INSAT	Module : Deep Learning
Département Mathématiques et Informatique	Sections : GL4/RT4
Année Universitaire : 2023 - 2024	Enseignante : Sana Hamdi

TP N°3 : Les Réseaux de Neurones Convolutifs

Objectif :

Ce TP a pour objectif de résoudre un problème de reconnaissance d'images en utilisant les réseaux de neurones convolutifs.

1. Problème :

Nous allons créer un réseau de neurones convolutionnels capable de reconnaître les chiffres écrits à la main ! Le principe est simple : à partir d'une banque d'images de chiffres écrits à la main (et annoté pour dire de quel chiffre il s'agit), fournie par Keras (un framework de deep learning), nous allons entraîner 3 IA différentes et les comparer.

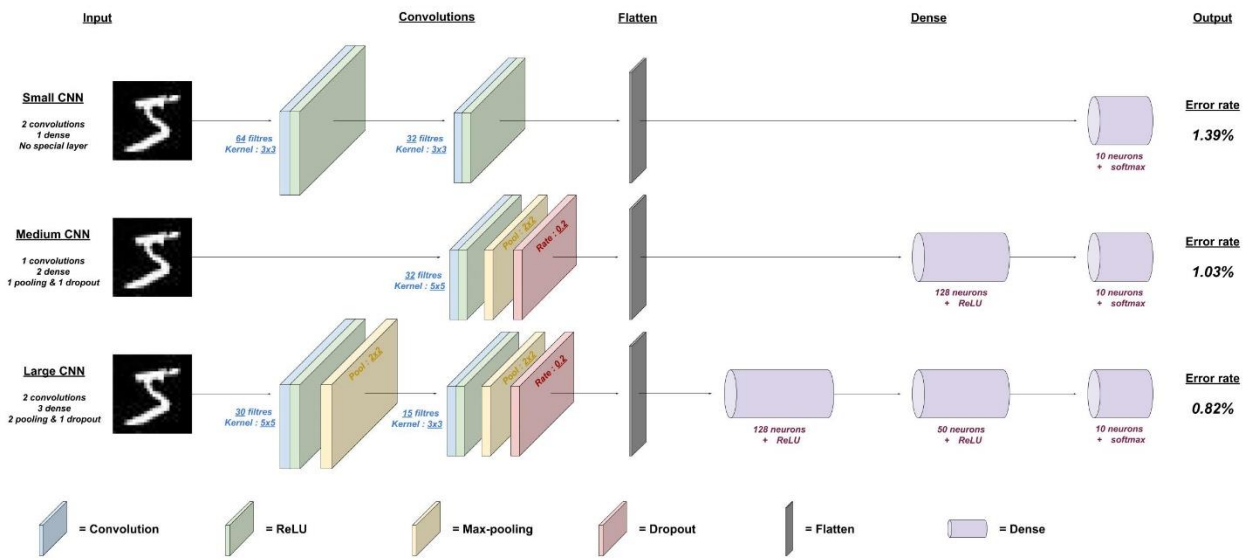
2. Installations préalables :

Nous installons Tensorflow et Keras.

Commençons par installer **Tensorflow**, le framework de Google pour faire du Deep Learning. Notez dès à présent que *nous ne nous en servirons pas*… mais on utilisera **Keras**, qui est un framework s'appuyant sur Tensorflow en backend (il en simplifie les commandes et ajoute certaines fonctionnalités).

3. Nos réseaux cibles :

Voici le schéma d'architecture des réseaux de neurones que nous allons réaliser dans ce TP. Vous remarquerez un certain nombre de différences, qui aboutissent à des résultats plus ou moins bons. Les **paramètres** ont été sélectionnés à la fois **par expérience** et de **manière empirique**, car bien souvent *on essaie plusieurs architectures* avant de trouver la meilleure !



3 CNN distincts pour un même problème : la reconnaissance de chiffres manuscrits (Lambert Rosique)

4. Construction du 1er réseau

a. Importer les données

De Keras, on va prendre le mnist (qui est le nom du dataset que l'on va utiliser), et le backend Tensorflow, nommé K, pour lequel on va spécifier le format de saisie des couches du modèle grâce à « channels_first ».

« channels_first » signifie que les kernels des convolutions auront pour format (depth, input_depth, rows, cols), à l'inverse de « channels_last » qui veut dire (rows, cols, input_depth, depth). Nous prendrons aussi le modèle, les couches (convolutions, pooling, etc…).

Lors de la génération du modèle, de nombreux paramètres sont choisis aléatoirement par Keras via Numpy. Afin d'avoir exactement le même modèle (et donc les mêmes résultats), il est courant de fixer le random grâce à **np.random.seed** (que vous devez donc ajouter).

```
import numpy as np
from keras.datasets import mnist
from keras.utils import np_utils
from keras import backend as K
K.set_image_data_format('channels_first')
fix random seed for reproducibility
seed = 7
np.random.seed(seed)
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
```

Commençons par charger les données, qui sont

- 60 000 images d'apprentissages (matrices carrées de taille 28×28) en niveaux de gris (donc 1 seul channel)
- Et 10 000 images de test (mêmes caractéristiques) depuis le jeu de données du MNIST. Keras nous apprend également que toutes les images sont des chiffres uniques entre 0 et 9 inclus.

```
#load data
(X_train, y_train),(X_test, y_test)= mnist.load_data()
```

- Ensuite, par rapport à notre configuration de TensorFlow, il va falloir modifier légèrement la structure des matrices pour qu'elles soient utilisables dans le modèle.
- On a pour l'instant du **60000x28x28**, et on voudrait du 60000x1x28x28 (nombre d'entrées, nombre de channels, largeur d'une entrée, hauteur d'une entrée). La commande « reshape » permet exactement de faire ça, à laquelle on ajoute « astype » pour convertir les matrices en float32 au lieu de uint8 (Unsigned integer (0 to 255)).

```
# reshape to be [samples][pixels][width][height]
    X_train = X_train.reshape(X_train.shape[0], 1, 28,28).astype
('float32')
    X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype
('float32')
```

- Travaillons maintenant les vecteurs de sorties pour qu'ils reflètent la valeur prédite par le réseau. En effet, il serait très compliqué (voire impossible de manière efficace et juste) d'avoir un CNN qui nous renverrait un seul et unique résultat entre 0 et 9 valant le chiffre détecté dans l'image. A l'inverse, il est plus facile pour un CNN de nous donner la probabilité que ce soit un 0, la probabilité que ce soit un 1, etc… jusqu'à 9.

```
#one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Question : Définissez une fonction **get_data_mnist()** qui sera appelée par notre 1er CNN, et qui retournera les différentes variables.

```
Def get_data_mnist() :
#load data
...
# reshape to be [samples][pixels][width][height]
...
#one hot encode outputs
```

```
...
return (X_train, y_train), (X_test, y_test), num_classes
```

b. Construire le modèle de CNN

Les données étant prêtes, on va travailler notre « **Small CNN** » conformément au schéma d'architecture présenté en début de partie.

Pour déclarer un nouveau modèle de deep learning dans Keras, on utilise l'instruction suivante, qui vaut aussi bien pour des ANN que des CNN qu'autre chose :

```
model = Sequential()
```

Ensuite, les méthodes à appeler sont très parlantes. Notre objectif est d'avoir successivement :

- Une convolution de 64 filtres en 3×3 suivie d'une couche d'activation ReLU
- Une convolution de 32 filtres en 3×3 suivie d'une couche d'activation ReLU
- Un flatten qui va créer le vecteur final à envoyer au réseau de neurones artificiels ne prend aucun paramètre, car il n'y a besoin de rien de particulier pour mettre toutes les images bout à bout.
- Un dense, réseau de neurones artificiels qui aura 10 neurones et sera suivi d'un softmax

```
  # create model
    model = Sequential()
    model.add(Conv2D(64, (3, 3), input_shape=(1, 28, 28),
activation='relu'))
    model.add(Conv2D(32, (3, 3), activation='relu'))
    model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax'))
```

Finalement, il faut compiler le modele via « **`model.compile`** » en précisant les informations suivantes, intrinsèques au réseau de neurones convolutifs…

- le **loss** : c'est une fonction qui va servir à mesurer l'écart entre les prédictions de notre modèle et les résultats attendus. Elle évalue donc la justesse du CNN et permet de mieux l'adapter aux données si besoin! Nous allons utiliser « `categorical_crossentropy` » comme loss, car on a des données de type « catégories » en sortie de l'algorithme.
- l'**optimizer** : c'est un algorithme qui va dicter comment mettre à jour le CNN pour diminuer le loss, et avoir donc de meilleures prédictions. Ici on s'appuiera sur « **adam** » (adaptive moment estimation), très souvent utilisé.
- Le paramètre **metrics** : c'est exactement comme le loss, sauf que metrics n'est PAS utilisée par le CNN, à l'inverse du loss qui sert pour la mise à jour des variables du

CNN via l'optimizer. On utilisera cette fois « accuracy », sans que cela ait de réelle importance pour nous.

Question : Créez une fonction **Small_model** qui construit le 1er CNN.

```
def small_model():
    # create model
    ...
    #compile model
    ...
    return model
```

c. Entrainer et évaluer notre Small CNN

Pour entraîner un modèle, il suffit d'appeler la méthode **fit** qui prend en arguments :

- Les données d'apprentissage, ici X_train
- Les prédictions attendues d'apprentissage, ici y_train
- Le paramètre validation_data : un vecteur qui contient les données d'entrées à tester et les valeurs à prédire, ici (X_test , y_test)
- Le nombre d'itérations de l'entraînement, c'est-à-dire le nombre de fois qu'il va répéter l'entraînement complet (prédire les 60 000 images), via epochs. Mettez 10.
- Et enfin le paramètre batch_size, qui est le nombre de données d'entrées à analyser « à la suite avant de mettre à jour le CNN », qui vaudra 200 (valeur standard). L'intérêt est que mettre à jour son CNN après chaque image demande du temps et a tendance à nous éloigner de la solution générale optimale (je rappelle qu'on veut un CNN capable d'identifier n'importe quel chiffre dessiné, pas seulement ceux de l'entraînement). En indiquant 200, on s'assure que le réseau ne se spécialise « pas trop »

Question1 : Donnez l'instruction permettant d'entrainer notre modèle.

A l'exécution (en utilisant un CPU et non le GPU), vous devriez obtenir un résultat similaire à celui-ci :

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 198s 3ms/step - loss: 0.5453 - accuracy: 0.9231
- val loss: 0.0768 - val accuracy: 0.9742
Epoch 2/10
60000/60000 [==============================] - 204s 3ms/step - loss: 0.0589 - accuracy: 0.9819
- val_loss: 0.0815 - val_accuracy: 0.9748
Epoch 3/10
60000/60000 [==============================] - 206s 3ms/step - loss: 0.0338 - accuracy: 0.9892
- val loss: 0.0713 - val accuracy: 0.9788
Epoch 4/10
60000/60000 [==============================] - 208s 3ms/step - loss: 0.0269 - accuracy: 0.9911
- val loss: 0.1070 - val accuracy: 0.9749
Epoch 5/10
60000/60000 [==============================] - 211s 4ms/step - loss: 0.0197 - accuracy: 0.9936
- val loss: 0.0878 - val accuracy: 0.9785
Epoch 6/10
```

```
60000/60000 [==============================] - 203s 3ms/step - loss: 0.0126 - accuracy: 0.9955
- val loss: 0.0971 - val accuracy: 0.9796
Epoch 7/10
60000/60000 [==============================] - 204s 3ms/step - loss: 0.0114 - accuracy: 0.9963
- val_loss: 0.1044 - val_accuracy: 0.9779
Epoch 8/10
60000/60000 [==============================] - 189s 3ms/step - loss: 0.0142 - accuracy: 0.9954
- val loss: 0.1227 - val accuracy: 0.9768
Epoch 9/10
60000/60000 [==============================] - 142s 2ms/step - loss: 0.0115 - accuracy: 0.9966
- val_loss: 0.1266 - val_accuracy: 0.9767
Epoch 10/10
60000/60000 [==============================] - 138s 2ms/step - loss: 0.0132 - accuracy: 0.9959
- val_loss: 0.1291 - val_accuracy: 0.9760
```

Question2 : Implémenter la méthode permettant d'évaluer notre modèle comme suit :

```
def print_model_error_rate(model, X_test, y_test):
    # Final evaluation of the model
    scores = model.evaluate(X_test, y_test, verbose=0)
    print("Model score : %.2f%%" % (scores[1]*100))
    print("Model error rate : %.2f%%" % (100-scores[1]*100))
```

« `evaluate` » permet d'évaluer un modèle comme son nom l'indique (plusieurs informations sont remontées par Keras). Pour avoir le taux d'erreur, il suffit de prendre 100% – le taux de succès (`scores[1]`).

Quel taux d'erreur avez-vous obtenu ?

5. Construction des 3 modèles après normalisation de données

Dans le 1er modèle on a conservé des entiers **entre 0 et 255** alors que dans les versions à améliorer, on va les convertir en float en les dévisant par 255 (pour avoir des valeurs **entre 0 et 1**). Cette conversion et normalisation est une étape très importante de la préparation des données, car elle permet de réduire l'écart entre les valeurs extrêmes et d'éviter les overflow dans les calculs car des nombres supérieurs à 1 peuvent rapidement tendre vers l'infini si on n'est pas prudent.

1. Modifier la fonction get_data_mnist() pour qu'elle prenne en considération la normalisation.
2. Quel est le nouveau taux d'erreur? Qu'est-ce que vous remarquez ?
3. Construire, entrainer et évaluer le medium_CNN défini comme suit :
    - Une convolution de 32 filtres en 5×5 avec un ReLU en sortie (n'oubliez pas le paramètre input_shape)
    - Un max-pooling de 2×2
    - Un dropout de 0.2
    - Un flatten
    - Un dense à 128 sorties avec un ReLU
    - Un dense final à 10 sorties avec un softmax

    Le max-pooling est ajouté grâce à `MaxPooling2D`, comme vous vous en doutez, qui prend en paramètre les dimensions du pooling (2,2) dans le paramètre `pool_size`.

- Le dropout pour sa part est ajouté via `Dropout`, qui reçoit directement le taux de dropout (i.e. 0.2). Cela signifie que 20% des neurones seront ignorés

4. Construire, entrainer et évaluer le large_CNN défini comme suit :
    - Une convolution de 30 filtres en 5×5 avec une activation ReLU (n'oubliez pas l'input_shape)
    - Un max-pooling de 2×2
    - Une convolution de 15 filtres en 3×3 avec ReLU
    - Un dropout de 0.2
    - Un flatten
    - Un dense de 128 sorties avec ReLU
    - Un dense de 50 sorties avec ReLU
    - Un dense de 10 sorties avec softmax

6. Sauvegarde et chargements des modèles Keras :

Pour la **sauvegarde**, il suffit d'utiliser la fonction « `to_json` » des modèles de Keras qui permet de réaliser l'export du modèle (**l'architecture**), à placer dans un fichier json. Il faut également appeler « `save_weights` » qui sauvegardera les paramètres du réseau (donc tout ce qui vient avec son entraînement) en h5. En aucun cas il faudra oublier l'un ou l'autre des fichiers, sinon vous devrez soit réécrire l'architecture du modèle, avec les convolutions, etc… (si vous perdez le json) soit ré-entraîner tout le CNN.

Voici la fonction de sauvegarde :

```
# This function saves a model on the drive using two files: a json and a h5
def save_keras_model(model, filename):
    # serialize model to JSON
    model_json = model.to_json()
    with open(filename+".json", "w") as json_file:
        json_file.write(model_json)
    # serialize weights to HDF5
    model.save_weights(filename+".h5")
```

Concernant le chargement, on a besoin d'un import de Keras :

```
from keras.models import model_from_json
```

Ensuite, il faudra lire le json pour charger l'architecture du modèle de CNN puis lire le h5 pour mettre à jour les variables du CNN et récupérer son entraînement :

```
This function loads a model from two files : a json and a h5
# BE CAREFUL : the model NEEDS TO BE COMPILED before any use !
def load_keras_model(filename):
    # load json and create model
    json_file = open(filename+".json", 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    loaded_model = model_from_json(loaded_model_json)
    # load weights into new model
    loaded_model.load_weights(filename+".h5")
    return loaded_model
```