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 taille $1 \times \boldsymbol{l} \cdot \boldsymbol{h}$, 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 : Base d'un réseau neuronal

Dans cette partie on va toujours utiliser l'image comme une matrice $\boldsymbol{m}$, et on va vouloir calculer à partir de celle-ci une **matrice** de taille 1×10, $\boldsymbol{p} = (p_0 \ p_1 \ \cdots \ p_8 \ p_9)$, 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 taille $\boldsymbol{l} \cdot \boldsymbol{h} \times 10$ et une matrice de biais $\boldsymbol{b}$ de taille 1×10.

1. On commence par initialiser $\mathbf{W}$ et $\mathbf{b}$ avec des 0, on pourrait aussi utiliser des distributions aléatoires, à vous de voir ce qui fonctionne le mieux.

Vous pouvez remarquer qu'en utilisant l'application affine $\boldsymbol{p} = \boldsymbol{m} \cdot \boldsymbol{W} + \boldsymbol{b}$ on peut obtenir la taille voulue pour $\mathbf{p}$.

2. On fait donc une fonction pour calculer ce résultat, on pensera à faire attention à bien utiliser le produit matriciel et à la taille de la matrice de sortie.

Mais on a un problème, les valeurs de $\mathbf{p}$ ne sont pas dans l'intervalle $[0, 1]$ et ne s'apparentent pas à des probabilités.

3. Pour régler ça, on peut utiliser la fonction sigmoïde dont l'expression est $\sigma(x) = \frac{1}{1+e^{-x}}$ et l'appliquer à nos prédictions $\mathbf{p}$.

De plus, dans ce calcul $\mathbf{W}$ et $\mathbf{b}$ ne nous apportent pour l'instant aucune information, il va falloir que notre réseau apprenne de ses erreurs pour qu'ils prennent du sens. Pour ça on va utiliser une fonction objectif, ce type de fonction permet d'évaluer la qualité de nos prédictions, et on modifiera nos paramètres $\mathbf{W}$ et $\mathbf{b}$ en se basant sur sa dérivée.
(Remarque : on s'intéresse à la dérivée car le but est de trouver un minimum, ce qui revient à avoir fait une bonne prédiction)

4. Ici on va utiliser la fonction d'erreur quadratique moyenne, pour faire simple sa dérivée est $\Delta(\boldsymbol{p}) = \boldsymbol{p} - (0 \ \ldots \ 0 \ 1 \ 0 \ \ldots \ 0)$ avec le 1 à l'indice n, où n est le chiffre à prédire. Vous pourrez essayer d'en trouver une meilleure.

Après l'avoir implémenter, on veut propager le résultat Δ en retournant en arrière dans notre réseau neuronal.

5. Pour ça on fait une fonction qui calcule la dérivée de la fonction sigmoïde en $\boldsymbol{p}$ et multiplie Δ par le résultat terme à terme.
   (Remarque : si on voulait retourner plus en arrière on pourrait continuer avec la dérivée de l'application affine $\boldsymbol{m} \cdot \boldsymbol{W} + \boldsymbol{b}$)

Tout en propageant le résultat, on veut aussi améliorer $\mathbf{W}$ et $\mathbf{b}$, on va pouvoir utiliser Δ après l'avoir calculé.

6. On crée une fonction pour modifier nos deux paramètres :
   - Le changement que reçoit $\mathbf{b}$ est simplement $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \cdot \Delta$
   - Le changement que reçoit $\mathbf{W}$ est donné par $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \cdot \boldsymbol{m}^t \cdot \Delta$

   Où $\alpha \in [0, 1]$ représente la **vitesse d'apprentissage**, et $\boldsymbol{m}^t$ est la transposée (pour que le produit matriciel soit possible).

7. Avec tout ça on peut faire une fonction intermédiaire pour s'entraîner sur une 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 et d'ajuster nos paramètres.

En répétant cette opération pour plusieurs 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, on peut répéter plusieurs fois en **mélangeant** les images 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 entre 80% et 90% de prédictions réussites. En bonus on peut essayer de sauvegarder nos paramètres **W** et **b** lorsque le pourcentage de réussite augmente.

Partie 3 : TODO