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 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 $\boldsymbol{m}$, 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 paramètres, 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. 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 $\boldsymbol{p} \mapsto \boldsymbol{p} - (0 \ \dots \ 0 \ 1 \ 0 \ \dots \ 0)$ 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'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 : Réseau neuronal séquentiel

Dans cette troisième partie, nous allons produire un réseau neuronal séquentiel, c'est à dire que chaque neurone aura un unique prédécesseur et successeur. À la fin, on peut imaginer que notre système neuronal ressemblera à quelque chose comme ça :

IMAGE

Notre code comprendra deux classes, le réseau neuronal et les neurones. On peut commencer avec la classe **Neurone** puis ensuite le **Réseau neuronal**.

1. Créons notre classe **Neurone**, avec une méthode permettant de connecter ce neurone à un autre. Cela permettra de les lier et de récupérer les données du suivant ou du précédent.

2. Lors de l'initialisation de notre Neurone, nous créerons un matrice de poids **W** de taille $entree\_dim \times sortie\_dim$ et une matrice de bias de taille $1 \times sortie\_dim$, où $entree\_dim$ et $sortie\_dim$ seront données en paramètres. Ces matrices pourront être initialisées avec une distribution uniforme entre -1 et 1.

Notre neurone possédera deux autres matrices bien distinctes, une d'entrée et une de sortie.

3. Ainsi, pour faciliter les prochaines exécution, nous pouvons crée une méthode *recupere_sortie_precedente* qui retournera la sortie du neurone précédent. À vous de traiter le cas où le neurone en question est le premier neurone. La matrice de sortie aura la même taille que le biais, cependant la matrice d'entrée sera de taille $1 \times entree\_dim$.

Vous pouvez remarquer qu'en multipliant notre matrice **W** avec la matrice d'entrée, nous retombons sur une matrice de taille $1 \times sortie\_dim$.

4. Cette question est une adaption des questions 2 et 3 de la partie 2, en suivant l'explication précédente, nous pouvons créer une fonction qui fera cette opération en y ajoutant la matrice bias, cela nous donnera la sortie de notre neurone.

Dans les prochaines questions nous allons nous concentrer sur l'apprentisage de notre Réseau neuronal.

5. Dans un premier temps, nous pouvons créer notre classe **ReseauNeuronal** avec comme seule variable notre liste de neurones. Nous pouvons également ajouter une méthode permettant d'ajouter un neurone au système neuronal et de le lier au dernier.

6. Nous pouvons par la suite, implémenter une méthode entrainant notre système neuronal sur une image, elle prendra comme paramètre notre image, le nombre associé à l'image, et la vitesse d'apprentissage α. Au sein de cette fonction, nous donnerons à notre premier neurone notre image comme entrée, puis on fera en sorte que notre image parcourt l'ensemble de notre réseau neuronal.

Une fois que l'image a été analysé par tous nos neurones, il va falloir évaluer le résultat et retourner en arrière.

7. Ainsi, nous pouvons créer une nouvelle variable *entree_delta* dans notre classe **Neurone**. Cette variable correspondra à l'erreur quadratique moyenne du neurone suivant (dans le sens de l'input) La valeur de cette variable pour notre dernier neurone sera l'erreur quadratique moyenne évaluée en sa sortie.

8. À la suite de la fonction du 6, nous pouvons appeler pour chacun des neurones dans le sens inverse la fonction *backward* :

9. Cette fonction *backward* dépendra de la vitesse d'apprentissage α, pour pouvoir bien définir cette fonction, nous devons utiliser la réciproque de la fonction sigmoïdale. Évidemment, nous n'allons pas vous la donner. Allez, on sort ses feuilles et au travail !! Cette fonction permettra également de modifier les paramètres de chacun de nos neurones de manière à ce qu'ils soient plus précis. Pour cela, nous aurons besoin du delta Δ précédemment calculé ainsi que de la sortie $\mathbf{p}$ du neurone, il sera ensuite indispensable de modifier ce delta en le multipliant par la dérivée de la sigmoïde de la sortie du neurone précédent. Nous allons ensuite appliquer les changements suivants à notre neurone :
   - $\boldsymbol{W} \mapsto \boldsymbol{W} - \alpha \cdot \boldsymbol{p} \cdot \Delta$
   - $\boldsymbol{b} \mapsto \boldsymbol{b} - \alpha \cdot \Delta$
   - $\Delta s \mapsto \Delta \cdot \boldsymbol{W}^t$

10. Dans cette avant dernière question de la partie 3, nous allons créer une fonction *train* qui prendra en paramètre les images d'entraînement sous forme de tuple, le nombre génération à passer, la vitesse d'apprentissage et les images de test afin de voir l'avancé de notre réseau neuronal. Il serait très certainement intéressant de reprendre les fonctions que vous avez effectué dans la partie 2.

11. Dans cette dernière question, vous pouvez initialiser votre réseau neuronal, y ajouter des neurones (attention aux dimensions de sortie et d'entrée), et ensuite l'entraîner sur le jeu de donnée que nous vous avons fourni.

Note: Si jamais 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 au contraire, le changement des paramètres dans l'ensemble du réseau neuronal sera moins important, ce qui permettra une meilleure précision.