Lab 2

Classification des chiffres manuscrits MNIST

Contenu

Dans ce laboratoire nous allons traiter je problème de classification des chiffres manuscrits du MNIST qui est un ensemble de données standard utilisé dans la vision par ordinateur et l'apprentissage en profondeur.

Ce problème, il peut être utilisé comme base pour apprendre et s'entraîner à développer, évaluer et utiliser des réseaux de neurones d'apprentissage en profondeur convolutifs pour la classification d'images à partir de zéro.

Cela comprend comment développer un harnais de test robuste pour estimer les performances du modèle, comment explorer les améliorations du modèle et comment enregistrer le modèle et le charger plus tard pour faire des prédictions sur de nouvelles données.

Vous allez découvrir comment développer un réseau de neurones convolutifs pour la classification des chiffres manuscrits à partir de zéro.

Après avoir terminé ce laboratoire, vous saurez :

• Comment développer un harnais de test pour développer une évaluation robuste d'un modèle et établir une référence de performance pour une tâche de classification.

• Comment explorer les extensions d'un modèle de base pour améliorer la capacité d'apprentissage et de modélisation.

• Comment développer un modèle finalisé, évaluer les performances du modèle final et l'utiliser pour faire des prédictions sur de nouvelles images.

Les sujets suivants seront traités dans ce laboratoire :

- Présentation de données MNIST
- Réseau multicouche profond
- Convolution
- Pooling
- Dropout
- Entraînement du modèle et test des modèles
- Comparer les résultats

2.1 Ensemble de données de classification de chiffres manuscrits MNIST

L'ensemble de données **MNIST** est un acronyme qui signifie l'ensemble de données **M**odifié du **N**ational **I**nstitute of **S**tandards and **T**echnology. 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.

L'exemple ci-dessous charge l'ensemble de données MNIST à l'aide de l'API Keras et crée un tracé des neuf premières images de l'ensemble de données d'apprentissage. Nous utiliserons les packages **`numpy`** , **`matplotlib`** , **`keras`** , **`scipy`** et **`tensorflow`** dans cet exercice. Ici, **TensorFlow** est utilisé comme backend pour **Keras**.

2.1.1 Importer les données

```
import numpy as np
# It is important that you set seed for reproducibility:
# set seed for reproducibility
seed_val = 9000
np.random.seed(seed_val)
#Importing the dataset for MNIST:
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

2.1.2 Explorer les données

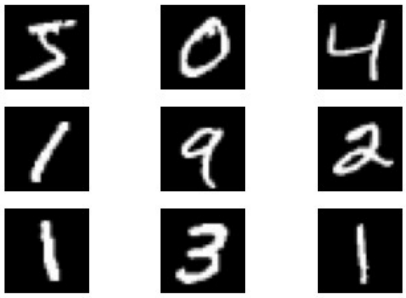
Maintenant que les données ont été importées, explorons ces chiffres :

```
print('Size of the training_set: ', X_train.shape)
print('Size of the test_set: ', X_test.shape)
print('Shape of each image: ', X_train[0].shape)
print('Total number of classes: ', len(np.unique(y_train)))
print('Unique class labels: ', np.unique(y_train))
```

```
Size of the training_set:  (60000, 28, 28)
Size of the test_set:  (10000, 28, 28)
Shape of each image:  (28, 28)
Total number of classes:  10
Unique class labels:  [0 1 2 3 4 5 6 7 8 9]
```

Nous pouvons voir 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**. Maintenant, traçons 9 chiffres manuscrits. Avant cela, nous devrons importer **`matplotlib`** pour tracer :

```
import matplotlib.pyplot as plt
# Plot of 9 random images
for i in range(0, 9):
      plt.subplot(3,3,1+i) # plot of 3 rows and 3 columns
      plt.axis('off') # turn off axis
      plt.imshow(X_train[i], cmap='gray')
      # gray scale
plt.show()
```



2.2 Définition des hyper-paramètres

Voici quelques-uns des **hyper-paramètres** que nous utiliserons tout au long de notre code. Ceux-ci sont totalement configurables :

```
# Number of epochs
epochs = 20
# Batchsizeo
batch_size = 128
# Optimizer for the generator
from tensorflow.keras.optimizers import Adam
optimizer = Adam(lr=0.01)
# try 0.0001
# Shape of the input image
input_shape = (28,28,1)
```

Si vous revenez au Lab 1 , vous verrez que l'optimiseur utilisé là-bas était **Adam** . Par conséquent, nous allons importer l'optimiseur Adam depuis le module **`keras`** et définir son **taux d'apprentissage**, comme indiqué dans le code précédent. Pour la plupart des cas qui suivront, nous nous entraînerons sur 20 époques pour faciliter la comparaison.

2.3 Construire et entraîner un simple réseau de neurones profonds

Maintenant que nous avons chargé les données en mémoire, nous devons créer un simple modèle de réseau neuronal pour prédire les chiffres MNIST.

2.3.1 Construire un modèle

Nous allons construire un modèle séquentiel. Alors, importons-le depuis Keras et initialisons-le avec le code suivant :

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

La prochaine chose que nous devons faire est de définir la couche **Dense**/**Perceptron**. Avec Keras, cela peut être fait en l'importation de la couche **`Dense`**, comme suit :

```
from keras.layers import Dense
```

Ensuite, nous devons ajouter la couche **`Dense`** au modèle séquentiel comme suit :

```
model.add(Dense(300, input_shape=(784,), activation = 'relu'))
```

La méthode **`add`** effectue le travail d'ajout d'une couche au modèle **`Sequential`**, dans ce cas, **`Dense`** .

Dans la couche **`Dense`** du code précédent, nous avons défini le **nombre de neurones** dans la **première couche cachée**, qui est de **300** . Nous avons également défini le paramètre **`input_shape`** comme étant égal à (**`784`**,) pour indiquer au modèle qu'il acceptera les **tableaux d'entrée de la forme** (**`784,`**).

Cela signifie que la couche d'entrée aura 784 neurones. Le type de fonction d'activation qui doit être appliqué au résultat peut être défini avec le **paramètre d'activation**. Dans ce cas, c'est **`relu`** .

Ajoutez une autre couche dense de 300 neurones en utilisant le code suivant :

```
model.add(Dense(300, activation='relu'))
```

Et la dernière couche **`Dense`** avec le code suivant :

```
model.add(Dense(10, activation='softmax'))
```

Ici, la **couche finale** a 10 neurones car nous en avons besoin pour prédire les scores pour **10 classes**. La fonction d'activation qui a été choisie ici est **`softmax`** afin que nous puissions limiter les scores entre 0 et 1, et la somme des scores à 1.

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

```
model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer ,
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.

Imprimez le résumé du modèle avec le code suivant :

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense (Dense)                (None, 300)               235500
 dense_1 (Dense)              (None, 300)               90300
 dense_2 (Dense)              (None, 10)                3010
=================================================================
```

```
Total params: 328,810
Trainable params: 328,810
Non-trainable params: 0
```

Notez que ce modèle a **328 810** paramètres pouvant être entraînés, ce qui est raisonnable. Maintenant, divisez les données d'entraînement (`train`) en données d'entraînement (`train`) et de validation en utilisant la fonction `train_test_split` que nous avons importée de `sklearn` :

```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, stratify =
y_train, test_size = 0.08333, random_state=42)
X_train = X_train.reshape(-1, 784)
X_val = X_val.reshape(-1, 784)
X_test = X_test.reshape(-1, 784)
print('Training Examples', X_train.shape[0])
print('Validation Examples', X_val.shape[0])
print('Test Examples', X_test.shape[0])
```

```
Training Examples 55000
Validation Examples 5000
Test Examples 10000
```

Nous avons divisé les données de manière à obtenir **55 000 images** pour **l'entraînement** et **5 000** images de **validation**.

Vous verrez également que nous avons remodelé les tableaux (`28,28,1`) pour que chaque image soit de forme (`784,`). C'est parce que nous avons défini le modèle pour accepter des images/arrays of **shape** (784,) .

Nous allons maintenant **entraîner** notre modèle sur **55 000** exemples d'entraînement, **valider** sur **5 000** exemples et **tester** sur **10 000** exemples.

Pendant l'entraînement, on stocke les variables, telles que la perte d'entraînement et de validation et la précision à chaque époque. Elle qui peuvent ensuite être utilisées pour tracer le processus d'apprentissage.

2.4 Entraînement d'un 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

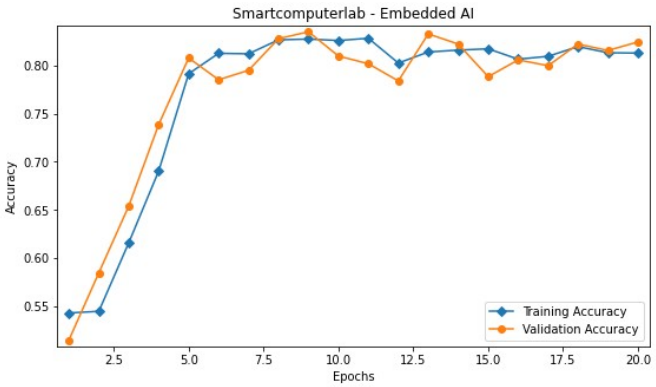
• **`validation_data`** : Le tuple de images de validation et étiquettes de validation

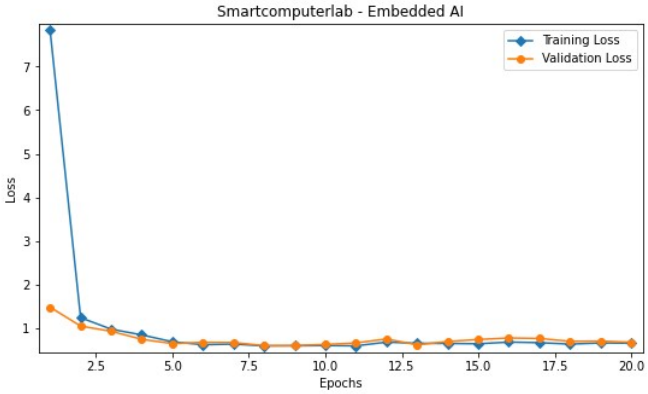
Consultez la section **Définition des hyper-paramètres** du laboratoire pour les valeurs définies des époques et du lot .

```
history = model.fit(X_train, y_train, epochs = epochs, batch_size=batch_size,
validation_data=(X_val, y_val))
# plot training loss
loss_plot(history)
```

Le résultat d'entraînement peut être affiché graphiquement par la fonction **`loss_plot()`**.

```
import matplotlib.pyplot as plt

def loss_plot(history):
train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
plt.figure(figsize=(9,5))
plt.plot(np.arange(1,len(train_acc)+1),train_acc,marker='D',label='Training Accuracy')
plt.plot(np.arange(1,len(train_acc)+1),val_acc,marker='o',label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Smartcomputerlab - Embedded AI')
plt.legend()
plt.margins(0.02)
plt.show()
train_loss = history.history['loss']
val_loss = history.history['val_loss']
plt.figure(figsize=(9,5))
plt.plot(np.arange(1,len(train_acc)+1),train_loss,marker='D',label='Training Loss')
plt.plot(np.arange(1,len(train_acc)+1),val_loss,marker='o',label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Smartcomputerlab - Embedded AI')
plt.legend()
plt.margins(0.02)
plt.show()
```





2.5 Modèle simple - Implémentation complète

Ce module met en œuvre la formation et l'évaluation d'un modèle multicouche simple (**`Dense`**) :

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

import matplotlib.pyplot as plt

def loss_plot(history):
train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
plt.figure(figsize=(9,5))
plt.plot(np.arange(1,len(train_acc)+1),train_acc,marker='D',label='Training Accuracy')
plt.plot(np.arange(1,len(train_acc)+1),val_acc,marker='o',label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Smartcomputerlab - Embedded AI')
plt.legend()
plt.margins(0.02)
plt.show()
train_loss = history.history['loss']
val_loss = history.history['val_loss']
plt.figure(figsize=(9,5))
plt.plot(np.arange(1,len(train_acc)+1),train_loss,marker='D',label='Training Loss')
plt.plot(np.arange(1,len(train_acc)+1),val_loss,marker='o',label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Smartcomputerlab - Embedded AI')
plt.legend()
plt.margins(0.02)
plt.show()
# Number of Convolution - Python fileepochs
epochs = 20
# Batchsize
batch_size = 128
# Optimizer for the generator
from tensorflow.keras.optimizers import Adam
optimizer = Adam(lr=0.0001) # in the receding example lr=0.01 !
# Shape of the input image
input_shape = (28,28,1)
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
stratify = y_train,
test_size = 0.08333,
random_state=42)
X_train = X_train.reshape(-1, 784)
X_val = X_val.reshape(-1, 784)
X_test = X_test.reshape(-1, 784)
model = Sequential()
model.add(Dense(300, input_shape=(784,), activation = 'relu'))
model.add(Dense(300, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=optimizer,
metrics = ['accuracy'])
history = model.fit(X_train, y_train, epochs = epochs, batch_size=batch_size,
validation_data=(X_val, y_val))
loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
loss_plot(history)
```

2.6 Convolution

La convolution peut être définie comme le processus consistant à parcourir un petit noyau/filtre/tableau sur un tableau cible et à obtenir la **somme de la multiplication** (produit scalaire) par élément entre le noyau et un sous-ensemble de taille égale du tableau cible **à cet emplacement**.

Considérez l'exemple suivant :

```
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
kernel = np.array([-1, 1, 0])
```

Ici, vous avez un **tableau cible de longueur 10** et un **noyau de longueur 3**. Lorsque vous démarrez la convolution, implémentez les étapes suivantes :

1. Le noyau sera multiplié par le sous-ensemble du tableau cible dans les indices 0 à 2. Ce sera entre [-1,1,0] (noyau) et [0,1,0] (de l'index 0 à 2 du tableau cible). Le résultat de cette multiplication élément par élément sera ensuite additionné pour obtenir ce qu'on appelle le **résultat de la convolution**.

2. Le noyau sera ensuite **échelonné d'une unité**, puis multiplié par le sous-ensemble du tableau cible dans les indices 1 à 3, comme à l'étape 1, et le résultat est obtenu.

L'étape 2 est répétée jusqu'à ce qu'un sous-ensemble égal à la longueur du noyau ne soit plus possible à un nouvel emplacement de foulée. Le résultat de la convolution à chaque foulée est stocké dans un tableau.

Ce tableau qui contient le résultat de la convolution s'appelle la **carte des caractéristiques**. La longueur de la carte de caractéristiques 1-D (avec pas/pas de 1) est égale à la différence de longueur du noyau et du tableau cible plus 1.

Dans ce cas seulement, nous devons prendre en compte l'équation suivante :

longueur de la carte des caractéristiques = longueur du tableau cible - longueur du noyau + 1

Voici un extrait de code implémentant la **convolution 1D** :

```
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
kernel = np.array([-1, 1, 0])
# empty feature map
conv_result = np.zeros(array.shape[0] - kernel.shape[0] +1).astype(int)
for i in range(array.shape[0] - kernel.shape[0] +1):
# convolving
conv_result[i] = (kernel * array[i:i+3]).sum()
print(kernel, '*', array[i:i+3], '=', conv_result[i])
print('Feature Map :', conv_result)
```

```
[-1  1  0] * [0 1 0] = 1
[-1  1  0] * [1 0 1] = -1
[-1  1  0] * [0 1 0] = 1
[-1  1  0] * [1 0 1] = -1
[-1  1  0] * [0 1 0] = 1
[-1  1  0] * [1 0 1] = -1
[-1  1  0] * [0 1 0] = 1
[-1  1  0] * [1 0 1] = -1
Feature Map : [ 1 -1  1 -1  1 -1  1 -1]
```

2.7 Convolution avec Keras

Maintenant que vous avez compris le fonctionnement de la convolution, utilisons-la et construisons un classificateur CNN sur les chiffres MNIST.

Pour cela, importez l'API **`Conv2D`** depuis le module couches de Keras. Vous pouvez le faire avec le code suivant :

```
from keras.layers import Conv2D
```

Puisque la convolution sera définie pour accepter des images de forme 28 * 28 * 1 , nous devons remodeler toutes les images pour qu'elles soient de 28 * 28 * 1 :

```
import numpy as np
from keras.datasets import mnist
# Shape of the input image
input_shape = (28,28,1)
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
stratify = y_train,
test_size = 0.08333,
random_state=42)
# reshape data
X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)
print('Train data shape:', X_train.shape)
print('Val data shape:', X_val.shape)
print('Test data shape:', X_test.shape)
```

```
Train data shape: (55000, 28, 28, 1)
Val data shape: (5000, 28, 28, 1)
Test data shape: (10000, 28, 28, 1)
```

Pour construire le modèle, comme nous l'avons fait précédemment, nous devons initialiser le modèle en tant que **`Sequential`** :

```
model = Sequential()
```

Now, add the **`Conv2D`** layer to the model with the following code:

```
model.add(Conv2D(32,kernel_size=(3,3),input_shape=input_shape,activation=
'relu'))
```

Dans l'API **`Conv2D`**, nous avons défini les paramètres suivants :

• **`units`** : 32 (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)

• activation **`relu`**

Le résultat de la convolution précédente est de **32 cartes de caractéristiques** de taille **`26*26`**.

Ces cartes de caractéristiques 2D doivent maintenant être converties en **une carte de caractéristiques 1D**. Cela peut être fait dans Keras avec le code suivant :

```
from keras.layers import Flatten
model.add(Flatten())
```

Le résultat de l'extrait précédent est comme une couche de neurones dans un simple réseau de neurones. La fonction **`Flatten`** convertit toutes les cartes de caractéristiques **2D** en une seule couche dense.

Dans cette couche, allons-nous ajouter une couche **Dense** avec 128 neurones :

```
model.add(Dense(128, activation = 'relu'))
```

Puisque nous devons obtenir des scores pour chacune des 10 classes possibles, nous devons ajouter une autre couche **Dense** avec 10 neurones, avec **softmax** comme **fonction d'activation** :

```
model.add(Dense(10, activation = 'softmax'))
```

Maintenant, tout comme dans le cas du réseau de neurones **Dense** simple que nous avons construit dans le code précédent, nous allons compiler et entraîner le modèle:

```
# compile model
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=optimizer,
metrics = ['accuracy'])
# print model summary
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320
 flatten (Flatten)           (None, 21632)             0
 dense_5 (Dense)             (None, 128)               2769024
 dense_6 (Dense)             (None, 10)                1290
=================================================================
Total params: 2,770,634
Trainable params: 2,770,634
Non-trainable params: 0
```

D'après le résumé du modèle, nous pouvons voir que ce classificateur de convolution a 2 770 634 paramètres. C'est beaucoup de paramètres par rapport au modèle Perceptron (MLP).

Entraînons ce modèle et évaluons ses performances.

```
# fit model
history = model.fit(X_train, y_train, epochs = epochs, batch_size=batch_size,
validation_data=(X_val, y_val))
model.save('Lab2.mnist.conv.h5')
```

```
..
Epoch 19/20
430/430 [==============================] - 4s 9ms/step - loss: 0.0022 - accuracy: 0.9995 - val_loss:
0.1278 - val_accuracy: 0.9796
Epoch 20/20
430/430 [==============================] - 4s 9ms/step - loss: 0.0013 - accuracy: 0.9996 - val_loss:
0.1195 - val_accuracy: 0.9786
```

2.7.1 Evaluation du modèle

Vous pouvez évaluer le modèle de convolution sur les données de test avec le code suivant :

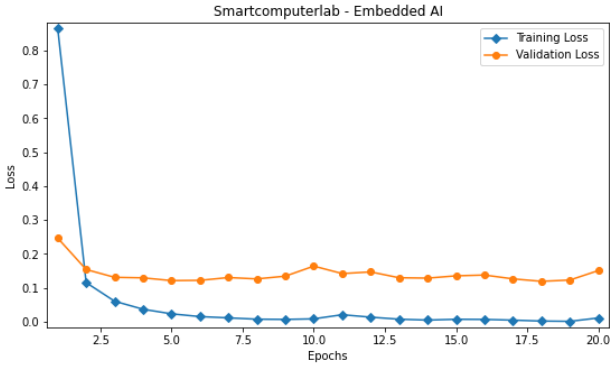
```
loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
```

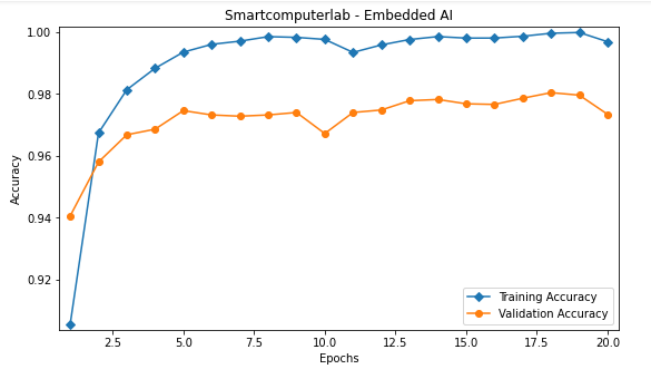
```
313/313 [==============================] - 1s 4ms/step - loss: 0.1689 -
accuracy: 0.9814
Test loss: 0.1689450442790985
Accuracy: 0.9814000129699707
```

Nous pouvons voir que le modèle est précis à 97,15 % sur les données de test, à 97,86 % sur les données de validation et à 99,96 % sur les données de l'entraînement. Il ressort également clairement de la perte que le modèle est légèrement sur-ajusté par rapport aux données de l'entraînement. Nous parlerons plus tard de la

façon de gérer le sur-apprentissage. Maintenant, traçons les métriques d'entraînement et de validation pour voir comment l'entraînement a progressé :

```
# plot training loss
loss_plot(history)
```





Précision et perte pour le modèle de convolution et `lr=0,0001` taux d'apprentissage

2.7.2 Implémentation complète

Le module suivant implémente la formation et l'évaluation d'un classificateur de convolution.

```
import numpy as np
from keras.layers import Conv2D
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import MaxPool2D
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
#from loss_plot import loss_plot
# Shape of the input image
input_shape = (28,28,1)
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_val, y_train, y_val =
train_test_split(X_train,y_train,stratify=y_train,test_size=0.08333,random_state=42)
# reshape data
X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)
print('Train data shape:', X_train.shape)
print('Val data shape:', X_val.shape)
print('Test data shape:', X_test.shape)
model = Sequential()
optimizer = Adam(lr=0.0001)
model.add(Conv2D(32,kernel_size=(3,3),input_shape=input_shape,activation='relu'))
#model.add(MaxPool2D(2,2))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
# compile model
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=optimizer,metrics = ['accuracy'])
# print model summary
model.summary()
# Number of epochs
epochs = 20
# Batchsize
batch_size = 128
history = model.fit(X_train, y_train,
epochs=epochs,batch_size=batch_size,validation_data=(X_val, y_val))
loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
loss_plot(history)
```

2.8 Pooling

La mise en commun maximale (**max pooling**) peut être définie comme le processus de récapitulation d'un groupe de valeurs avec la valeur maximale au sein de ce groupe. De même, si vous calculiez la moyenne, il s'agirait d'une mise en commun moyenne. Les opérations de **pooling** sont généralement effectuées sur les cartes de caractéristiques générées après convolution **pour réduire le nombre de paramètres**.

Prenons l'exemple de tableau que nous avons considéré pour la convolution :

```
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
```

Maintenant, si vous deviez effectuer une **mise en pool maximale** sur ce tableau avec la taille du pool définie sur la taille 12 et une foulée de 2, le résultat serait un tableau de `[1,1,1,1,1]`. Le tableau de taille 10 a été réduit à une taille de 5 en raison de la mise en commun maximale. Ici, puisque la taille du pool est de forme 2, vous prendriez le sous-ensemble du tableau cible de l'index 0 à l'index 2, qui sera `[0,1]`, et calculeriez le maximum de ce sous-ensemble comme 1.

Vous feriez le idem pour le sous-ensemble de l'indice 2 à l'indice 4, de l'indice 4 à l'indice 6, de l'indice 6 à l'indice 8, et enfin de l'indice 8 à 10. De même, une mutualisation moyenne peut être mise en œuvre en calculant la valeur moyenne de la **section poolée**.

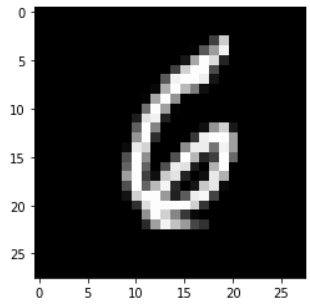
Dans ce cas, il en résulterait le tableau `[0.5, 0.5, 0.5, 0.5, 0.5]`.

Voici quelques extraits de code qui implémentent la mise en commun maximale et moyenne :

```
# 1D Max Pooling
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
result = np.zeros(len(array)//2)
for i in range(len(array)//2):
result[i] = np.max(array[2*i:2*i+2])
print(result)

# 1D Max Pooling
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
result = np.zeros(len(array)//2)
for i in range(len(array)//2):
result[i] = np.mean(array[2*i:2*i+2])
print(result)

[1. 1. 1. 1. 1.]
[0.5 0.5 0.5 0.5 0.5]
```

Considérez le code suivant pour un chiffre :

```
plt.imshow(X_train[0].reshape(28,28), cmap='gray')
```

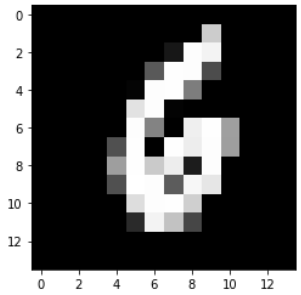


Cette image est de forme **28x28 .** Maintenant, si vous deviez effectuer une opération de regroupement de 2x2 max, l'image résultante aurait une forme de **14x14** .

Maintenant, écrivons une fonction pour implémenter une opération de pooling **2*2 max** sur un chiffre MNIST :

```
def square_max_pool(image, pool_size=2):
result = np.zeros((14,14))
for i in range(result.shape[0]):
```

```
for j in range(result.shape[1]):
result[i,j] = np.max(image[i*pool_size : i*pool_size+pool_size,j*pool_size :
j*pool_size+pool_size])
return result
# plot a pooled image
plt.imshow(square_max_pool(X_train[0].reshape(28,28)), cmap='gray')
plt.show()
```



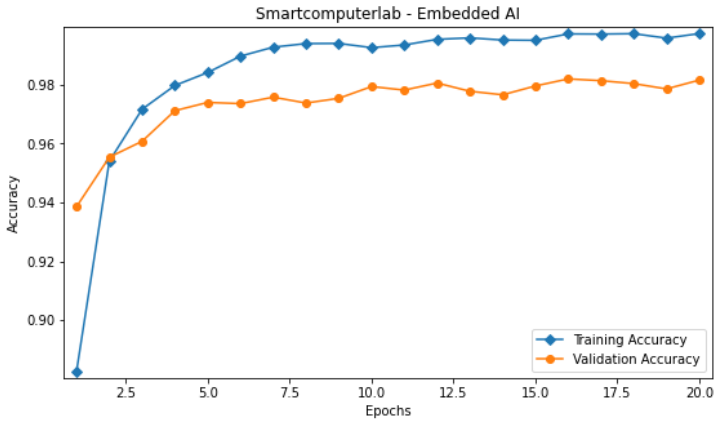
Vous avez peut-être remarqué que le classificateur à convolution que nous avons construit dans la section précédente contient environ **2,7 millions de paramètres**. Il a été prouvé que le fait d'avoir beaucoup de paramètres peut conduire à un sur-apprentissage dans de nombreux cas. C'est là qu'intervient la mise en commun. Elle nous aide à conserver les caractéristiques importantes des données ainsi qu'à réduire le nombre de paramètres.

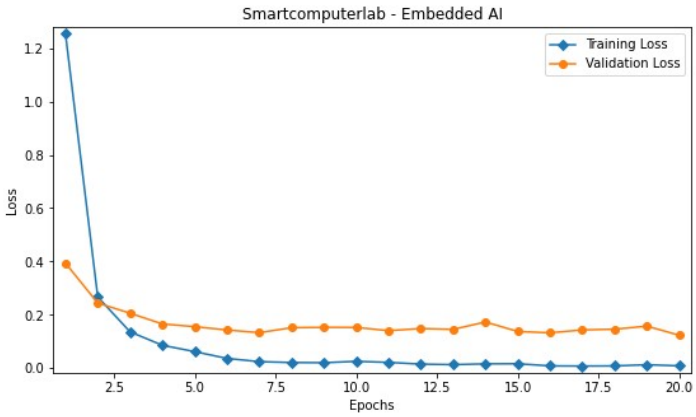
Implémentons maintenant un classificateur à convolution avec **pooling maximal**.

```
import numpy as np
from keras.layers import Conv2D
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import MaxPool2D
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Shape of the input image
input_shape = (28,28,1)
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_val, y_train, y_val =
train_test_split(X_train,y_train,stratify=y_train,test_size=0.08333,random_state=42)
# reshape data
X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)
print('Train data shape:', X_train.shape)
print('Val data shape:', X_val.shape)
print('Test data shape:', X_test.shape)
model = Sequential()
optimizer = Adam(lr=0.0001)
model.add(Conv2D(32,kernel_size=(3,3),input_shape=input_shape,activation='relu'))
model.add(MaxPool2D(2,2))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
# compile model
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=optimizer,metrics = ['accuracy'])
# print model summary
model.summary()
# Number of epochs
epochs = 20
# Batchsize
batch_size = 128
history = model.fit(X_train, y_train,
epochs=epochs,batch_size=batch_size,validation_data=(X_val, y_val))
loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
loss_plot(history)
```

```
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)            (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D)  (None, 13, 13, 32)          0
```

```
 )

 flatten_3 (Flatten)          (None, 5408)              0

 dense_14 (Dense)             (None, 128)           692352

 dense_15 (Dense)             (None, 10)              1290

=================================================================
Total params: 693,962
Trainable params: 693,962
Non-trainable params: 0

..
Epoch 19/20
430/430 [==============================] - 4s 9ms/step - loss: 0.0129 - accuracy: 0.9959 - val_loss:
0.1577 - val_accuracy: 0.9786
Epoch 20/20
430/430 [==============================] - 4s 8ms/step - loss: 0.0087 - accuracy: 0.9974 - val_loss:
0.1232 - val_accuracy: 0.9816
313/313 [==============================] - 1s 4ms/step - loss: 0.1268 - accuracy: 0.9803
Test loss: 0.12684263288974762
Accuracy: 0.9803000092506409
```






```
loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
```

```
313/313 [==============================] - 1s 4ms/step - loss: 0.1268 -
accuracy: 0.9803
Test loss: 0.12684263288974762
Accuracy: 0.9803000092506409
```

2.8.1 Exercice

Commentez les différences entre le modèle de **conv** simple et le modèle de **conv** avec **pooling**.

2.9 Dropout

Un seul modèle peut être utilisé pour simuler un grand nombre d'architectures de réseau différentes en supprimant de manière aléatoire des nœuds pendant l'apprentissage. C'est ce qu'on appelle l'abandon (**dropout**) et offre une méthode de régularisation très bon marché et remarquablement efficace pour réduire le sur-apprentissage et améliorer l'erreur de généralisation dans les réseaux de neurones profonds de toutes sortes.

2.9.1 Problème de sur-apprentissage

Les grands réseaux de neurones entraînés sur des ensembles de données relativement petits peuvent surcharger les données d'entraînement. Cela a pour effet que le modèle apprend le bruit statistique dans les données d'apprentissage, ce qui entraîne de mauvaises performances lorsque le modèle est évalué sur de nouvelles données, par ex. un jeu de données de test. L'erreur de généralisation augmente en raison du sur-apprentissage.

Une approche pour réduire le sur-apprentissage consiste à ajuster tous les différents réseaux de neurones possibles sur le même ensemble de données et à faire la moyenne des prédictions de chaque modèle.

Ce n'est pas faisable dans la pratique et peut être approché en utilisant une **petite collection de différents modèles**, appelée un **ensemble**.

Un problème, même avec l'approximation d'ensemble, est qu'elle nécessite l'ajustement et le stockage de plusieurs modèles, ce qui peut être un défi si les modèles sont volumineux, nécessitant des jours ou des semaines pour s'entraîner et se régler.

2.9.2 Suppression aléatoire de nœuds

Dropout est une méthode de régularisation qui se rapproche de la formation d'un grand nombre de réseaux de neurones avec différentes architectures en parallèle. Pendant l'entraînement, un certain nombre de sorties de couche sont ignorées de manière aléatoire ou « abandonnées ». Cela a pour effet de faire en sorte que la couche ressemble et soit traitée comme une couche avec un nombre de nœuds et une connectivité différents par rapport à la couche précédente.

En effet, chaque mise à jour d'une couche pendant l'apprentissage est effectuée avec une « vue » différente de la couche configurée. Par abandon d'une unité, nous entendons la retirer temporairement du réseau, ainsi que toutes ses connexions entrantes et sortantes.

Cette conceptualisation suggère que peut-être l'abandon rompt les situations où les couches réseau s'adaptent pour corriger les erreurs des couches précédentes, ce qui rend le modèle plus robuste.

2.9.2.1 Comment abandonner

L'abandon est implémenté par couche dans un réseau de neurones. Il peut être utilisé avec la plupart des types de couches, telles que les couches denses entièrement connectées, les couches convolutives et les couches récurrentes telles que la couche réseau de mémoire à long terme.

L'abandon peut être mis en œuvre sur une ou toutes les couches cachées du réseau ainsi que sur la couche visible ou d'entrée. **Il n'est pas utilisé sur la couche de sortie.**

Un nouvel hyper-paramètre est introduit qui spécifie la **probabilité** à laquelle les sorties de la couche sont **abandonnées**, ou inversement, la probabilité à laquelle les sorties de la couche sont conservées.

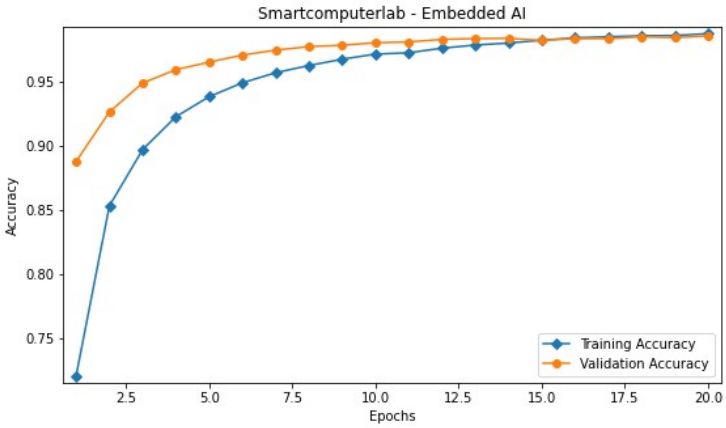
Une valeur commune est une probabilité de 0,5 pour conserver la sortie de chaque nœud dans une couche cachée et une valeur proche de 1,0, telle que 0,8, pour conserver les entrées de la couche visible.

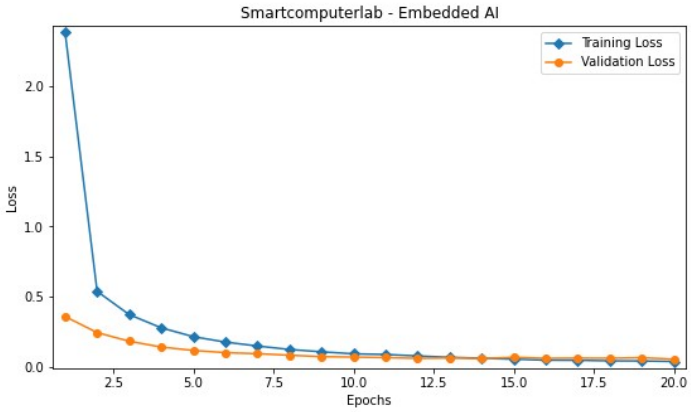
2.9.2.2 Une implémentation complète

```
import numpy as np
from keras.layers import Conv2D
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import MaxPool2D
from keras.layers import Dropout
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Shape of the input image
input_shape = (28,28,1)
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_val, y_train, y_val =
train_test_split(X_train,y_train,stratify=y_train,test_size=0.08333,random_state=42)
# reshape data
X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)
print('Train data shape:', X_train.shape)
print('Val data shape:', X_val.shape)
print('Test data shape:', X_test.shape)
model = Sequential()
optimizer = Adam(lr=0.0001)
model.add(Conv2D(32,kernel_size=(3,3),input_shape=input_shape,activation='relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation = 'softmax'))
# compile model
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=optimizer,metrics = ['accuracy'])
# print model summary
model.summary()
# Number of epochs
epochs = 20
# Batchsize
batch_size = 128
history = model.fit(X_train, y_train,
epochs=epochs,batch_size=batch_size,validation_data=(X_val, y_val))
loss,acc = model.evaluate(X_test, y_test)
model.save('Lab2.conv.dropout.h5')
print('Test loss:', loss)
print('Accuracy:', acc)
loss_plot(history)
```

```
Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 26, 26, 32)        320

 max_pooling2d_1 (MaxPooling  (None, 13, 13, 32)       0
 2D)

 dropout (Dropout)           (None, 13, 13, 32)        0

 flatten_4 (Flatten)         (None, 5408)              0

 dense_16 (Dense)            (None, 128)               692352

 dropout_1 (Dropout)         (None, 128)               0

 dense_17 (Dense)            (None, 10)                1290

=================================================================
Total params: 693,962
Trainable params: 693,962
Non-trainable params: 0
```

```
..
Epoch 19/20
430/430 [==============================] - 4s 10ms/step - loss: 0.0419 - accuracy: 0.9862 -
val_loss: 0.0651 - val_accuracy: 0.9846
Epoch 20/20
430/430 [==============================] - 4s 9ms/step - loss: 0.0377 - accuracy: 0.9877 - val_loss:
0.0541 - val_accuracy: 0.9858
313/313 [==============================] - 1s 4ms/step - loss: 0.0584 - accuracy: 0.9859
Test loss: 0.0584438182413578
Accuracy: 0.9858999848365784
```






```
loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
```

```
313/313 [==============================] - 1s 4ms/step - loss: 0.0584 -
accuracy: 0.9859
Test loss: 0.0584438182413578
Accuracy: 0.9858999848365784
```

Exercice – écrire un résumé comparatif pour les modèles élaborés :

Modèle **convolutive simple**:Test loss, Test accuracy, ..

Modèle **convolutive avec pooling** :Test loss, Test accuracy, ..

Modèle **convolutive avec dropout** :Test loss, Test accuracy, ..