


<b>Année Universitaire :</b> 2021-2022  <b>Niveau:</b> 2 GII	<b>Module :</b> Intelligence artificielle  <b>TP N°2</b> <b>Réseaux à couches (Multi-couches)</b> <b>(Feedforward Neural Networks)</b>	
<b>Responsable de cours :</b> Pr. Mounir Ben Naser		<b>Responsable de TP :</b> Amira Echtioui

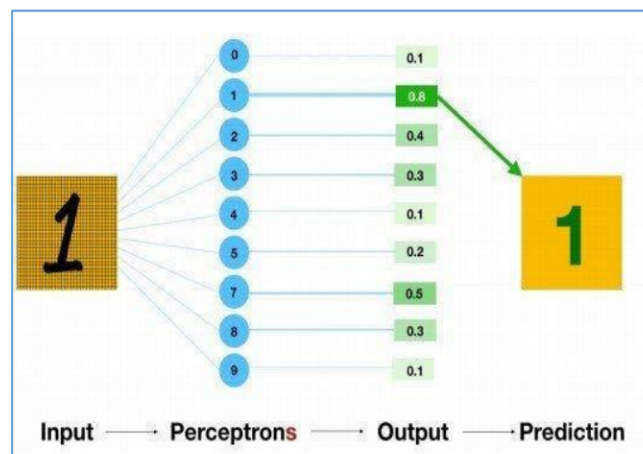
### ➤ Objectifs:

Lors de cette séance, vous serez capable de :

- Comprendre le fonctionnement du réseau multicouche.
- Implémentation du réseau multicouche

### Ensembles de données MNIST

MNIST signifie « Institut national modifié des normes et de la technologie ». Il s'agit d'un ensemble de données de 70 000 images manuscrites. Chaque image fait 28x28 pixels soit environ 784 éléments. Chaque caractéristique représente l'intensité d'un seul pixel, c'est-à-dire . de 0 (blanc) à 255 (noir). Cette base de données est ensuite divisée en 60 000 images d'entraînement et 10 000 images de test.



### Importer les bibliothèques

Tout d'abord, nous avons importé toutes les bibliothèques que nous allons utiliser.

```
import tensorflow as tf
from matplotlib import pyplot as plt
import numpy as np
```

Nous avons importé **TensorFlow** qui est une *bibliothèque gratuite open source* utilisée pour les applications d'apprentissage automatique telles que les réseaux de neurones, etc. De plus, nous avons importé la fonction **pyplot** , qui est essentiellement utilisée pour le *traçage*, à partir de la bibliothèque **matplotlib** qui est utilisée à des fins de *visualisation* . Après cela, nous avons importé **NumPy** , c'est-à-dire Numerical Python, qui est utilisé pour *effectuer diverses opérations mathématiques*.

### Charger le jeu de données

```
from keras.datasets import mnist
objects=mnist
(train_img,train_lab),(test_img,test_lab)=objects.load_data()
```

Le jeu de données MNIST en fait également partie. Donc, nous l'avons importé de *keras.datasets* et l'avons chargé dans des "objets" variables. La méthode *objects.load\_data()* nous renvoie les données d'apprentissage (*train\_img*), ses étiquettes (*train\_lab*) ainsi que les données de test (*test\_img*) et ses étiquettes (*test\_lab*). Sur les 70 000 images fournies dans l'ensemble de données, 60 000 sont données pour la formation et 10 000 sont données pour les tests.

Avant de prétraiter les données, nous avons d'abord affiché les 20 premières images de l'ensemble d'apprentissage à l'aide de la boucle *for*.

```
for i in range(20):
    plt.subplot(4,5,i+1)
    plt.imshow(train_img[i],cmap='gray_r')
    plt.title("Digit : {}".format(train_lab[i]))
    plt.subplots_adjust(hspace=0.5)
    plt.axis('off')
```

**subplot()** est utilisé pour ajouter une sous-parcelle ou une structure de type grille à la figure actuelle. Le premier argument est pour « *non* », deuxième pour " *no. de lignes* » et le troisième pour l'index de position dans la grille.

**imshow()** est utilisé pour afficher les données sous forme d'image, c'est-à-dire d'image d'entraînement (*train\_img[i]*) alors que **cmap** représente la carte des couleurs. *Cmap* est une fonctionnalité facultative. Fondamentalement, si l'image est dans le tableau de forme (M, N), alors le **cmap** contrôle la carte de couleurs utilisée pour afficher les valeurs. *cmap='gray'* affichera l'image en niveaux de gris tandis que *cmap='gray\_r'* est utilisé pour afficher l'image en niveaux de gris inverses.

**title()** définit le titre de chaque image. Nous avons défini « Digit : *train\_lab[i]* » comme titre pour chaque image de la sous-parcelle.

**subplots\_adjust()** est utilisé pour ajuster la disposition des sous-parcelles. Afin de modifier l'espace prévu entre deux lignes, nous avons utilisé **hspace**. Si vous souhaitez modifier l'espace entre deux colonnes, vous pouvez utiliser **wspace**.

**Après cela, nous avons affiché la forme de la section de formation et de test.**

```
print('Training images shape : ',train_img.shape)
print('Testing images shape : ',test_img.shape)
```

**(60000,28,28)** signifie qu'il y a 60 000 images dans l'ensemble d'apprentissage et que chaque image a une taille de 28x28 pixels. De même, il y a 10 000 images de même taille dans l'ensemble de test.

Ainsi chaque image est de taille 28x28 soit 784 traits, et chaque trait représente l'intensité de chaque pixel de 0 à 255.

Vous pouvez utiliser *print(train\_img[0])* pour imprimer la première image de l'ensemble d'apprentissage sous la forme matricielle de 28x28.

Nous avons tracé la première image d'entraînement sur un histogramme. Avant la normalisation,

## Création du modèle

```
from keras.models import Sequential
from keras.layers import Flatten,Dense
```

```

model=Sequential()
input_layer= Flatten(input_shape=(28,28))
model.add(input_layer)
hidden_layer1=Dense(512,activation='relu')
model.add(hidden_layer1)
hidden_layer2=Dense(512,activation='relu')
model.add(hidden_layer2)
output_layer=Dense(10,activation='softmax')
model.add(output_layer)

```

Il existe 3 façons de créer un modèle dans Keras :

Le **modèle séquentiel** est très direct et simple. Il permet de construire un modèle couche par couche.

L' **API fonctionnelle** qui est une API facile à utiliser et complète qui prend en charge les architectures de modèles arbitraires. C'est le modèle Keras "force de l'industrie".

**Sous-classement de modèles** où vous implémentez tout à partir de zéro par vous-même.

Ici, nous avons utilisé le **modèle séquentiel**. Ce modèle a une couche d'entrée, une couche de sortie et deux couches cachées.

**Sequential()** est utilisé pour créer une couche du réseau en séquence.

**.add()** est utilisé ici pour ajouter la couche dans le modèle.

Dans la première couche (couche d'entrée), nous alimentons l'image en entrée. Puisque chaque image est de taille 28x28, nous avons donc utilisé **Flatten()** pour compresser l'entrée.

Nous avons utilisé **Dense()** dans les autres couches. Il garantit que chaque neurone de la couche précédente est connecté à chaque neurone de la couche suivante.

Le modèle est un simple réseau de neurones à deux couches cachées avec **512 neurones**. Une fonction d'activation d'unité linéaire de redresseur (**ReLU**) est utilisée pour les neurones dans les couches cachées. La meilleure chose à ce sujet est que son gradient est toujours égal à 1, de cette façon nous pouvons transmettre le maximum d'erreur à travers le réseau lors de la rétro-propagation.

La couche de sortie a 10 neurones, c'est-à-dire pour chaque classe de 0 à 9. Une **fonction d'activation softmax** est utilisée sur la couche de sortie pour transformer les sorties en valeurs de type probabilité.

**Remarque :** Vous pouvez ajouter plus de neurones dans les couches cachées. Vous pouvez même augmenter le non. de couches cachées dans le modèle pour augmenter l'efficacité. Cependant, cela prendra plus de temps pendant la formation.

## Compilation du réseau

```

#compiling the sequential model
model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy',
              metrics=['accuracy'])

```

Ensuite, nous devons compiler notre modèle. La compilation du modèle prend trois paramètres : optimiseur, perte et métriques. L'optimiseur contrôle le taux d'apprentissage. Nous utilisons **"adam"** comme optimiseur. C'est généralement un bon optimiseur à utiliser dans de nombreux cas. Il ajuste le rythme d'apprentissage tout au long de la formation.

Nous utiliserons **'Sparse\_Categorical\_Crossentropy'** pour notre fonction de perte car cela économise du temps en mémoire ainsi que des calculs car il utilise simplement un seul entier pour une classe, plutôt qu'un vecteur entier. Un score inférieur indique que le modèle fonctionne mieux.

Afin de déterminer la précision, nous utiliserons la **métrique "précision"** pour voir le score de précision sur l'ensemble de validation lorsque nous formerons le modèle.

### Former le modèle

```
model.fit(train_img, train_lab, epochs=100)
```

Nous allons former le modèle à l'aide de la **fonction fit()** . Il aura des paramètres tels que les données d'entraînement (train\_img), les étiquettes d'entraînement (train\_lab) et le nombre d'époques. Le nombre d'époques est le nombre de fois que le modèle parcourt les données. Plus nous courrons d'époques, plus le modèle s'améliorera, jusqu'à un certain point. Après ce point, le modèle cessera de s'améliorer à chaque époque.

Nous allons enregistrer le modèle sous **project.h5**

```
model.save('project.h5')
```

### Évaluer le modèle

```
loss_and_acc=model.evaluate(test_img, test_lab, verbose=2)
print("Test Loss", loss_and_acc[0])
print("Test Accuracy", loss_and_acc[1])
```

La méthode **model.evaluate()** *calcule la perte et toute métrique définie* lors de la compilation du modèle. Ainsi, dans notre cas, la précision est calculée sur les 10 000 exemples de test en utilisant les poids de réseau donnés par le modèle enregistré.

Verbose peut être 0, 1 ou 2. Par défaut, verbose est 1.

verbose = 0, signifie silencieux.

verbose = 1, qui inclut à la fois la barre de progression et une ligne par époque.

verbose = 2, une ligne par époque, c'est-à-dire n° d'époque/n° total. d'époques.

Après avoir évalué le modèle, nous allons maintenant vérifier le modèle pour la section de test.

```
plt.imshow(test_img[0], cmap='gray_r')
plt.title('Actual Value: {}'.format(test_lab[0]))
prediction=model.predict(test_img)
plt.axis('off')
print('Predicted Value: ', np.argmax(prediction[0]))
if (test_lab[0]==(np.argmax(prediction[0]))):
    print('Successful prediction')
else:
    print('Unsuccessful prediction')
```

**model.predict()** est utilisé pour faire des prédictions sur l'ensemble de test.

**np.argmax()** renvoie les indices des valeurs maximales le long d'un axe.