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`)
Remarque : Les parties sont organisées de manière linéaire, il sera utile de réutiliser ce que vous aurez fait.

Préparation :

Avant de reconnaître des chiffres on doit pouvoir lire et manipuler des images en Python. Dans toute la feuille on considère une image de largeur **l** et de hauteur **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 première 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 deuxième partie on va toujours utiliser l'image comme une matrice $\mathbf{m}$, 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 paramètres, une matrice de poids $\mathbf{W}$ de taille $\boldsymbol{l} \cdot \boldsymbol{h} \times 10$ et une matrice de biais $\mathbf{b}$ 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. On va utiliser la fonction sigmoïde dont l'expression est $\sigma : x \mapsto \frac{1}{1+e^{-x}} \in [0, 1]$ et l'appliquer à notre matrice $\mathbf{p}$ pour la rendre correcte.

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 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 donnée par $\boldsymbol{p} \mapsto \boldsymbol{p} - (0 \ \ldots \ 0 \ 1 \ 0 \ \ldots \ 0)$ 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 implémenté, on veut propager le résultat en retournant en arrière dans notre réseau neuronal.

5. Pour ça on multiplie le résultat précédent par la dérivée de la fonction sigmoïde en $\boldsymbol{p}$, terme à terme, on appelle ce produit Δ.
   (*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 $\mathbf{W}$ est $\boldsymbol{W} \mapsto \boldsymbol{W} - \alpha \cdot \boldsymbol{m}^t \cdot \Delta$
   - Le changement que reçoit $\mathbf{b}$ est simplement $\boldsymbol{b} \mapsto \boldsymbol{b} - \alpha \cdot \Delta$

   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'entraîne 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 : Réseau neuronal séquentiel

Dans cette troisième partie, nous allons construire un réseau neuronal séquentiel, contrairement à la partie précédente où l'on avait une seule couche, on aura plusieurs couches de tailles différentes ayant chacune leurs poids et leurs biais, chaque couche aura un unique prédécesseur et un unique successeur.
À la fin, on peut imaginer que notre système neuronal ressemblera à ça :

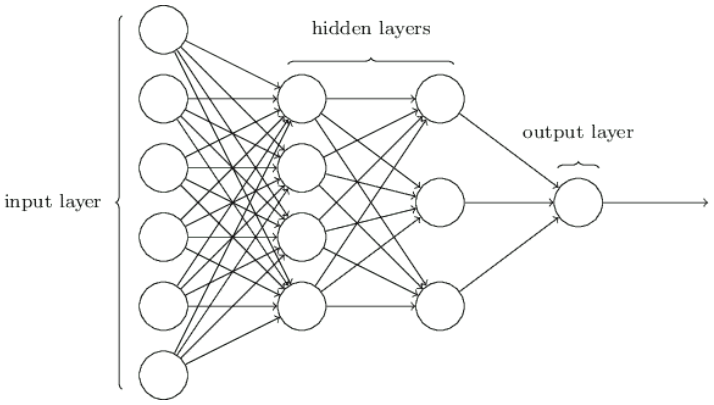

Figure 1: Un réseau neuronal à deux couches

Par exemple, dans un réseau avec 3 couches, au lieu de faire directement $\boldsymbol{p} = \boldsymbol{m} \cdot \boldsymbol{W} + \boldsymbol{b}$, on va calculer $\boldsymbol{p} = ((\boldsymbol{m} \cdot \boldsymbol{W_1} + \boldsymbol{b_1}) \cdot \boldsymbol{W_2} + \boldsymbol{b_2}) \cdot \boldsymbol{W_3} + \boldsymbol{b_3}$, où $\boldsymbol{W_n}$ et $\boldsymbol{b_n}$ sont les matrices de poids et de biais de la couche n, l'idée est d'avoir plusieurs étapes au lieu de passer directement de l'image à une prédiction.
(*Exemple* : $\boldsymbol{m} = 1 \times \mathbf{784} \to 1 \times \mathbf{392} \to 1 \times \mathbf{196} \to 1 \times \mathbf{10} = \boldsymbol{p}$)

1. Créons d'abord une classe pour représenter une **couche**. Une couche aura donc des données en entrée $\boldsymbol{e}$ et en sortie $\boldsymbol{s}$, elle aura également un **prédécesseur** et un **successeur** (`None` si elle n'en a pas), ainsi qu'une matrice de poids **W** et une matrice de biais **b**.
   (*Remarque* : la tailles de **W** et **b** dépendra de la taille voulue pour l'entrée et la sortie)

Pour commencer on a besoin de calculer la sortie d'une couche à partir de son entrée.

2. On peut faire une méthode pour faire ce calcul à partir de **W** et **b**, on peut se référer aux questions 2 et 3 de la partie 2.

Pour la suite nos couches ont besoin de pouvoir s'échanger leurs résultats.

3. On a d'abord besoin d'une méthode permettant à une couche de se **connecter** à une autre, cela permettra de récupérer les données venant de la suivante ou de la précédente.

4. Ensuite, pour faciliter les prochaines exécutions, on peut créé une méthode pour récupérer les données en **entrée** d'une couche, c'est à dire la sortie de la couche précédente (pensez à traiter le cas où la couche n'a pas de prédécesseur).

Maintenant que nos différentes couches peuvent interagir, on peut les utiliser dans un réseau neuronal.

5. Dans un premier temps, nous pouvons créer une classe pour représenter un **réseau neuronal**. un réseau neuronal est simplement composée d'une liste de **couches**. On ajoutera une méthode permettant d'ajouter une couche au réseau neuronal et de la connecter à la dernière.

Dans les prochaines questions nous allons nous concentrer sur l'apprentissage de notre réseau neuronal.

6. Nous pouvons implémenter une méthode entrainant notre système neuronal sur une image, elle prendra comme paramètre notre **image**, le **chiffre** associé à l'image, et la vitesse d'apprentissage $\boldsymbol{\alpha}$. Dans cette fonction, nous donnerons à notre première couche l'image comme entrée, puis on transmettra le résultat à chaque couche l'une après l'autre, jusqu'à obtenir la prédiction $\boldsymbol{p}$.

Une fois que l'image est passée par toutes nos couches, il va falloir évaluer le résultat et retourner en arrière.

7. Ainsi, on peut modifier nos **couches** pour qu'elles aient deux nouveaux attributs, le delta d'entrée Δ_e et le delta de sortie Δ_s, les deltas nous indiquent les changements à appliquer à chaque couche.

8. Ensuite, pour faciliter les prochaines exécutions, on peut créé une méthode pour récupérer le delta en **entrée** d'une couche, c'est à dire le delta en sortie de la couche suivante (pensez à traiter le cas où la couche n'a pas de successeur).

Maintenant on peut utiliser ces deux attributs pour retourner en arrière et modifier nos paramètres pour chacune de nos couches.

9. On peut créer une méthode qui nous permet de faire ça en se basant sur le delta en entrée Δ_e de notre couche. On définit $\Delta = \sigma'(\boldsymbol{e} \cdot \boldsymbol{W} + \boldsymbol{b}) \times \Delta_e$ (la multplication $\times$ est faite terme à terme) et les paramètres de chaque couche sont modifiées comme suit :
   - Le changement que reçoit $\boldsymbol{W}$ est $\boldsymbol{W} \mapsto \boldsymbol{W} - \alpha \cdot \boldsymbol{m}^t \cdot \Delta$
   - Le changement que reçoit $\boldsymbol{b}$ est simplement $\boldsymbol{b} \mapsto \boldsymbol{b} - \alpha \cdot \Delta$

   Enfin on définit le delta de sortie $\Delta_s = \Delta \cdot \boldsymbol{W}^t$

10. Maintenant on peut changer notre méthode d'entrainement pour qu'elle retourne en arrière après avoir évalué une image, notre premier Δ_e sera obtenu comme dans la question 4 de la partie 2.

11. Pour suivre on peut faire une fonction qui s'entraîne sur un ensemble d'image comme dans la question 8 de la partie 2.

12. Pour finir vous pouvez initialiser votre réseau neuronal, ajouter des neurones (attention aux dimensions d'entrée et de sortie), et l'entraîner sur le jeu de données fourni.

Remarque : Si vous ajoutez un unique neurone, le résultat sera très similaire à la partie 2. Cependant, ajouter plusieurs neurones prendra plus de temps au niveau de l'entraînement, mais le changement des paramètres dans l'ensemble du réseau neuronal sera plus précis.