Compte rendu du TP de deep learning

Objectif : Entrainement de modèles de réseaux neurones à reconnaître des images de chiffres manuscrits

- L'objectif de ce TP est de créer des modèles de deep learning capable de reconnaître des images de chiffres manuscrits et les comparer afin de tirer le meilleur modèle avec un pourcentage de précision élevé.

Dans cette perspective, nous traçons notre plan de travail qui sera présenter comme suit :

1. Chargement et présentation des données sur lesquelles le travail sera fait.
2. Définir l'architecture des modèles choisis pour l'entrainement.
3. Compilation et entrainement des modèles.
4. Sauvegarde et chargement des modèles définis et évaluation de leur performance.
5. Surveillance du processus d'entrainement avec l'interface Tenserboard.

Dans ce TP, on va utiliser la bibliothèque **keras** pour sa facilité à créer des réseaux de neurones. Pour notre dataset, elle est aussi fournie par **keras.**

Les modèles que nous allons entrainer sont :

1. Un classifieur linéaire
2. Un réseau de neurones entièrement connecté avec deux couches cachées
3. Un réseau de neurones convolutif (a vanilla convolutional neural network)

- ### Description de la dataset

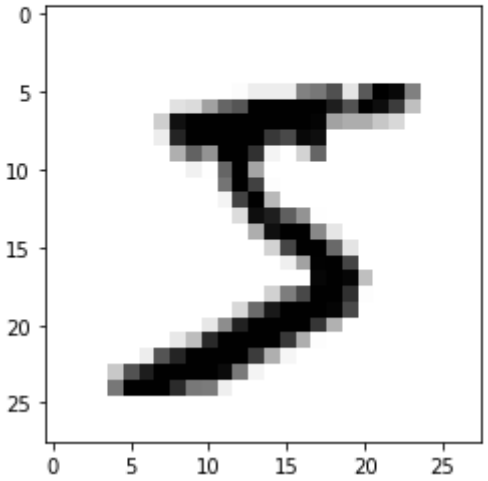
On va travailler sur la base de données **MNIST** qui contient un jeu d'entrainement de 60000 images en niveaux de gris de résolution 28x28, représentant 10 chiffres de 0 à 9, ainsi qu'un jeu de test de 10000 images. Nous sommes dans le cas supervisé et donc notre base de données contient aussi dans labels pour chaque image (**i.e.** chaque image est associée à un chiffre entre 0 et 9). Chargeons nos données.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import Input
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.models import Model
import os
from tensorflow.keras.callbacks import TensorBoard
from tensorflow.keras.callbacks import ModelCheckpoint
import h5py
from tensorflow.keras.models import load_model
from tensorflow.keras.layers import Lambda
from tensorflow.keras import regularizers
from tensorflow.keras.layers import Dropout
import matplotlib.pyplot as plt
import numpy as np


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

X_train et **X_test** contiennent les images et **y_train** et **y_test** contiennent les labels. Nous allons vérifier à présent les tailles de chaque jeu de données et nous visualiserons la première image.

```
print("X_train.shape = ", X_train.shape)
print("y_train.shape = ", y_train.shape)
print("X_test.shape = ", X_test.shape)
print("y_test.shape = ", y_test.shape)

# ces deux ligne de code nous permettent de visualiser la premiére image
plt.imshow(X_train[0], cmap = plt.cm.binary)
plt.show()
```

```
X_train.shape =  (60000, 28, 28)
y_train.shape =  (60000,)
X_test.shape =  (10000, 28, 28)
y_test.shape =  (10000,)
```



1. Classifieur linéaire

Comme on a pu le remarquer, nos jeux de données X_train et X_test sont des images 28x28. Or, notre modèle est linéaire (**i.e.** un réseau de neurones avec une seule couche cachée) et qui attends en entrée un vecteur. On va alors modifier nos données en les transformant en vecteurs sans perdre le contenu (au lieu d'une matrice 28x28, on aura un vecteur avec 784 pixels). Pour cela, on utilise la fonction **reshape** dans le code suivant :

```
num_train  = X_train.shape[0]
num_test   = X_test.shape[0]

img_height = X_train.shape[1]
img_width  = X_train.shape[2]
X_train = X_train.reshape((num_train, img_width * img_height))
X_test  = X_test.reshape((num_test, img_width * img_height))
```

Vérifions maintenant la taille de nos jeux de données

```
print("X_train.shape = ", X_train.shape)
print("X_test.shape = ", X_train.shape)
```

```
X_train.shape =  (60000, 784)
X_test.shape =  (60000, 784)
```

Après l'entrainement, le modèle va être évalué sur la base de test, cela veut dire qu'il va prédire à quelle classe appartient chaque image de X_test. Les classes étant de 0 à 9. Le score qu'on obtiendra est le taux de bonnes réponses par rapport au vrai labels présents dans y_test.

Enfin comme notre modèle s'attend à qu'on lui donne des vecteurs comme entrées (input), il est normal qu'en sortie (output), on s'attend aussi à avoir des vecteurs. Les outputs dans notre cas sont nos chiffres de 0 à 9. Donc on va modifier nos labels en les transformant en vecteurs (**i.e.** chaque chiffre devient un vecteur qui le représenta). On utilise pour cela la fonction to_categorical. Elle va représenter chaque chiffre par un vecteur constitué d'un seul un et de zéros. Visualisant nos 3 premiers labels maintenant. Le premier vecteur par exemple est un 5.

```
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
print(y_train[:3])
```

```
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

1.1. Architecture du modèle

Maintenant qu'on a chargé nos données et les a modifiés afin de les utiliser, nous allons à présent construire notre classifieur linéaire. Il est linéaire car il y a une seule couche cachée (Dense layer). Pour cela on a besoin d'une couche input, une couche cachée et une couche output. Notre modèle va calculer le score de chaque classe comme produit des poids associés à chaque neurone et la valeur que contient le neurone dans la première couche (**i.e.** $w_k^T x_i$), le biais étant ajouté dans la structure, qu'on lui appliquera une fonction de transfère **softmax** pour avoir les probabilités sur chaque classe. Cette fonction est utilisée dans les cas de classification multi-classe (dans notre cas, on a 10 classes) de plus la somme de tous les probabilités égale à 1.

```
num_classes = 10
xi      = Input(shape=(img_height*img_width,))
xo      = Dense(num_classes)(xi)
yo      = Activation('softmax')(xo)
model   = Model(inputs=[xi], outputs=[yo])


model.summary()
```

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 784)]             0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                7850
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
_________________________________________________________________
```

La fonction **model.summary** nous donne un résumé de notre modèle :

- On obtient un tableau avec en première colonne le nombre de couches. Dans la deuxième, la taille de sortie de chaque couche et dans la troisième, le nombre de paramètres qu'on va entrainer, associés à chaque couche.

- Pour la couche cachée, on a une taille de sortie égale à dix qui est son nombre de neurones. Son nombre de paramètres est égale à 7850. On a ce nombre de paramètres car il existe pour chaque neurone de la couche d'entrée 10 connexion (poids). En l'occurrence ici, on a 7840 neurones correspondant à une image, c'est à dire 7840 paramètres. Avec **keras**, on n'a pas besoin d'ajouter une constante en première couche car le biais est prédéfini dans la couche cachée, qui nous permettra d'avoir 7850 en ajoutant les 10 paramètres du biais.

- Les outputs de la couche cachée sont passés par la fonction **softmax** et on aura au final des outputs dans la couche de sortie, de même taille que la couche précédente, entre 0 et 1 et leur sommation égale à 1.

1.2. Compilation et entrainement des modèles.

L'étape suivante est celle de la compilation de ce modèle. On va définir alors une fonction loss qu'on cherche à minimiser, un optimiseur et une métrique pour surveiller tout ça. On utilise ici la loss **categorical_crossentropy** car on un nombre de classe strictement supérieur à deux. Si on avait un problème de classification binaire, on aurait pu choisir **binary_crossentropy**. Pour l'optimiseur, on choisit **adam** et pour la métrique, on choisit **accuracy** qu'on cherche à maximiser. Le code est le suivant (on utilise la fonction **model.compile**) :

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

Maintenant, nous allons lancer l'étape de l'apprentissage. Mais avant cela, on doit expliquer la manière dont l'entrainement est fait. Premièrement, on définit le **batch** qui est un échantillon de données de notre jeu. On définit aussi l'epoche et durant une **epoche**, le modèle va parcourir tous les batchs. Ici, on définit 20 epoches et la taille du **batch** égale à 128, c'est à dire que le modèle va parcourir notre jeu de données 20 fois. Ici on va prendre 10 % de notre jeu d'entrainement pour valider à chaque fois notre entrainement et on va choisir notre meilleur modèle se basant sur la perte de validation (**val_loss**), qui doit être la plus minimale possible. Enfin, on évalue la performance de notre modèle sur notre jeu de test avec la fonction **model.evaluate**.

```
model.fit(X_train, y_train,
          batch_size=128,
          epochs=20,
          verbose=1,
          validation_split=0.1)

score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/20
54000/54000 [==============================] - 2s 42us/sample - loss: 13.7199 - acc: 0.7961 - val_loss: 5.0058 - val_acc: 0.
8835
Epoch 2/20
54000/54000 [==============================] - 2s 31us/sample - loss: 5.0897 - acc: 0.8721 - val_loss: 3.6368 - val_acc: 0.8
938
```

```
Epoch 3/20
54000/54000 [==============================] - 2s 30us/sample - loss: 4.1614 - acc: 0.8802 - val_loss: 3.5067 - val_acc: 0.8
890
Epoch 4/20
54000/54000 [==============================] - 2s 29us/sample - loss: 3.6023 - acc: 0.8837 - val_loss: 3.3869 - val_acc: 0.8
940
Epoch 5/20
54000/54000 [==============================] - 2s 31us/sample - loss: 3.4056 - acc: 0.8850 - val_loss: 3.0565 - val_acc: 0.8
877
Epoch 6/20
54000/54000 [==============================] - 2s 35us/sample - loss: 3.1631 - acc: 0.8873 - val_loss: 2.6831 - val_acc: 0.9
018
Epoch 7/20
54000/54000 [==============================] - 2s 35us/sample - loss: 2.8999 - acc: 0.8889 - val_loss: 2.6758 - val_acc: 0.8
997
Epoch 8/20
54000/54000 [==============================] - 2s 32us/sample - loss: 2.8201 - acc: 0.8881 - val_loss: 2.6761 - val_acc: 0.9
008
Epoch 9/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.7347 - acc: 0.8903 - val_loss: 3.4197 - val_acc: 0.8
585
Epoch 10/20
54000/54000 [==============================] - 2s 29us/sample - loss: 2.5311 - acc: 0.8928 - val_loss: 3.2258 - val_acc: 0.8
828
Epoch 11/20
54000/54000 [==============================] - 2s 31us/sample - loss: 2.5957 - acc: 0.8900 - val_loss: 2.4004 - val_acc: 0.9
147
Epoch 12/20
54000/54000 [==============================] - 2s 29us/sample - loss: 2.5105 - acc: 0.8921 - val_loss: 2.4758 - val_acc: 0.9
022
Epoch 13/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.5000 - acc: 0.8903 - val_loss: 2.7425 - val_acc: 0.8
903
Epoch 14/20
54000/54000 [==============================] - 2s 29us/sample - loss: 2.4214 - acc: 0.8920 - val_loss: 2.1776 - val_acc: 0.9
112
Epoch 15/20
54000/54000 [==============================] - 2s 29us/sample - loss: 2.4168 - acc: 0.8930 - val_loss: 2.2538 - val_acc: 0.8
967
Epoch 16/20
54000/54000 [==============================] - 2s 32us/sample - loss: 2.3990 - acc: 0.8918 - val_loss: 2.7066 - val_acc: 0.8
938
Epoch 17/20
54000/54000 [==============================] - 2s 36us/sample - loss: 2.4549 - acc: 0.8917 - val_loss: 2.1989 - val_acc: 0.9
038
Epoch 18/20
54000/54000 [==============================] - 2s 32us/sample - loss: 2.3771 - acc: 0.8924 - val_loss: 2.1625 - val_acc: 0.9
152
Epoch 19/20
54000/54000 [==============================] - 2s 31us/sample - loss: 2.3224 - acc: 0.8938 - val_loss: 2.5430 - val_acc: 0.8
970
Epoch 20/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.3625 - acc: 0.8918 - val_loss: 3.5931 - val_acc: 0.8
625
Test loss: 3.8547970748061053
Test accuracy: 0.8555
```

On obtient une **val_loss** égale à 3,5931, dans 20 sur 20 **epoches** et une **val_acc** de 0,8625 et une **Test Loss** égale à 3,8547970748061053 et une **Test accuracy** égale à 0.8555. Si on se pencher juste sur l'accuracy, on pourrait dire que ce modèle n'est pas mal mais en remarquant la **Test loss**, on s'aperçoit que notre modèle n'est pas performant car la **val_loss** est trop grande, ce qui se confirme en évaluation sur le jeu de test avec un **Test loss** encore plus grande.

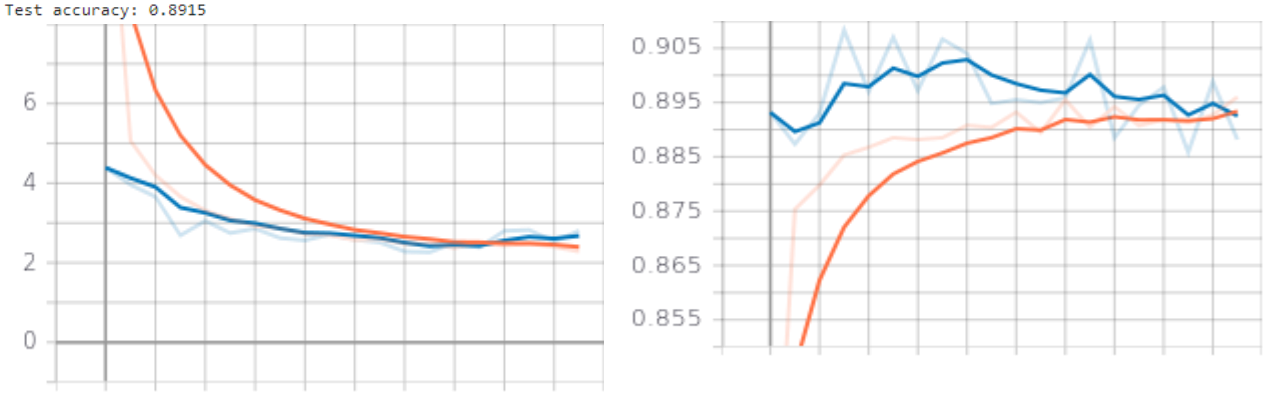
1.3. Sauvegarde et chargement des modèles

Maintenant, on se concentre sur la partie sauvegarde et surveillance du meilleur modèle. On va utiliser les calbacks qui est une particularité de **TensorFlow**. Un de sauvegarde (**ModelCheckpoint**) qui va prendre le meilleur modèle et un autre de surveillance sur la plateforme de **tensorboard** qui va nous permettre de visualiser les loss et accuracy et les comparer entre les différents résultats de notre modèle. On va définir une fonction qui créera un chemin vers où la data sera sauvegardées. Cette fonction vient avant la partie entrainement. Ensuite, on définira le code qui nous permettra de surveiller la progression de notre modèle et le sauvegarder avant le **model.fit** aussi.

```python
def generate_unique_logpath(logdir, raw_run_name):
    i = 0
    while(True):
        run_name = raw_run_name + "-" + str(i)
        log_path = os.path.join(logdir, run_name)
        if not os.path.isdir(log_path):
            return log_path
        i = i + 1

# définition du callback de surveillance et de sauvegarde sur tensorboard
run_name = "linear"
logpath = generate_unique_logpath("./logs_linear", run_name)
tbcb = TensorBoard(log_dir=logpath)
# définition du callback de sauvegarde du meilleur modèle
checkpoint_filepath = os.path.join(logpath,  "best_model.h5")
checkpoint_cb = ModelCheckpoint(checkpoint_filepath, save_best_only=True)

model.fit(X_train, y_train,
          batch_size=128,
      epochs=20,
      verbose=1,
      validation_split=0.1,
      callbacks=[tbcb, checkpoint_cb])

score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/20
54000/54000 [==============================] - 2s 36us/sample - loss: 2.4851 - acc: 0.8919 - val_loss: 2.3313 - val_acc: 0.9
032
Epoch 2/20
54000/54000 [==============================] - 2s 33us/sample - loss: 2.3100 - acc: 0.8938 - val_loss: 2.3501 - val_acc: 0.9
040
Epoch 3/20
54000/54000 [==============================] - 2s 31us/sample - loss: 2.3428 - acc: 0.8935 - val_loss: 2.5013 - val_acc: 0.8
997
Epoch 4/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.2712 - acc: 0.8936 - val_loss: 2.9234 - val_acc: 0.8
763
Epoch 5/20
54000/54000 [==============================] - 2s 29us/sample - loss: 2.4940 - acc: 0.8923 - val_loss: 2.3820 - val_acc: 0.9
082
Epoch 6/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.2358 - acc: 0.8963 - val_loss: 2.9696 - val_acc: 0.8
740
Epoch 7/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.2122 - acc: 0.8951 - val_loss: 2.3816 - val_acc: 0.9
018
Epoch 8/20
54000/54000 [==============================] - 2s 31us/sample - loss: 2.4150 - acc: 0.8924 - val_loss: 2.6876 - val_acc: 0.8
973
Epoch 9/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.3864 - acc: 0.8928 - val_loss: 2.5438 - val_acc: 0.9
012
Epoch 10/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.3758 - acc: 0.8933 - val_loss: 2.4737 - val_acc: 0.9
105
Epoch 11/20
54000/54000 [==============================] - 2s 32us/sample - loss: 2.2851 - acc: 0.8954 - val_loss: 3.4959 - val_acc: 0.8
773
```

```
Epoch 12/20
54000/54000 [==============================] - 2s 35us/sample - loss: 2.4237 - acc: 0.8940 - val_loss: 3.0242 - val_acc: 0.8
868
Epoch 13/20
54000/54000 [==============================] - 2s 29us/sample - loss: 2.2718 - acc: 0.8957 - val_loss: 2.5265 - val_acc: 0.8
988
Epoch 14/20
54000/54000 [==============================] - 2s 29us/sample - loss: 2.3989 - acc: 0.8944 - val_loss: 2.5031 - val_acc: 0.8
965
Epoch 15/20
54000/54000 [==============================] - 2s 29us/sample - loss: 2.2991 - acc: 0.8977 - val_loss: 2.5846 - val_acc: 0.8
930
Epoch 16/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.4110 - acc: 0.8943 - val_loss: 2.4275 - val_acc: 0.9
018
Epoch 17/20
54000/54000 [==============================] - 2s 31us/sample - loss: 2.3220 - acc: 0.8967 - val_loss: 2.6538 - val_acc: 0.8
977
Epoch 18/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.3655 - acc: 0.8950 - val_loss: 2.7125 - val_acc: 0.9
005
Epoch 19/20
54000/54000 [==============================] - 2s 30us/sample - loss: 2.3312 - acc: 0.8943 - val_loss: 2.6411 - val_acc: 0.9
020
Epoch 20/20
54000/54000 [==============================] - 2s 29us/sample - loss: 2.3593 - acc: 0.8952 - val_loss: 2.5972 - val_acc: 0.8
985
Test loss: 3.066965817362076
Test accuracy: 0.8915
```



Sur ses deux graphes, on a la **val_loss** et la **loss** à gauche et la **val_acc** et la **accuracy** à droite.

On remarque qu'après une deuxième exécution, on est toujours au niveau de loss et de accuracy de la première exécution. On pourra exécuter le modèle plusieurs fois et le surveiller sur tensorboard et ainsi à chaque fois garder le meilleur modèle selon la validation loss.

Pour charger le meilleur modèle, on utilise la fonction **load_model**. Pour des raisons techniques, on ajoutera une partie du code pour parvenir à charger le modèle.

```
with h5py.File(checkpoint_filepath, 'a') as f:
    if 'optimizer_weights' in f.keys():
        del f['optimizer_weights']

model = load_model(checkpoint_filepath)
score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 2.8240551112762255
Test accuracy: 0.9006
```

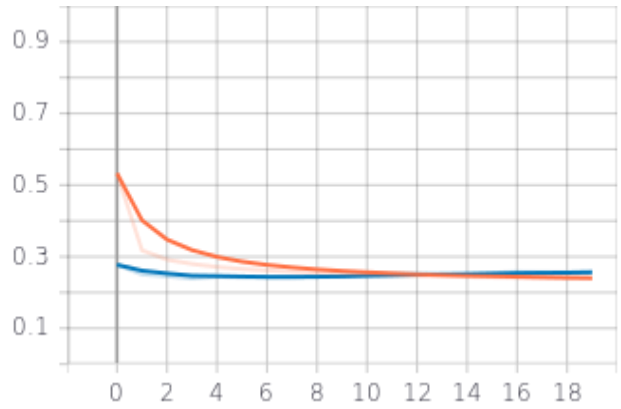
On remarque qu'après plusieurs itérations, le modèle s'améliore jusqu'à 90 % d'accuracy et une **Test loss** de 2,8240551112762255

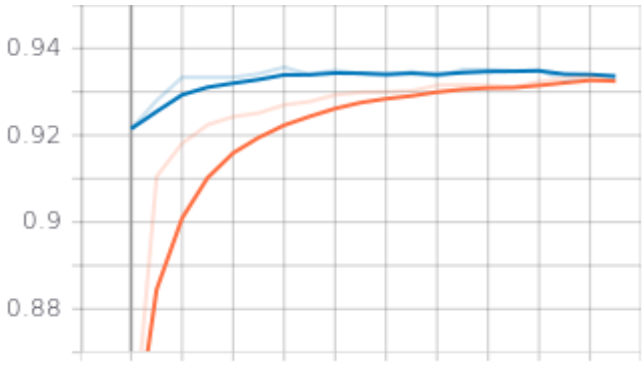
1.4. Normalisation des données

Jusqu'à maintenant, on travailler avec les données brutes de notre jeu (i.e. des pixels d'une valeur entre 0 et 255. On normalisera ces données afin de rendre l'entrainement de nos paramètres encore plus rapide et l'améliorer en accuracy. On remplacera le code suivant au début de notre script au niveau de la définition de la structure de notre modèle. Les pixels désormais aurons une valeur entre 0 et 1.

```
xi = Input(shape=(img_height*img_width,), name="input")

mean = X_train.mean(axis=0) # normalisation des données
std = X_train.std(axis=0) + 1.0
xl = Lambda(lambda image, mu, std: (image - mu) / std,
            arguments={'mu': mean, 'std': std})(xi)
xo = Dense(num_classes, name="y")(xl)
yo = Activation('softmax', name="y_act")(xo)
model = Model(inputs=[xi], outputs=[yo])

model.summary()
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/20
54000/54000 [==============================] - 3s 47us/sample - loss: 0.5209 - acc: 0.8479 - val_loss: 0.2786 - val_acc: 0.9
225
Epoch 2/20
54000/54000 [==============================] - 2s 41us/sample - loss: 0.3151 - acc: 0.9112 - val_loss: 0.2503 - val_acc: 0.9
302
...
Epoch 19/20
54000/54000 [==============================] - 2s 43us/sample - loss: 0.2391 - acc: 0.9334 - val_loss: 0.2539 - val_acc: 0.9
378
Epoch 20/20
54000/54000 [==============================] - 2s 44us/sample - loss: 0.2383 - acc: 0.9335 - val_loss: 0.2541 - val_acc: 0.9
332
Test loss: 0.32559393405914305
Test accuracy: 0.9247
Test loss: 0.2923996151328087
Test accuracy: 0.9249
```





On remarque une nette amélioration au niveau de la **val_loss** et la **Test_loss** et une accuracy au-dessus de 92 %.

2. Réseau de neurones entièrement connecté avec deux couches cachées

2.1. Architecture du modèle

Pour cette partie, on va changer la structure de notre modèle et ajouter deux couches cachées. Le modèle n'est plus linéaire. Cela permettra d'améliorer la performance du modèle. On définira deux fonctions d'activation **Relu** après chaque couche cachée car on est plus dans le cas linéaire et la fonction de loss reste la **softmax**. En ce qui concerne le nombre de neurones des couches cachées. On pourra les modifier selon nos résultats. L'avant dernière couche qui était là originellement restera avec le même nombre de neurones égales à 10. On travaillera ici avec les données normalisées.

```
num_classes = 10
nhidden1=150
nhidden2=150
input_shape = (img_height*img_width)

xi = Input(shape=input_shape)

mean = X_train.mean(axis=0)
std = X_train.std(axis=0) + 1.0
x = Lambda(lambda image, mu, std: (image - mu) / std,
           arguments={'mu': mean, 'std': std})(xi)

x = Dense(nhidden1)(x)
x = Activation('relu')(x)
x = Dense(nhidden2)(x)
x = Activation('relu')(x)
x = Dense(num_classes)(x)
y = Activation('softmax')(x)
model = Model(inputs=[xi], outputs=[y])


model.summary()
```

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 784)]             0
_________________________________________________________________
lambda_2 (Lambda)            (None, 784)               0
_________________________________________________________________
dense_6 (Dense)              (None, 150)               117750
_________________________________________________________________
activation_6 (Activation)    (None, 150)               0
_________________________________________________________________
dense_7 (Dense)              (None, 150)               22650
_________________________________________________________________
activation_7 (Activation)    (None, 150)               0
_________________________________________________________________
dense_8 (Dense)              (None, 10)                1510
_________________________________________________________________
activation_8 (Activation)    (None, 10)                0
=================================================================
Total params: 141,910
Trainable params: 141,910
Non-trainable params: 0
_________________________________________________________________
```

Après compilation et entrainement de ce modèle, on interprète les résultats.

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/20
54000/54000 [==============================] - 5s 91us/sample - loss: 0.2664 - acc: 0.9197 - val_loss: 0.1224 - val_acc: 0.9
617
Epoch 2/20
54000/54000 [==============================] - 4s 74us/sample - loss: 0.0963 - acc: 0.9705 - val_loss: 0.1006 - val_acc: 0.9
677
```

...

```
Epoch 19/20
54000/54000 [==============================] - 4s 75us/sample - loss: 0.0069 - acc: 0.9978 - val_loss: 0.1267 - val_acc: 0.9
790
Epoch 20/20
54000/54000 [==============================] - 4s 72us/sample - loss: 0.0082 - acc: 0.9974 - val_loss: 0.1621 - val_acc: 0.9
783
Test loss: 0.17071686534380595
Test accuracy: 0.9751
Test loss: 0.1061718339617364
Test accuracy: 0.9725
```

On arrive à un **val_loss** égale à 0,1621 et une **val_acc** égale à 0,973 ainsi qu'une **Test loss** du meilleur modèle encore meilleur avec 0.1061718339617364 et une accuracy de 0,9725. On peut dire que notre modèle est bien précis au niveau de l'accuracy.

2.2. Régularisation

Cette étape on va ajouter des régularisateur afin d'améliorer la performance et combattre le sur-apprentissage. Il y a deux types de régularisateur :

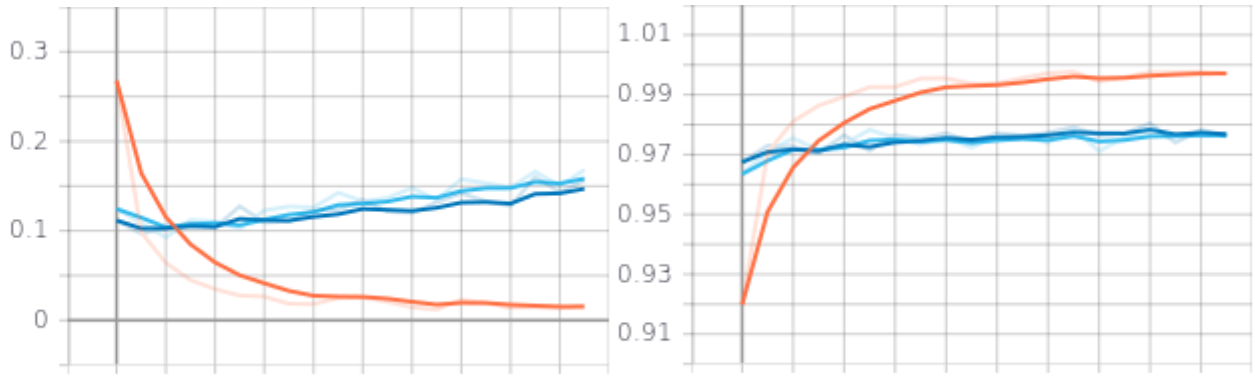
- Un régularisateur L2.
- Un régularisateur Dropout.

La régularisation L2, c'est de rajouter un terme dans la fonction de loss et la minimiser de la forme $\lambda \sum_i w_i^2$. Le paramètre λ va être trouver après plusieurs expériences en surveillant la performance à chaque fois. Il est assez petit (**i.e.** 10^{-5}). On peut le définir directement sur la première couche de la façon suivante :

```
...
x = Dense(nhidden1, kernel_regularizer=regularizers.l2(l2_reg))(x)
```

Après entrainement, on les résultats suivants :

```
Epoch 19/20
54000/54000 [==============================] - 5s 87us/sample - loss: 0.0159 - acc: 0.9972 - val_loss: 0.1284 - val_acc: 0.9
780
Epoch 20/20
54000/54000 [==============================] - 5s 97us/sample - loss: 0.0122 - acc: 0.9983 - val_loss: 0.1333 - val_acc: 0.9
775
Test loss: 0.1309506074987352
Test accuracy: 0.9797
Test loss: 0.09943590401709079
Test accuracy: 0.9726
```



Il est clair que la performance s'est nettement améliorée. On a une **Test loss** de 0,130950607 4987352 est une **Test accuracy** de 97 %. Encore mieux, si on évalue le meilleur modèle qui n ous donne un **Test loss** de 0,09943590401709079 et une performance de 97% aussi.

La régularisation Dropout est une technique qui va aléatoirement ignorer un sous ensembl e de neurones. Ça améliore la performance est combat le sur-apprentissage. On peut choisi r quel nombre de neurone on va ignorer et comme pour la L2, on modifie à chaque fois en surveillant la performance. On rajoute une couche de Dropout après la couche d'activation **relu**.

```
x = Dense(nhidden1)(x)
x = Activation('relu')(x)
if(args.dropout):
   x = Dropout(0.5)(x)
```

0,5 veut dire, on ignorer 50% de nos neurones

3. Réseau de neurones convolutif VANILLA CNN

3.1. Architecture du modèle.

Nous allons maintenant implémenter un réseau de neurones convolutionnels. Ces réseaux de neurones sont constitués de couches de convolution suivies de couches max-pooling.

Pour cet exercice, nous avons besoin des méthodes de **keras** suivantes, en plus de celles déjà vue précédemment :

- Conv2D
- MaxPooling2D

Premièrement, on va recharger notre dataset telle quelle avec **mnist.load_data** car contrairement aux modèles précédents, on n'a besoin d'aplatir nos images sur des vecteurs. Les couches de convolution s'attendent à des matrices.

Notre modèle sera constitué d'une couche de convolution suivit d'une couche de max-pooling suivit de deux couches denses. On commence par ajouter à notre modèle la première couche qui est une couche de convolution qui utilise des filtre 5x5 et ces filtres, ils vont balayer toute l'image afin de détecter des patterns présents sur l'image. On choisit 64 filtres avec une fonction d'activation **relu**. Ensuite, on rajoute une couche de max-pooling. Ça permet de réduire les dimensions. En effet, on va choisir la taille du filtre 2x2 et ça va permettre de réduire le nombre de paramètres dans le réseau et au modèle de voir plus largement sur l'image. On rajoute une couche dense de 128 neurones mais avant cela, il faut rajouter une couche **Flatten** puisque les couches denses acceptent des vecteurs. On rajoute une fonction d'activation **relu** une autre couche dense de 64 neurones suivie de l'ouput avec 10 neurones nombre de classes avec un fonction d'activation **softmax.**

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

num_train  = X_train.shape[0]
num_test   = X_test.shape[0]

img_height = X_train.shape[1]
img_width  = X_train.shape[2]
X_train = X_train.reshape((num_train, img_height, img_width))
X_test  = X_test.reshape((num_test, img_height, img_width))
img_rows = img_height
img_cols = img_width
input_shape = (img_rows, img_cols, 1)
```

```
xi = Input(shape=input_shape)
mean = X_train.mean(axis=0)
std = X_train.std(axis=0) + 1.0
x = Lambda(lambda image, mu, std: (image - mu) / std,
           arguments={'mu': mean, 'std': std})(xi)
x = Conv2D(filters=64,
       kernel_size=5, strides=1)(x)
x = Activation('relu')(x)
x = MaxPooling2D(pool_size=2, strides=2)(x)

x = Flatten()(x)
x = Dense(128)(x)
x = Activation('relu')(x)

x = Dense(64)(x)
x = Activation('relu')(x)
y = Dense(10, activation='softmax')(x)
```