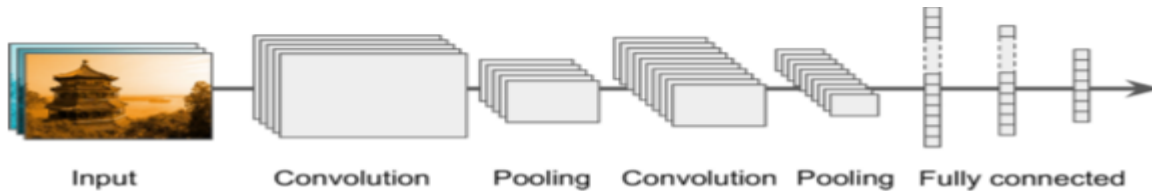
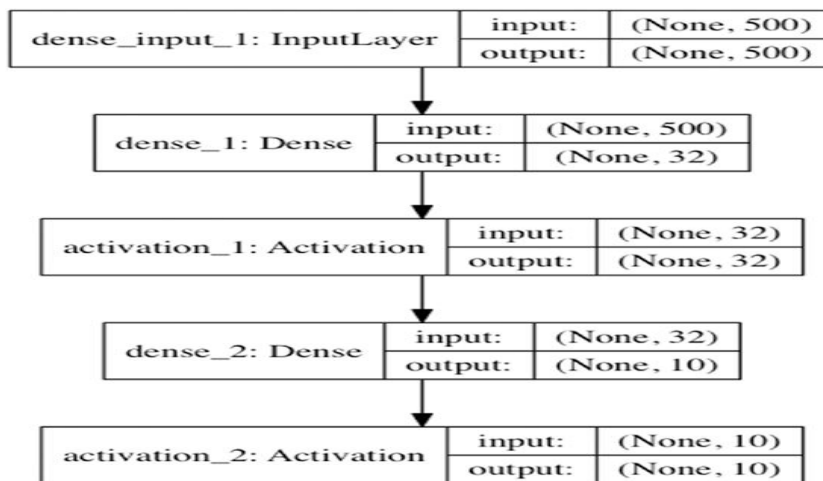


TP n2 en Apprentissage profond

Reseaux de neurones convolutifs



Exercice 1 : Entraîner, évaluer, et compiler le modèle CNN multiclasse illustré dans la figure suivante, utiliser l'outil TensorFlow



Le modèle est défini à l'aide de la construction séquentielle,

- La dimensionnalité des données d'entrées est 500
- La première couche est une couche Dense, de dimensionnalité d'entrée égale 500, et produira une sortie de dimensionnalité 32
- La fonction d'activation est relu.
- La deuxième couche Dense, avec la dimensionnalité de sortie à 10.
- La deuxième fonction d'activation est softmax et l'entropie catégorielle comme fonction de perte.

1. Entraîner le modèle avec des données d'entrées aléatoires entre 0 et 1, de dimension 1000×500, et le nombre de classes égale à 10.

```
data = np.random.random((1000, 500))
labels = to_categorical(np.random.randint(10, size=(1000, 1)))
```

2. Compiler le modèle, utiliser l'optimiseur 'rmsprop', et la fonction de perte entropie croisée **loss** : c'est une fonction qui va servir à mesurer l'écart entre les prédictions de notre IA et les résultats attendus. Elle évalue donc la justesse du CNN et permet de mieux l'adapter aux données si besoin ! Nous allons utiliser « categorical_crossentropy » comme loss, car on a des données de type « catégories » en sortie de l'algorithme.

optimizer : c'est un algorithme qui va dicter comment mettre à jour le CNN pour diminuer le loss, et avoir donc de meilleures prédictions. Ici on s'appuiera sur « adam » (adaptive moment estimation), très souvent utilisé.

metrics : c'est exactement comme le loss, sauf que la metrics n'est PAS utilisée par le CNN, à l'inverse du loss qui sert pour la mise à jour des variables du CNN via l'optimizer. On utilisera cette fois « accuracy », sans que cela ait de réelle importance pour nous.

```
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils.np_utils import to_categorical
import matplotlib.pyplot as plot
import pandas as pd
import numpy as np
from keras.utils.np_utils import to_categorical

model = Sequential()
model.add(Dense(32, input_dim=500))
...
...
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
              metrics=['categorical_accuracy'])
```

3. Evaluer le modèle : TensorFlow a mis de coté les informations de précision et perte lors de la phase d'apprentissage et pour chaque epochs. Il nous suffit de les récupérer :

```
loss = pd.DataFrame(mon_cnn.history.history)
```

Visualiser graphiquement les resultats de Accuracy avant et apres la compilation du modele

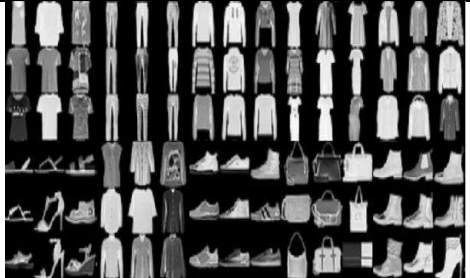
```
loss[['categorical_accuracy', 'val_categorical_accuracy']].plot()
loss[['loss', 'val_loss']].plot()
```

4. Visualiser graphiquement les resultats de la fonction de perte avant et apres la compilation du modele

Exercice 2 : Dataset fashion_mnsit

Ce jeu de données contient plus de 70000 images en niveau de gris : Chaque image est un carré de 28×28 pixels.

Ce jeu de données permet d'identifier 10 types d'objets (étiquettes). Ces étiquettes sont codifiées avec des nombres de 0 à 9:

0 – T-shirt/haut	5 – Sandale	
1 – Pantalon	6 – Chemise	
2 – Pullover	7 – Sneaker	
3 – Robe	8 – Sac	
4 – Manteau	9 – Bottine	

1. Preparation de donnees

- 1.1. Charger ce jeu de données

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

dataset_fashion_mnsit = tf.keras.datasets.fashion_mnist
(X_train, y_train), (X_test, y_test) = dataset_fashion_mnsit.load_data()
```

- 1.2. Lister le nombre d'occurrences de chaque étiquette,
- 1.3. Visualiser la première image, et leur étiquette

```
plt.imshow(X_train[0])
print('étiquette de image 1:', y_train[0])
```

2. Normaliser les données

Les réseaux de neurones sont très sensibles à la normalisation des données. Dans le cas d'images en niveau de gris c'est très simple et comme les pixels vont de 0 à 255, diviser tous les pixels par 255, et afficher la taille de données d'apprentissage et de test

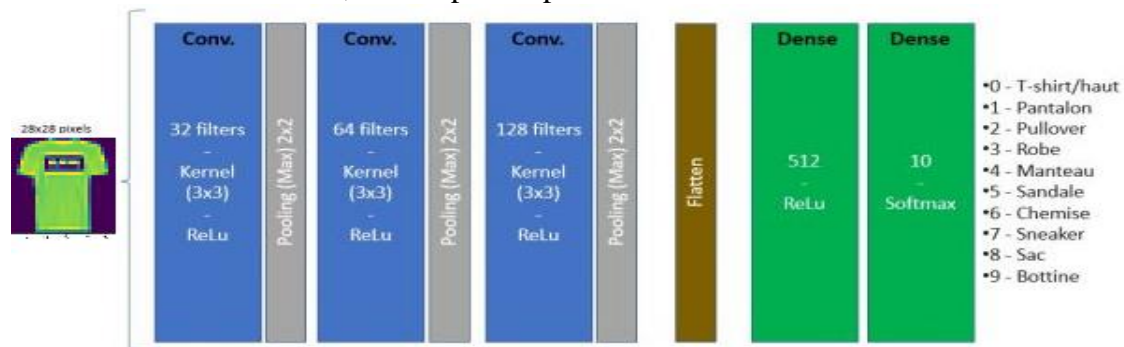
```
X_train = X_train / 255
X_test = X_test / 255

print(f'Taille de données d'apprentissage: {X_train.shape}, Test: {X_test.shape}')
```

3. Etant donné que l'on a des images en niveau de gris (couleur vert), il nous manque une dimension (couleur : RVB). Ajouter cette dimension aux données d'apprentissage et de test

```
X_train = X_train.reshape(60000, 28, 28, 1)
X_test = X_test.reshape(10000, 28, 28, 1)
```

5. Créer le Modèle CNN, défini par les paramètres suivants :



```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Conv2D, Input, Flatten, Dropout, MaxPooling2D
from tensorflow.keras.models import Model
from sklearn.metrics import classification_report, confusion_matrix
from tensorflow.keras.callbacks import EarlyStopping
cnn = tf.keras.Sequential()
```

4. Entraîner le modèle avec un nombre de filtres progressif 32, 64 puis 128

Pour déclarer une convolution, la syntaxe est d'appeler **Conv2D** avec en paramètres le nombre de filtres, les dimensions du noyau (3x3), en précisant `input_shape` : la taille des données d'entrée (uniquement sur la 1ère couche du CNN) et enfin l'activation.

Pourquoi Conv2D et pas juste « Conv » ? Dans Keras, la possibilité existe d'utiliser des [convolutions à 1, 2 ou 3 dimensions](#), les convolutions s'appliquent séparément à chaque channel de l'image. Elles sont donc en 2D et non en 3D.

```
cnn.add(Conv2D(filters=32, kernel_size=(3,3), input_shape=(28, 28, 1), activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
....
# Couche de sortie (classes de 0 à 9)
cnn.add(Dense(10, activation='softmax'))
```

- Utiliser la technique du **callback EarlyStopping** qui permet d'arrêter l'apprentissage dès lors que le modèle commence à faire du sur-apprentissage.

Afin de ne pas tâtonner sur le nombre d'epochs à réaliser,

```
early_stop = EarlyStopping(monitor='val_loss', patience=2)
```

early_stop : Arrêt l'apprentissage lorsqu'une métrique surveillée a cessé de s'améliorer.

En supposant que le but d'une formation est de minimiser la perte. Avec cela, la métrique à surveiller serait « perte » et le mode serait « min ». Une boucle d'apprentissage **model.fit()** vérifiera à la fin de chaque époque si la perte ne diminue plus, en tenant compte du min_delta et de la patience, le cas échéant. Une fois qu'il ne diminue plus, **model.stop_training** est marqué **True** et la formation se termine. Le paramètre à surveiller doit être disponible dans les logs dict. Pour ce faire, passer la perte ou la métrique à **model.compile()**.

Syntaxe :

```
tf.keras.callbacks.EarlyStopping(  
    monitor="val_loss",  
    min_delta=0,  
    patience=0,  
    verbose=0,  
    mode="auto",  
    baseline=None,  
    restore_best_weights=False,  
)
```

monitor : metrique a surveiller

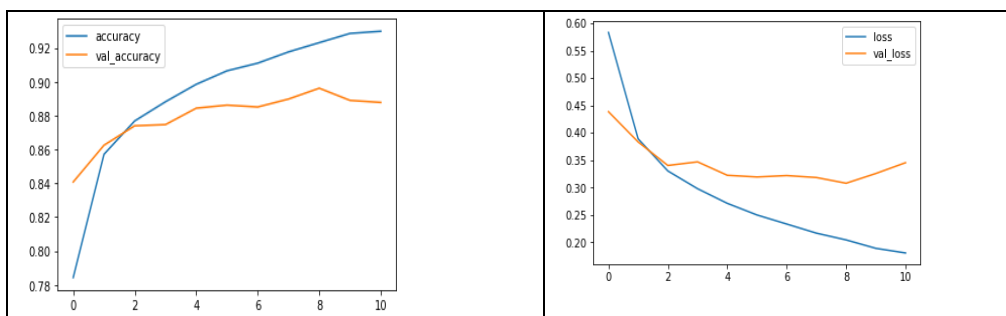
patiente : Nombre d'époques sans amélioration après lesquelles l'apprentissage sera arrêté.

- 5.1. Compiler le modèle, utiliser l'optimiseur 'adam', et la fonction de perte entropie croisée
- 5.2. Afficher le détail des paramètres du modèle entraîné

6. Exécuter le modèle sur les données d'apprentissage, avec le nombre d'epochs (itérations / rétro-propagation) égale 25:

Qu'est-ce que vous remarquez à propos de la condition d'earlystopping ?

7. Évaluer le modèle : afficher les valeurs de perte, et de l'accuracy



8. Prediction : Déterminer la classe de la première image, et afficher uniquement l'étiquette de prob max.