La reconnaissance des nombres manuscrites

Ensemble de données de classification de chiffres manuscrits MNIST :

L'ensemble de données MNIST est un acronyme qui signifie l'ensemble de données Modifié du National Institute of Standards and Technology. Il s'agit d'un ensemble de données de 60 000 petites images carrées en niveaux de gris de 28×28 pixels de chiffres manuscrits uniques entre 0 et 9. La tâche consiste à classer une image donnée d'un chiffre manuscrit dans l'une des 10 classes représentant des valeurs entières de 0 à 9, inclusivement. Il s'agit d'un ensemble de données largement utilisé et bien compris et, pour la plupart, est «résolu». Les modèles les plus performants sont les réseaux de neurones convolutifs d'apprentissage en profondeur qui atteignent une précision de classification supérieure à 99 %, avec un taux d'erreur compris entre 0,4 % et 0,2 % sur l'ensemble de données de test retenu.

Les bibliothèques utiliseés

Pour la réalisation de notre projet nous avons choisi le langage de programmation Python, ainsi que les différents bibliothèques Tensorflow, Keras, Numpy,mtplotlib.

La première des choses à faire consiste tout d'abord à configurer l'environnement et installer toutes les bibliothèques puis Importer les bibliothèques installées.

Importation des bibliothèques nécessaires :

On a importer le type de modèle séquentiel de Keras. Il s'agit simplement d'un empilement linéaire de couches de réseaux de neurones.

On a importer aussi les couches "principales" de Keras :

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten, Conv2D, MaxPooling2D

- Dense : régulateur de couche, chaque couche reçoit une entré de chaque neurone de la couche précédente.
- Flatten : ça permet de rendre un modèle sous forme d'une colonne.
- Dropout : elle permet d'éviter le sur apprentissage « overfitting » en désactivant une partie de neurones.

Pour un entrainement plus efficace sur les données d'image on a importé les couches CNN de Keras. Ce sont les couches convolutives qui nous aideront à former notre modèle.

- Convolution2D : agit comme un filtrage.
- MaxPooling2D : est utilisé pour réduire les dimensions spatiales du volume de sortie.

from keras.models import load_model **Les**

librairies utilisées pour la visualisation :

import matplotlib.pyplot as plt

Les librairies utilisées pour les tableaux multidimensionnels :

Import numpy as np

Les librairies utilisées pour l'interface

: from keras.models import load_model

from tkinter import * import tkinter as tk

import win32gui import os import cv2

from PIL import ImageGrab,

Image import numpy as np from

PIL import ImageTk, Image from

tkinter import filedialog

Importation de la base de donnée MNIST :

On doit charger l'ensemble de données MNIST à l'aide de l'API Keras.

On remarque que nous avons 60 000 images de train, 10 000 images de test, chaque image ayant une taille de 28*28 et un total de 10 classes prévisibles.

```
import tensorflow as tf

# Loading the dataset
mnist = tf.keras.datasets.mnist

# Divide into training and test dataset
(x_train, y_train),(x_test, y_test) = mnist.load_data()
```

```
x_train.shape
```

```
(60000, 28, 28)
```

```
y_train.shape
x_test.shape
y_test.shape
```

```
(10000,)
```

```
import matplotlib.pyplot as plt
plt.imshow(x_train[0])
plt.show()
```

Construire le modèle :

Nous allons construire un modèle séquentiel. Il s'agit simplement d'un empilement linéaire de couches de réseaux de neurones.

Ensuite, nous devons ajouter les différentes couches au modèle séquentiel comme suit :

```
# 64 -> number of filters, (3,3) -> size of each kernal,
model.add(Conv2D(64, (3,3), input_shape = x_trainr.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

### Second Convolution Layer
model.add(Conv2D(64, (3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

### Third Convolution Layer
model.add(Conv2D(64, (3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

### Fully connected layer 1
model.add(Flatten())
model.add(Dense(64))
model.add(Activation("relu"))

### Fully connected layer 2
model.add(Dense(32))
model.add(Activation("relu"))

### Fully connected layer 3, output layer must be equal to number of classes
model.add(Dense(10))
```

La méthode add effectue le travail d'ajout d'une couche au modèle Sequential. Dans l'API Conv2D, nous devons définir les paramètres suivants : units (nombre de noyaux/filtres), kernel_size : (3,3) (taille de chaque noyau) input_shape (28,28,1 forme du tableau d'entrée qu'il recevra).

La fonction Flatten convertit toutes les cartes de caractéristiques 2D en une seule couche dense. La couche finale a 10 neurones car nous en avons besoin pour prédire les scores pour 10 classes.

La compilation du modèle dans Keras est super facile et peut se faire avec le code suivant :

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=['accuracy'])
```

Les métriques qui doivent être surveillées au cours de ce processus d'apprentissage doivent être spécifiées sous forme de liste dans le paramètre metrics de la méthode de compilation.

Pour compiler, on utilise est maintenant temps de définir un optimiseur et une fonction de perte « Loss» pour ajuster le modèle. Puis il faut l'évaluer avec x_test et y_test en utilisant les instructions suivantes :

Optimiser : permet de réduire le poids des erreurs

Loss : désigne le taux d'erreur.

Accuracy : désigne le taux de précision.

Entraînement du modèle

Pour entraîner un modèle dans Keras, on appelle la méthode d'ajustement du modèle (fit()) avec les paramètres suivants : **epochs** : Le nombre d'époques

batch_size : Le nombre d'images dans chaque lot

validation_data : Le tuple de images de validation et étiquettes de validation

Evaluation du modèle :

Pour évaluer le modèle, on utlise la commande suivante :

```
# Evaluating the accuracy on the test data
test_loss, test_acc = model.evaluate(x_testr, y_test)
print("Test Loss on 10,000 test samples", test_loss)
print("Test Accuracy on 10,000 test samples", test_acc)
```

```
313/313 [==============================] - 5s 16ms/step - loss: 0.0560 - accuracy: 0.9839
Test Loss on 10,000 test samples 0.05595068261027336
Test Accuracy on 10,000 test samples 0.9839000105857849
```