Entreposage et fouille de données (STA211)
TME : apprentissage profond

On va travailler avec la base de données image MNIST, constituée d'images de caractères manuscrits (60000 images en apprentissage, 10000 en test).

Voici un bout de code pour récupérer les données :

```
from keras.datasets import mnist
# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

Les travaux pratiques sont structurés de la manière suivante :

- L'exercice 0 consiste à prendre en main et visualiser quelques images de la base MNIST
- Dans la suite, on prendra en main la librairie `Keras` pour utiliser et entraîner des réseaux de neurones profonds.
  1. L'exercice 1 présente un premier réseau de neurone simple (et non profond) : la régression logistique. Sa mise en place et son apprentissage seront très formateurs pour utiliser des modèles plus évolués.
  2. L'exercice 2 propose de mettre en place un perceptron, *i.e.* un réseau de neurone complètement connecté à une couche cachée.
  3. L'exercice 3 propose une extension à des réseaux de neurones convolutifs profonds (ConvNets), modèles qui constituent les réseaux de neurones modernes, et qui sont particulièrement efficaces pour la reconnaissance de données de type signal bas niveau (image, son, *etc*).
- Enfin, on s'intéressera à l'analyse des représentations internes apprises par les modèles d'apprentissage profond. En particulier, on utilisera la méthode t-SNE [3] pour visualiser en 2D les couches cachées des réseaux de neurones et illustrer le phénomène de "manifold untangling".

Exercice 0 Chargement de la base

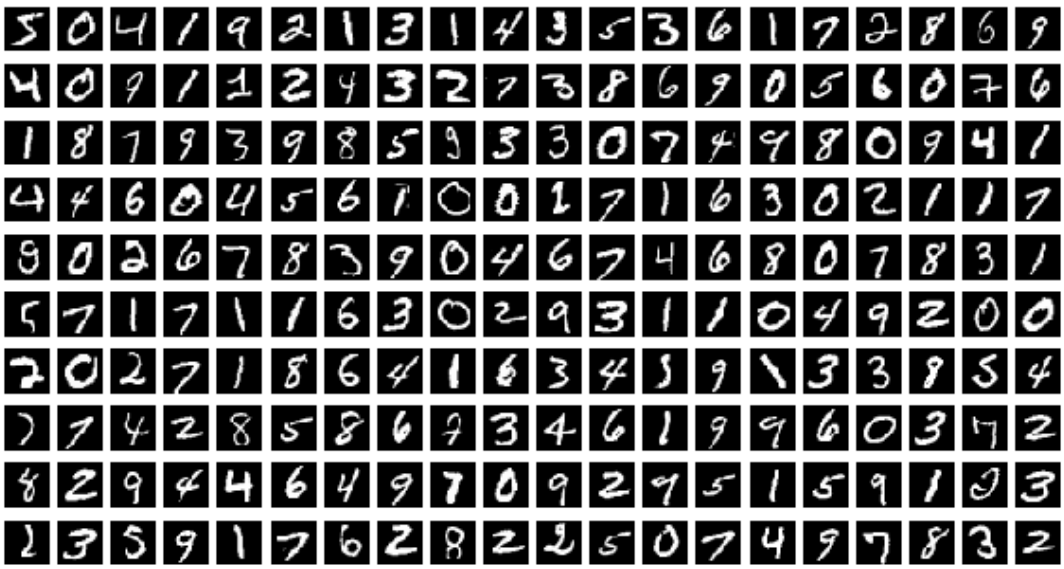
On va commencer par afficher les 200 premières images de la base d'apprentissage.

▷ **Écrire un script `exo0.py` qui va récupérer les données avec le code précédent**

▷ **Compléter `exo0.py` pour permettre l'affichage demandé en utilisant le code suivant :**

```
import matplotlib as mpl
mpl.use('TKAgg')
import matplotlib.pyplot as plt
plt.figure(figsize=(7.195, 3.841), dpi=100)
for i in range(200):
    plt.subplot(10,20,i+1)
    plt.imshow(X_train[i,:].reshape([28,28]), cmap='gray')
    plt.axis('off')
plt.show()
```

Le script `exo0.py` doit produire l'affichage ci-dessous :



► **Quel est l'espace dans lequel se trouvent les images ? Quel est sa taille ?**

I Apprentissage profond (deep learning) avec Keras

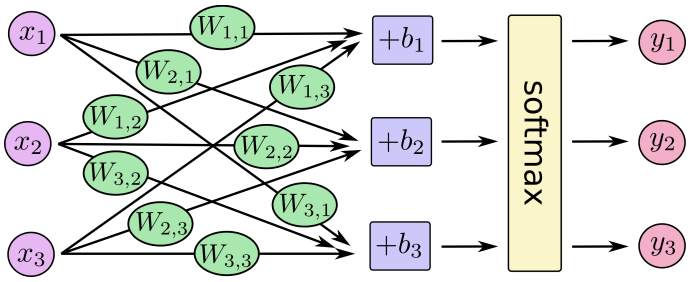
L'objectif est maintenant d'utiliser la librairie `Keras`[1] afin d'entraîner des réseaux de neurones pour reconnaître les catégories ($\{0;1;2;3;4;5;6;7;8;9\}$) des images de chiffres manuscrits de la base MNIST.

Exercice 1 Régression logistique

On va d'abord commencer par créer un modèle de classification linéaire populaire, la régression logistique. Ce modèle correspond à un réseau de neurones à une seule couche, qui va projeter le vecteur d'entrée $\mathbf{x}_i$ pour une image MNIST (taille $28^2 = 784$) avec un vecteur de de paramètres $\mathbf{w_c}$ pour chaque classe (plus un biais b_c). En regroupant l'ensemble des jeux de paramètres $\mathbf{w_c}$ pour les 10 classes dans une matrice $\mathbf{W}$ (taille 10x784), et les biais dans un vecteur $\mathbf{b}$, on obtient un vecteur $\hat{\mathbf{s_i}} = \mathbf{W}\mathbf{x}_i + \mathbf{b}$ de taille 10. Une fonction d'activation de type soft-max sur $\mathbf{s_i}$ permet d'obtenir le vecteur de sortie prédit par le modèle $\hat{\mathbf{y}}_\mathbf{i}$ (de taille 10), qui représente la probabilité *a posteriori* $p(\hat{\mathbf{y}}_\mathbf{i}|\mathbf{x}_i)$ pour chacune des 10 classes:

$$p(\hat{y_{c,i}}|\mathbf{x}_i)) = \frac{e^{\langle \mathbf{x}_i;\mathbf{w_c}\rangle + b_c}}{\sum_{c'=1}^{10} e^{\langle \mathbf{x}_i;\mathbf{w_{c'}}\rangle + b_{c'}}} \tag{1}$$

Le schéma ci-dessous illustre le modèle de régression logistique avec un réseau de neurones.



▶ Quel est le nombre de paramètres du modèle ? Justifier le calcul.

Avec `Keras`, les réseaux de neurones avec une structure de chaîne (réseaux "feedforward"), s'utilisent de la manière suivante:

```
from keras.models import Sequential
model = Sequential()
```

On créé ainsi un réseau de neurones vide. On peut alors ajouter des couches avec la fonction `add`. Par exemple, l'ajout d'une couche de projection linéaire (couche complètement connectée) de taille 10, suivi de l'ajout d'une couche d'activation de type `softmax`, peuvent effectuer de la manière suivante:

```
from keras.layers import Dense, Activation
model.add(Dense(10, input_dim=784, name='fc1'))
model.add(Activation('softmax'))
```

▷ Écire un script `exo1.py` permettant de créer un réseau de neurone pour apprendre un modèle de régression logistique comme expliqué ci-dessus. On peut ensuite visualiser l'architecture du réseau avec la méthode `summary()` du modèle. Vérifier le nombre de paramètres du réseau à apprendre.

[1] https://keras.io/

Afin d'entraîner le réseau de neurones, on va comparer, pour chaque exemple d'apprentissage, la sortie prédite $\hat{\mathbf{y}}_\mathbf{i}$ par le réseau (équation (1)) pour l'image $\mathbf{x}_i$, avec la sortie réelle $\mathbf{y}_\mathbf{i}^*$ issue de la supervision qui correspond à la catégorie de l'image $\mathbf{x}_i$: on utilisera en encodage de type "one-hot" pour $\mathbf{y}_\mathbf{i}^*$, *i.e.*

$$y_{c,i}^* = \begin{cases} 1 & \text{si c correspond à l'indice de la classe de } \mathbf{x}_i \\ 0 & \text{sinon} \end{cases} \tag{2}$$

Pour mesurer l'erreur de prédiction, on utilisera une fonction de coût de type entropie croisée ("cross-entropy") entre $\hat{\mathbf{y}}_\mathbf{i}$ et $\mathbf{y}_\mathbf{i}^*$ [2] :

$$\mathcal{L}(\hat{\mathbf{y}}_\mathbf{i}, \mathbf{y}_\mathbf{i}^*) = -\sum_{c=1}^{10} y_{c,i}^* log(\hat{y}_{c,i}) = -log(\hat{y}_{c^*,i}) \tag{3}$$

où c^* correspond à l'indice de la classe donné par la supervision pour l'image $\mathbf{x}_i$. La fonction de coût, moyennée sur l'ensemble d'apprentissage, peut alors être minimisée par exemple par une technique de descente de gradient.

▶ La fonction de coût de l'Eq. (3) est-elle convexe par rapports aux paramètres $\mathbf{W}$, $\mathbf{b}$ du modèle ? Avec un pas de gradient bien choisi, peut-on assurer la convergence vers le minimum global de la solution ?

Avec Keras, on va compiler le modèle en lui passant un loss (ici l' entropie croisée), une méthode d'optimisation (ici une descente de gradient stochastique, stochastic gradient descent, sgd), et une métrique d'évaluation (ici le taux de bonne prédiction des catégories, accuracy):

```
from keras.optimizers import SGD
learning_rate = 0.5
sgd = SGD(learning_rate)
model.compile(loss='categorical_crossentropy',optimizer=sgd,metrics=['accuracy'])
```

Enfin, l'apprentissage du modèle sur des données d'apprentissage est mis en place avec la méthode `fit` (batch_size correspond au nombre d'exemple utilisé pour estimer le gradient de la fonction de coût, et epochs le nombre d'itération lors de la descente de gradient). À partir du code permettant de récupérer les données de la base MNIST (début de l'énoncé), on transformera les labels afin d'avoir le format de type "one-hot encoding" de l'équation (2):

```
from keras.utils import np_utils
batch_size = 300
nb_epoch = 10
# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
model.fit(X_train, Y_train,batch_size=batch_size, epochs=nb_epoch,verbose=1)
```

On peut ensuite évaluer les performances du modèle dur l'ensemble de test avec la fonction `evaluate`

```
scores = model.evaluate(X_test, Y_test, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[0], scores[0]*100))
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Le premier élément de score renvoie la fonction de coût sur la base de test, le second élément renvoie le taux de bonne détection (accuracy).

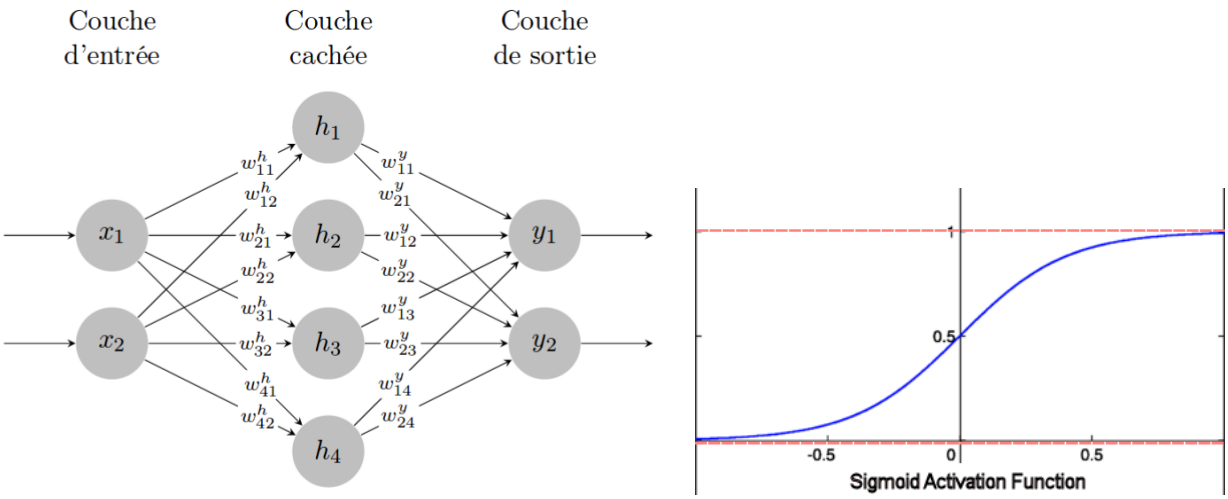
▷ Implémenter l'apprentissage du modèle de régression logistique sur la base de train de la base MNIST.

▷ Évaluer les performances du réseau sur la base de test. **Vous devez obtenir un score de l'ordre de** 92% **pour ce modèle de régression logistique.**

[2] l'entropie croisée est lié à la divergence de Kullback-Leiber, qui mesure une dissimilarité entre distribution de probabilités.

Exercice 2 Perceptron multi-couche

On va maintenant enrichir le modèle de régression logistique en créant une couche de neurones cachée supplémentaire, afin de former un modèle appelé perceptron, cf Figure 1.



a) Architecture du réseau b) Fonction d'activation sigmoïde

Figure 1: Perceptron à une couche cachée. l'architecture générale montrée en a) consiste à augmenter le modèle de régression logistique de l'exercice 4 avec une nouvelle couche de neurones **h**. L'activation des neurones **h** est obtenue en appliquant une projection linéaire complètement connectée depuis l'entrée **x**, suivi d'une étape de non linéarité de type sigmoïde, comme illustrée en b).

- ▶ On va utiliser 100 neurones dans la couche cachée du nouveau réseau. Quel est maintenant le nombre de paramètres du modèle MLP ? Justifier le calcul.
- ▷ Écire un script `exo2.py` qui va enrichir le modèle de régression logistique de l'exercice précédent afin de créer le réseau MLP. **N.B.** : on ajoutera on couche complètement connectée suivie d'une non linéarité de type sigmoid.

Une fois le modèle MLP créer, la façon de l'entraîner va être strictement identique à ce qui a été écrit dans l'exercice 1. En effet, on peut calculer l'erreur (entropie croisée décrite à l'équation 3) pour chaque exemple d'apprentissage à partir de la sortie prédite $\hat{\mathbf{y}}_\mathbf{i}$ et de la supervision $\mathbf{y}^*_\mathbf{i}$. **L'algorithme de rétro-propagation du gradient de cette erreur permet alors de mettre à jour l'ensemble des paramètres du réseau.**

- ▷ Compléter le script `exo2.py` afin d'effectuer l'entraînement du réseau MLP.
- ▶ Avec ce modèle MLP à une couche cachée, la fonction de coût de l'Eq. (3) est-elle convexe par rapports aux paramètres du modèle ? Avec un pas de gradient bien choisi, peut-on assurer la convergence vers le minimum global de la solution ?
- ▷ Évaluer les performances du réseau sur la base de test. **Vous devez obtenir un score de l'ordre de** 98% **pour ce modèle MLP.**
- ▷ Utiliser la fonction `saveModel` pour stocker le résultat de votre modèle appris.

Exercice 3 Réseaux de neurones convolutifs

On propose maintenant d'étendre le perceptron de l'exercice précédent pour mettre en place un réseau de neurones convolutif profond. Les réseaux convolutifs manipulent des images multi-dimensionnelles en entrée (tenseurs). On va donc commencer par reformater les données d'entrée afin que chaque exemple soit de taille $28 \times 28 \times 1$.

```
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
input_shape = (28, 28, 1)
```

Par rapport aux réseaux complètement connectés, les réseaux convolutifs utilisent les briques élémentaires suivantes :

- Des couches de convolution, qui transforment un tenseur d'entrée de taille $n_x \times n_y \times p$ en un tenseur de sortie $n_{x'} \times n_{y'} \times n_H$, où n_H est le nombre de filtres choisi. Par exemple, une couche de convolution pour traiter les images d'entrée de MNIST peut être créée de la manière suivante :

```
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.layers import Conv2D, MaxPooling2D
Conv2D(32,kernel_size=(5, 5),activation='sigmoid',input_shape=(28, 28, 1),padding='same')
```

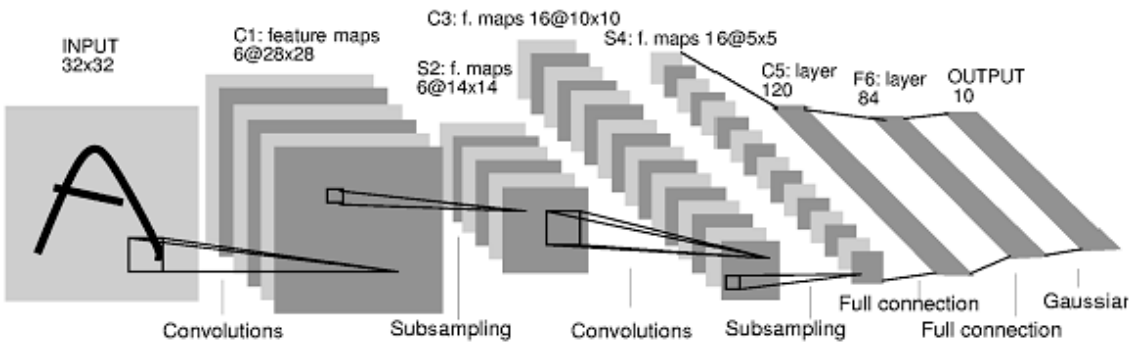
  - 32 est le nombre de filtres.
  - $(5, 5)$ est la taille spatiale de chaque filtre (masque de convolution).
  - padding='same' correspond à garder la même taille spatiale en sortie de la convolution.
  - **N.B.** : on peut directement inclure dans la couche de convolution la non-linéarité en sortie de la convolution, comme illustré ici dans l'exemple avec une fonction d'activation de type `sigmoid`.

- Des couches d'agrégation spatiale (pooling), afin de permettre une invariance aux translations locales. Voici par exemple la manière de déclarer une couche de max-pooling:

```
pool = MaxPooling2D(pool_size=(2, 2))
```

  - $(2, 2)$ est la taille spatiale sur laquelle l'opération d'agrégation est effectuée.
  - **N.B.** : par défaut, le pooling est effectué avec un décalage de 2 neurones, dans l'exemple précédent on obtient donc des cartes de sorties avec des tailles spatiales divisées par deux par rapport à la taille d'entrée.

L'architecture LeNet [1], montrée à la figure 3, constitue un modèle de reconnaissance historique pour la lecture de chiffres manuscrits comme MNIST.



On propose de mettre en place une variante du réseau LeNet, qui va être constituée d'une succession de deux blocs [Conv + Non linéarité + pooling], suivie de deux couches complètement connectées.

Plus précisément, l'architecture sera structurée séquentiellement de la manière suivante :

- Une couche de convolution avec 32 filtres de taille 5×5, suivie d'une non linéarité de type sigmoïde puis d'une couche de max pooling de taille 2×2.
- Une seconde couche de convolution avec 64 filtres de taille 5×5, suivie d'une non linéarité de type sigmoïde puis d'une couche de max pooling de taille 2×2.
- Comme dans le réseau LeNet, on considérera la sortie du second bloc convolutif comme un vecteur, ce que revient à "mettre à plat" les couches convolutives précédentes (`model.add(Flatten())`).
- Une couche complètement connectée de taille 100, suivie d'une non linéarité de type sigmoïde.
- Une couche complètement connectée de taille 10, suivie d'une non linéarité de type softmax.

▷ **Écrire un script `exo3.py` afin de définir l'architecture du réseau convolutif précédemment indiquée et d'effectuer l'entraînement de ce modèle.**

▷ **Évaluer les performances du réseau sur la base de test. Vous devez obtenir un score de l'ordre de 99% pour ce modèle MLP.**

▷ **Utiliser la fonction `saveModel` pour stocker le résultat de votre modèle appris.**

II Visualisation de l'effet de "manifold untangling"

On va maintenant s'intéresser à visualiser les représentations internes apprises par les réseaux de neurones.

Exercice 4 t-Distributed Stochastic Neighbor Embedding (t-SNE)

La méthode t-Distributed Stochastic Neighbor Embedding (t-SNE) [3] est une réduction de dimension non linéaire, dont l'objectif est d'assurer que des points proches dans l'espace de départ aient des position proches dans l'espace (2D) projeté.

On va tour d'abord appliquer la méthode t-SNE sur les données brutes **de la base de test de MNIST** en utilisant la classe `TSNE` du module `sklearn.manifold`[3].

▷ **Créer un script `exo4.py` dont l'objectif va être d'effectuer une réduction de dimension en 2D des données de la base de test de MNIST en utilisant la méthode t-SNE.**

- ▷ Créer une instance de type `TSNE`.
  N.B. : on choisira 2 composantes et les paramètres suivants : `init='pca'` (réduire la dimension préalablement avec une ACP), `perplexity=30` (lié au nombre de voisins dans le calcul des distances), `verbose=2` (pour l'affichage lors de l'apprentissage).
- ▷ appliquer la transformation pour obtenir les données projetées en 2D (fonction `fit_transform`).
  N.B. : essayer tout d'abord avec un sous ensemble de la base (*e.g.* 1000 exemples) pour tester l'algorithme, l'apprentissage avec l'ensemble de la base de test pouvant être long.

[3] http://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html

Exercice 5 Visualisation et métrique de séparation des classes

On va maintenant compléter le script `exo4.py` précédent afin de visualiser l'ensemble des points projetés en 2D, et de définir des critères pour analyser la séparabilité des classes dans l'espace projeté.

▷ **Calcul de l'enveloppe convexe des points projetés pour chacune des classe classe.** On utilisera pour cela la la classe `ConvexHull` du module `scipy.spatial`[4]. On pourra donc utiliser le code suivant pour calculer les enveloppes convexes pour les 10 classes :

```
def convexHulls(points, labels):
    # computing convex hulls for a set of points with asscoiated labels
    convex_hulls = []
    for i in range(10):
        convex_hulls.append(ConvexHull(points[labels==i,:]))
    return convex_hulls
# Function Call
convex_hulls = convexHulls(xpca, labels)
```

▷ **Calcul de l'ellipse de meilleure approximation des points.** On utilisera pour cela la la classe `GaussianMixture` du module `sklearn.mixture`[5]. On pourra donc utiliser le code suivant pour calculer les ellipses pour les 10 classes :

```
def best_ellipses(points, labels):
    # computing best fiiting ellipse for a set of points with asscoiated labels
    gaussians = []
    for i in range(10):
        gaussians.append(GaussianMixture(n_components=1, covariance_type='full').fit(points[labels==i, :]))
    return gaussians
# Function Call
ellipses = best_ellipses(xpca, labels)
```

▷ **Métrique de séparation des classes. Utiliser la fonction `neighboring_hit(points, labels)` pour calculer le le "Neighborhood Hit" (NH) [2].** Pour chaque point, la métrique NH consiste à calculer, pour les k plus proches voisins (`k-nn`) de ce point, le taux des voisins qui sont de la même classe que le point considéré. La métrique NH est ensuite moyennée sur l'ensemble de la base.

▷ **Affichage :** Utiliser la fonction `Visualization(points2D, labels, convex_hulls, ellipses, name, nh)` fournie pour effectuer l'affichage des points et des labels, des enveloppes convexes et des ellipses de meilleure approximation de chaque classe. Un exemple de résultat attendu est montré à la Figure 2a).

▶ **Interprétation des résultats :** discuter la différence entre les critères (enveloppe convexe, ellipse, NH) et leur impact en fonction de l'application finale (visualisation, classification linéaire ou non linéaire, *etc*).

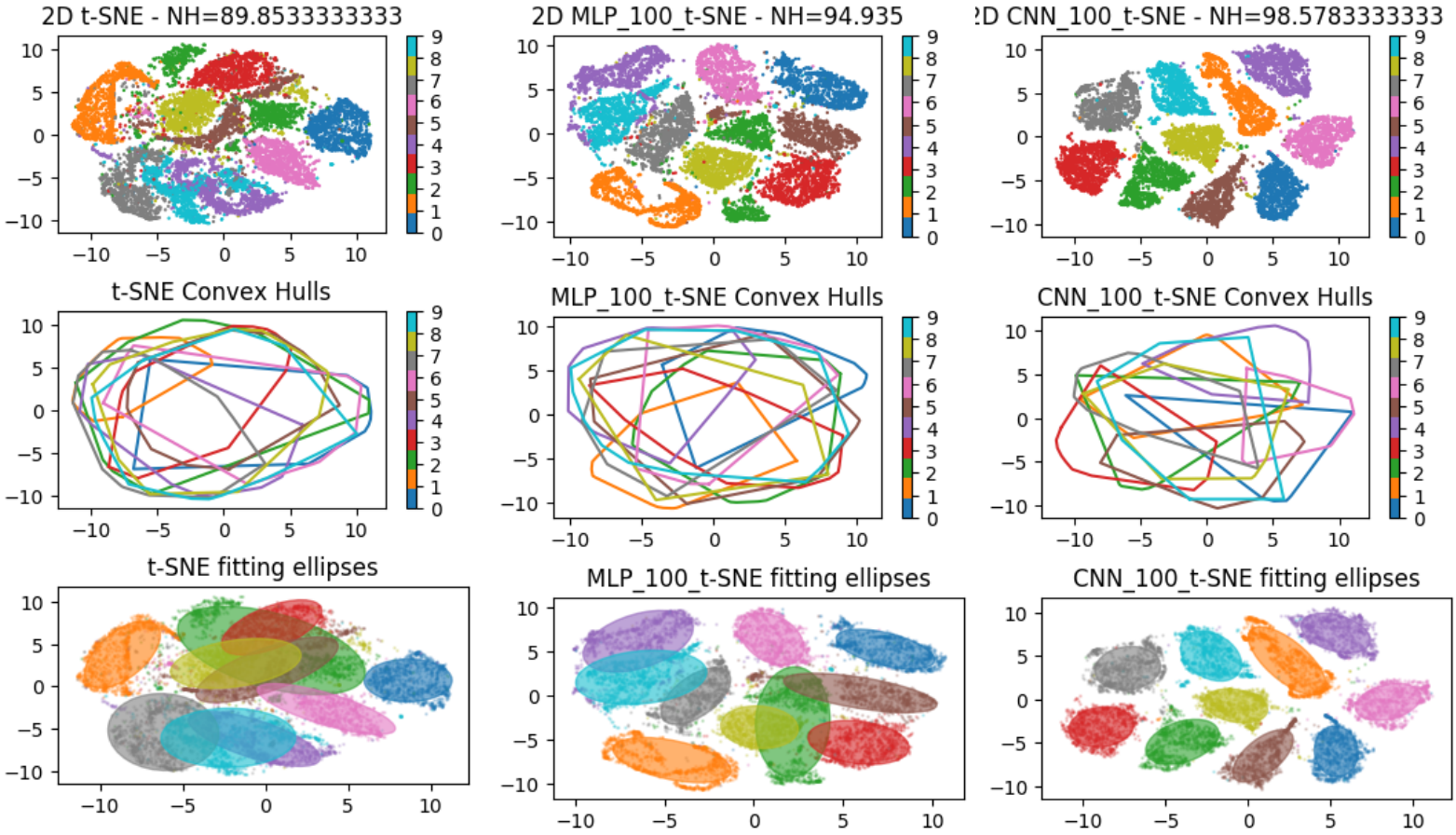
[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.ConvexHull.html
[5] http://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html#sklearn.mixture.GaussianMixture

Exercice 6 MLP et visualisation de la couche cachée

On va maintenant chercher à utiliser la méthode de visualisation t-SNE mise en place dans l'exercice 4 pour visualiser les représentations internes encodées dans la couche caché du perceptron de l'exercice 2 et du réseau convolutif de l'exercice 3.

On décrit dans la suite la mise en place de la visualisation pour le perceptron, le principe est similaire pour le réseau convolutif.

- ▷ Commencer par charger le modèle préalablement appris à exercice 3. On pourra vérifier l'architecture du modèle chargé avec la méthode `summary()`.
  - On pourra également évaluer les performances du modèle chargé sur la base de test de MNIST pour vérifier son comportement.
    N.B. : il faudra avoir compilé le modèle au préalable, comme effectué à exercice 3.
- ▷ On veut maintenant extraire la couche cachée (donc un vecteur de dimension 100) pour chacune des images de la base de test. Pour cela, on va utiliser la méthode `model.pop()` (permettant de supprimer la couche au sommet du modèle) deux fois (on supprime la couche d'activation softmax et la couche complètement connectée). Ensuite on peut appliquer la méthode `model.predict(X_test)` sur l'ensemble des données de test.
- ▷ Une fois le vecteur latent extrait pour toutes les images de la base de test, on va utiliser la méthode de visualisation t-Distributed Stochastic Neighbor Embedding (t-SNE) de l'exercice 4 pour visualiser en 2D les représentations internes du réseau. On visualisera également les différents critères vus dans l'exercice 5 pour interpréter la séparation des classes dans l'espace latent, en particulier les enveloppes convexes, l'ellipse de meilleure approximation pour chaque classe et le critère "Neighborhood Hit" (NH).
- ▶ Interprétation des résultats : que peut-on conclure sur la capacité du réseau de neurones à "détordre" (disuntangling) la variété des données ? Justifier.
  Un exemple de résultat attendu pour le MLP de l'exercice 5 est montré à la figure 2b), et pour le réseau convolutif de l'exercice 3 à la figure 2c).



a) Espace d'entrée b) MLP de l'exercice 3 c) Réseau convolutif de l'exercice 4

Figure 2: Visualisation de l'effet de manifold disuntangling permis par les réseaux de neurones. En a), on montre la visualisation des points dans l'espace de départ. en b), le résultat est obtenu pour le perceptron à un couche caché (taille 100) l'exercice 2, et en c) pour le réseau de neurones convolutif avec deux couches de convolution et une couche complètement connectée de l'exercice 3.

References

[1] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[2] Fernando Vieira Paulovich, Luis Gustavo Nonato, Rosane Minghim, and Haim Levkowitz. Least square projection: A fast high-precision multidimensional projection technique and its application to document mapping. *IEEE Trans. Vis. Comput. Graph.*, 14(3):564–575, 2008.

[3] Laurens van der Maaten and Geoffrey E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008.