Implémentation d’un **réseau de neurones simple à une couche cachée** en Python, en utilisant la bibliothèque **NumPy** pour la manipulation de données et des matrices.

Objectif du code

L'objectif est de prédire la couleur d'une fleur (rouge ou bleue) en fonction de deux caractéristiques d'entrée. Les données d'entrée (`x_entrener`) contiennent des exemples de caractéristiques (par exemple, la longueur et la largeur de la fleur), et les données de sortie (`y`) indiquent la couleur de la fleur (1 pour rouge, 0 pour bleu).

---

1. Données d'entrée et de sortie

```
python

x_entrener = np.array([[3, 1.5], [2, 1], [4, 1.5], [3, 1], [3.5, 0.5], [2, 0.5], [5.5, 1], [1, 1], [1.5, 1.5]], dtype=float)
y = np.array([[1], [0], [1], [0], [1], [0], [1], [0], [0]], dtype=float)  # données de sortie 1 = Rouge / 0 = Bleu
```

- **`x_entrener`** : Données d'entrée, où chaque sous-liste représente des caractéristiques de la fleur.
- **`y`** : Étiquettes de sortie, où `1` signifie "rouge" et `0` signifie "bleu".

Dans ce contexte, lorsque l’on parle de **caractéristiques** (ou **features** en anglais), on fait référence aux **attributs mesurables** ou **propriétés** qui décrivent chaque fleur et qui seront utilisées par le modèle pour faire la prédiction.

Dans le cas de ce code, chaque sous-liste de `x_entrener` contient deux valeurs, qui représentent les **caractéristiques** de chaque fleur.

Concrètement, voici ce que ces caractéristiques peuvent représenter :

1. **Première caractéristique (par exemple, la taille de la fleur)** : Cela pourrait être la longueur de la fleur ou la taille moyenne d'un aspect particulier de la fleur, comme la longueur d'un pétale ou d'une feuille.
2. **Deuxième caractéristique (par exemple, la largeur de la fleur)** : Cela pourrait être la largeur d'un pétale, la largeur d'une feuille ou un autre aspect mesurable de la fleur.

Exemple pour mieux comprendre

Supposons que :

- La **première valeur** de chaque sous-liste représente la **longueur du pétale**.
- La **deuxième valeur** représente la **largeur du pétale**.

Ainsi, pour la première entrée `x_entrener = [3, 1.5]`, cela signifierait que :

- La longueur du pétale est **3 unités**.
- La largeur du pétale est **1.5 unités**.

Objectif de ces caractéristiques

Ces caractéristiques permettent au modèle d'**apprendre à différencier les fleurs**. Le réseau de neurones utilise ces valeurs pour apprendre un modèle qui associe certains types de caractéristiques à la classe de la fleur (Rouge ou Bleu). Par exemple :

- Une fleur avec des pétales longs et fins pourrait être classée comme **bleue**.
- Une fleur avec des pétales courts et larges pourrait être classée comme **rouge**.

Ces caractéristiques sont donc des informations essentielles pour aider le réseau à **reconnaître des motifs** et à faire des prédictions basées sur des données similaires.

---

2. Normalisation des données

```
python

x_entrener /= np.amax(x_entrener, axis=0)
X, xPrediction = np.split(x_entrener, [8])
y, yPrediction = np.split(y, [8])
```

- Les données d'entrée sont **normalisées** pour avoir des valeurs comprises entre 0 et 1, en divisant chaque valeur par le maximum de sa colonne.
- `X` et `y` sont les données pour l'entraînement, tandis que `xPrediction` et `yPrediction` sont utilisées pour la prédiction après l'entraînement.

---

3. Définition de la classe du réseau de neurones

La classe `Neural_Network` représente un réseau de neurones à une couche cachée.

```
python

class Neural_Network(object):
    def __init__(self):
        # Paramètres
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3

        # Poids (les matrices de poids W1 et W2)
        self.W1 = np.random.randn(self.inputSize, self.hiddenSize)  #
Matrice 2x3
        self.W2 = np.random.randn(self.hiddenSize, self.outputSize)  #
Matrice 3x1
```

- **`inputSize`** : Nombre de neurones dans la couche d'entrée (2).
- **`outputSize`** : Nombre de neurones dans la couche de sortie (1).
- **`hiddenSize`** : Nombre de neurones dans la couche cachée (3).
- **`W1`** : Poids entre la couche d'entrée et la couche cachée (matrice 2x3).
- **`W2`** : Poids entre la couche cachée et la couche de sortie (matrice 3x1).

---

4. Propagation avant (Forward Propagation)

```
python

def forward(self, X):
    self.z = np.dot(X, self.W1)
    self.z2 = self.sigmoid(self.z)
    self.z3 = np.dot(self.z2, self.W2)
    o = self.sigmoid(self.z3)
    return o
```

- **Étape 1** : Calculer la sortie de la couche cachée (`z`) en multipliant les entrées `X` par les poids `W1` et en appliquant la fonction sigmoïde (`z2`).
- **Étape 2** : Calculer la sortie finale (`o`) en multipliant `z2` par `W2` et en appliquant la fonction sigmoïde à nouveau.

---

5. Fonction d'activation Sigmoïde

```
python

def sigmoid(self, s):
    return 1 / (1 + np.exp(-s))
```

- La **fonction sigmoïde** est utilisée pour normaliser les valeurs entre 0 et 1, ce qui est utile pour des prédictions probabilistes.

6. Dérivée de la fonction Sigmoïde

```
python

def sigmoidPrime(self, s):
    return s * (1 - s)
```

- La **dérivée de la sigmoïde** est utilisée pour la rétropropagation, pour ajuster les poids en fonction de l'erreur.

---

7. Rétropropagation (Backward Propagation)

```
python

def backward(self, X, y, o):
    self.o_error = y - o  # Erreur en sortie
    self.o_delta = self.o_error * self.sigmoidPrime(o)

    self.z2_error = self.o_delta.dot(self.W2.T)  # Erreur à z2
    self.z2_delta = self.z2_error * self.sigmoidPrime(self.z2)

    self.W1 += X.T.dot(self.z2_delta)
    self.W2 += self.z2.T.dot(self.o_delta)
```

- **Calcul de l'erreur en sortie (`o_error`)** : Différence entre la sortie prévue (`o`) et la sortie réelle (`y`).
- **Calcul des décalages pour ajuster les poids (`o_delta` et `z2_delta`)** : En appliquant la dérivée de la sigmoïde aux erreurs, on obtient les ajustements nécessaires.
- **Mise à jour des poids** (`W1` et `W2`) : Ajustement des poids en fonction des décalages calculés.

---

8. Entraînement du réseau de neurones

```
python

def train(self, X, y):
    o = self.forward(X)
    self.backward(X, y, o)
```

- La fonction `train` exécute une étape de propagation avant puis de rétropropagation pour mettre à jour les poids.

---

9. Prédiction

```
python

def predict(self):
    print("Donnée prédite après entraînement : ")
    print("Entrée : \n" + str(xPrediction))
    print("Sortie : \n" + str(self.forward(xPrediction)))
```

```
    if (self.forward(xPrediction) < 0.5):
        print("La fleur est BLEUE ! \n")
    else:
        print("La fleur est ROUGE ! \n")
```

- La fonction `predict` utilise le réseau de neurones entraîné pour faire une prédiction sur `xPrediction`.
- Si la sortie est inférieure à 0,5, elle est interprétée comme "BLEU", sinon comme "ROUGE".

---

10. Création et Entraînement du Réseau de Neurones

```
python

NN = Neural_Network()

for i in range(15000):
    NN.train(X, y)
```

- **Boucle d'entraînement** : Le réseau est entraîné pendant 15 000 itérations pour ajuster les poids et minimiser l'erreur.

Conclusion

Ce code met en œuvre un **réseau de neurones simple** pour classer des fleurs en fonction de deux caractéristiques d'entrée. Il utilise une couche cachée et applique une **fonction d'activation sigmoïde** pour la normalisation. Après l'entraînement, le réseau peut prédire si une fleur est bleue ou rouge en fonction des valeurs d'entrée.

Le réseau ajuste ses poids à chaque itération en utilisant **la rétropropagation** pour minimiser l'erreur de prédiction.