المدرسة الوطنية للعلوم التطبيقية
ⵜⵉⵎⵔⵙⵉⵜ ⵜⴰⵏⴰⵎⵓⵔⵜ ⵏ ⵜⵙⴳⵎⵉⵍⵜ ⵜⵉⴷⵍⴰⵙⵉⵜ
Ecole Nationale des Sciences Appliquées

Cycle Ingénieur
Filière : Génie Informatique et Intelligence Artificielle GIIA
Ecole Nationale des Sciences Appliquées de Safi Université Cadi Ayyad

Compte rendu TP : Utilisation d'un réseau de neurones artificiels pour classer les différentes espèces d'iris

Module : Machine Learning

ENSA de Safi

Réalisé par :

Nom et prénom : BENSALTANA HASSAN

Encadré par :

❑ Mr. **MADIAFI Mohammed**

Année universitaire : 2022 - 2023

Introduction :

Le traitement et l'analyse de données sont de plus en plus importants dans de nombreux domaines, notamment en biologie. L'une des tâches courantes en analyse de données biologiques est la classification d'espèces. Dans ce TP, nous nous intéressons en particulier à la classification de trois espèces d'iris à partir de mesures de leurs caractéristiques.

Pour résoudre ce problème de classification, nous utiliserons un réseau de neurones artificiels avec une fonction d'activation logistique et la descente de gradient avec rétropropagation pour l'algorithme d'entraînement. L'objectif est de construire un modèle qui sera capable de prédire l'espèce d'iris à partir des caractéristiques mesurées avec une précision élevée.

Implémentation :

Importer les données :

```python
# Dans cette partie, on utilise le module csv pour lire les données d'une base de données de test.
# Cette base est utilisée pour entraîner/tester un réseau de neurone à apprentissage supervisé.
fichier = open("iris.csv")
import csv
lecteur = csv.reader(fichier)
for ligne in lecteur:
    if(lecteur.line_num == 1):
        n = int(ligne[0])
        p = int(ligne[1])
        c = int(ligne[2])
        x = []
        d = []
    else:
        # print(ligne): chaque ligne est une liste de chaînes de caractères
        x.append([float(ligne[j]) for j in range(p)])
        tmp = [0 for i in range(c)]
        tmp[int(ligne[p])-1] = 1
        d.append(tmp)

print(x[50])
print(d[50])
```

Output :

```
[7.0, 3.2, 4.7, 1.4]
[0, 1, 0]
```

Définir la classe Neurone :

```python
# Implémentation d'un neurone formel
from random import random
from math import exp
class Neurone:
    def __init__(self, nbrPoids):
        self.nbrPoids = nbrPoids
        self.sortie = random()
        self.poidsSynap = [random() for i in range(nbrPoids)]
        self.biais = random()
        self.entree = []
        self.delta = 0.
    def chargeEnt(self, vectEnt):
        self.entree = [vectEnt[i] for i in range(self.nbrPoids)]
    def calculSortie(self): #Nécissite l'exécution de la fonction chargeEnt
        self.sortie = self.biais
        for i in range(self.nbrPoids):
            self.sortie += self.poidsSynap[i]*self.entree[i]
        self.sortie = 1./(1.+exp(-self.sortie))
    def ajustPoids(self, eta, delta):
        self.delta = delta
        for i in range(self.nbrPoids):
            self.poidsSynap[i] += eta*self.delta*self.entree[i]
        self.biais += eta*delta
```

Définir la classe Couche :

```python
# Implémentation d'une couche neuronale
class Couche:
    def __init__(self, nbrNrn, nbrPoids):
        self.nbrNrn = nbrNrn
        self.nbrPoids = nbrPoids
        self.neurone = [Neurone(nbrPoids) for i in range(nbrNrn)]
    def chargeEnt(self, vectEnt):
        for i in range(self.nbrNrn):
            self.neurone[i].chargeEnt(vectEnt)
    def calculSortie(self): # Prérequis: exécution de la fonction chargerE
        for i in range(self.nbrNrn):
            self.neurone[i].calculSortie()
    def affichSortie(self):
        print([self.neurone[i].sortie for i in range(self.nbrNrn)])
    def ajustPoids(self, eta, delta):
        for i in range(self.nbrNrn):
            self.neurone[i].ajustPoids(eta, delta[i])
```

Définir la classe PMC :

```
class PMC:
    def __init__(self, nbrCch, nbrNrn, nbrPoids): #nbrNrn: tab, nbrPoids: tab
        #la première composante du vect nbrNrn contient le nombre de neurones de la première couche
        self.couche = [Couche(nbrNrn[i], nbrPoids[i]) for i in range(nbrCch)]
        self.nbrCch = nbrCch
        self.nbrNrn = [nbrNrn[i] for i in range(self.nbrCch)]
        self.nbrPoids = [nbrPoids[i] for i in range(self.nbrCch)]
    def propager(self, vectEnt): # c'est calculerSortie pour le réseau de neurones
        self.couche[0].chargeEnt(vectEnt)
        self.couche[0].calculSortie()
        for i in range(1, self.nbrCch):
            sortCchPrec = [self.couche[i-1].neurone[j].sortie for j in range(self.nbrNrn[i-1])]
            self.couche[i].chargeEnt(sortCchPrec)
            self.couche[i].calculSortie()
    def affichSortie(self):
        self.couche[self.nbrCch-1].affichSortie()
    def entrainer(self, X, D, eta, tmax):
        for r in range(tmax):
            for q in range(len(X)):
                # Propager les signaux d'entrée pour chaque exemple
                self.propager(X[q])
                # Ajuster les poids synaptiques des neurones de la couche de sortie
                delta = [
                    (D[q][j] - self.couche[self.nbrCch-1].neurone[j].sortie)*self.couche[self.nbrCch-
                    1].neurone[j].sortie*(1. - self.couche[self.nbrCch-1].neurone[j].sortie) for j in
                    range(self.couche[self.nbrCch-1].nbrNrn)
                ]
                for j in range(self.couche[self.nbrCch-1].nbrNrn):
                    self.couche[self.nbrCch-1].neurone[j].ajustPoids(eta, delta[j])
                # Ajuster les poids synaptiques des neurones des couches cachées et de la couche d'entrée
                for s in range(self.nbrCch-2, -1, -1):
                    delta = [random() for i  in range(self.couche[s].nbrNrn)]
                    for k in range(self.couche[s].nbrNrn):
                        delta[k] = 0.
                        for i in range(self.couche[s+1].nbrNrn):
                            delta[k] += self.couche[s+1].neurone[i].poidsSynap[k]*self.couche[s+1].neurone[i].delta
                        delta[k] *= self.couche[s].neurone[k].sortie*(1. - self.couche[s].neurone[k].sortie)
                    for j in range(self.couche[s].nbrNrn):
                        self.couche[s].neurone[j].ajustPoids(eta, delta[j])
```

Matrice de confusion :

La fonction est mise en œuvre en tant que méthode de la classe PMC. Elle calcule le taux de réussite et affiche la matrice de confusion, où les lignes représentent les classifications réelles actuelles et les colonnes représentent les classifications prédites.

```
def mtrcConfusion(self, x, d):
    n = len(d[0])
    matrice = [[0 for k in range(n)] for l in range(n)]
    for q in range(len(x)):
        self.propager(x[q])
        y = 0 # utilisé comme l'argument maximal de la liste d: sortie désirée
        z = 0 # utilisé comme l'argument maximal des sorties calculées
        for k in range(1, 3):
            if self.couche[self.nbrCch-1].neurone[k].sortie > self.couche[self.nbrCch-1].neurone[z].sortie:
                z = k
            if d[q][k] > d[q][y]:
                y = k
        matrice[y][z] = matrice[y][z] + 1
    tauxReuss = ((matrice[0][0] + matrice[1][1] + matrice[2][2]) / 150.) * 100
    print("Taux de réussite est =", tauxReuss, "%")
    print()
    print("Matrice de confusion :")
    for ligne in matrice:
        print(ligne)
```

Je teste ma fonction avec :

```
pmc = PMC (2 , [3 , c] , [p , 3])
pmc.entrainer (x , d , 0.2 , 10000)
pmc.mtrcConfusion (x , d)
```

Output :

```
Taux de réussite est = 98.66666666666667 %

Matrice de confusion :
[50, 0, 0]
[0, 48, 2]
[0, 0, 50]
```

Je change les valeurs de taux d'apprentissage **μ** et le nombre d'itérations N

```
Taux de réussite est = 97.33333333333334 %

Matrice de confusion :
[50, 0, 0]
[0, 46, 4]
[0, 0, 50]
```

*N=10000, **μ=0.1***

```
Taux de réussite est = 98.66666666666667 %

Matrice de confusion :
[50, 0, 0]
[0, 48, 2]
[0, 0, 50]
```

*N=10000, **μ=0.01***

```
Taux de réussite est = 95.33333333333334 %

Matrice de confusion :
[50, 0, 0]
[0, 43, 7]
[0, 0, 50]
```

N=10000, ***μ=0.99***

```
Taux de réussite est = 95.33333333333334 %

Matrice de confusion :
[50, 0, 0]
[0, 43, 7]
[0, 0, 50]
```

N=10, ***μ=0.2***

```
Taux de réussite est = 98.0 %

Matrice de confusion :
[50, 0, 0]
[0, 47, 3]
[0, 0, 50]
```

N=100, ***μ=0.2***

```
Taux de réussite est = 98.0 %

Matrice de confusion :
[50, 0, 0]
[0, 47, 3]
[0, 0, 50]
```

N=1000, ***μ=0.2***