

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 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 $1 \times l \cdot h$, par exemple avec la librairie `numpy`. On devra aussi transformer les couleurs en une valeur dans $[0, 1]$.
(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 $[0, 1]^{l \cdot h}$. 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 $[0, 1]^{l \cdot h}$.
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 $[0, 1]^{l \cdot h}$.
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 \mathbf{m} , et on va vouloir calculer à partir de celle-ci une **matrice** de taille 1×10 , $\mathbf{p} = (p_0 \ p_1 \ \dots \ p_8 \ p_9)$, 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 $\mathbf{l} \cdot \mathbf{h} \times 10$ 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 $\mathbf{p} = \mathbf{m} \cdot \mathbf{W} + \mathbf{b}$ 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 $\mathbf{p} \mapsto \mathbf{p} - (0 \ \dots \ 0 \ 1 \ 0 \ \dots \ 0)$ 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 \mathbf{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 $\mathbf{m} \cdot \mathbf{W} + \mathbf{b}$)

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 $\mathbf{W} \mapsto \mathbf{W} - \alpha \cdot \mathbf{m}^t \cdot \Delta$

- Le changement que reçoit \mathbf{b} est simplement $\mathbf{b} \mapsto \mathbf{b} - \alpha \cdot \Delta$
Où $\alpha \in [0, 1]$ représente la **vitesse d'apprentissage**, et \mathbf{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 $\mathbf{p} = \sigma(\mathbf{m} \cdot \mathbf{W} + \mathbf{b})$, 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 : Système neuronal séquentiel

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

Notre fichier final comprendra deux classes, le système neuronal et les neurones. Pour éviter toute copie, vous pouvez copier-coller les fonctions suivantes : la sigmoïde et sa dérivée ainsi que les fonction pour récupérer les images d'entraînement, celle qui génère l'erreur quadratique moyenne. Nous allons nous attaquer à la class Neurone puis ensuite au système neuronal.

1. Créons une classe **Neurone**, avec une méthode permettant de connecter ce neurone avec son précédent. Cela permettra de les lier et de récupérer les données du suivant ou du précédent.
2. Lors de l'initialisation de notre Neuron, nous créerons une 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 donnés en paramètres. Ces matrices seront remplies de valeurs aléatoires comprises entre -1 et 1.

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

3. Ainsi, pour faciliter les prochaines exécution, nous pouvons créer 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 bias, 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 (qui est un field de la classe), nous retompons sur une matrice de taille $1 \times sortie_dim$.

4. Cette question est une adaptation de la question 3 de la partie 2, en suivant l'explication précédente, nous pouvons créer une fonction **forward** qui fait cette opération en y ajoutant la matrice bias. Or les valeurs obtenus ne sont pas dans l'intervalle $[0, 1]$. Il va donc falloir appliquer vous savez quelle fonction au résultat précédent, ce sera la donnée de sortie de notre neurone.

Dans les prochaines questions nous allons nous attaquer à l'apprentissage de notre système neuronal

5. Dans un premier temps, nous pouvons créer notre classe avec comme seule variable notre liste de neurones. Nous pouvons également créer 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 entraînant notre système neuronal sur une image, elle prendra comme paramètre un tuple contenant notre image sous forme de matrice 1×784 avec le nombre associé à l'image, et la vitesse d'apprentissage. Au sein de cette fonction, nous donnerons à notre premier neurone notre image comme entrée, puis on appellera forward pour chaque neurone.

Une fois que les données de l'image sont passés ??? dans tous nos neurones, il va falloir que notre système neuronal apprenne.

7. Ainsi, nous pouvons utiliser la fonction d'erreur quadratique moyenne, pour évaluer la prédiction du nombre.

Après avoir ajouté cette fonction à votre code, il nous faut propager l'erreur dans l'ensemble de nos neurones.

8. Ainsi, nous pouvons créer une nouvelle variable *entree_delta* dans notre classe Neurone. La valeur de cette variable pour notre dernier neurone sera l'erreur quadratique moyenne évaluée en la sortie du dernier neurone.

Maintenant que nous avons implémenté le calcul du manque de prédictions, nous devons remonter notre système neuronaux.

10. Cette fonction backward dépendra de la vitesse d'apprentissage α , elle permettra également de modifier les poids 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{d} du neurone précédent (`self.precedent`), il sera ensuite indispensable de modifier ce delta en multipliant avec la dérivée de la sigmoïde de la sortie du neurone précédent. Nous allons ensuite modifier les variables de notre neurone de la manière suivante :

- $\Delta_s \leftarrow \Delta \cdot \mathbf{W}^t$
- $\mathbf{W} \leftarrow \mathbf{W} - \alpha \cdot \mathbf{d} \cdot \Delta$
- $\mathbf{b} \leftarrow \mathbf{b} - \alpha \cdot \Delta$

Ne pas donner un unique image mais plusieurs