TP 2 - Deep Learning avec Keras et Manifold Untangling

In [1]:
```
nom="jai"
prenom="ilyass"
```

In [2]:
```
from keras.models import Sequential
model = Sequential()
```

Exercice 1 : Régression Logistique avec `Keras`

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 s'effectuer de la manière suivante:

In [4]:
```
from keras.layers import Dense, Activation
model.add(Dense(10,  input_dim=784, name='fc1'))
model.add(Activation('softmax'))
```

```
C:\Users\jaimo\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [5]:
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
fc1 (Dense)	(None, 10)	7,850
activation (Activation)	(None, 10)	0

Total params: 7,850 (30.66 KB)

Trainable params: 7,850 (30.66 KB)

Non-trainable params: 0 (0.00 B)

- Structure du modèle
  - entrée : Un vecteur de dimension 784
  - couche dense : 10 neurones, appliquant une transformation linéaire.
  - fonction d'activation : Softmax pour produire des probabilités pour 10 classes.
  - nombre total de paramètres entraînables : 7,850.
- On retrouve le modèle à 10 classes du TP1.

In [6]:
```
from keras.optimizers import SGD
learning_rate = 0.1
sgd = SGD(learning_rate)
model.compile(loss='categorical_crossentropy',optimizer=sgd,metrics=['accuracy']
```

In [7]:
```
from keras.datasets import mnist
(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')
```

```
60000 train samples
10000 test samples
```

Enfin, l’apprentissage du modèle sur des données d’apprentissage est mis en place avec la méthode `fit` :

In [8]:
```
from keras.utils import to_categorical
batch_size = 100
nb_epoch = 20
# convert class vectors to binary class matrices
Y_train = to_categorical(y_train, 10)
Y_test = to_categorical(y_test, 10)
```

In [9]:
```
model.fit(X_train, Y_train,batch_size=batch_size, epochs=nb_epoch,verbose=1)
```

```
Epoch 1/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8031 - loss: 0.7805
Epoch 2/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9000 - loss: 0.3630
Epoch 3/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9059 - loss: 0.3369
Epoch 4/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9105 - loss: 0.3225
Epoch 5/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9153 - loss: 0.3041
Epoch 6/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9142 - loss: 0.3055
Epoch 7/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9176 - loss: 0.2955
Epoch 8/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9187 - loss: 0.2924
Epoch 9/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9190 - loss: 0.2877
Epoch 10/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9216 - loss: 0.2810
Epoch 11/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9216 - loss: 0.2872
Epoch 12/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9216 - loss: 0.2776
Epoch 13/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9217 - loss: 0.2781
Epoch 14/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9214 - loss: 0.2819
Epoch 15/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9237 - loss: 0.2704
Epoch 16/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9238 - loss: 0.2756
Epoch 17/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9236 - loss: 0.2738
Epoch 18/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9249 - loss: 0.2686
Epoch 19/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9265 - loss: 0.2676
Epoch 20/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9260 - loss: 0.2700
```

Out[9]: <keras.src.callbacks.history.History at 0x224db412850>

In [10]:
```
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))
```

```
loss: 27.20%
compile_metrics: 92.28%
```

- les mêmes performances avec les deux méthodes (logique!!)

Exercice 2 : Perceptron avec Keras

On va maintenant enrichir le modèle de régression logistique en insérant une couche de neurones cachés complètement connectée (suivie d’une fonction d’activation non linéaire de type sigmoïde) entre la couche d'entrée et la couche de sortie. On va ainsi obtenir un réseau de neurones à une couche cachée, le Perceptron (cf. TP2).

La première couche de ce réseau peut être obtenue de la manière suivante en `Keras` :

- Sur un réseau séquentiel vide, on va ajouter la méthode `add` pour insérer une couche cachée (de dimension 100):

In [11]:
```
model=Sequential()
model.add(Dense(100,  input_dim=784, name='fc1'))
```

- La non-linéarité de type sigmoïde sera obtenue de la manière suivante :

In [12]:
```
model.add(Activation('sigmoid'))
```

Question : Quel est maintenant le nombre de paramètres du modèle MLP? Justifier le calcul et le vérifier avec la méthode summary().

On reprendra le même raisonnement du TP1 : Entre la couche d'entrée et la couche cachée, on a Nb_param = card(Wi) + card(bi) = Nb_input ⋆ Nb_neurones_cachs + Nb_neurones_cachs = 784 ⋆ 100 + 100 = 78500, et pour la prochaine couche : Nb_param = card(Wi) + card(bi) = Nb_neurones_cachs ⋆ Nb_output + Nb_output = 100⋆10+10 = 1010 et donc on a en tout 79510 paramètres.

In [13]:
```
model.add(Dense(10, name='fc2'))
model.add(Activation('softmax'))
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
fc1 (Dense)	(None, 100)	78,500
activation_1 (Activation)	(None, 100)	0
fc2 (Dense)	(None, 10)	1,010
activation_2 (Activation)	(None, 10)	0

Total params: 79,510 (310.59 KB)

Trainable params: 79,510 (310.59 KB)

Non-trainable params: 0 (0.00 B)

VOILAA !! 79,510 = 79510

In [14]:
```
learning_rate = 0.1
nb_epoch = 100
sgd = SGD(learning_rate)
```

```
model.compile(loss='categorical_crossentropy',optimizer=sgd,metrics=['accuracy']
model.fit(X_train, Y_train,batch_size=batch_size, epochs=nb_epoch,verbose=1)
```

```
Epoch 1/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - accuracy: 0.6931 - loss: 1.3186
Epoch 2/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8867 - loss: 0.4292
Epoch 3/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9014 - loss: 0.3558
Epoch 4/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9094 - loss: 0.3191
Epoch 5/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9161 - loss: 0.2946
Epoch 6/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9189 - loss: 0.2813
Epoch 7/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9230 - loss: 0.2679
Epoch 8/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9288 - loss: 0.2452
Epoch 9/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9325 - loss: 0.2382
Epoch 10/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9334 - loss: 0.2280
Epoch 11/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9382 - loss: 0.2163
Epoch 12/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9392 - loss: 0.2153
Epoch 13/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9416 - loss: 0.2077
Epoch 14/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9432 - loss: 0.1951
Epoch 15/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9464 - loss: 0.1887
Epoch 16/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9464 - loss: 0.1837
Epoch 17/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9497 - loss: 0.1764
Epoch 18/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9505 - loss: 0.1729
Epoch 19/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9515 - loss: 0.1695
Epoch 20/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9545 - loss: 0.1579
Epoch 21/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9550 - loss: 0.1561
Epoch 22/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9571 - loss: 0.1505
Epoch 23/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9578 - loss: 0.1482
Epoch 24/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9572 - loss: 0.1480
Epoch 25/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9598 - loss: 0.1409
Epoch 26/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9607 - loss: 0.1381
Epoch 27/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9612 - loss: 0.1353
Epoch 28/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9616 - loss: 0.1370
Epoch 29/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9632 - loss: 0.1284
Epoch 30/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9636 - loss: 0.1264
```

```
Epoch 31/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9642 - loss: 0.1257
Epoch 32/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9669 - loss: 0.1184
Epoch 33/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9659 - loss: 0.1183
Epoch 34/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9668 - loss: 0.1144
Epoch 35/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9685 - loss: 0.1124
Epoch 36/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9685 - loss: 0.1104
Epoch 37/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9704 - loss: 0.1066
Epoch 38/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9705 - loss: 0.1033
Epoch 39/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9716 - loss: 0.1009
Epoch 40/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9714 - loss: 0.1004
Epoch 41/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9717 - loss: 0.1009
Epoch 42/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9716 - loss: 0.0987
Epoch 43/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9730 - loss: 0.0959
Epoch 44/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9744 - loss: 0.0934
Epoch 45/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9734 - loss: 0.0934
Epoch 46/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9748 - loss: 0.0904
Epoch 47/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9747 - loss: 0.0909
Epoch 48/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9752 - loss: 0.0905
Epoch 49/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9749 - loss: 0.0872
Epoch 50/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9772 - loss: 0.0824
Epoch 51/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9769 - loss: 0.0829
Epoch 52/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9777 - loss: 0.0806
Epoch 53/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9774 - loss: 0.0795
Epoch 54/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9771 - loss: 0.0801
Epoch 55/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9784 - loss: 0.0786
Epoch 56/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9783 - loss: 0.0785
Epoch 57/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9783 - loss: 0.0756
Epoch 58/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9798 - loss: 0.0759
Epoch 59/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9786 - loss: 0.0757
Epoch 60/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9801 - loss: 0.0713
```

```
Epoch 61/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9805 - loss: 0.0714
Epoch 62/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9815 - loss: 0.0693
Epoch 63/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9822 - loss: 0.0685
Epoch 64/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9812 - loss: 0.0699
Epoch 65/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9822 - loss: 0.0681
Epoch 66/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9818 - loss: 0.0680
Epoch 67/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9836 - loss: 0.0618
Epoch 68/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9826 - loss: 0.0653
Epoch 69/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9841 - loss: 0.0613
Epoch 70/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9835 - loss: 0.0631
Epoch 71/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9839 - loss: 0.0606
Epoch 72/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9842 - loss: 0.0594
Epoch 73/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9839 - loss: 0.0619
Epoch 74/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9848 - loss: 0.0584
Epoch 75/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9854 - loss: 0.0568
Epoch 76/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9850 - loss: 0.0593
Epoch 77/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9857 - loss: 0.0547
Epoch 78/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9849 - loss: 0.0559
Epoch 79/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9860 - loss: 0.0545
Epoch 80/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9853 - loss: 0.0562
Epoch 81/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9855 - loss: 0.0553
Epoch 82/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9859 - loss: 0.0541
Epoch 83/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9864 - loss: 0.0526
Epoch 84/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9865 - loss: 0.0521
Epoch 85/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9873 - loss: 0.0509
Epoch 86/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9865 - loss: 0.0507
Epoch 87/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9870 - loss: 0.0515
Epoch 88/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9869 - loss: 0.0504
Epoch 89/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9878 - loss: 0.0481
Epoch 90/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9880 - loss: 0.0478
```

```
Epoch 91/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9880 - loss: 0.0473
Epoch 92/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9884 - loss: 0.0474
Epoch 93/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9878 - loss: 0.0478
Epoch 94/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9877 - loss: 0.0465
Epoch 95/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9895 - loss: 0.0444
Epoch 96/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9895 - loss: 0.0438
Epoch 97/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9885 - loss: 0.0460
Epoch 98/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9896 - loss: 0.0444
Epoch 99/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9897 - loss: 0.0434
Epoch 100/100
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9897 - loss: 0.0428
```

Out[14]:
```
<keras.src.callbacks.history.History at 0x224db8d2a10>
```

In [15]:
```python
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))
```

```
loss: 7.77%
compile_metrics: 97.54%
```

Comme indiqué dans la documentation de Keras "https://keras.io/api/layers/initializers/", cette bibliothèque utilise par défaut l'initialisation de Xavier (Glorot Uniform). Ainsi, si nous comparons cela avec notre modèle du TP1 utilisant également l'initialisation Xavier, nous obtenons les mêmes performances.

- On pourra utiliser la méthode suivante pour sauvegarder le modèle appris :

In [28]:
```python
def saveModel(model, savename):
    # Sauvegarder l'architecture et les poids du modèle
    model.save(savename + ".h5")
    print(f"Modèle complet sauvegardé sous le nom {savename}.h5")

    # Sauvegarder uniquement les poids si nécessaire
    model.save_weights(savename + ".weights.h5")
    print(f"Poids sauvegardés sous le nom {savename}.weights.h5")
```

In [29]:
```python
saveModel(model, "MLP")
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `ker
as.saving.save_model(model)`. This file format is considered legacy. We recommend
using instead the native Keras format, e.g. `model.save('my_model.keras')` or `ke
ras.saving.save_model(model, 'my_model.keras')`.
Modèle complet sauvegardé sous le nom MLP.h5
Poids sauvegardés sous le nom MLP.weights.h5
```

In [32]:
```python
from keras.models import model_from_yaml
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[32], line 1
----> 1 from keras.models import model_from_yaml

ImportError: cannot import name 'model_from_yaml' from 'keras.models' (C:\Users\jaimo\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\keras\api\models\__init__.py)
```

je pense la méthode to_yaml a été supprimée dans les versions récentes de Keras "https://keras.io/guides/serialization_and_saving/"

Exercice 3 : Réseaux de neurones convolutifs avec Keras

Écrire un script pour mettre en place un ConvNet.

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 $28 \times 28 \times 1$.

In [34]:
```
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 :

1. 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 :

In [35]:
```
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.layers import Conv2D, MaxPooling2D
conv1 = Conv2D(32,kernel_size=(5, 5),activation='relu',input_shape=(28, 28, 1),p
```

```
C:\Users\jaimo\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [36]:
```
pool1 = MaxPooling2D(pool_size=(2, 2))
```

Compléter le script pour mettre en place un ConvNet à l'architecture suivante, proche du modèle historique LeNet5

In [37]:
```
model = Sequential()
model.add(Conv2D(16,kernel_size=(5, 5),activation='relu',input_shape=(28, 28, 1)
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
model.add(Conv2D(32,kernel_size=(5, 5),activation='relu',padding='valid'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(100, name='fc1'))
model.add(Activation('sigmoid'))
model.add(Dense(10, name='fc2'))
model.add(Activation('softmax'))
model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 24, 24, 16)	416
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 16)	0
conv2d_2 (Conv2D)	(None, 8, 8, 32)	12,832
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 32)	0
flatten (Flatten)	(None, 512)	0
fc1 (Dense)	(None, 100)	51,300
activation_3 (Activation)	(None, 100)	0
fc2 (Dense)	(None, 10)	1,010
activation_4 (Activation)	(None, 10)	0

Total params: 65,558 (256.09 KB)

Trainable params: 65,558 (256.09 KB)

Non-trainable params: 0 (0.00 B)

In [38]:
```
learning_rate = 0.1
nb_epoch = 10
sgd = SGD(learning_rate)
model.compile(loss='categorical_crossentropy',optimizer=sgd,metrics=['accuracy']
model.fit(X_train, Y_train,batch_size=batch_size, epochs=nb_epoch,verbose=1)
```

```
Epoch 1/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 7s 9ms/step - accuracy: 0.6634 - loss: 1.0735
Epoch 2/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - accuracy: 0.9589 - loss: 0.1474
Epoch 3/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 9s 14ms/step - accuracy: 0.9712 - loss: 0.0978
Epoch 4/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 8s 13ms/step - accuracy: 0.9771 - loss: 0.0766
Epoch 5/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 8s 13ms/step - accuracy: 0.9819 - loss: 0.0647
Epoch 6/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.9846 - loss: 0.0547
Epoch 7/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 9s 15ms/step - accuracy: 0.9862 - loss: 0.0489
Epoch 8/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 7s 12ms/step - accuracy: 0.9882 - loss: 0.0431
Epoch 9/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.9897 - loss: 0.0392
Epoch 10/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 6s 10ms/step - accuracy: 0.9899 - loss: 0.0360
```

Out[38]: `<keras.src.callbacks.history.History at 0x224ea011d10>`

In [39]:
```
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))
```

```
loss: 4.35%
compile_metrics: 98.60%
```

In [40]:
```
saveModel(model, "CNN")
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `ker
as.saving.save_model(model)`. This file format is considered legacy. We recommend
using instead the native Keras format, e.g. `model.save('my_model.keras')` or `ke
ras.saving.save_model(model, 'my_model.keras')`.
Modèle complet sauvegardé sous le nom CNN.h5
Poids sauvegardés sous le nom CNN.weights.h5
```

Quelle est le temps d'une époque avec ce modèle convolutif ?

du 9 a 15 ms

- L'utilisation d'un CNN pour traiter les données de test a permis d'obtenir les meilleures performances comparé aux autres architectures. Malgré un nombre de paramètres inférieur (65558, donné par summary()) à celui d'un réseau MLP entièrement connecté (79510), le CNN est capable de sélectionner efficacement les caractéristiques pertinentes et de les coder de manière autonome lors de son apprentissage.
- Le temps d'une époque est plus lent pour le CNN, mais il nécessite bien moins d'époques (<10) pour converger, par rapport au MLP (>60).
- Je n'ai pas testé sur un GPU, mais cela devrait être beaucoup mieux en raison de ses capacités de traitement parallèle.

Exercice 4 : Visualisation avec t-SNE

On va maintenant illustrer la capacité des réseaux de neurones profonds à apprendre des représentations internes capables de résoudre le problème connu sous le nom de « manifold untangling » en neuroscience, c'est à dire de séparer les exemples des différentes classes dans l'espace de représentations appris.

Pour cela, on va utiliser des outils de visualisation qui vont vont permettre de représenter chaque donnée (par exemple une image de la base MNIST) par un point dans l'espace 2D. Ces même outils vont permettre de projeter en 2D les représentations internes des réseaux de neurones, ce qui va permettre d'analyser la séparabilité des points et des classes dans l'espace d'entrée et dans les espaces de représentions appris par les modèles.

On aura besoin des modules suivants qu'on pourra importer en début de script :

In [41]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
from scipy.spatial import ConvexHull
from sklearn.mixture import GaussianMixture
from scipy import linalg
from sklearn.neighbors import NearestNeighbors
from sklearn.manifold import TSNE
```

Créer un script `exo1.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.

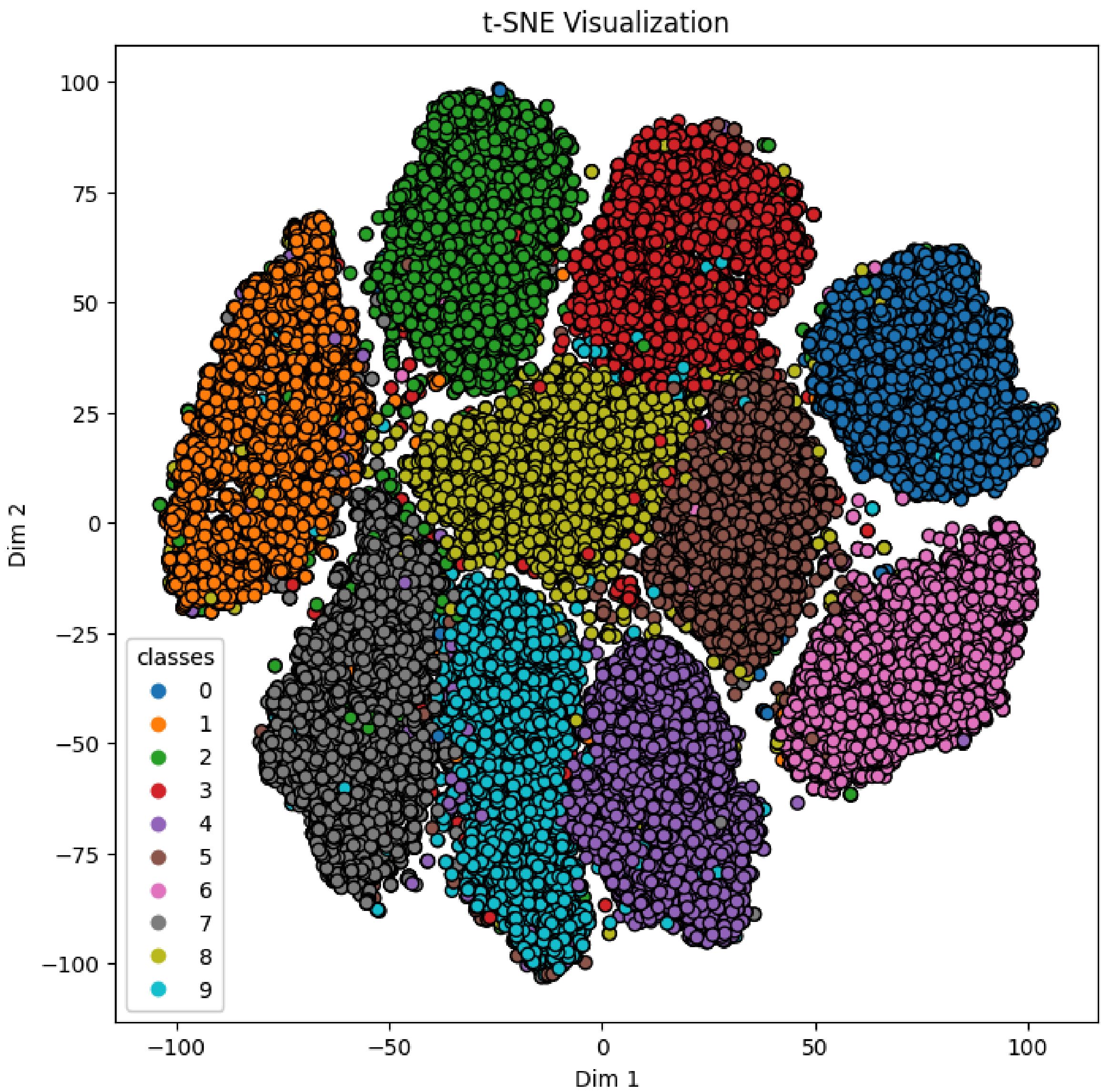
In [42]:
```
from keras.datasets import mnist

(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
```

In [43]:
```
# Calcul TSNE

tsne = TSNE(n_components=2, init='pca', perplexity=30, verbose=2)
X_tsne = tsne.fit_transform(X_train)
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 60000 samples in 0.037s...
[t-SNE] Computed neighbors for 60000 samples in 78.969s...
[t-SNE] Computed conditional probabilities for sample 1000 / 60000
[t-SNE] Computed conditional probabilities for sample 2000 / 60000
[t-SNE] Computed conditional probabilities for sample 3000 / 60000
[t-SNE] Computed conditional probabilities for sample 4000 / 60000
[t-SNE] Computed conditional probabilities for sample 5000 / 60000
[t-SNE] Computed conditional probabilities for sample 6000 / 60000
[t-SNE] Computed conditional probabilities for sample 7000 / 60000
[t-SNE] Computed conditional probabilities for sample 8000 / 60000
[t-SNE] Computed conditional probabilities for sample 9000 / 60000
[t-SNE] Computed conditional probabilities for sample 10000 / 60000
[t-SNE] Computed conditional probabilities for sample 11000 / 60000
[t-SNE] Computed conditional probabilities for sample 12000 / 60000
[t-SNE] Computed conditional probabilities for sample 13000 / 60000
[t-SNE] Computed conditional probabilities for sample 14000 / 60000
[t-SNE] Computed conditional probabilities for sample 15000 / 60000
[t-SNE] Computed conditional probabilities for sample 16000 / 60000
[t-SNE] Computed conditional probabilities for sample 17000 / 60000
[t-SNE] Computed conditional probabilities for sample 18000 / 60000
[t-SNE] Computed conditional probabilities for sample 19000 / 60000
[t-SNE] Computed conditional probabilities for sample 20000 / 60000
[t-SNE] Computed conditional probabilities for sample 21000 / 60000
[t-SNE] Computed conditional probabilities for sample 22000 / 60000
[t-SNE] Computed conditional probabilities for sample 23000 / 60000
[t-SNE] Computed conditional probabilities for sample 24000 / 60000
[t-SNE] Computed conditional probabilities for sample 25000 / 60000
[t-SNE] Computed conditional probabilities for sample 26000 / 60000
[t-SNE] Computed conditional probabilities for sample 27000 / 60000
[t-SNE] Computed conditional probabilities for sample 28000 / 60000
[t-SNE] Computed conditional probabilities for sample 29000 / 60000
[t-SNE] Computed conditional probabilities for sample 30000 / 60000
[t-SNE] Computed conditional probabilities for sample 31000 / 60000
[t-SNE] Computed conditional probabilities for sample 32000 / 60000
[t-SNE] Computed conditional probabilities for sample 33000 / 60000
[t-SNE] Computed conditional probabilities for sample 34000 / 60000
[t-SNE] Computed conditional probabilities for sample 35000 / 60000
[t-SNE] Computed conditional probabilities for sample 36000 / 60000
[t-SNE] Computed conditional probabilities for sample 37000 / 60000
[t-SNE] Computed conditional probabilities for sample 38000 / 60000
[t-SNE] Computed conditional probabilities for sample 39000 / 60000
[t-SNE] Computed conditional probabilities for sample 40000 / 60000
[t-SNE] Computed conditional probabilities for sample 41000 / 60000
[t-SNE] Computed conditional probabilities for sample 42000 / 60000
[t-SNE] Computed conditional probabilities for sample 43000 / 60000
[t-SNE] Computed conditional probabilities for sample 44000 / 60000
[t-SNE] Computed conditional probabilities for sample 45000 / 60000
[t-SNE] Computed conditional probabilities for sample 46000 / 60000
[t-SNE] Computed conditional probabilities for sample 47000 / 60000
[t-SNE] Computed conditional probabilities for sample 48000 / 60000
[t-SNE] Computed conditional probabilities for sample 49000 / 60000
[t-SNE] Computed conditional probabilities for sample 50000 / 60000
[t-SNE] Computed conditional probabilities for sample 51000 / 60000
[t-SNE] Computed conditional probabilities for sample 52000 / 60000
[t-SNE] Computed conditional probabilities for sample 53000 / 60000
[t-SNE] Computed conditional probabilities for sample 54000 / 60000
[t-SNE] Computed conditional probabilities for sample 55000 / 60000
[t-SNE] Computed conditional probabilities for sample 56000 / 60000
[t-SNE] Computed conditional probabilities for sample 57000 / 60000
```

```
[t-SNE] Computed conditional probabilities for sample 58000 / 60000
[t-SNE] Computed conditional probabilities for sample 59000 / 60000
[t-SNE] Computed conditional probabilities for sample 60000 / 60000
[t-SNE] Mean sigma: 1.644729
[t-SNE] Computed conditional probabilities in 1.593s
[t-SNE] Iteration 50: error = 108.4333344, gradient norm = 0.0066326 (50 iteratio
ns in 30.309s)
[t-SNE] Iteration 100: error = 101.7633514, gradient norm = 0.0019024 (50 iterati
ons in 38.962s)
[t-SNE] Iteration 150: error = 100.1725769, gradient norm = 0.0010414 (50 iterati
ons in 28.814s)
[t-SNE] Iteration 200: error = 99.4438553, gradient norm = 0.0006378 (50 iteratio
ns in 43.404s)
[t-SNE] Iteration 250: error = 99.0592346, gradient norm = 0.0004818 (50 iteratio
ns in 35.030s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 99.059235
[t-SNE] Iteration 300: error = 4.3485985, gradient norm = 0.0055195 (50 iteration
s in 30.677s)
[t-SNE] Iteration 350: error = 3.7609806, gradient norm = 0.0048841 (50 iteration
s in 28.258s)
[t-SNE] Iteration 400: error = 3.4756989, gradient norm = 0.0045309 (50 iteration
s in 28.015s)
[t-SNE] Iteration 450: error = 3.2954876, gradient norm = 0.0043271 (50 iteration
s in 27.467s)
[t-SNE] Iteration 500: error = 3.1678538, gradient norm = 0.0040859 (50 iteration
s in 26.711s)
[t-SNE] Iteration 550: error = 3.0721292, gradient norm = 0.0038483 (50 iteration
s in 27.294s)
[t-SNE] Iteration 600: error = 2.9979355, gradient norm = 0.0035970 (50 iteration
s in 26.694s)
[t-SNE] Iteration 650: error = 2.9389958, gradient norm = 0.0033532 (50 iteration
s in 27.109s)
[t-SNE] Iteration 700: error = 2.8910339, gradient norm = 0.0031298 (50 iteration
s in 25.832s)
[t-SNE] Iteration 750: error = 2.8512802, gradient norm = 0.0029393 (50 iteration
s in 26.750s)
[t-SNE] Iteration 800: error = 2.8177412, gradient norm = 0.0027683 (50 iteration
s in 24.767s)
[t-SNE] Iteration 850: error = 2.7888935, gradient norm = 0.0026146 (50 iteration
s in 26.065s)
[t-SNE] Iteration 900: error = 2.7638860, gradient norm = 0.0024690 (50 iteration
s in 24.293s)
[t-SNE] Iteration 950: error = 2.7419538, gradient norm = 0.0023426 (50 iteration
s in 25.630s)
[t-SNE] Iteration 1000: error = 2.7224588, gradient norm = 0.0022318 (50 iteratio
ns in 26.610s)
[t-SNE] KL divergence after 1000 iterations: 2.722459
```

In [44]:
```
plt.figure(figsize=(8, 8))
scatter = plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y_train, cmap=cm.tab10, edge
# Assurez-vous que y_train est votre vecteur de labels/classes correspondant à X
legend=plt.legend(*scatter.legend_elements(),title="classes")
plt.gca().add_artist(legend)
# Ajouter des étiquettes et un titre
plt.title('t-SNE Visualization')
plt.xlabel('Dim 1')
plt.ylabel('Dim 2')

# Afficher le graphique
plt.show()
```



- Il y a une assez bonnes séparation des classes, mais on remarque quand même que quelques classes ont des voisinages plutôt rapprochés : comme les 4 et les 9, les 7 et les 9, ainsi que les 3, 8 et 5. Mais on comprend pourquoi quand on regarde les imagettes : elle se ressemblent vraiment, et on pourrai nous même les confondre peut-être.

Métrique de séparation des classes

1. **Calcul de l'enveloppe convexe des points projetés pour chacune des classe classe.**

In [45]:
```
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
```

où `points` (*resp.* `labels`) dans la méthode `convexHulls(points, labels)` correspond aux images projetées dans le plan 2D avec la méthode t-SNE de l'exercice 1 (*resp.* aux labels, *i.e.* classes, des images).

2. **Calcul de l'ellipse de meilleure approximation des points.** On utilisera pour cela la classe `GaussianMixture` du module `sklearn.mixture` http://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html#sklearn.mixtu
On pourra donc utiliser le code suivant pour calculer les ellipses de meilleure approximation pour les 10 classes :

In [46]:
```
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',init
  return gaussians
```

3. **Calcul du « Neighborhood Hit » (NH)** [PNML08]. 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. Le code suivant permet de calculer la métrique NH, en utilisant la classe `NearestNeighbors` du module `sklearn.neighbors` :

In [47]:
```
def neighboring_hit(points, labels):
  k = 6
  nbrs = NearestNeighbors(n_neighbors=k+1, algorithm='ball_tree').fit(points)
  distances, indices = nbrs.kneighbors(points)

  txs = 0.0
  txsc = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
  nppts = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

  for i in range(len(points)):
    tx = 0.0
    for j in range(1,k+1):
      if (labels[indices[i,j]]== labels[i]):
        tx += 1
    tx /= k
    txsc[labels[i]] += tx
    nppts[labels[i]] += 1
    txs += tx

  for i in range(10):
    txsc[i] /= nppts[i]

  return txs / len(points)
```

Question :

Analyse des Méthodes de Séparabilité des Classes

L'enveloppe convexe

L'enveloppe convexe est une forme géométrique qui englobe tous les points d'un ensemble de données. Dans le contexte de la séparabilité des classes, une enveloppe convexe bien définie autour des points d'une classe peut indiquer une certaine facilité à séparer cette classe des autres.

L'enveloppe convexe considère la distribution globale des points et peut aider à identifier des tendances générales dans la distribution des classes. Cependant, elle ne permet pas de déterminer si deux classes sont bien séparées l'une de l'autre.

L'ellipse de meilleure approximation

L'ellipse de meilleure approximation est une ellipse qui tente de capturer la distribution des points d'une classe de manière plus spécifique qu'une simple enveloppe convexe. Elle fournit des informations plus fines sur la forme et la distribution d'une classe.

Par rapport à l'enveloppe convexe, l'ellipse de meilleure approximation est mieux adaptée pour décrire la géométrie précise d'une classe, en particulier lorsque la distribution des points est complexe.

Le Neighborhood Hit

Le Neighborhood Hit est une métrique qui mesure à quel point les voisins d'un point partagent le même label que lui. Dans le contexte de la séparabilité des classes, une valeur élevée de Neighborhood Hit indique que les points d'une classe sont regroupés et bien séparés des points des autres classes.

Contrairement à l'enveloppe convexe et à l'ellipse de meilleure approximation, qui se basent sur la géométrie globale ou locale des classes, le Neighborhood Hit évalue la séparabilité des classes en analysant les relations de proximité entre les points dans l'espace des caractéristiques.

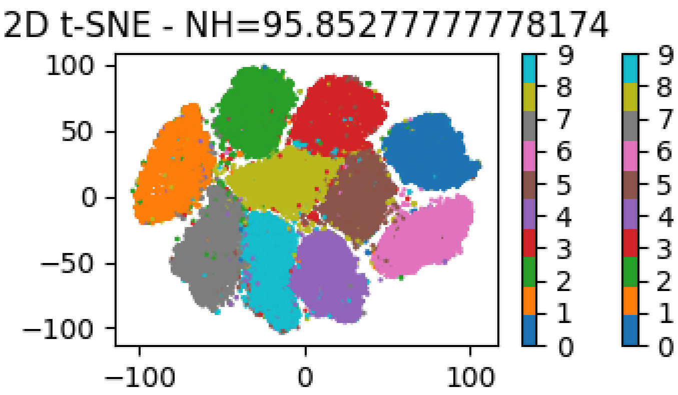
In [48]:
```python
# Computing convex hulls, best fitting ellipses & NH
convex_hulls = convexHulls(X_tsne, y_train)
ellipses = best_ellipses(X_tsne, y_train)
nh = neighboring_hit(X_tsne, y_train)
```

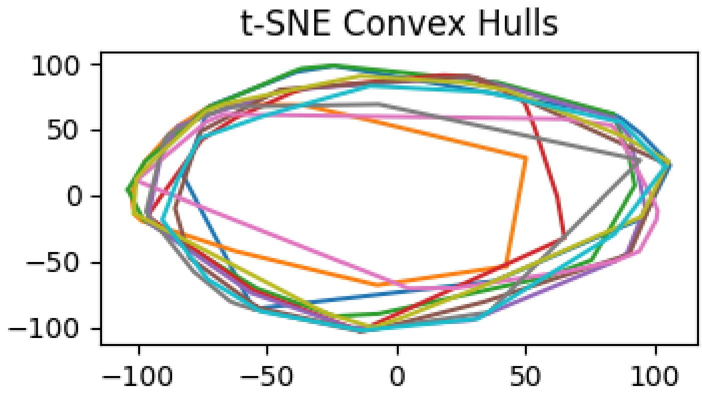
In [62]:
```python
def visualization(points2D, labels, convex_hulls, ellipses ,projname, nh):
    points2D_c= []
    for i in range(10):
        points2D_c.append(points2D[labels==i, :])
    # Data Visualization
    cmap =cm.tab10

    plt.figure(figsize=(3.841, 7.195), dpi=100)
    plt.set_cmap(cmap)
    plt.subplots_adjust(hspace=0.4 )
    plt.subplot(311)
    plt.scatter(points2D[:,0], points2D[:,1], c=labels,  s=3,edgecolors='none',
    plt.colorbar(ticks=range(10))
    plt.title("2D "+projname+" - NH="+str(nh*100.0))

    vals = [ i/10.0 for i in range(10)]
    sp2 = plt.subplot(312)
```

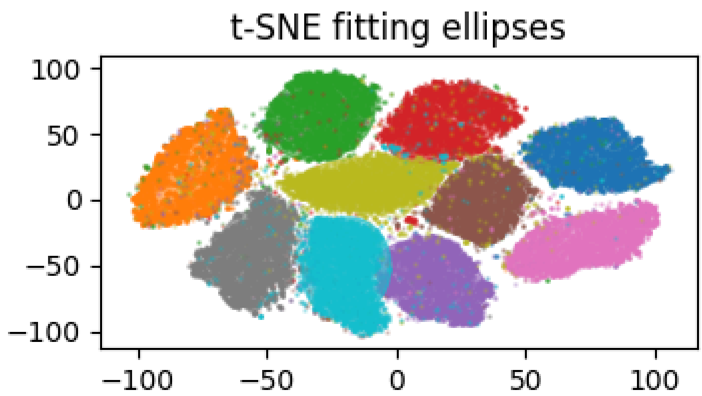
```
    for i in range(10):
        ch = np.append(convex_hulls[i].vertices,convex_hulls[i].vertices[0])
        sp2.plot(points2D_c[i][ch, 0], points2D_c[i][ch, 1], '-',label='$%i$'%i,

    plt.colorbar(ticks=range(10))
    plt.title(projname+" Convex Hulls")

    def plot_results(X, Y_, means, covariances, index, title, color):
        splot = plt.subplot(3, 1, 3)
        for i, (mean, covar) in enumerate(zip(means, covariances)):
            v, w = linalg.eigh(covar)
            v = 2. * np.sqrt(2.) * np.sqrt(v)
            u = w[0] / linalg.norm(w[0])
            # as the DP will not use every component it has access to
            # unless it needs it, we shouldn't plot the redundant
            # components.
            if not np.any(Y_ == i):
              continue
            plt.scatter(X[Y_ == i, 0], X[Y_ == i, 1], .8, color=color, alpha = 0

            # Plot an ellipse to show the Gaussian component
            angle = np.arctan(u[1] / u[0])
            angle = 180. * angle / np.pi  # convert to degrees
            ell = mpl.patches.Ellipse(
    xy=mean,
    width=v[0],
    height=v[1],
    angle=180. + angle,
    color=color
)
            ell.set_clip_box(splot.bbox)
            ell.set_alpha(0.6)
            splot.add_artist(ell)

        plt.title(title)

    plt.subplot(313)
    for i in range(10):
        plot_results(points2D[labels==i, :], ellipses[i].predict(points2D[labels

    plt.savefig(projname+".png", dpi=100)
    plt.show()
```

In [63]:

```
# visualization
visualization(X_tsne, y_train, convex_hulls, ellipses, "t-SNE", nh)
```







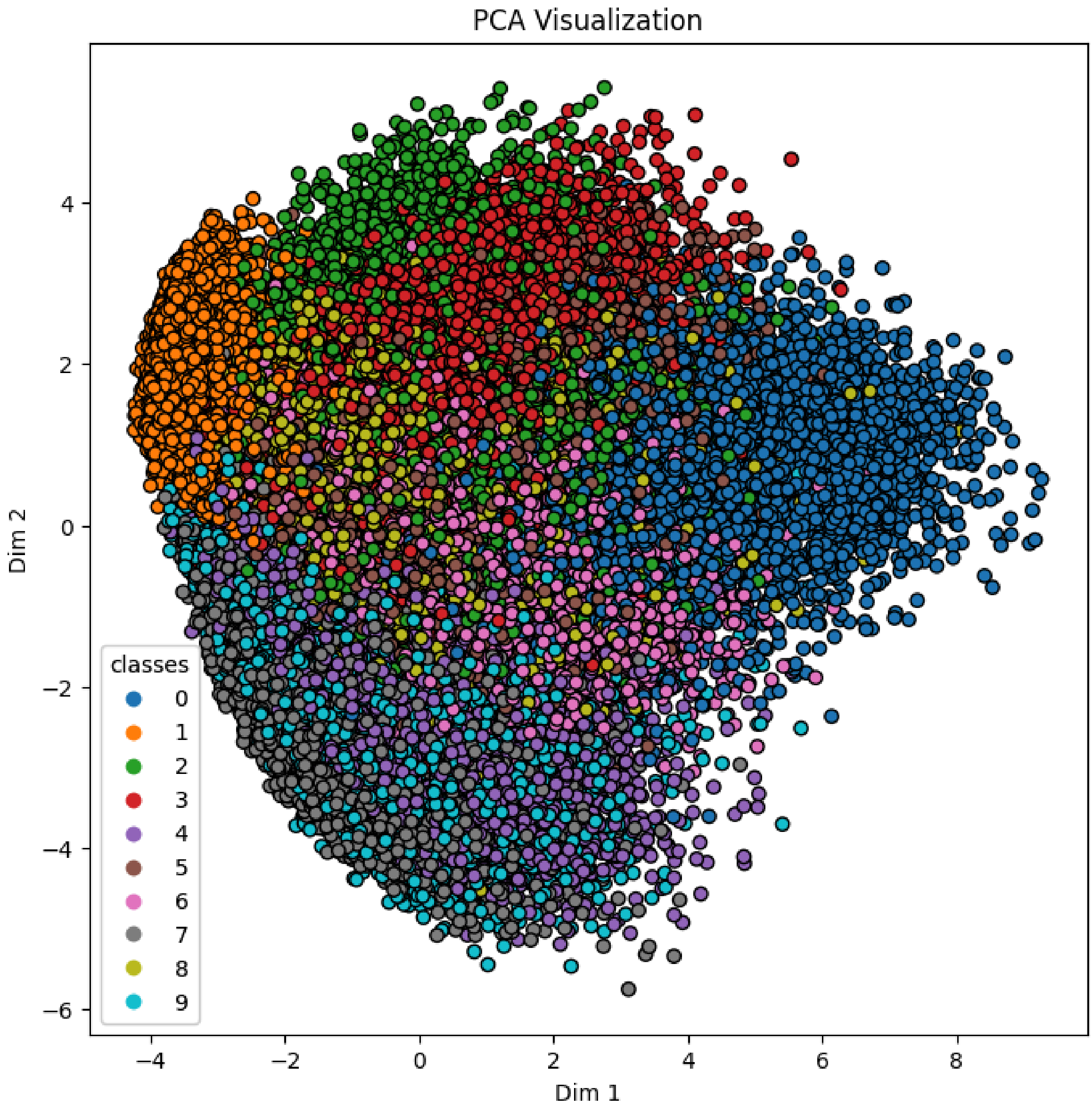
- Comparer la méthode t-SNE à une Analyse en Composantes Principales (ACP) [Hot33]. On pourra utiliser la classe `PCA` du module `sklearn.decomposition` http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

- Analyser la distribution des points et des classes : que peut-on en conclure ?

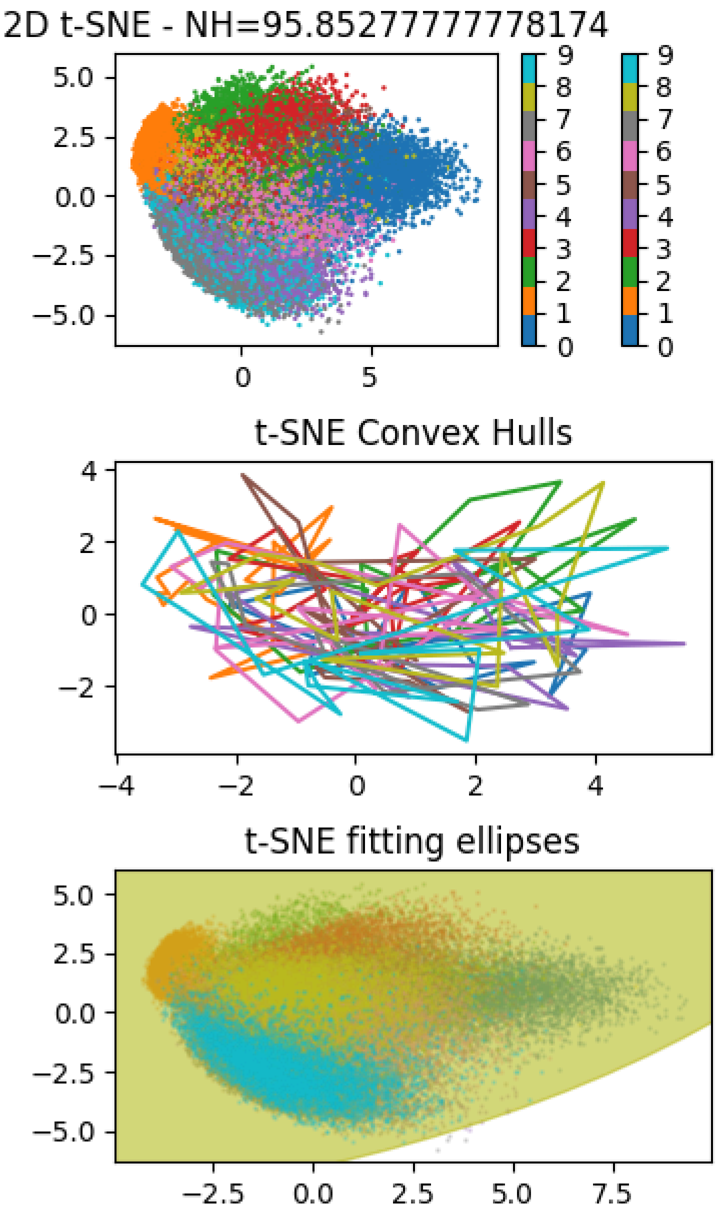
In [64]:
```
from sklearn.decomposition import PCA

acp = PCA(n_components=2)
X_pca = acp.fit_transform(X_train)
```

In [65]:
```
plt.figure(figsize=(8, 8))
plt.scatter(X_pca[:,0], X_pca[:,1], c=y_train, cmap=cm.tab10, edgecolors='k')
# Assurez-vous que y_train est votre vecteur de labels/classes correspondant à X
legend=plt.legend(*scatter.legend_elements(),title="classes")
plt.gca().add_artist(legend)
# Ajouter des étiquettes et un titre
plt.title('PCA Visualization')
plt.xlabel('Dim 1')
plt.ylabel('Dim 2')
```

```python
# Afficher le graphique
plt.show()
```



In [66]:
```python
visualization(X_pca, y_train, convex_hulls, ellipses, "t-SNE", nh)
```



- Les résultats montrent que t-SNE offre une meilleure séparation visuelle des données par rapport à l’ACP. Cela s’explique par le fait que t-SNE conserve la structure locale des données, tandis que l’ACP se concentre sur la maximisation de la variance globale. Cette supériorité est confirmée par les valeurs du "Neighborhood Hit", qui atteint 93,3 % avec t-SNE contre seulement 38,5 % avec l’ACP. En revanche, les enveloppes convexes ne sont pas adaptées pour évaluer la séparabilité des classes, car elles incluent tous les points, y compris les valeurs extrêmes ou aberrantes.

Exercice 5 : Visualisation des représentations internes des réseaux de neurones

On va maintenant s’intéresser à visualisation de l’effet de « manifold untangling » permis par les réseaux de neurones.

Créer un script dont l'objectif va être d'utiliser la méthode t-SNE de l'exercice 2 pour projeter les couches cachés des réseaux de neurones dans un espace de dimension 2, ce qui permettra de visualiser la distribution des représentations internes et des labels.

- Commencer par charger le Perceptron entraîné avec `Keras` dans la partie précédente, en utilisant la méthode `loadModel(savename)` suivante:

In [85]:
```
from keras.models import load_model

def loadModel(savename):
    # Charger le modèle complet depuis le fichier .h5
    model = load_model(savename + ".h5")
    model.load_weights(savename + ".weights.h5")
    print(f"Modèle complet chargé depuis le fichier {savename}.h5.")
    print(f"Poids chargés depuis le fichier {savename}.weights.h5.")
    return model
```

In [86]:
```
model_MLP_appele = loadModel("MLP")
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be b
uilt. `model.compile_metrics` will be empty until you train or evaluate the mode
l.
Modèle complet chargé depuis le fichier MLP.h5.
Poids chargés depuis le fichier MLP.weights.h5.
```

In [87]:
```
model_MLP_appele.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
fc1 (Dense)	(None, 100)	78,500
activation_1 (Activation)	(None, 100)	0
fc2 (Dense)	(None, 10)	1,010
activation_2 (Activation)	(None, 10)	0

Total params: 79,512 (310.60 KB)

Trainable params: 79,510 (310.59 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 2 (12.00 B)

In [88]:
```
learning_rate = 0.1
sgd = SGD(learning_rate)
model_MLP_appele.compile(loss='categorical_crossentropy',optimizer=sgd,metrics=[
```

In [89]:
```
scores = model_MLP_appele.evaluate(X_test, Y_test, verbose=0)
print("%s: %.2f%%" % (model_MLP_appele.metrics_names[0], scores[0]*100))
print("%s: %.2f%%" % (model_MLP_appele.metrics_names[1], scores[1]*100))
```

```
loss: 7.77%
compile_metrics: 97.54%
```

- 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.

In [90]:
```
# Chargement du modèle
from keras import utils
from keras.optimizers import SGD
```

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.
- Finalement, on va utiliser la méthode t-SNE mise en place à l'exercice 2 pour visualiser les représentations internes des données.

En plus du Perceptron précédent, on pourra visualiser les représentations internes apprises par un réseau convolutif de type LeNet de la partie précédente. Conclure sur la capacité des réseaux de neurones à résoudre le problème du Manifold Untangling.

[Hot33] H. Hotelling. *Analysis of a Complex of Statistical Variables Into Principal Components*. Warwick & York, 1933. URL: https://books.google.fr/books?id=qJfXAAAAMAAJ.

[LBD+89] 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.

[PNML08] 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.

[vdMH08] Laurens van der Maaten and Geoffrey E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

In [91]:
```
# On extrait la couche cachée
model_MLP_appele.pop()
model_MLP_appele.pop()
```

Out[91]:
```
<Dense name=fc2, built=True>
```

In [92]:
```
representation = model_MLP_appele.predict(X_test)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
```

In [93]:
```
representation.shape
```

Out[93]:
```
(10000, 100)
```

In [94]:
```
tsne = TSNE(n_components=2, random_state=0, init='pca',  perplexity=30, verbose=
 # Fit and transform on the first 1000 data points
X_test_tsne = tsne.fit_transform(representation)
```

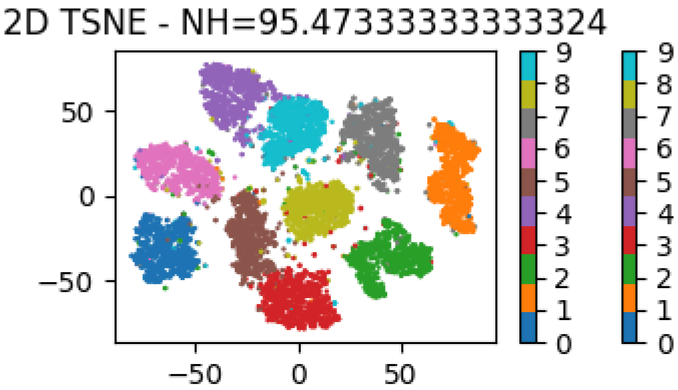
```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 10000 samples in 0.004s...
[t-SNE] Computed neighbors for 10000 samples in 0.638s...
[t-SNE] Computed conditional probabilities for sample 1000 / 10000
[t-SNE] Computed conditional probabilities for sample 2000 / 10000
[t-SNE] Computed conditional probabilities for sample 3000 / 10000
[t-SNE] Computed conditional probabilities for sample 4000 / 10000
[t-SNE] Computed conditional probabilities for sample 5000 / 10000
[t-SNE] Computed conditional probabilities for sample 6000 / 10000
[t-SNE] Computed conditional probabilities for sample 7000 / 10000
[t-SNE] Computed conditional probabilities for sample 8000 / 10000
[t-SNE] Computed conditional probabilities for sample 9000 / 10000
[t-SNE] Computed conditional probabilities for sample 10000 / 10000
[t-SNE] Mean sigma: 0.713628
[t-SNE] Computed conditional probabilities in 0.261s
[t-SNE] Iteration 50: error = 86.3063812, gradient norm = 0.0167252 (50 iteration
s in 4.434s)
[t-SNE] Iteration 100: error = 79.7664642, gradient norm = 0.0052940 (50 iteratio
ns in 4.005s)
[t-SNE] Iteration 150: error = 78.2772141, gradient norm = 0.0027242 (50 iteratio
ns in 3.453s)
[t-SNE] Iteration 200: error = 77.6221390, gradient norm = 0.0017427 (50 iteratio
ns in 3.296s)
[t-SNE] Iteration 250: error = 77.2509308, gradient norm = 0.0013103 (50 iteratio
ns in 3.210s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 77.250931
[t-SNE] Iteration 300: error = 2.7438259, gradient norm = 0.0124555 (50 iteration
s in 3.042s)
[t-SNE] Iteration 350: error = 2.2362640, gradient norm = 0.0108709 (50 iteration
s in 4.362s)
[t-SNE] Iteration 400: error = 2.0027003, gradient norm = 0.0097518 (50 iteration
s in 2.827s)
[t-SNE] Iteration 450: error = 1.8657129, gradient norm = 0.0089213 (50 iteration
s in 3.195s)
[t-SNE] Iteration 500: error = 1.7750139, gradient norm = 0.0083183 (50 iteration
s in 3.787s)
[t-SNE] Iteration 550: error = 1.7103765, gradient norm = 0.0078250 (50 iteration
s in 3.829s)
[t-SNE] Iteration 600: error = 1.6619165, gradient norm = 0.0073380 (50 iteration
s in 3.619s)
[t-SNE] Iteration 650: error = 1.6247270, gradient norm = 0.0069369 (50 iteration
s in 3.825s)
[t-SNE] Iteration 700: error = 1.5954602, gradient norm = 0.0064281 (50 iteration
s in 4.286s)
[t-SNE] Iteration 750: error = 1.5723265, gradient norm = 0.0059328 (50 iteration
s in 4.435s)
[t-SNE] Iteration 800: error = 1.5537850, gradient norm = 0.0053861 (50 iteration
s in 3.143s)
[t-SNE] Iteration 850: error = 1.5389080, gradient norm = 0.0049494 (50 iteration
s in 4.082s)
[t-SNE] Iteration 900: error = 1.5269073, gradient norm = 0.0043344 (50 iteration
s in 3.985s)
[t-SNE] Iteration 950: error = 1.5169708, gradient norm = 0.0039597 (50 iteration
s in 2.746s)
[t-SNE] Iteration 1000: error = 1.5088512, gradient norm = 0.0034874 (50 iteratio
ns in 3.092s)
[t-SNE] KL divergence after 1000 iterations: 1.508851
```

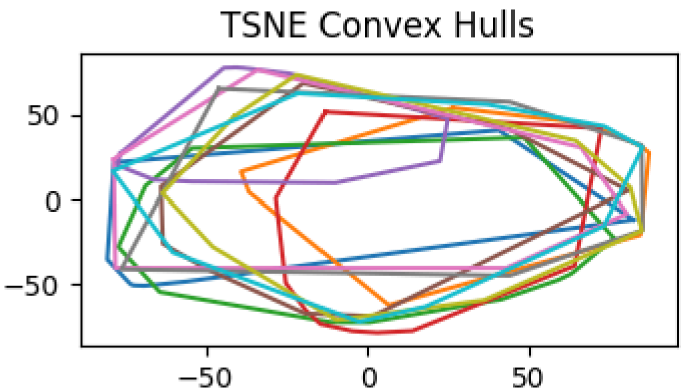
In [95]:

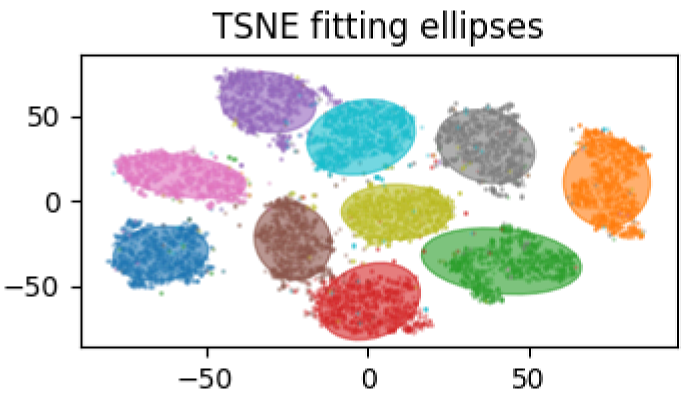
```
# Computing convex hulls, best fitting ellipses & NH
conv_hulls = convexHulls(X_test_tsne, y_test)
```

```python
best_ells = best_ellipses(X_test_tsne, y_test)
nh = neighboring_hit(X_test_tsne, y_test)
```

In [96]:
```python
visualization(X_test_tsne, y_test, conv_hulls, best_ells, "TSNE", nh)
```







In [97]:
```python
model_CNN_appele = loadModel("CNN")
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be b
uilt. `model.compile_metrics` will be empty until you train or evaluate the mode
l.
Modèle complet chargé depuis le fichier CNN.h5.
Poids chargés depuis le fichier CNN.weights.h5.
```

In [98]:
```python
learning_rate = 0.1
sgd = SGD(learning_rate)
model_CNN_appele.compile(loss='categorical_crossentropy',optimizer=sgd,metrics=[
```

In [105…
```python
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)
```

In [108...

```
print(f"Forme de X_test : {X_test.shape}")
print(f"Forme de Y_test : {Y_test.shape}")
```

```
Forme de X_test : (10000, 28, 28, 1)
Forme de Y_test : (10000, 10)
```

In [109...

```
model_CNN_appele.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 24, 24, 16)	416
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 16)	0
conv2d_2 (Conv2D)	(None, 8, 8, 32)	12,832
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 32)	0
flatten (Flatten)	(None, 512)	0
fc1 (Dense)	(None, 100)	51,300
activation_3 (Activation)	(None, 100)	0
fc2 (Dense)	(None, 10)	1,010
activation_4 (Activation)	(None, 10)	0

Total params: 65,558 (256.09 KB)

Trainable params: 65,558 (256.09 KB)

Non-trainable params: 0 (0.00 B)

In [110...

```
print(f"Type de X_test : {X_test.dtype}")
print(f"Valeurs min et max de X_test : {X_test.min()}, {X_test.max()}")
```

```
Type de X_test : float32
Valeurs min et max de X_test : 0.0, 1.0
```

In [111...

```
print(f"Forme de Y_test : {Y_test.shape}")
print(f"Valeurs uniques de Y_test : {np.unique(Y_test)}")
```

```
Forme de Y_test : (10000, 10)
Valeurs uniques de Y_test : [0. 1.]
```

In [112...

```
sample_input = X_test[0:1]  # Prendre une seule image
prediction = model_CNN_appele.predict(sample_input)
print(f"Prédiction pour un échantillon : {prediction}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 410ms/step
Prédiction pour un échantillon : [[1.6318791e-05 1.3223232e-04 3.4253023e-04 6.18
96906e-05 1.7231714e-06
  3.1119691e-06 6.9401764e-08 9.9936229e-01 4.5798984e-06 7.5194330e-05]]
```

In [115...

```
import tensorflow as tf
tf.config.experimental_run_functions_eagerly(True)
```

```
WARNING:tensorflow:From C:\Users\jaimo\AppData\Local\Temp\ipykernel_35668\5868888
08.py:2: experimental_run_functions_eagerly (from tensorflow.python.eager.polymor
phic_function.eager_function_run) is deprecated and will be removed in a future v
ersion.
Instructions for updating:
Use `tf.config.run_functions_eagerly` instead of the experimental version.
```


```
WARNING:tensorflow:From C:\Users\jaimo\AppData\Local\Temp\ipykernel_35668\5868888
08.py:2: experimental_run_functions_eagerly (from tensorflow.python.eager.polymor
phic_function.eager_function_run) is deprecated and will be removed in a future v
ersion.
Instructions for updating:
Use `tf.config.run_functions_eagerly` instead of the experimental version.
```


!! j'ai un probleme pour load model cnn

In [116…

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

```
C:\Users\jaimo\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2
kfra8p0\LocalCache\local-packages\Python311\site-packages\tensorflow\python\data
\ops\structured_function.py:258: UserWarning: Even though the `tf.config.experime
ntal_run_functions_eagerly` option is set, this option does not apply to tf.data
functions. To force eager execution of tf.data functions, please use `tf.data.exp
erimental.enable_debug_mode()`.
  warnings.warn(
loss: 4.35%
compile_metrics: 98.60%
```

In [117…

```
# On extrait la couche cachée
model_CNN_appele.pop()
model_CNN_appele.pop()
```

Out[117… `<Dense name=fc2, built=True>`

In [118…

```
representation2 = model_CNN_appele.predict(X_test)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step
```

In [119…

```
tsne = TSNE(n_components=2, random_state=0, init='pca',  perplexity=30, verbose=
 # Fit and transform on the first 1000 data points
X_test_tsne2 = tsne.fit_transform(representation2)
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 10000 samples in 0.005s...
[t-SNE] Computed neighbors for 10000 samples in 0.585s...
[t-SNE] Computed conditional probabilities for sample 1000 / 10000
[t-SNE] Computed conditional probabilities for sample 2000 / 10000
[t-SNE] Computed conditional probabilities for sample 3000 / 10000
[t-SNE] Computed conditional probabilities for sample 4000 / 10000
[t-SNE] Computed conditional probabilities for sample 5000 / 10000
[t-SNE] Computed conditional probabilities for sample 6000 / 10000
[t-SNE] Computed conditional probabilities for sample 7000 / 10000
[t-SNE] Computed conditional probabilities for sample 8000 / 10000
[t-SNE] Computed conditional probabilities for sample 9000 / 10000
[t-SNE] Computed conditional probabilities for sample 10000 / 10000
[t-SNE] Mean sigma: 0.768223
[t-SNE] Computed conditional probabilities in 0.327s
[t-SNE] Iteration 50: error = 81.5583801, gradient norm = 0.0271650 (50 iteration
s in 5.112s)
[t-SNE] Iteration 100: error = 74.7478485, gradient norm = 0.0083636 (50 iteratio
ns in 4.291s)
[t-SNE] Iteration 150: error = 72.9799881, gradient norm = 0.0051071 (50 iteratio
ns in 3.412s)
[t-SNE] Iteration 200: error = 72.1063080, gradient norm = 0.0037275 (50 iteratio
ns in 2.988s)
[t-SNE] Iteration 250: error = 71.5759354, gradient norm = 0.0029148 (50 iteratio
ns in 2.791s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 71.575935
[t-SNE] Iteration 300: error = 2.5787828, gradient norm = 0.0108199 (50 iteration
s in 2.852s)
[t-SNE] Iteration 350: error = 2.1130977, gradient norm = 0.0106002 (50 iteration
s in 2.903s)
[t-SNE] Iteration 400: error = 1.8786185, gradient norm = 0.0096964 (50 iteration
s in 2.793s)
[t-SNE] Iteration 450: error = 1.7395205, gradient norm = 0.0089037 (50 iteration
s in 2.946s)
[t-SNE] Iteration 500: error = 1.6470879, gradient norm = 0.0083262 (50 iteration
s in 2.665s)
[t-SNE] Iteration 550: error = 1.5809027, gradient norm = 0.0078539 (50 iteration
s in 2.695s)
[t-SNE] Iteration 600: error = 1.5311134, gradient norm = 0.0074001 (50 iteration
s in 4.008s)
[t-SNE] Iteration 650: error = 1.4929729, gradient norm = 0.0069786 (50 iteration
s in 3.013s)
[t-SNE] Iteration 700: error = 1.4630275, gradient norm = 0.0064638 (50 iteration
s in 3.007s)
[t-SNE] Iteration 750: error = 1.4390739, gradient norm = 0.0059592 (50 iteration
s in 3.807s)
[t-SNE] Iteration 800: error = 1.4198716, gradient norm = 0.0054352 (50 iteration
s in 4.307s)
[t-SNE] Iteration 850: error = 1.4045199, gradient norm = 0.0049811 (50 iteration
s in 4.128s)
[t-SNE] Iteration 900: error = 1.3919624, gradient norm = 0.0043489 (50 iteration
s in 3.679s)
[t-SNE] Iteration 950: error = 1.3821222, gradient norm = 0.0039124 (50 iteration
s in 4.106s)
[t-SNE] Iteration 1000: error = 1.3740087, gradient norm = 0.0034474 (50 iteratio
ns in 4.064s)
[t-SNE] KL divergence after 1000 iterations: 1.374009
```

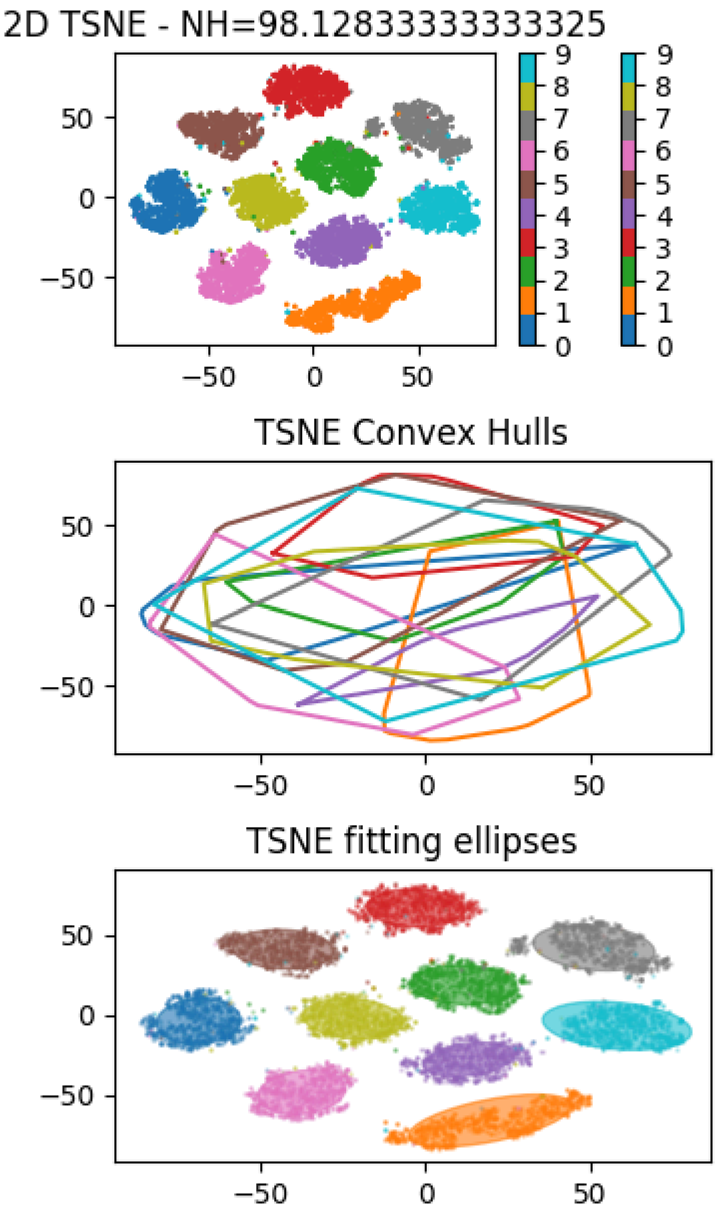
In [120…

```
# Computing convex hulls, best fitting ellipses & NH
conv_hulls = convexHulls(X_test_tsne2, y_test)
```

```
best_ells = best_ellipses(X_test_tsne2, y_test)
nh = neighboring_hit(X_test_tsne2, y_test)
```

In [121…

```
visualization(X_test_tsne2, y_test, conv_hulls, best_ells, "TSNE", nh)
```



- En comparant les projections des données initiales avec celles des caractéristiques profondes, nous constatons que les réseaux neuronaux sont capables de séparer les classes de manière efficace. Cette aptitude se manifeste par une augmentation du nombre d'occurrences de NH dans les projections des caractéristiques profondes. Par ailleurs, la comparaison entre les MLP et les CNN révèle que les classes sont mieux compactées dans le cas des CNN. Cela se traduit par un NH plus élevé ainsi que par des ellipses plus denses et éloignées les unes des autres, offrant ainsi une meilleure représentation au niveau de la dernière couche. En revanche, les enveloppes convexes ne montrent pas d'amélioration significative, leur sensibilité aux valeurs aberrantes restant un facteur limitant.