gradient_Phi(alpha, beta, gamma)
    Phi1 = Phi(alpha_1, beta_1, gamma_1)
    n_iteration = 1

    while Phi1 < Phi0 and n_iteration < 100:
        n_iteration += 1
        Phi0 = Phi1
        alpha, beta, gamma = alpha_1, beta_1, gamma_1
        alpha_1, beta_1, gamma_1 = gradient_Phi(alpha, beta, gamma)
        Phi1 = Phi(alpha_1, beta_1, gamma_1)

    return alpha, beta, gamma, n_iteration
```

2.1.1 Question 1.1

Avec un pas de 0.1, l'algorithme diverge. J'ai donc pris un pas de 0.01 Avec un pas de 0.01 et des valeurs de départ de (alpha=2, beta=2, gamma=0), j'obtiens une erreur de 3.74 (il s'agit de $\phi(a, b, c)$).

2.1.2 Question 1.2

Avec un pas de 0.1, l'algorithme diverge. J'ai donc pris un pas de 0.01 Avec un pas de 0.01 et des valeurs de départ de (alpha=1, beta=1, gamma=0), j'obtiens une erreur plus faible de 2.54 (il s'agit de $\phi(a, b, c)$).

2.2 Méthode du perceptron : approximation universelle

Avec cette méthode, nous n'avons pas besoin de modéliser Y en fonction de X au travers d'une équation.

2.2.1 Question 2.1

On utilise donc un réseau de neurones à deux couches et la fonction d'activation donnée dans le sujet.

On utilise les fonctions puissance et exponentielle de la librairie math. De même, on utilise la librairie random (choix aléatoire pour la méthode du gradient stochastique). La librairie pandas nous sert encore à importer les données (fichiers csv). La librairie numpy nous sert à manipuler des matrices (le code peut être simplifié pour se passer de cette librairie et utiliser une liste).

```
from math import pow, exp
import random
import pandas as pd
import numpy as np


global pas
pas = 0.1

```

```
global X, Y
X = pd.read_csv("p.csv", header=None)
Y = pd.read_csv("d.csv", header=None)
```

Les 6 fonctions suivantes permettent dans l'ordre :

- P_1j : calcule la sortie du jème perceptron de la première couche (j=0 ou 1).
- P_2 : calcule la sortie du perceptron de la seconde couche.
- Neural_prediction : calcule la sortie du réseau de neurones pour le ième élément de la matrice X (p1 et p2).
- prediction : calcule la sortie du réseau de neurones en prenant p1 et p2 en entrée.
- activation : c'est la fonction d'activation.
- dev_activation : c'est la dérivée de la fonction d'activation.

```
def P_1j(X, coefficients, j, individu):
    x = coefficients[3*j] * X[0][individu]
    x += coefficients[3*j+1] * X[1][individu]
    x -= coefficients[3*j+2]
    return activation(x)


def P_2(x1, x2, coefficients):
    x = coefficients[3*2] * x1
    x += coefficients[3*2+1] * x2
    x -= coefficients[3*2+2]
    return activation(x)


def Neural_prediction(X, coefficients, individu):
    x1 = P_1j(X, coefficients, 0, individu)
    x2 = P_1j(X, coefficients, 1, individu)
    return P_2(x1, x2, coefficients)


def prediction(x1, x2, coefficients):
    x_11 = coefficients[0] * x1
    x_11 += coefficients[1] * x2
    x_11 -= coefficients[2]
    x_11 = activation(x_11)
    x_12 = coefficients[3*1] * x1
    x_12 += coefficients[3*1+1] * x2
    x_12 -= coefficients[3*1+2]
    x_12 = activation(x_12)
    return P_2(x_11, x_12, coefficients)


def activation(x):
    return exp(2*x) / (1 + exp(2*x))


def dev_activation(x):
    numerateur = 2*exp(2*x)*(1+exp(2*x)) - 2*exp(4*x)
    denominateur = pow(1+exp(2*x), 2)
    return numerateur / denominateur
```

Il y a donc 9 paramètres à déterminer (3 paramètres pour chaque perceptron). Avec la méthode du gradient, il faut déterminer la dérivée de l'erreur par rapport à chacun des 9 paramètres

(2 poids pour les deux variables d'entrée et 1 biais).

J'ai calculé les dérivées, que je ne mettrai pas sur ce rapport. A partir de ces formules j'ai donc pu coder les 9 fonctions suivantes. Par la suite, il a fallu coder

— der_S_a111 : correspond à la dérivée de l'erreur S par rapport à $a^1_{1,1}$
— der_S_a112 : correspond à la dérivée de l'erreur S par rapport à $a^1_{1,2}$
— der_S_b11 : correspond à la dérivée de l'erreur S par rapport à b^1_1
— der_S_a122 : correspond à la dérivée de l'erreur S par rapport à $a^1_{2,2}$
— der_S_a122 : correspond à la dérivée de l'erreur S par rapport à $a^1_{2,2}$
— der_S_b12 : correspond à la dérivée de l'erreur S par rapport à b^1_2
— der_S_a21 : correspond à la dérivée de l'erreur S par rapport à a^2_1
— der_S_a22 : correspond à la dérivée de l'erreur S par rapport à a^2_2
— der_S_b2 : correspond à la dérivée de l'erreur S par rapport à b^2

Pour ne pas surcharger ce rapport, je ne mets pas le code ici, les fonctions sont dans le fichier nn_descente_gradient_stochastique.py.

Enfin, la dernière fonction main, est celle que l'on appelle pour optimiser les paramètres du réseau de neurones.

```
53 def main(coefficients):
54     functions_derivees = [der_S_a111, der_S_a112, der_S_b11, \
55                           der_S_a121, der_S_a122, der_S_b12, \
56                           der_S_a12, der_S_a22, der_S_b2]
57     for i in range(100):
58         # Choix du paramètre
59         indice = random.choice(range(len(coefficients)))
60         # Calcul des paramètres à optimiser
61         coefficients[indice] += - pas * functions_derivees[indice](X, Y, coefficients)
62     return coefficients
```

2.2.2 Question 2.2

En choisissant 100 itérations, et avec en entrée (p[1] = 4 et p[2] = 5), on trouve :

```
63 coefficients = main(coefficients_mat_reshape)
64 print(coefficients)
65 print(prediction(4, 5, coefficients))
```

La matrice de coefficients est :

$$\begin{bmatrix} 0.20660909 & 0.02581564 & -0.01239829 \\ 0.05778275 & 0.05441899 & -0.00351334 \\ -0.4847807 & -0.16796324 & 0.39664471 \end{bmatrix} \tag{7}$$

Soit pour A :

$$\begin{bmatrix} 0.20660909 & 0.02581564 \\ 0.05778275 & 0.05441899 \\ -0.4847807 & -0.16796324 \end{bmatrix} \tag{8}$$

et B :

$$\begin{bmatrix} -0.01239829 \\ -0.00351334 \\ 0.39664471 \end{bmatrix} \tag{9}$$

La prédiction pour le couple (4, 5) est : 0.13158101484239562, c'est 57% plus grand que 1/12.

2.3 Comparaison des deux méthodes

(Question 2.3)

La comparaison des deux méthodes amène à dire que la première méthode est bien plus efficace, non pas à cause de la méthode d'optimisation (gradient à pas constant versus gradient stochastique) mais à cause de la modélisation, principalement.

Dans le premier cas, on utilise une fonction, qui semble bien correspondre aux données. Ainsi, il n'y a que 3 paramètres, que l'on améliore tous à chaque itération. Dans la seconde méthode, nous n'avons pas de fonction mais un réseau de neurones qui peut (non prouvé) être universel et donc modéliser toute fonction. Cela nécessite donc bien plus de calcul (itérations). Cette méthode génère aussi 9 paramètres à optimiser. De plus, la méthode choisie (stochastique), ne met à jour qu'un seul des paramètres à chaque itération. Cela correspond environ à 11 itérations pour chaque paramètres (certains plus, certains moins).