Reconnaissance de chiffres en Python

Objectif :

Le but des exercices de cette feuille est de réussir à créer un algorithme en Python permettant de reconnaître un chiffre à travers une image.
On implémentera ainsi différentes méthodes, plus ou moins simples, pour ce qui est des images on pourra utiliser cette archive. (Pour l'extraire : `tar -xzf chiffres.tar.gz`)

Préparation :

Avant de reconnaître des chiffres on doit pouvoir lire et manipuler des images en Python. Dans toute la feuille on considére une image de largeur $\boldsymbol{l}$ et de hauteur $\boldsymbol{h}$.

1. Tout d'abord on aura besoin d'une fonction pour lire cette image depuis un fichier, on pourra par exemple utiliser la classe `Image` de la librairie `PIL`.
   (Pour l'installer : `pip3 install Pillow`)
2. Ensuite il faut une fonction pour passer de l'image à un objet plus simple à manipuler, on pourra la représenter comme une **matrice** de pixels de la forme $(1, \boldsymbol{l} \cdot \boldsymbol{h})$, par exemple avec la librairie `numpy`. On devra aussi transformer les couleurs en une valeur dans $[\mathbf{0}, \mathbf{1}]$.
   (Remarque : On fera attention à ce que notre matrice soit bien en 2 dimensions pour pouvoir utiliser le produit matriciel correctement)
3. Enfin on pourra faire une fonction pour récupérer une liste contenant tous les couples de référence `(image, chiffre)` depuis un répertoire.
   (Remarque : pour lister tous les fichiers d'un répertoire on pourra utiliser la fonction `listdir(repertoire)` de la librairie `os`)

Partie 1 : Les k plus proches voisins

Dans cette partie on va considérer une image comme un **point** ou un **vecteur** de $[\mathbf{0}, \mathbf{1}]^{\boldsymbol{l} \cdot \boldsymbol{h}}$.
Ainsi en calculant la **distance** entre deux points ou le **produit scalaire** de deux vecteurs, on peut obtenir une mesure de la ressemblance d'une image à une autre et utiliser la méthode des k plus proches voisins.

1. D'abord on crée une fonction pour calculer la **distance** entre deux points de $[\mathbf{0}, \mathbf{1}]^{\boldsymbol{l} \cdot \boldsymbol{h}}$.
   On pourra appliquer la formule de la distance euclidienne de la même manière que dans l'espace, ou une autre si vous trouvez plus intéressant.
   (Remarque : pour les performances on pourra enlever la racine carré dans la formule de la distance euclidienne qui n'est pas nécessaire ici)
2. On fera aussi une fonction pour calculer le **produit scalaire** de deux vecteurs de $[\mathbf{0}, \mathbf{1}]^{\boldsymbol{l} \cdot \boldsymbol{h}}$.
   Encore une fois, la formule s'applique de la même manière que dans l'espace.
3. Enfin on peut créer une fonction pour reconnaître un chiffre depuis une image, pour ça on peut trouver les **k** images parmi les références dont les points sont les **moins distants**, ou dont les vecteurs se **ressemblent le plus**, et en déduire le chiffre.
   On pourra comparer la méthode avec la distance et celle avec le produit scalaire pour voir laquelle fonctionne le mieux.

Partie 2 : Un réseau neuronal basique

Dans cette partie on va toujours utiliser l'image sous forme d'une matrice $\boldsymbol{m}$, et on va vouloir calculer à partir de cette matrice un vecteur de taille 10, $\boldsymbol{p} = (p_0 \ p_1 \ \cdots \ p_8 \ p_9)$, où $p_n \in [0,1]$ représente la probabilité que n soit le chiffre de l'image.
Pour ça on va utiliser deux matrices, une matrice de poids $\boldsymbol{W}$ de la forme $(\boldsymbol{l} \cdot \boldsymbol{h}, 10)$, et une matrice de biais $\boldsymbol{b}$ de la forme $(1, 10)$.

1. On peut utiliser le module `random` de la librairie `numpy` pour initialiser $\mathbf{W}$ et $\mathbf{b}$ avec des distributions aléatoires, par exemple une distribution uniforme entre -1 et 1, à vous de trouver le mieux.

Vous pouvez remarquer qu'en faisant $\boldsymbol{p} = \boldsymbol{m} \cdot \boldsymbol{W} + \boldsymbol{b}$ on peut obtenir la forme voulue.

2. On fait donc une fonction pour calculer ce résultat, on pensera à faire attention à la forme de la matrice de sortie.

Mais il y a deux problèmes, d'abord les valeurs de $\mathbf{p}$ ne sont pas dans l'intervalle $[0,1]$, et en plus en faisant ce calcul $\mathbf{W}$ et $\mathbf{b}$ ne nous apportent aucune information.

3. Pour régler le premier problème, on peut utiliser la fonction sigmoïde dont l'expression est $\sigma(x) = \frac{1}{1+e^{-x}}$

Pour régler le deuxième problème il va falloir que notre programme apprenne de ses erreurs. Pour ça on va utiliser une fonction objectif, cette fonction permet d'évaluer la qualité de nos prédictions, et on modifiera nos poids $\mathbf{W}$ et nos biais $\mathbf{b}$ en fonction.

4. Ici on va utiliser la fonction d'erreur quadratique moyenne, pour faire simple ce qui nous intéresse est sa dérivée $\Delta(\boldsymbol{p}) = \boldsymbol{p} - (\ldots \ 0 \ 1 \ 0 \ \ldots)$ avec le 1 à l'indice n, le chiffre à prédire. Après l'avoir implémenter, on pourra propager le résultat en arrière pour ajuster $\mathbf{W}$ et $\mathbf{b}$.

Pour apprendre en évitant de faire trop de calculs, on fait un certains nombre de prédictions pour des matrices $\boldsymbol{m_k}$ pour lesquels on calcule Δ_k avant de définir $\Delta\boldsymbol{W}$ et $\Delta\boldsymbol{b}$ comme suit :

$$\Delta\boldsymbol{W} = \sum_{k=1}^{10} {\boldsymbol{m_k}}^t \cdot \Delta_k$$

$$\Delta\boldsymbol{b} = \sum_{k=1}^{10} \Delta_k$$

Ici on a fait 10 prédictions, avec ${\boldsymbol{m_k}}^t$ la transposée.

5. Ainsi pour revenir en arrière on fait une fonction qui multiplie d'abord chaque terme de Δ_k par ceux de la **dérivée** de la fonction sigmoïde appliquée à $\boldsymbol{m_k} \cdot \boldsymbol{W} + \boldsymbol{b}$, puis ajoute respectivement ${\boldsymbol{m_k}}^t \cdot \Delta_k$ et Δ_k à $\Delta\boldsymbol{W}$ et $\Delta\boldsymbol{b}$.

6. Ensuite on peut faire une fonction pour mettre à jour $\mathbf{W}$ et $\mathbf{b}$ en leur soustrayant respectivement $\alpha \cdot \Delta\boldsymbol{W}$ et $\alpha \cdot \Delta\boldsymbol{b}$, où α représente la **vitesse d'apprentissage** (on pourra prendre un nombre entre 0 et 1)

7. Avec tout ça on peut faire une fonction intermédiaire pour s'entraîner sur un groupe d'images, pour chaque image on commence par calculer une prédiction $\boldsymbol{p} = \sigma(\boldsymbol{m} \cdot \boldsymbol{W} + \boldsymbol{b})$, puis on lui applique Δ avant de retourner en arrière pour ajuster nos paramètres.

En répétant cette opération pour plusieurs groupes d'images on améliore petit à petit nos paramètres et nos prédictions deviennent meilleures.

8. Pour terminer on peut donc faire une fonction qui s'entraînent sur un ensemble d'images en les séparant en petits groupes, on peut répéter cet entraînement plusieurs fois et calculer le pourcentage de réussite pour chaque génération sur des images de test.

En fonction de l'ensemble de base utilisé on peut rapidement arriver à 80~90% de prédictions réussites. En bonus on peut essayer de sauvegarder nos paramètres $\mathbf{W}$ et $\mathbf{b}$ lorsque le pourcentage de réussite augmente.