Lab 2: Développement des modèles CNN pour la classification des chiffres manuscrits du MNIST

Le problème de classification des chiffres manuscrits du MNIST est un ensemble de données standard utilisé en **vision par ordinateur** et en **apprentissage profond**.

Ce laboratoire à été préparé pour apprendre et pratiquer comment développer, évaluer et utiliser des réseaux de neurones d'apprentissage profond convolutionnels pour la classification d'images à partir de zéro. Cela comprend comment développer un 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 les nouvelles données.

Dans ce laboratoire, vous découvrirez comment développer un réseau neuronal convolutif pour la classification des chiffres manuscrits à partir de zéro.

Après avoir terminé ce laboratoire, vous saurez:

• Comment développer un ensemble de test pour développer une évaluation robuste d'un modèle et établir une ligne de base de performance pour une tâche de classification.

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

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

Le matériel: **`Jetson Nano, Jetson Xavier`** (mode serveur)

Le logiciel: **`Python3`**, **`TensorFlow 2.0`** et **`Keras 2.3.`**

Présentation du laboratoire

Ce laboratoire est divisé en cinq parties; elles sont:

1. Ensemble de données de classification manuscrite du MNIST

2. Méthodologie d'évaluation du modèle

3. Comment développer un modèle de référence

4. Comment développer les modèles améliorés

5. Comment finaliser le modèle et faire des prédictions

6. Comment créer et mettre en œuvre le modèle « standard » - **LeNet-5**

Table of Contents

1. Ensemble de données de classification manuscrite du MNIST

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

Il s'agit d'un ensemble de données de **60000** petites images carrées de **28×28** pixels en niveaux de gris de chiffres manuscrits compris 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.

A noter que les modèles les plus performants sont des réseaux de neurones convolutionnels 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.

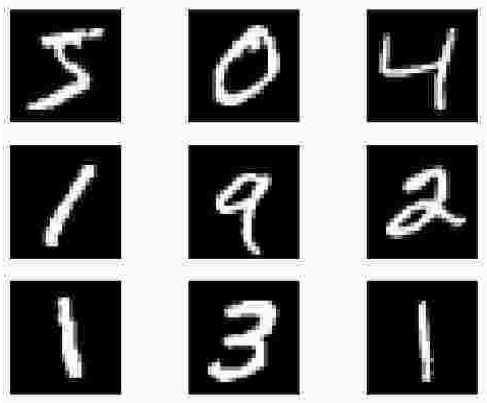
L'exemple ci-dessous charge l'ensemble de données **MNIST** à l'aide de l'API **Keras** et crée un tracé des n**euf premières images** de l'ensemble de données d'apprentissage.

```
# example of loading the mnist dataset
from keras.datasets import mnist
from matplotlib import pyplot
# load dataset
(trainX, trainy), (testX, testy) = mnist.load_data()
# summarize loaded dataset
print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))
print('Test: X=%s, y=%s' % (testX.shape, testy.shape))
# plot first few images
for i in range(9):
	# define subplot
	pyplot.subplot(3,3,0 + 1 + i)
	# plot raw pixel data
	pyplot.imshow(trainX[i], cmap=pyplot.get_cmap('gray'))
# show the figure
pyplot.show()
```

Nous pouvons voir qu'il y a 60000 exemples dans le jeu de données d'apprentissage et 10000 dans le jeu de données de test et que les images sont en effet carrées avec **28×28** pixels.

```
Train: X = (60000, 28, 28), y = (60000,) Test: X = (10000, 28, 28), y = (10000,)
```

Un tracé des neuf premières images de l'ensemble de données est également créé, montrant la nature manuscrite des images à classer.



2. Méthodologie d'évaluation du modèle

Bien que l'ensemble de données MNIST soit effectivement résolu, il peut être un point de départ utile pour **développer et pratiquer une méthodologie** pour résoudre des tâches de **classification d'images** à l'aide de **réseaux de neurones convolutifs**.

Dans ce laboratoire nous allons développer un nouveau modèle à partir de zéro. Nous pouvons utiliser l'ensemble de données d'entrainement et de test bien défini disponible sur le site de MNIST.

Afin d'estimer les performances d'un modèle pour un cycle d'entraînement donné, nous pouvons diviser l'ensemble d'entraînement en plusieurs séries d'entrainements. Les performances du modèle en entrainement sur chaque série peuvent ensuite être tracées pour fournir des courbes d'apprentissage et un aperçu de la façon dont un modèle apprend le problème.

L'API **`Keras`** prend en charge cela en spécifiant l'argument **`validation_data`** dans la fonction **`model.fit()`** lors de la formation du modèle, qui, à son tour, renverra un objet qui décrit les performances du modèle et les métriques à chaque **époque** de formation (entrainement).

```
# record model performance on a validation dataset during training
history = model.fit(..., validation_data=(valX, valY))
```

Afin d'estimer les performances d'un modèle sur le problème en général, nous pouvons utiliser plusieurs fois (5) la **validation croisée k (*k-fold*)**. Cela rendra compte de la **variance des modèles** à la fois en ce qui concerne les différences dans les ensembles de données d'apprentissage et de test, et en termes de **nature stochastique de l'algorithme d'apprentissage**. La performance d'un modèle peut être considérée comme la performance moyenne entre les **plis k** (***k-fold***), étant donné l'écart-type, qui pourrait être utilisée pour estimer un i**ntervalle de confiance**.

Nous pouvons utiliser la classe **`KFold`** de l'API scikit-learn pour implémenter l'évaluation de validation croisée **`k-fold`** d'un modèle de réseau neuronal donné. Il existe de nombreuses façons d'y parvenir, bien que nous puissions choisir une approche flexible où la classe **`KFold`** n'est utilisée que pour spécifier les index de ligne utilisés pour chaque fragment du code.

```
# example of k-fold cv for a neural net
data = ...
# prepare cross validation
kfold = KFold(5, shuffle=True, random_state=1)
# enumerate splits
for train_ix, test_ix in kfold.split(data):
        model = ...
      …
```

Nous retiendrons l'ensemble de données de test réel et l'utiliserons comme une évaluation de notre modèle final.

3. Comment développer un modèle de référence

La première étape consiste à développer un modèle de base.

Ceci est essentiel car il implique à la fois de développer l'infrastructure de la base de test afin que tout modèle que nous concevons puisse être évalué sur l'ensemble de données, et il établit une **base de référence** pour les performances du modèle , par laquelle toutes les améliorations peuvent être comparées.

La conception de la base de test est modulaire et nous pouvons développer une fonction distincte pour chaque partie. Cela donne un aspect donné modulaire et interchangéable.

Nous pouvons développer ce modèle de réference avec **cinq éléments clés**.

1. Chargement de l'ensemble de données
2. Préparation de l'ensemble de données
3. Définition du modèle
4. L'évaluation du modèle
5. Présentation des résultats

3.1 Charger le jeu de données

Nous savons certaines choses sur l'ensemble de données.

Par exemple, nous savons que les images sont toutes pré-alignées (par exemple, chaque image ne contient qu'un chiffre dessiné à la main), que les images ont toutes la même taille carrée de 28 × 28 pixels et que les images sont en niveaux de gris.

Par conséquent, nous pouvons charger les images et remodeler les tableaux de données pour avoir un seul canal de couleur.

```
# load dataset
(trainX, trainY), (testX, testY) = mnist.load_data()
# reshape dataset to have a single channel
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))
```

Nous savons également qu'il existe **10 classes** et que les classes sont représentées sous forme d'entiers uniques.

Nous pouvons donc utiliser un **codage à chaud** (***one hot encoding***) pour l'élément de classe de chaque échantillon, transformant l'entier en un **vecteur binaire à 10 éléments** avec un 1 pour l'index de la valeur de classe et 0 pour toutes les autres classes. Nous pouvons y parvenir avec la fonction utilitaire **`to_categorical ()`**.

```
# one hot encode target values
trainY = to_categorical(trainY)
testY = to_categorical(testY)
```

```
The load_dataset() function implements these behaviors and can be used to load
the dataset.
# load train and test dataset
def load_dataset():
      # load dataset
      (trainX, trainY), (testX, testY) = mnist.load_data()
      # reshape dataset to have a single channel
      trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
      testX = testX.reshape((testX.shape[0], 28, 28, 1))
      # one hot encode target values
      trainY = to_categorical(trainY)
      testY = to_categorical(testY)
      return trainX, trainY, testX, testY
```

3.2 Préparer les données de pixels

Nous savons que les valeurs de pixels pour chaque image dans l'ensemble de données sont des entiers non signés dans la plage entre le **noir** et le **blanc**, ou **0** et **255**.

Nous ne connaissons pas la meilleure façon de mettre à l'échelle les valeurs des pixels pour la modélisation, mais nous savons qu'une certaine mise à l'échelle sera nécessaire.

Un bon point de départ est de **normaliser** les valeurs de pixels des images en niveaux de gris, par ex. remettez-les à l'échelle **[0,1]**. Cela implique tout d'abord de convertir le type de données d'**entiers non signés** en **flottants**, puis de **diviser** les valeurs de pixels par la **valeur maximale**.

```
# convert from integers to floats
train_norm = train.astype('float32')
test_norm = test.astype('float32')
# normalize to range 0-1
train_norm = train_norm / 255.0
test_norm = test_norm / 255.0
```

La **fonction** `prep_pixels ()` ci-dessous implémente ces comportements et est fournie avec les valeurs de pixels pour les jeux de données d'entrainement et de test qui devront être mis à l'échelle.

```
# scale pixels
def prep_pixels(train, test):
	# convert from integers to floats
	train_norm = train.astype('float32')
	test_norm = test.astype('float32')
	# normalize to range 0-1
	train_norm = train_norm / 255.0
	test_norm = test_norm / 255.0
	# return normalized images
	return train_norm, test_norm
```

Cette fonction doit être appelée pour préparer les valeurs des pixels avant toute modélisation.

3.3 Définir le modèle

Ensuite, nous devons définir un modèle de réseau neuronal convolutionnel de base.

Le modèle a **deux aspects principaux**:

1. **le frontal d'extraction** d'entités composé de couches convolutionnelles et de mise en commun
2. **le *backend* classificateur** qui fera une **prédiction**

Pour la partie frontale convolutionnelle, nous pouvons commencer avec une seule **couche convolutionnelle** avec une petite taille de **filtre** `(3,3)` et un nombre modeste de filtres **(32)** suivie d'une couche de regroupement maximale (**MaxPool)** . Les projections de filtres peuvent ensuite être aplaties (***flatened***) pour fournir des fonctionnalités au classificateur.

Étant donné que nontre tâche est la classification multi-classes, nous savons que nous aurons besoin d'une couche de sortie à **10 nœuds** afin de prédire la distribution de probabilité d'une image appartenant à chacune des 10 classes. Cela nécessitera également l'utilisation d'une fonction d'activation **`softmax`**.

Entre l'extracteur d'entités et la couche de sortie, nous pouvons ajouter une couche **`Dense`** pour interpréter les entités ; dans ce cas nous utiliserons une couche **`Dense`** avec 100 nœuds.

Toutes les couches utiliseront la fonction d'activation **`ReLU`** et le schéma d'initialisation du poids **`He`**, les deux meilleures pratiques.

Nous utiliserons une configuration conservatrice pour l'optimiseur de **descente de gradient stochastique** avec un **taux d'apprentissage de 0,01** et un **momentum de 0,9**. La fonction de perte d'entropie croisée catégorielle sera optimisée, adaptée à la classification multi-classes, et nous surveillerons la métrique de précision de la classification, ce qui est approprié étant donné que nous avons **le même nombre d'exemples dans chacune des 10 classes**.

La fonction **`define_model ()`** ci-dessous va définir et retourner ce modèle.

```
# define cnn model
def define_model():
	model = Sequential()
	model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
	model.add(MaxPooling2D((2, 2)))
	model.add(Flatten())
	model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
	model.add(Dense(10, activation='softmax'))
	# compile model
	opt = SGD(lr=0.01, momentum=0.9)
	model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
	return model
```

3.4 Évaluer le modèle

Une fois le modèle défini, nous devons l'évaluer.

Le modèle sera évalué en utilisant une **validation croisée cinq fois**. La valeur de **k = 5** a été choisie pour fournir une base de référence à la fois pour une **évaluation répétée** et pour ne pas être suffisamment grande pour nécessiter une longue durée de fonctionnement. Chaque ensemble de tests représentera **20%** de l'ensemble de données d'apprentissage, soit environ **12 000** exemples, proche de la taille de l'ensemble de tests réel pour ce problème.

Le jeu de données d'apprentissage est mélangé avant d'être portionné, et le mélange d'échantillons est effectué à chaque fois, de sorte que tout modèle que nous évaluons aura le même jeux de données d'entrainement et de test dans chaque **pli** (***fold***).

Nous entrainerons le modèle de base avec 10 époques avec une taille de **lot** (***batch***) par défaut de 32 exemples. L'ensemble de tests pour chaque **pli** sera utilisé pour évaluer le modèle à la fois à chaque époque de la séance d'entraînement, afin que nous puissions plus tard créer des courbes d'apprentissage pour estimer les performances du modèle.

Nous garderons une trace de **l'historique** résultant de chaque exécution, ainsi que de la précision de classification du **pli**.

La fonction **`evaluate_model ()`** ci-dessous implémente ces comportements, en prenant le **jeu de données d'apprentissage** comme arguments et en renvoyant une liste de scores de précision et d'histoires d'apprentissage qui peuvent être résumés ultérieurement.

```
# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
	scores, histories = list(), list()
	# prepare cross validation
	kfold = KFold(n_folds, shuffle=True, random_state=1)
	# enumerate splits
	for train_ix, test_ix in kfold.split(dataX):
		# define model
		model = define_model()
		# select rows for train and test
		trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix],
dataX[test_ix], dataY[test_ix]
		# fit model
		history = model.fit(trainX, trainY, epochs=10, batch_size=32,
validation_data=(testX, testY), verbose=0)
		# evaluate model
		_, acc = model.evaluate(testX, testY, verbose=0)
		print('> %.3f' % (acc * 100.0))
		# stores scores
		scores.append(acc)
		histories.append(history)
	return scores, histories
```

3.5 Présentation des résultats

Une fois le modèle évalué, nous pouvons présenter les résultats.

Il y a deux aspects clés à présenter: le diagnostic du **comportement d'apprentissage** du modèle pendant la formation et l'**estimation des performances** du modèle. Ceux-ci peuvent être implémentés à l'aide de fonctions distinctes.

Tout d'abord, les diagnostics impliquent la création d'un tracé de ligne montrant les performances du modèle en entrainement et l'ensemble de test pendant chaque **pli** de la validation croisée. Ces tracés sont utiles pour se faire une idée si un modèle est **sur-ajusté**, **sous-ajusté** ou bien adapté à l'ensemble de données.

Nous allons créer une seule figure avec deux sous-tracés, une pour la perte (***loss***) et une pour la précision. Les lignes bleues indiqueront les performances du modèle sur l'ensemble de données d'apprentissage et les lignes orange indiqueront les performances sur l'ensemble de données de test. La fonction **`summarize_diagnostics()`** ci-dessous crée et affiche ce tracé en fonction des historiques d'entraînement.

```
# plot diagnostic learning curves
def summarize_diagnostics(histories):
	for i in range(len(histories)):
		# plot loss
		pyplot.subplot(2, 1, 1)
		pyplot.title('Cross Entropy Loss')
		pyplot.plot(histories[i].history['loss'], color='blue',
label='train')
		pyplot.plot(histories[i].history['val_loss'], color='orange',
label='test')
		# plot accuracy
		pyplot.subplot(2, 1, 2)
		pyplot.title('Classification Accuracy')
		pyplot.plot(histories[i].history['accuracy'], color='blue',
label='train')
		pyplot.plot(histories[i].history['val_accuracy'], color='orange',
label='test')
	pyplot.show()
```

Les ***scores*** d'exactitude de classification collectés lors de chaque **pli** peuvent être résumés en calculant la **moyenne** et l'**écart type**. Cela fournit une estimation de la performance moyenne attendue du modèle entrainé sur cet ensemble de données, avec une estimation de la **variance moyenne** de la **moyenne**. Nous résumerons également la distribution des ***scores*** en créant et en montrant une **graphique en boîte** .

> Un graphique en boîte, affiche le résumé à cinq chiffres d'un ensemble de données. Le résumé à cinq chiffres est le minimum, le **premier quartile**, la **médiane**, le **troisième quartile** et le **maximum**.
>
> Dans le graphique, nous dessinons une boîte du premier quartile au troisième quartile. Une ligne qui verticale traverse la boîte est une médiane.

La fonction **`summarize_performance()`** ci-dessous implémente cela pour une liste donnée de ***scores*** collectés lors de l'évaluation du modèle.

```
# summarize model performance
def summarize_performance(scores):
	# print summary
	print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100,
std(scores)*100, len(scores)))
	# box and whisker plots of results
	pyplot.boxplot(scores)
	pyplot.show()
```

3.6 Exemple complet

Nous avons besoin d'une fonction qui pilotera toutes les fonctions qu'on vient de définir.

```
# run the test harness for evaluating a model
def run_test_harness():
	# load dataset
	trainX, trainY, testX, testY = load_dataset()
	# prepare pixel data
	trainX, testX = prep_pixels(trainX, testX)
	# evaluate model
	scores, histories = evaluate_model(trainX, trainY)
	# learning curves
	summarize_diagnostics(histories)
	# summarize estimated performance
	summarize_performance(scores)
```

Nous avons maintenant tout ce dont nous avons besoin; l'exemple de code complet pour un **modèle de réseau de neurones convolutionnel de base** sur l'ensemble de données MNIST.

Exercice :

Avant de lancer l'exécution du programme ci-dessous ajouter le paramètre **`sys.argv[1]`** pour rendre flexible ne nombre de phases d'entraînement - **Kfold** . Modifier également le paramètre **`vorbose`** à **`1`** pour pouvoir suivre les époques d'entraînement.

```
# baseline cnn model for mnist
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
	# load dataset
	(trainX, trainY), (testX, testY) = mnist.load_data()
	# reshape dataset to have a single channel
	trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
	testX = testX.reshape((testX.shape[0], 28, 28, 1))
	# one hot encode target values
	trainY = to_categorical(trainY)
	testY = to_categorical(testY)
	return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
	# convert from integers to floats
	train_norm = train.astype('float32')
	test_norm = test.astype('float32')
	# normalize to range 0-1
	train_norm = train_norm / 255.0
	test_norm = test_norm / 255.0
	# return normalized images
	return train_norm, test_norm

# define cnn model
def define_model():
	model = Sequential()
	model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
```

```
	model.add(MaxPooling2D((2, 2)))
	model.add(Flatten())
	model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
	model.add(Dense(10, activation='softmax'))
	# compile model
	opt = SGD(lr=0.01, momentum=0.9)
	model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
	model.summary()
	return model

# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
	scores, histories = list(), list()
	# prepare cross validation
	kfold = KFold(n_folds, shuffle=True, random_state=1)
	# enumerate splits
	for train_ix, test_ix in kfold.split(dataX):
		# define model
		model = define_model()
		# select rows for train and test
		trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix],
dataX[test_ix], dataY[test_ix]
		# fit model
		history = model.fit(trainX, trainY, epochs=10, batch_size=32,
validation_data=(testX, testY), verbose=0)
		# evaluate model
		_, acc = model.evaluate(testX, testY, verbose=0)
		print('> %.3f' % (acc * 100.0))
		# stores scores
		scores.append(acc)
		histories.append(history)
	return scores, histories

# plot diagnostic learning curves
def summarize_diagnostics(histories):
	for i in range(len(histories)):
		# plot loss
		pyplot.subplot(2, 1, 1)
		pyplot.title('Cross Entropy Loss')
		pyplot.plot(histories[i].history['loss'], color='blue',
label='train')
		pyplot.plot(histories[i].history['val_loss'], color='orange',
label='test')
		# plot accuracy
		pyplot.subplot(2, 1, 2)
		pyplot.title('Classification Accuracy')
		pyplot.plot(histories[i].history['accuracy'], color='blue',
label='train')
		pyplot.plot(histories[i].history['val_accuracy'], color='orange',
label='test')
	pyplot.show()

# summarize model performance
def summarize_performance(scores):
	# print summary
	print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100,
std(scores)*100, len(scores)))
	# box and whisker plots of results
	pyplot.boxplot(scores)
	pyplot.show()

# run the test harness for evaluating a model
def run_test_harness():
	# load dataset
	trainX, trainY, testX, testY = load_dataset()
	# prepare pixel data
	trainX, testX = prep_pixels(trainX, testX)
	# evaluate model
	scores, histories = evaluate_model(trainX, trainY)
```

```
        # learning curves
        summarize_diagnostics(histories)
        # summarize estimated performance
        summarize_performance(scores)

# entry point, run the test harness
run_test_harness()
```

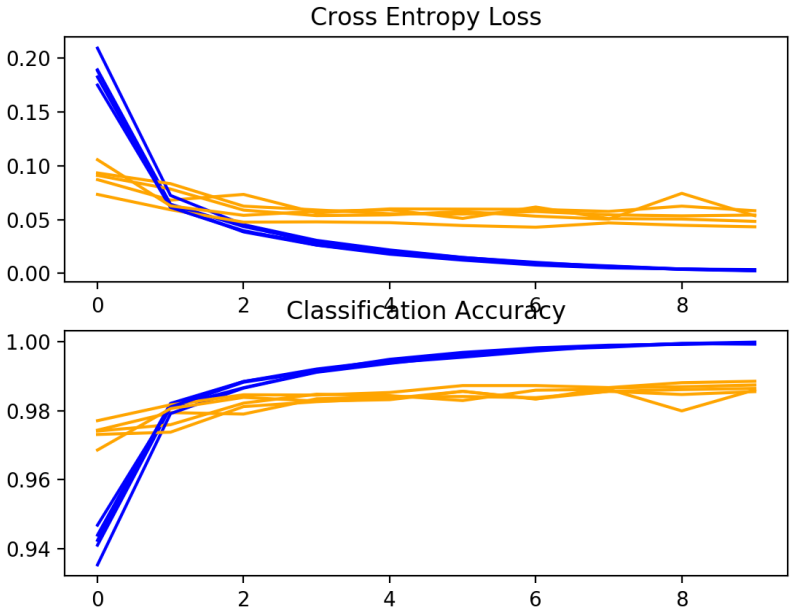
L'exécution de l'exemple imprime la précision de classification pour chaque pli du processus de validation croisée. Ceci est utile pour se faire une idée de la progression de l'évaluation du modèle.

Nous pouvons voir deux cas où le modèle atteint une compétence près parfaite et un cas où il a atteint une précision inférieure à 98%. Ce sont de bons résultats.

```
> 98.550
> 98.600
> 98.642
> 98.850
> 98.742
```

Ensuite, un tracé de diagnostic est montré, donnant un aperçu du comportement d'apprentissage du modèle à travers chaque pli.

Dans ce cas, nous pouvons voir que le modèle atteint généralement un bon ajustement, avec des courbes d'apprentissage d'entrainement et de test convergentes. Il n'y a aucun signe évident de **sur-** ou **sous-ajustement**.



Ensuite, un résumé des performances du modèle est calculé.

Nous pouvons voir dans ce cas, le modèle a une précision estimée à environ 98,6%, ce qui est raisonnable.

```
Accuracy: mean=98.677 std=0.107, n=5
```

Nous avons maintenant un résultat de test robuste et un modèle de base performant.

4. Comment développer un modèle amélioré

Il existe de nombreuses façons d'explorer les améliorations du modèle de référence. Nous examinerons les **domaines de configuration** du modèle qui se traduisent souvent par une amélioration. Le premier est une **modification de l'algorithme d'apprentissage** et le second une **augmentation de la profondeur** du modèle.

4.1 Amélioration de l'apprentissage

Il existe de nombreux aspects de l'algorithme d'apprentissage qui peuvent être explorés pour être améliorés. Peut-être le point de levier le plus important est le **taux d'apprentissage**, comme l'impact que des **valeurs** plus ou moins grandes du taux d'apprentissage, et le fonctionnement des **ordonnanceurs** qui modifient le taux d'apprentissage pendant l'entrainement.

Une autre approche qui peut accélérer rapidement l'apprentissage d'un modèle et entraîner de grandes améliorations des performances est la **normalisation par lots (*batch normalization*)**. Nous évaluerons l'effet de la normalisation des lots sur notre modèle de base.

La **normalisation par lots** peut être utilisée après des couches convolutionnelles et entièrement connectées. Il a pour effet de modifier la distribution de la sortie de la couche, notamment en standardisant les sorties. Cela a pour effet de stabiliser et d'accélérer le processus d'apprentissage.

Nous pouvons mettre à jour la définition du modèle et utiliser la normalisation par lots après la fonction d'activation pour les couches convolutionnelles et denses de notre modèle de base.

La version du modèle (fonction **`define_model()`**)avec la normalisation par lots est présentée ci-dessous.

Exercice

Attention : Le modèle ci-dessous est trop gourmand en mémoire RAM pour la carte Jetson Nano.

Il faut le modifier en mettent à la place de la couche **`BatchNormalization`** la couche de **`Dropout`** positionnée après le **`pooling`**.

Avant de lancer l'exécution du programme ci-dessous ajouter le paramètre **`sys.argv[1]`** pour rendre flexible ne nombre de phases d'entraînement - **Kfold** . Modifier également le paramètre **`vorbose`** à **`1`** pour pouvoir suivre les époques d'entraînement.

```
# define cnn model
def define_model():
	model = Sequential()
	model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
	model.add(BatchNormalization())
	model.add(MaxPooling2D((2, 2)))
	model.add(Flatten())
	model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
	model.add(BatchNormalization())
	model.add(Dense(10, activation='softmax'))
	# compile model
	opt = SGD(lr=0.01, momentum=0.9)
	model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
	return model
```

Le listing complet avec cette modification est fournie ci-dessous.

```
# cnn model with batch normalization for mnist
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import mnist
from keras.utils import to_categorical
```

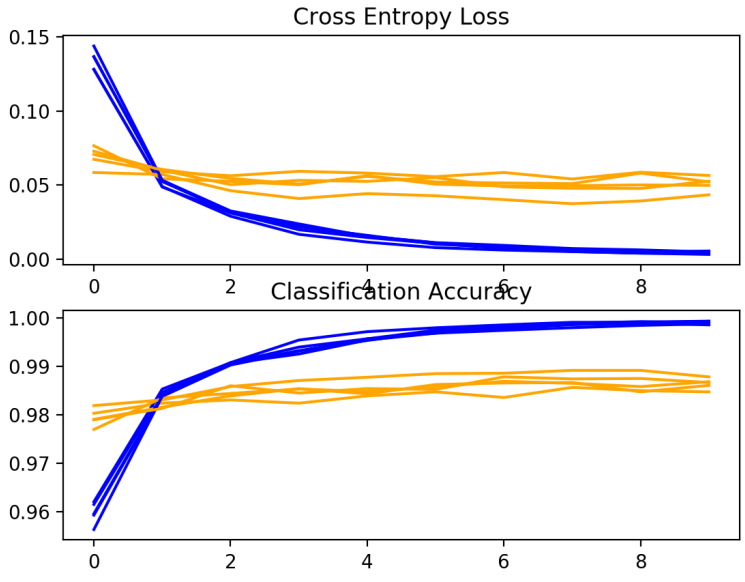
```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.layers import BatchNormalization

# load train and test dataset
def load_dataset():
	# load dataset
	(trainX, trainY), (testX, testY) = mnist.load_data()
	# reshape dataset to have a single channel
	trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
	testX = testX.reshape((testX.shape[0], 28, 28, 1))
	# one hot encode target values
	trainY = to_categorical(trainY)
	testY = to_categorical(testY)
	return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
	# convert from integers to floats
	train_norm = train.astype('float32')
	test_norm = test.astype('float32')
	# normalize to range 0-1
	train_norm = train_norm / 255.0
	test_norm = test_norm / 255.0
	# return normalized images
	return train_norm, test_norm

# define cnn model
def define_model():
	model = Sequential()
	model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
	model.add(BatchNormalization())
	model.add(MaxPooling2D((2, 2)))
	model.add(Flatten())
	model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
	model.add(BatchNormalization())
	model.add(Dense(10, activation='softmax'))
	# compile model
	opt = SGD(lr=0.01, momentum=0.9)
	model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
	model.summary()
	return model

# evaluate a model using k-fold cross-validation  - 5 fold - 5 times
def evaluate_model(dataX, dataY, n_folds=5):
	scores, histories = list(), list()
	# prepare cross validation
	kfold = KFold(n_folds, shuffle=True, random_state=1)
	# enumerate splits
	for train_ix, test_ix in kfold.split(dataX):
		# define model
		model = define_model()
		# select rows for train and test
		trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix],
dataX[test_ix], dataY[test_ix]
		# fit model
		history = model.fit(trainX, trainY, epochs=10, batch_size=32,
validation_data=(testX, testY), verbose=0)
		# evaluate model
		_, acc = model.evaluate(testX, testY, verbose=0)
		print('> %.3f' % (acc * 100.0))
		# stores scores
		scores.append(acc)
		histories.append(history)
```

```
	return scores, histories

# plot diagnostic learning curves
def summarize_diagnostics(histories):
	for i in range(len(histories)):
		# plot loss
		pyplot.subplot(2, 1, 1)
		pyplot.title('Cross Entropy Loss')
		pyplot.plot(histories[i].history['loss'], color='blue',
label='train')
		pyplot.plot(histories[i].history['val_loss'], color='orange',
label='test')
		# plot accuracy
		pyplot.subplot(2, 1, 2)
		pyplot.title('Classification Accuracy')
		pyplot.plot(histories[i].history['accuracy'], color='blue',
label='train')
		pyplot.plot(histories[i].history['val_accuracy'], color='orange',
label='test')
	pyplot.show()

# summarize model performance
def summarize_performance(scores):
	# print summary
	print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100,
std(scores)*100, len(scores)))
	# box and whisker plots of results
	pyplot.boxplot(scores)
	pyplot.show()

# run the test harness for evaluating a model
def run_test_harness():
	# load dataset
	trainX, trainY, testX, testY = load_dataset()
	# prepare pixel data
	trainX, testX = prep_pixels(trainX, testX)
	# evaluate model
	scores, histories = evaluate_model(trainX, trainY)
	# learning curves
	summarize_diagnostics(histories)
	# summarize estimated performance
	summarize_performance(scores)

# entry point, run the test harness
run_test_harness()
```

L'exécution à nouveau de l'exemple rapporte les performances du modèle pour chaque pli du processus de validation croisée. Nous pouvons peut-être constater une légère baisse des performances du modèle par rapport au modèle de base.

```
> 98.475
> 98.608
> 98.683
> 98.783
> 98.667
```

Les graphiques suggèrent que la normalisation des lots, au moins telle qu'elle est mise en œuvre dans ce cas, n'offre aucun avantage.



Ensuite, les performances estimées du modèle smontrent les performances avec une légère diminution de la précision moyenne du modèle: 98,643 par rapport à 98,677 du modèle de base.

```
Accuracy: mean=98.643 std=0.101, n=5
```

4.2 Augmentation de la profondeur du modèle

Il existe de nombreuses façons de modifier la configuration du modèle afin d'explorer les améliorations par rapport au modèle de base.

Deux approches courantes consistent à modifier la capacité de la partie d'extraction de caractéristiques du modèle ou à modifier la capacité ou la fonction de la partie classifieur du modèle. Peut-être que le point le plus influent est le changement de l'extracteur de fonctionnalités.

Nous pouvons augmenter la **profondeur de la partie extracteur** de fonctionnalités du modèle, en suivant un modèle de type VGG consistant à ajouter plus de couches convolutionnelles et de mise en commun avec le même filtre de taille, tout en augmentant le nombre de filtres. Dans ce cas, nous ajouterons une **double couche convolutionnelle avec 64 filtres** chacun, suivie d'une autre couche de ***pool max***.

La version mise à jour de la fonction `define_model()` avec cette modification est présentée ci-dessous.

```
# define cnn model
def define_model():
	model = Sequential()
	model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
	model.add(MaxPooling2D((2, 2)))
	model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
	model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
	model.add(MaxPooling2D((2, 2)))
	model.add(Flatten())
	model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
	model.add(Dense(10, activation='softmax'))
	# compile model
	opt = SGD(lr=0.01, momentum=0.9)
	model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
	return model
```

Pour être complet, le listing complet, y compris cette modification, est fournie ci-dessous.

Exercice

Avant de lancer l'exécution du programme ci-dessous ajouter le paramètre `sys.argv[1]` pour rendre flexible ne nombre de phases d'entraînement - **Kfold** . Modifier également le paramètre `vorbose` à `1` pour pouvoir suivre les époques d'entraînement.

```
# deeper cnn model for mnist
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
# load train and test dataset
def load_dataset():
	# load dataset
	(trainX, trainY), (testX, testY) = mnist.load_data()
```

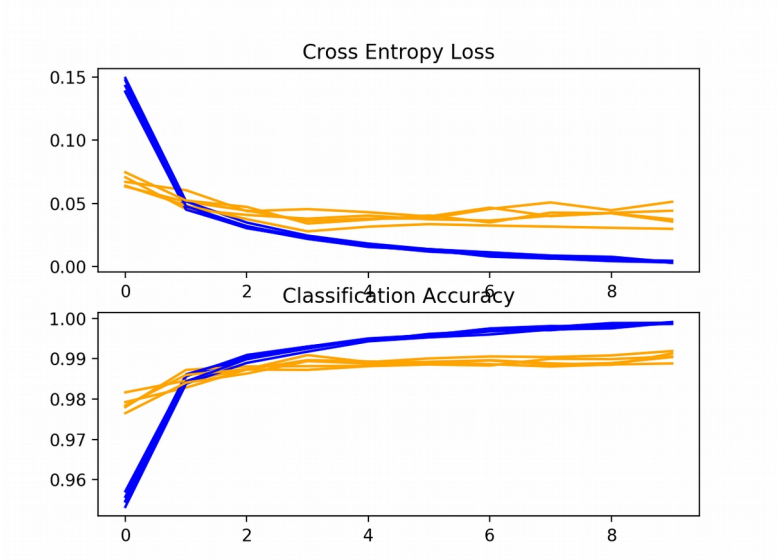
```
	# reshape dataset to have a single channel
	trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
	testX = testX.reshape((testX.shape[0], 28, 28, 1))
	# one hot encode target values
	trainY = to_categorical(trainY)
	testY = to_categorical(testY)
	return trainX, trainY, testX, testY
# scale pixels
def prep_pixels(train, test):
	# convert from integers to floats
	train_norm = train.astype('float32')
	test_norm = test.astype('float32')
	# normalize to range 0-1
	train_norm = train_norm / 255.0
	test_norm = test_norm / 255.0
	# return normalized images
	return train_norm, test_norm
# define cnn model
def define_model():
	model = Sequential()
	model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
	model.add(MaxPooling2D((2, 2)))
	model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
	model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
	model.add(MaxPooling2D((2, 2)))
	model.add(Flatten())
	model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
	model.add(Dense(10, activation='softmax'))
	# compile model
	opt = SGD(lr=0.01, momentum=0.9)
	model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
	model.summary()
	return model
# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
	scores, histories = list(), list()
	# prepare cross validation
	kfold = KFold(n_folds, shuffle=True, random_state=1)
	# enumerate splits
	for train_ix, test_ix in kfold.split(dataX):
		# define model
		model = define_model()
		# select rows for train and test
		trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix],
dataX[test_ix], dataY[test_ix]
		# fit model
		history = model.fit(trainX, trainY, epochs=10, batch_size=32,
validation_data=(testX, testY), verbose=0)
		# evaluate model
		_, acc = model.evaluate(testX, testY, verbose=0)
		print('> %.3f' % (acc * 100.0))
		# stores scores
		scores.append(acc)
		histories.append(history)
	return scores, histories

# plot diagnostic learning curves
def summarize_diagnostics(histories):
	for i in range(len(histories)):
		# plot loss
		pyplot.subplot(2, 1, 1)
		pyplot.title('Cross Entropy Loss')
		pyplot.plot(histories[i].history['loss'], color='blue',
label='train')
		pyplot.plot(histories[i].history['val_loss'], color='orange',
label='test')
```

```
            # plot accuracy
            pyplot.subplot(2, 1, 2)
            pyplot.title('Classification Accuracy')
            pyplot.plot(histories[i].history['accuracy'], color='blue',
label='train')
            pyplot.plot(histories[i].history['val_accuracy'], color='orange',
label='test')
      pyplot.show()
# summarize model performance
def summarize_performance(scores):
      # print summary
      print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100,
std(scores)*100, len(scores)))
      # box and whisker plots of results
      pyplot.boxplot(scores)
      pyplot.show()
# run the test harness for evaluating a model
def run_test_harness():
      # load dataset
      trainX, trainY, testX, testY = load_dataset()
      # prepare pixel data
      trainX, testX = prep_pixels(trainX, testX)
      # evaluate model
      scores, histories = evaluate_model(trainX, trainY)
      # learning curves
      summarize_diagnostics(histories)
      # summarize estimated performance
      summarize_performance(scores)

# entry point, run the test harness
run_test_harness()
```

L'exécution de l'exemple signale les performances du modèle pour chaque étape du processus de validation croisée. Les scores par pli peuvent suggérer une certaine amélioration par rapport au modèle de base.

```
> 99.058
> 99.042
> 98.883
> 99.192
> 99.133
```

Un tracé des courbes d'apprentissage est créé, dans ce cas montrant que les modèles ont toujours un bonne précision, sans signe clair de sur-ajustement. Les graphiques peuvent même suggérer que des périodes de formation supplémentaires pourraient être utiles.



Les performances estimées du modèle montrent une légère amélioration des performances par rapport au modèle de base de 98,677 à 99,062, avec une légère baisse de l'écart-type également.

```
Accuracy: mean=99.062 std=0.104, n=5
```

5. Comment finaliser le modèle et faire des prédictions

Le processus d'amélioration du modèle peut se poursuivre tant que nous avons des idées et le temps et les ressources pour les tester. À un moment donné, une configuration finale du modèle doit être choisie et adoptée. Dans ce cas, nous choisirons le modèle plus profond comme modèle final.

Tout d'abord, nous allons finaliser notre modèle en ajustant un modèle sur l'ensemble de données de formation et en enregistrant le modèle dans un fichier pour une utilisation ultérieure. Nous chargerons ensuite le modèle et évaluerons ses performances sur l'ensemble de données de test pour avoir une idée de la performance réelle du modèle choisi.

Enfin, nous utiliserons le modèle enregistré pour faire une prédiction sur une seule image.

5.1 Enregistrer le modèle final

Un modèle final est généralement adapté à toutes les données disponibles, telles que la combinaison de tous les jeux de données d'entrainement et de test.

Dans ce laboratoire, nous retenons intentionnellement un ensemble de données de test afin de pouvoir estimer les performances du modèle final, ce qui peut être une bonne idée dans la pratique.

En tant que tel, nous adapterons notre modèle uniquement à l'ensemble de données d'entrainement.

```
# fit model
model.fit(trainX, trainY, epochs=10, batch_size=32, verbose=0)
```

Once fit, we can save the final model to an H5 file by calling the *save()* function on the model and pass in the chosen filename.

Une fois ajusté, nous pouvons enregistrer le modèle final dans un fichier **`H5`** en appelant la fonction **`save()`** sur le modèle et passer le nom de fichier choisi

```
# save model
model.save('final_model.h5')
```

Ci-dessous vous trouvez l'exemple complet de l'ajustement du modèle profond final sur le jeu de données d'apprentissage et de son enregistrement dans un fichier type **`.h5`**.

Exercice :

Modifier également le paramètre **`vorbose`** à **`1`** pour pouvoir suivre les époques d'entraînement.

```
# save the final model to file
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
	# load dataset
	(trainX, trainY), (testX, testY) = mnist.load_data()
	# reshape dataset to have a single channel
	trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
	testX = testX.reshape((testX.shape[0], 28, 28, 1))
	# one hot encode target values
	trainY = to_categorical(trainY)
	testY = to_categorical(testY)
	return trainX, trainY, testX, testY
```

```
# scale pixels
def prep_pixels(train, test):
	# convert from integers to floats
	train_norm = train.astype('float32')
	test_norm = test.astype('float32')
	# normalize to range 0-1
	train_norm = train_norm / 255.0
	test_norm = test_norm / 255.0
	# return normalized images
	return train_norm, test_norm

# define cnn model
def define_model():
	model = Sequential()
	model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
	model.add(MaxPooling2D((2, 2)))
	model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
	model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
	model.add(MaxPooling2D((2, 2)))
	model.add(Flatten())
	model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
	model.add(Dense(10, activation='softmax'))
	# compile model
	opt = SGD(lr=0.01, momentum=0.9)
	model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
	return model

# run the test harness for evaluating a model
def run_test_harness():
	# load dataset
	trainX, trainY, testX, testY = load_dataset()
	# prepare pixel data
	trainX, testX = prep_pixels(trainX, testX)
	# define model
	model = define_model()
	# fit model
	model.fit(trainX, trainY, epochs=10, batch_size=32, verbose=0)
	# save model
	model.save('final_model.h5')

# entry point, run the test harness
run_test_harness()
```

5.2 Évaluer le modèle final

Nous pouvons maintenant charger le modèle final et l'évaluer sur l'ensemble de données de test .

Le modèle peut être chargé via la fonction `load_model()`.

Ci-dessous nous donnons l'exemple complet de chargement du modèle enregistré et d'évaluation de celui-ci sur l'ensemble de données de test.

```
# evaluate the deep model on the test dataset
from keras.datasets import mnist
from keras.models import load_model
from keras.utils import to_categorical

# load train and test dataset
def load_dataset():
	# load dataset
	(trainX, trainY), (testX, testY) = mnist.load_data()
	# reshape dataset to have a single channel
	trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
	testX = testX.reshape((testX.shape[0], 28, 28, 1))
	# one hot encode target values
```

```
	trainY = to_categorical(trainY)
	testY = to_categorical(testY)
	return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
	# convert from integers to floats
	train_norm = train.astype('float32')
	test_norm = test.astype('float32')
	# normalize to range 0-1
	train_norm = train_norm / 255.0
	test_norm = test_norm / 255.0
	# return normalized images
	return train_norm, test_norm

# run the test harness for evaluating a model
def run_test_harness():
	# load dataset
	trainX, trainY, testX, testY = load_dataset()
	# prepare pixel data
	trainX, testX = prep_pixels(trainX, testX)
	# load model
	model = load_model('final_model.h5')
	# evaluate model on test dataset
	_, acc = model.evaluate(testX, testY, verbose=0)
	print('> %.3f' % (acc * 100.0))

# entry point, run the test harness
run_test_harness()
```

L'exécution de l cet exemple charge le modèle enregistré et évalue le modèle pour l'ensemble de données de test .

La précision de classification du modèle sur l'ensemble de données de test est calculée et imprimée. Dans ce cas, nous pouvons voir que le modèle a atteint une précision de 99,090%.

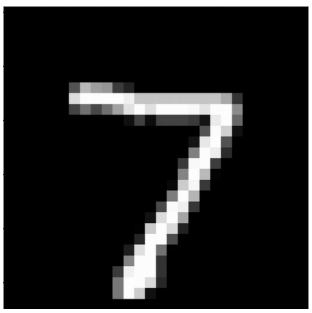
```
> 99.090
```

5.3 Faire une prédiction

Nous pouvons utiliser notre modèle enregistré pour faire une prédiction sur de nouvelles images.

Le modèle suppose que les nouvelles images sont en niveaux de gris, qu'elles ont été alignées de manière à ce qu'une image contienne un chiffre manuscrit centré et que la forme de l'image soit carrée avec la taille de **28×28** pixels.

Ci-dessous, une image extraite de l'ensemble de données de test MNIST. Vous pouvez l'enregistrer dans votre répertoire de travail actuel avec le nom de fichier **`sample_image.jpg`**.



Nous suposons qu'il s'agit d'une image entièrement nouvelle et invisible, préparée de la manière requise, et verrons comment nous pourrions utiliser notre modèle enregistré pour prédire l'entier que l'image représente (par exemple, nous nous attendons à détecter **7**).

Tout d'abord, nous pouvons charger l'image, transformer en niveaux de gris et en taille de 28 × 28 pixels. L'image chargée peut être redimensionnée pour avoir un seul canal et représenter un seul échantillon dans un ensemble de données. La fonction **`load_image()`** effectue cela et retournera l'image chargée prête pour la classification.

Il est important de noter que les valeurs de pixels sont préparées de la même manière que les valeurs de pixels préparées pour l'ensemble de données d'apprentissage lors de l'entrainement du modèle final.

```
# load and prepare the image
def load_image(filename):
	# load the image
	img = load_img(filename, grayscale=True, target_size=(28, 28))
	# convert to array
	img = img_to_array(img)
	# reshape into a single sample with 1 channel
	img = img.reshape(1, 28, 28, 1)
	# prepare pixel data
	img = img.astype('float32')
	img = img / 255.0
	return img
```

Ensuite, nous pouvons charger le modèle comme dans la section précédente et appeler la fonction **`predict_classes()`** pour prédire le chiffre que l'image représente.

```
# predict the class
digit = model.predict_classes(img)
```

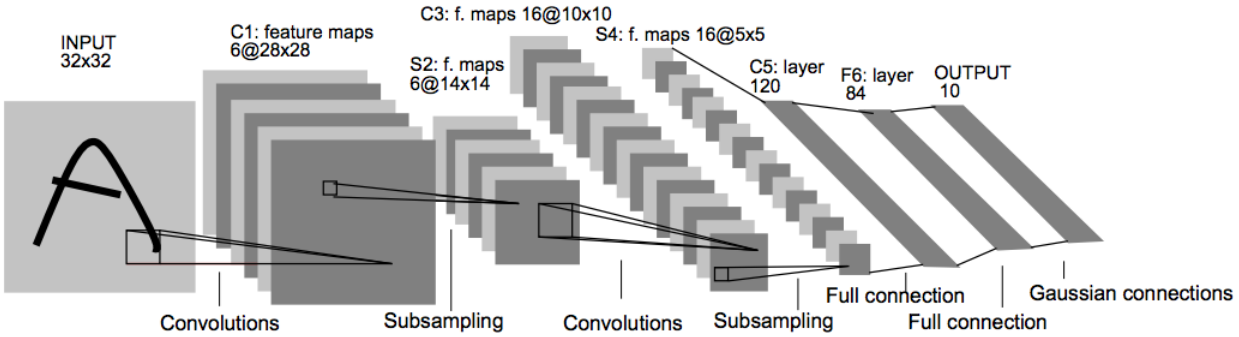
L'exemple complet est donné ci-dessous.

```
# make a prediction for a new image.
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.models import load_model

# load and prepare the image
def load_image(filename):
	# load the image
	img = load_img(filename, grayscale=True, target_size=(28, 28))
	# convert to array
	img = img_to_array(img)
	# reshape into a single sample with 1 channel
	img = img.reshape(1, 28, 28, 1)
	# prepare pixel data
	img = img.astype('float32')
	img = img / 255.0
	return img

# load an image and predict the class
def run_example():
	# load the image
	img = load_image('sample_image.png')
	# load model
	model = load_model('final_model.h5')
	# predict the class
	digit = model.predict_classes(img)
	print(digit[0])

# entry point, run the example
run_example()
```

L'exécution de ce code charge et prépare l'image, puis charge le modèle, enfin il prédit correctement que l'image chargée représente le chiffre «7».

6. Exercice final : LeNet-5

Ci-dessous l'architecture d'un réseau de type **LeNet-5** avec deux niveaux de convolution (**`Conv2D`**) et 3 niveaux denses (**`Dense`**) plus les opérations des mise à plat (**`Flatten`**) et les réductions type **`AveragePooling`**.




```
model = keras.Sequential()
model.add(layers.Conv2D(filters=6, kernel_size=(3, 3), activation='relu',
input_shape=(32,32,1)))
model.add(layers.AveragePooling2D())
model.add(layers.Conv2D(filters=16, kernel_size=(3, 3),
activation='relu'))
model.add(layers.AveragePooling2D())
model.add(layers.Flatten())
model.add(layers.Dense(units=120, activation='relu'))
model.add(layers.Dense(units=84, activation='relu'))
model.add(layers.Dense(units=10, activation = 'softmax'))
```

Exercice

A partir de la présentation ci-dessus mettez en ouevre trois programmes Python pour :

1. Construire, entraîner et sauvegarder le modèle de type LeNet-5 (**`lenet-5_model.h5`**)
2. Charger et évaluer le modèle généré
3. Faire une prédiction sur les images fournies

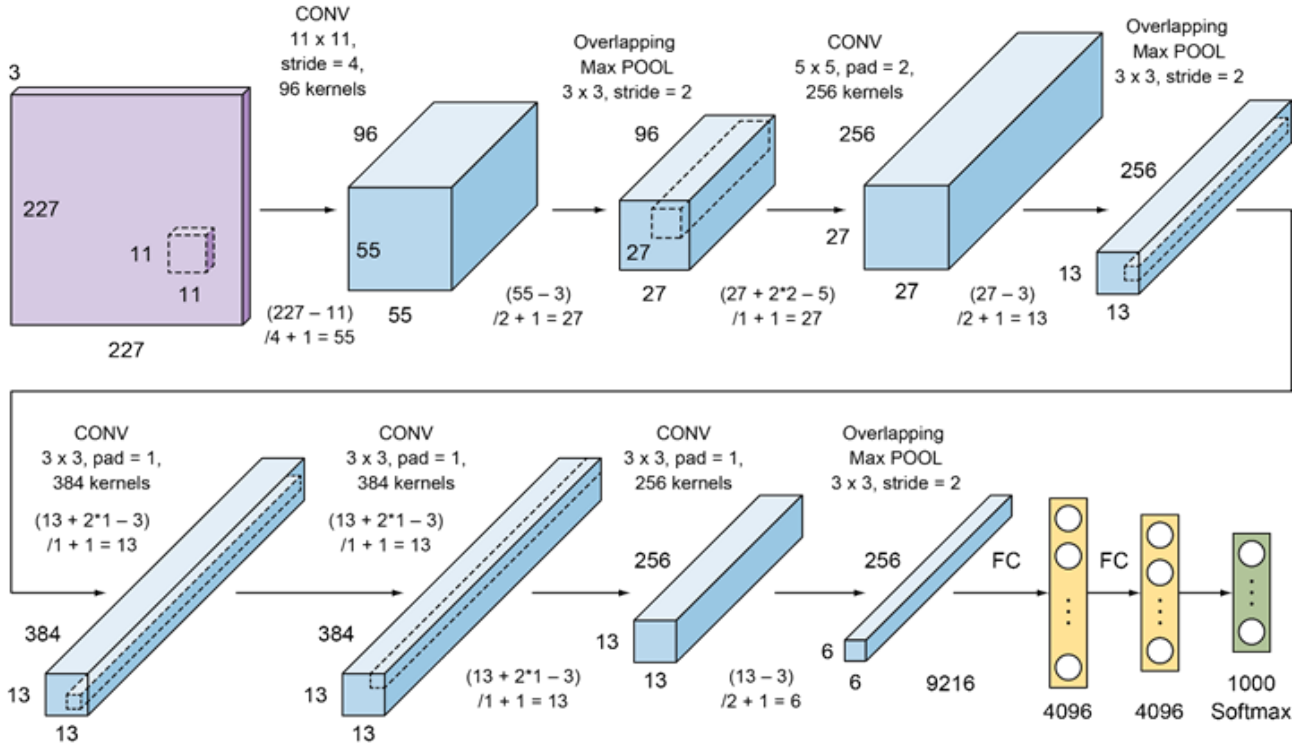
Au-delà de Lab2

Un modèle puissant pour la reconnaissance des images– AlexNet

LeNet-5 fonctionne très bien sur l'ensemble de données MNIST. Mais il s'avère que l'ensemble de données MNIST est très simple car il contient des images en niveaux de gris (1 canal) et n'est classé qu'en 10 classes, ce qui en fait un défi plus simple. La principale motivation derrière **`AlexNet`** était de construire un réseau plus profond qui peut apprendre des fonctions plus complexes.

- Quelle est la taille (shape) des données (images) ?
- Quel est le nombre de classes diiférenciées ?
- Quel est le nombre de paramètres d'entrainement ?

..




```
# Instantiate an empty sequential model
model = Sequential()
# 1st layer (conv + pool + batchnorm)
model.add(Conv2D(filters= 96, kernel_size= (11,11), strides=(4,4),
padding='valid',
input_shape = (224,224,3)))
model.add(Activation('relu')) <---- activation function can be added on its own
layer or
within the Conv2D function as we did in previous implementations
model.add(MaxPool2D(pool_size=(3,3), strides=(2,2)))
model.add(BatchNormalization())
# 2nd layer (conv + pool + batchnorm)
model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same',
kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(MaxPool2D(pool_size=(3,3), strides=(1,1)))
model.add(BatchNormalization())
# layer 3 (conv + batchnorm) <--- note that the authors did not add a POOL layer
here
```

```
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same',
kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())
# layer 4 (conv + batchnorm) <--- similar to layer 4
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same',
kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())
# layer 5 (conv + batchnorm)
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same',
kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(3,3), strides=(2,2)))

# Flatten the CNN output to feed it with fully connected layers
model.add(Flatten())

# layer 6 (Dense layer + dropout)
model.add(Dense(units = 4096, activation = 'relu'))
model.add(Dropout(0.5))

# layer 7 (Dense layers)
model.add(Dense(units = 4096, activation = 'relu'))
model.add(Dropout(0.5))
# layer 8 (softmax output layer)
model.add(Dense(units = 1000, activation = 'softmax'))

# print the model summary
model.summary()
```

Attention

Ce modèle ne peut pas être entraîné sur une carte **Jetson Nano** . Pour le faire il vous faut une carte type **Jetson Xavier** et **5 heures** de patience !