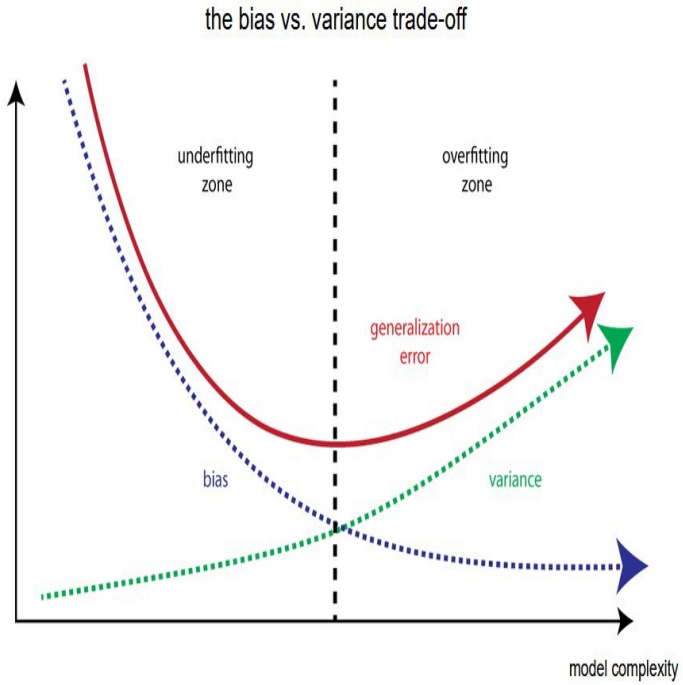
# Biais et Variance

Le but de ce notebook est de mieux estimer la qualité du modèle. En apprentissage automatique, certains modèles sont vraiment trop simples et ne tiennent pas compte des relations qui peuvent exister entre les données d'apprentissage afin de pouvoir améliorer les prédictions. Ces modèles possèdent un **Biais élevé**. Au contraire, d'autres modèles sont trop complexes et en cherchant des relations dans les variables vont au final ajouter du bruit. Ainsi en modifiant un peu les données, les prédictions s'avèrent très différentes car le modèle est trop sensible et réagit de manière excessive au changement de données, i.e. **forte Variance**. Ils ne sont donc pas généralisable. La difficulté est donc d'arriver à trouver un bon compromis entre Biais et Variance comme l'illustre la figure suivante :




## Installation

Avant de commencer, il est nécessaire de déjà posséder dans son environnement toutes les librairies utiles. Dans la seconde cellule nous importons toutes les librairies qui seront utiles à ce notebook. Il se peut que, lorsque vous lanciez l'éxecution de cette cellule, une soit absente. Dans ce cas il est nécessaire de l'installer. Pour cela dans la cellule suivante utiliser la commande :

*! pip install nom_librairie*

**Attention :** il est fortement conseillé lorsque l'une des librairies doit être installer de relancer le kernel de votre notebook.

**Remarque :** même si toutes les librairies sont importées dès le début, les librairies utiles pour des fonctions présentées au cours de ce notebook sont ré-importées de manière à indiquer d'où elles viennent et ainsi faciliter la réutilisation de la fonction dans un autre projet.

```
# utiliser cette cellule pour installer les librairies manquantes
# pour cela il suffit de taper dans cette cellule : !pip install nom_librairie_manquante
# d'exécuter la cellule et de relancer la cellule suivante pour voir si tout se passe bien
# recommencer tant que toutes les librairies ne sont pas installées ...

# sous Colab il faut déjà intégrer ces deux librairies

#!pip install

# eventuellement ne pas oublier de relancer le kernel du notebook
```

```
# Importation des différentes librairies utiles pour le notebook

#Sickit learn met régulièrement à jour des versions et
#indique des futurs warnings.
#ces deux lignes permettent de ne pas les afficher.
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

# librairies générales
import pickle
import pandas as pd
from scipy.stats import randint
import numpy as np
```

```
import string
import time
import base64
import re
import sys
import copy


# librairie affichage
import matplotlib.pyplot as plt
import seaborn as sns
from pylab import rcParams
from matplotlib import rc

# TensorFlow et keras
import tensorflow as tf
from keras import layers
from keras import models
from keras import optimizers
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from keras.preprocessing.image import ImageDataGenerator
#from keras.preprocessing.image import img_to_array, load_img
from tensorflow.keras.utils import img_to_array
from keras.callbacks import ModelCheckpoint, EarlyStopping
import keras
from sklearn.metrics import confusion_matrix
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.preprocessing import image
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.model_selection import ShuffleSplit
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.model_selection import learning_curve
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier

sns.set(style='whitegrid', palette='muted')

rcParams['figure.figsize'] = 20, 12
```

Pour pouvoir sauvegarder sur votre répertoire Google Drive, il est nécessaire de fournir une autorisation. Pour cela il suffit d'éxecuter la ligne suivante et de saisir le code donné par Google.

```
# pour monter son drive Google Drive local
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content/gdrive", force_r
```

Corriger éventuellement la ligne ci-dessous pour mettre le chemin vers un répertoire spécifique dans votre répertoire Google Drive :

```
my_local_drive='/content/gdrive/My Drive/Colab Notebooks/ML2_2023_2024'
# Ajout du path pour les librairies, fonctions et données
sys.path.append(my_local_drive)
# Se positionner sur le répertoire associé
%cd $my_local_drive

%pwd
```

```
/content/gdrive/My Drive/Colab Notebooks/ML2_2023_2024
'/content/gdrive/My Drive/Colab Notebooks/ML2_2023_2024'
```

```
# fonctions utilities (affichage, confusion, etc.)
from MyNLPUtilities import *
```

Fonction d'affichage des courbes loss et accuracy :

```
def plot_learningcurve(train_scores, test_scores):
  #
  # training and test mean and std
  #
  train_mean = np.mean(train_scores, axis=1)
  train_std = np.std(train_scores, axis=1)
  test_mean = np.mean(test_scores, axis=1)
  test_std = np.std(test_scores, axis=1)
  plt.figure(1,figsize=(16,6))
  # Plot the learning curve

  plt.plot(train_sizes, train_mean, color='blue', marker='o', markersize=5, label='Training Accuracy')
  plt.fill_between(train_sizes, train_mean + train_std, train_mean - train_std, alpha=0.15, color='blue')
  plt.plot(train_sizes, test_mean, color='green', marker='+', markersize=5, linestyle='--', label='Validation Accuracy')
  plt.fill_between(train_sizes, test_mean + test_std, test_mean - test_std, alpha=0.15, color='green')
  plt.title('Learning Curve')
  plt.xlabel('Training Data Size')
  plt.ylabel('Model accuracy')
  plt.grid()
  plt.legend(loc='lower right')
  plt.show()

def plot_curves_confusion (history,confusion_matrix,class_names):
  plt.figure(1,figsize=(16,6))
  plt.gcf().subplots_adjust(left = 0.125, bottom = 0.2, right = 1,
                            top = 0.9, wspace = 0.25, hspace = 0)

  # division de la fenêtre graphique en 1 ligne, 3 colonnes,
  # graphique en position 1 - loss fonction

  plt.subplot(1,3,1)
```

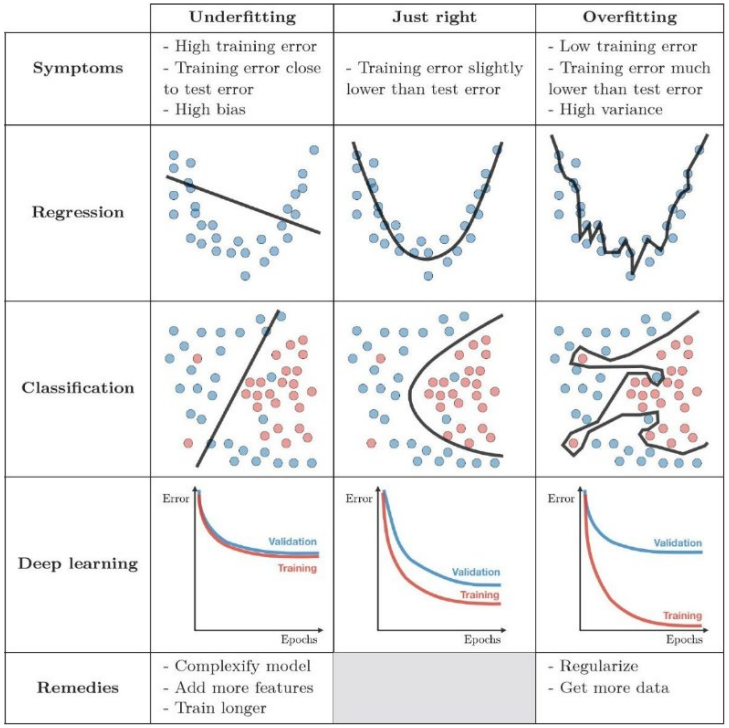
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['Training loss', 'Validation loss'], loc='upper left')
# graphique en position 2 - accuracy
plt.subplot(1,3,2)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['Training accuracy', 'Validation accuracy'], loc='upper left')

# matrice de correlation
plt.subplot(1,3,3)
sns.heatmap(conf,annot=True,fmt="d",cmap='Blues',xticklabels=class_names, yticklabels=class_names)# label=class_names)
# labels, title and ticks
plt.xlabel('Predicted', fontsize=12)
#plt.set_label_position('top')
#plt.set_ticklabels(class_names, fontsize = 8)
#plt.tick_top()
plt.title("Correlation matrix")
plt.ylabel('True', fontsize=12)
#plt.set_ticklabels(class_names, fontsize = 8)
plt.show()
```

# Bias-Variance Tradeoff

Les modèles à **biais élevé** sont *sous-adaptés* (**Underfit**) aux données d'apprentissage et l'erreur de prédiction est élevée à la fois sur les données d'entraînement et les données de test. Les modèles à **forte variance** sont *surajustés* (**Overfit**) aux données d'apprentissage et leur

erreur de prédiction est généralement faible sur les données d'entraînement mais élevée sur les données de test, i.e. ils manquent de généralisation.

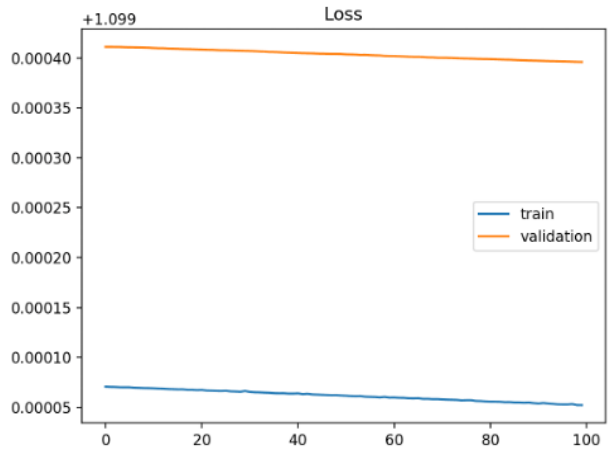
La figure suivante illustre différents cas en considérant des approches de type régression (voir M1), de type modèles de classification traditionnels (voir M1) ou bien de modèles de deep learning. Les courbes obtenues permettent de mettre en évidence où se situe un modèle :



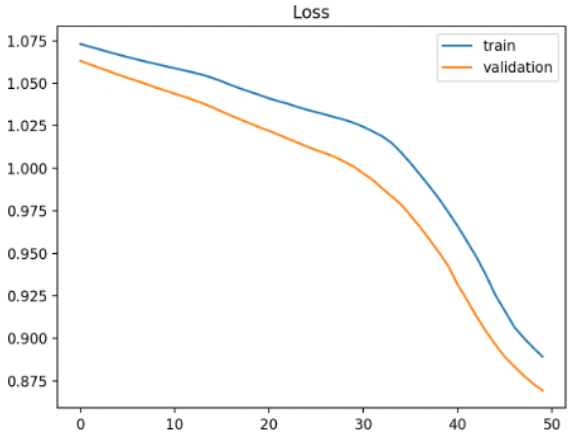

(source : https://i.pinimg.com/originals/72/e2/22/72e222c1542539754df1d914cb671bd7.png)

Les figures ci-dessous illustrent différents exemples d'**underfit**, d'**overfit** et de **goodfit** que nous allons voir. Elles sont données pour vous permettre, par la suite, de rapidement vérifier où se situe votre modèle et donc éventuellement de l'améliorer.

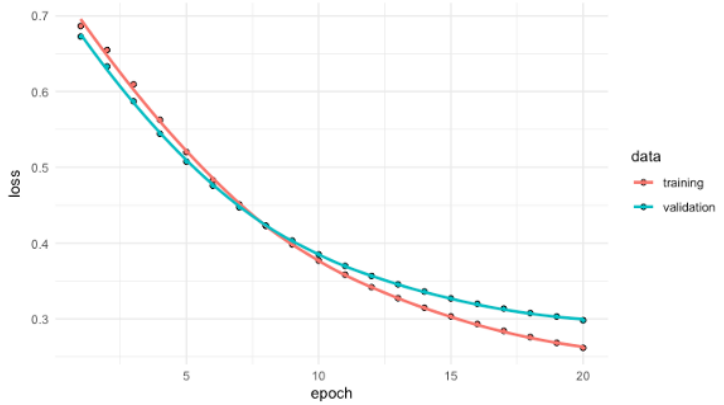
La figure ci-dessous illustre des exemples d'**underfit** en utilisant la courbe de *loss*:




(a)




(b)



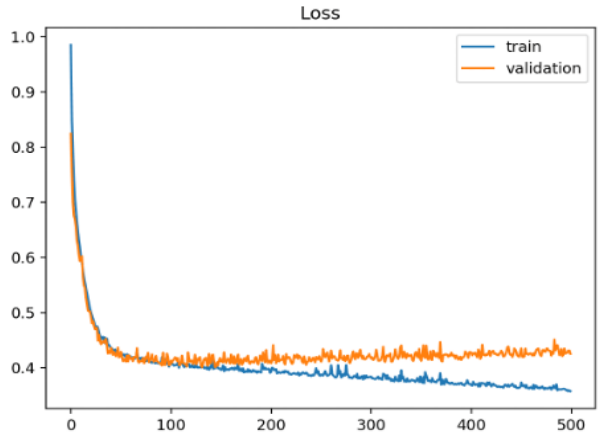

(c)

La partie *(a)* illustre un cas où le modèle est trop simple. Il n'est pas capable de prendre en compte la complexité du jeu de données. La partie *(b)* illustre un cas où la *loss* diminue et continue à diminuer jusqu'en bas. Cela indique que le modèle est capable d'apprendre encore et qu'il a été arrêté prématurément. Enfin la partie *(c)* montre également un exemple où le processus s'est arrêté trop tôt.

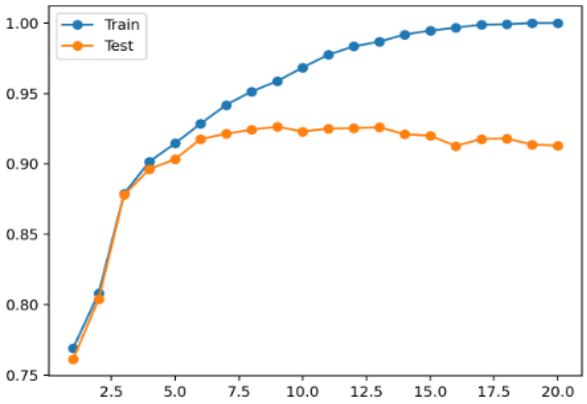
En résumé, une courbe d'apprentissage montre un **underfitting** si :

- La courbe *loss* reste stable quelque soit l'entraînement.
- La courbe *loss* continue de diminuer jusqu'à la fin de l'entraînement.

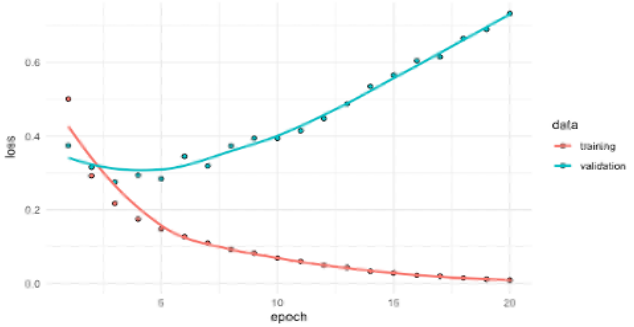
La figure ci-dessous illustre des exemples d'**overfit** :




(a)




(b)




(c)

La partie *(a)* illustre un exemple d'overfitting où nous voyons bien sur la courbe de *loss* qu'à un moment donné la validation ne perd plus rien. Le modèle apprend "trop bien" les données d'entrainement et même les bruits associés. Il faut, dans ce cas là, stopper l'entraînement plus tôt (e.g. Early Stop). La partie *(b)* illustre le même cas mais sur la courbe de l'*accuracy*. Enfin la partie *(c)* montre un overfitting qui démarre très tôt et où la courbe a une forme de *U*. Généralement cela indique le fait que le modèle apprend trop vite et que le learning rate est trop élevé.

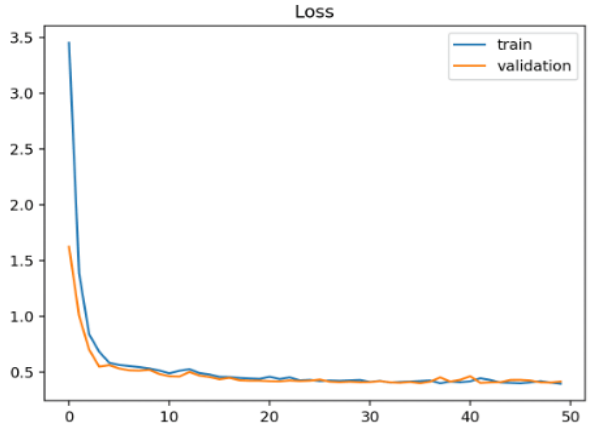
En résumé, les courbes d'apprentissage montrent un **overfitting** si :

- La courbe *loss* pour l'entraînement continue de diminuer.
- La courbe *loss* pour la validation diminue jusqu'à un certain point et recommence à augmenter.

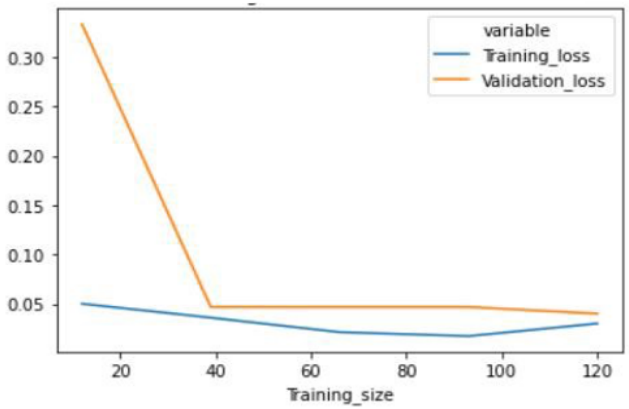
Remarque : le point d'inflexion de *loss* pour la validation peut être le point auquel l'apprentissage pourrait être interrompu.

Un **goodfitting** peut se reprérer lorque la *loss* pour l'apprentissage et le test diminue jusqu'à un point de stabilité avec un écart minimal entre les deux valeurs.

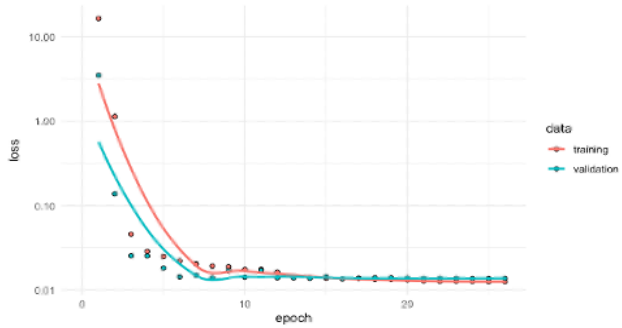
La figure ci-dessous illustre des exemples de **goodfitting** :




(a)




(b)




(c)

La partie *(a)* montre que la *loss* décroit vers un point de stabilité aussi bien pour le test que pour l'apprentissage. L'écart entre les deux s'appelle « **écart de généralisation** ». La partie *(b)* illustre un exemple où, au début, la *loss* est assez faible pour l'entraînement et que pour la validation elle diminue rapidement puis très progressivement lors de l'ajout d'exemples d'apprentissage. Par la suite elle s'applatit diminuant ainsi l'écart de généralisation. Enfin, la partie *(c)* montre également que la *loss* pour l'apprentissage et le test décroit vers un point stable avec un écart de généralisation très faible.

En résumé, les courbes d'apprentissage montrent un **goodfitting** si :

- La courbe *loss* pour l'entraînement diminue jusqu'à un point de stabilité.
- La courbe *loss* pour la validation diminue jusqu'à un point de stabilité et présente un petit écart avec la *loss* pour l'entraînement.

**Solutions :** Il existe bien entendu des moyens pour corriger le modèle :

- **Underfitting :**

1. Comme l'underfitting est lié au fait que le modèle est trop simple, il ne capture pas les relations dans les données. Dans cette situation, la meilleure stratégie consiste à augmenter la complexité du modèle. Il peut s'agir d'ajouter d'autres features, par exemple en classification traditionnelle, ou bien d'augmenter le nombre de paramètres du modèle d'apprentissage profond. L'augmentation de la complexité du

modèle va forcément entraîner une amélioration des performances. Il faut toutefois veiller à ne pas avoir une erreur d'entraînement de zéro, i.e. dans ce cas on arrive en **overfitting**.

2. Augmentation du nombre d'epochs. Si le modèle montre qu'il peut apprendre plus, il est possible d'augmenter le nombre d'epochs. Comme la courbe montre qu'il faut du temps pour atteindre un minimum, il faut peut être aussi regarder le *learning rate* pour accélérer le processus.
3. Bien vérifier que les données soient bien mélangées à chaque epoch. Penser à la validation croisée.

- **Overfitting :**

1. Ajouter plus de données d'entraînement. Le fait d'avoir un ensemble de données plus important et diversifié aide généralement à améliorer les performances du modèle, i.e. un modèle qui généralise mieux.
2. Augmentation. S'il n'est pas possible de rajouter des données, il est peut être possible d'utiliser les données présentes et d'en faire des variations. Pour des données numériques on peut penser à générer des données en tenant compte de la moyenne et de la variance initiales. Il existe par exemple pour les images des librairies spécifiques (déformation, rotation, zoom, etc.). La difficulté est d'avoir des données augmentées qui restent proches du réel.
3. Early Stopping. Keras propose par exemple de pouvoir arrêter l'apprentissage lorsque des métriques (loss, accuracy, etc.) n'évoluent pas pendant un certain temps.
4. Ajout de dropout. Les couches de type dropout permettent de "bloquer" des neurones différents à chaque étape pendant l'apprentissage et ainsi éviter que le modèle soit trop proche des données. Attention les dropout sont utilisés lors de l'apprentissage et non pas avec lors du test de validation. La conséquence est que les courbes sont un peu différentes et que l'accuracy pour l'apprentissage peut être un peu moins bon.
5. Régularisation. La régularisation est un terme supplémentaire ajouté à la fonction de perte pour imposer une pénalité sur les poids importants des paramètres de réseau afin de réduire le surapprentissage. Le dropout ou le early stopping sont des sortes de méthodes de régularisation. Il en existe d'autres.

## Jeux de données non représentatif

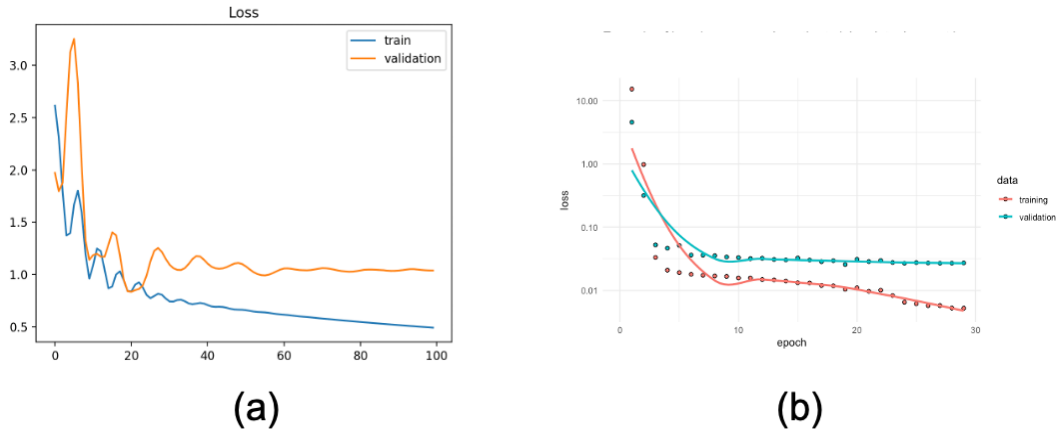
Dans cette section, nous abordons des courbes qui peuvent apparaître parce que les jeux de données (apprentissage et validation) ne sont pas représentatifs. Cela peut arriver par exemple lors d'un *test_train_split* avec de mauvais paramètres si les classes ne sont pas équilibrées ou bien s'il manque des données.

Lors de l'apprentissage, le modèle va avoir un jeu d'apprentissage et un jeu de validation. Nous considérons les conséquences que cela peut avoir sur ces deux jeux.

**Jeu d'apprentissage non représentatif**

Un jeu d'apprentissage non représentatif signifie que l'ensemble de données d'apprentissage ne fournit pas suffisamment d'informations pour apprendre le problème, par rapport à l'ensemble des données de validation utilisée pour l'évaluer. Cela peut se produire si l'ensemble de données d'apprentissage comporte trop peu d'exemples par rapport à l'ensemble de données de validation.

La figure ci-dessous illustre des exemples de **jeux d'apprentissage non représentatifs** :




La partie *(a)* illustre un cas où les *loss* pour l'entraînement et la validation montrent une amélioration mais qu'il reste un écart important entre les deux courbes. La partie *(b)* montre un autre exemple ou il reste toujours un fort écart entre les deux *loss*.

Il existe différentes solutions.

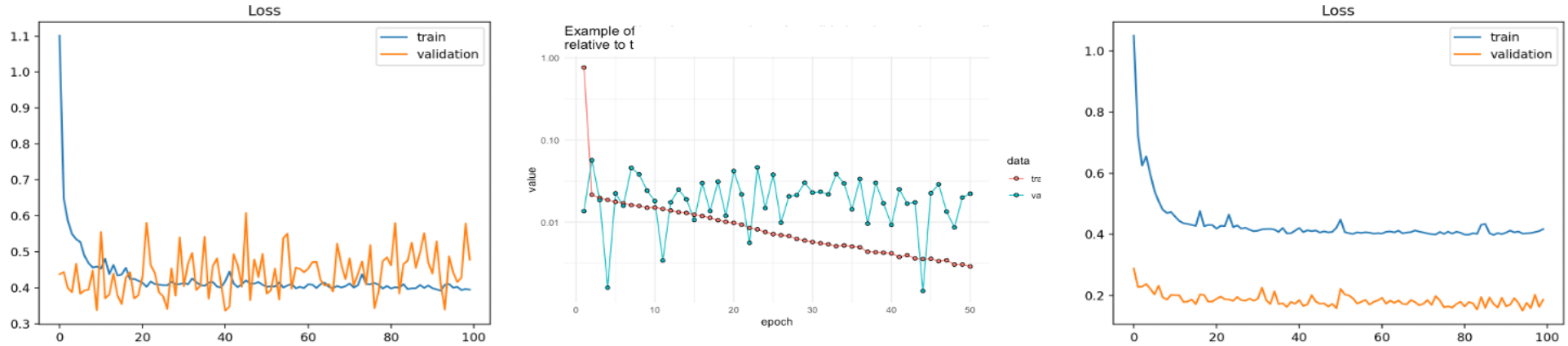
**Solutions :**

1. Ajouter plus de données. Il n'y a peut être pas assez de données pour capturer ce qu'il y a dans les données d'apprentissage et de validation. Eventuellement faire de la data augmentation : ajouts de données similaires, déformation d'images, etc.
2. Vérifier que l'échantillonage est bien fait. Si les classes sont déséquilibrées s'assurer que les échantillons sont représentatifs, par exemple utiliser (*stratify=y*) dans train_test_split ou faire un *StratifiedKfold* plutôt qu'un Kfold.
3. Faire de la cross validation pour être certain que toutes les données puissent être dans le jeu d'apprentissage et de validation.

**Jeu de validation non représentatif**

Un jeu de validation non représentatif signifie qu'il ne fournit pas suffisamment d'informations pour évaluer la capacité du modèle à généraliser. Cela peut se produire si l'ensemble de données de validation contient trop peu d'exemples par rapport à l'ensemble de données d'apprentissage.

La figure ci-dessous illustre des exemples de **jeux de validation non représentatifs** :




(a) (b) (c)

Les parties *(a)* et *(b)* illustrent deux cas où même si la *loss* d'entraînement se comporte bien, la *loss* en validation montre de grandes variations et pas ou peu d'amélioration. La partie *(c)* montre une *loss* validation inférieure à la *loss* d'apprentissage. Cela indique que l'ensemble de données de validation peut être plus facile à prédire pour le modèle que l'ensemble de données d'apprentissage.

Les solutions sont assez similaires aux précédentes. **Solutions :**

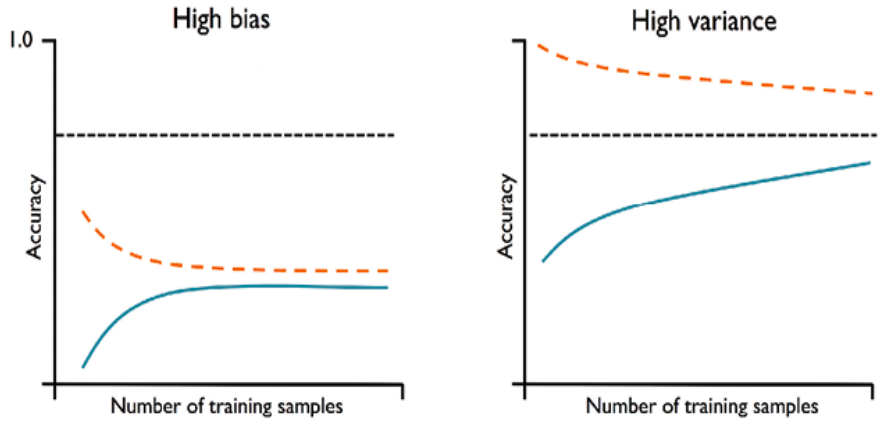
1. Ajouter plus de données dans le jeu de validation (cf. taille de l'échantillon)
2. Effectuer une cross validation.

Dans ce notebook nous présentons les librairies ou les moyens qui peuvent être utilisés pour vérifier comment se situe un modèle. Nous allons dans un premier temps voir les learning curves de scikit learn et enfin nous examinons comment afficher les courbes dans le cas d'un réseau profond.

Nous illustrons, par la suite avec différentes jeux de données classiques des cas d'overfitting ou d'underfitting à l'aide de différents modèles et via différentes méthodes.

# Utilisation des learning curve (Scikit Learn)

La courbe d'apprentissage est utilisée pour évaluer les performances des modèles avec un nombre variable d'échantillons d'apprentissage. Ceci est réalisé en surveillant les scores d'apprentissage et de validation (précision du modèle) avec un nombre croissant d'échantillons d'apprentissage. Dans la figure suivante, l'entraînement correspond à la ligne pointillée orange, la validation à la ligne bleue et la précision en pointillée noire).



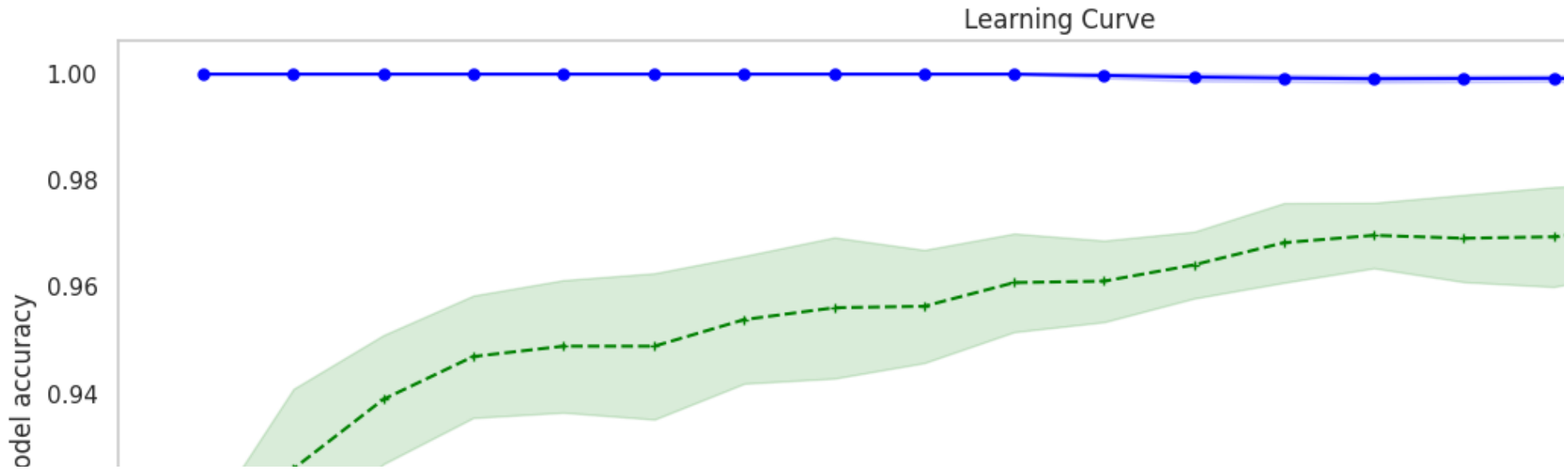

La méthode *learning_curve* utilise comme paramètres un estimateur (classifieur), les jeux de données et la validation croisée. Il est également nécessaire de spécifier les tailles du train_sizes qui seront testées.

Par la suite, pour illustrer différents comportements nous utiliserons le jeu de donnéees digits disponible dans Scickit Learn. Il contient des données correspondant à des images de chiffres écrits à la main : 10 classes où chaque classe fait référence à un chiffre.

```
dataset = datasets.load_digits()

# X contient les variables prédictives et y la variable à prédire
X, y = dataset.data, dataset.target

# normalisation du jeu de données
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)
```

### Cas d'une regression

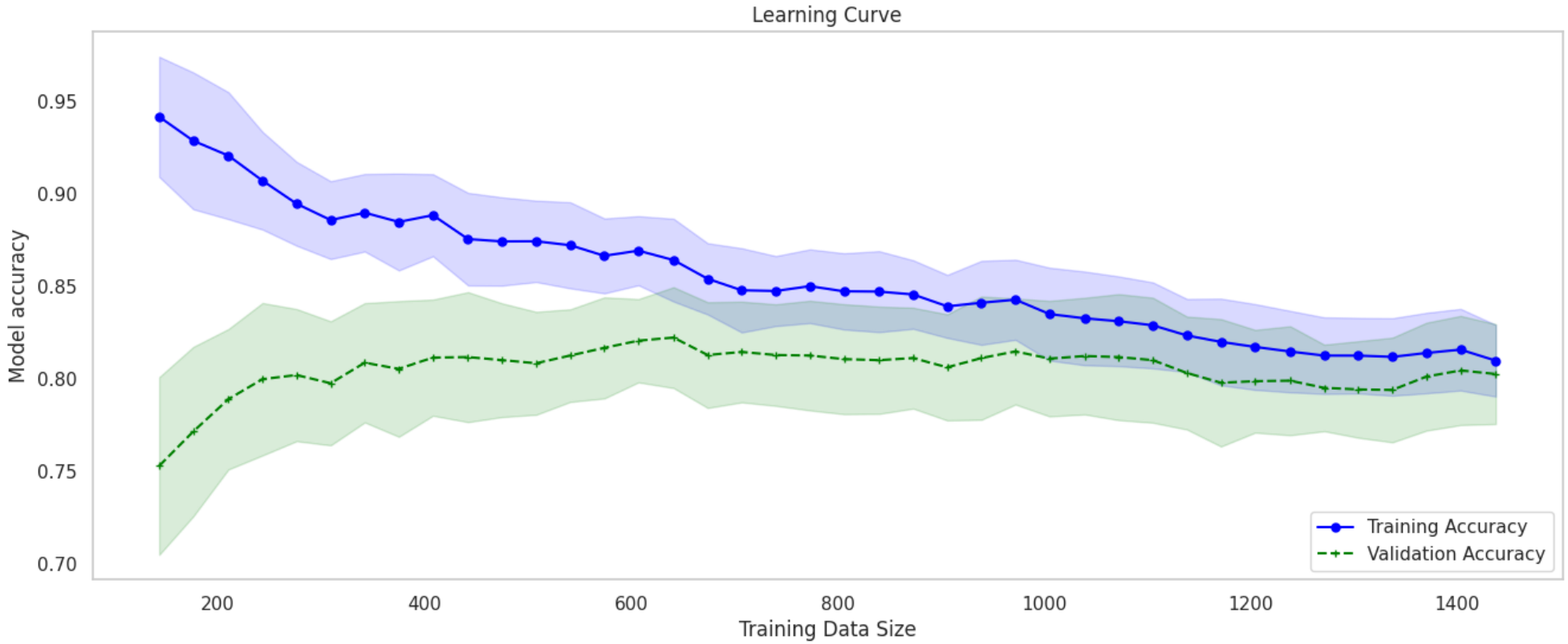
```
cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
estimator=LogisticRegression(solver='lbfgs', random_state=1, max_iter=1000)
train_sizes, train_scores, test_scores = learning_curve(estimator=estimator, X=X, y=y,
                                                        cv=cv, train_sizes=np.linspace(0.1, 1.0, 20),
                                                       n_jobs=1)

plot_learningcurve(train_scores,test_scores)
```




La courbe montre que quelque soit la taille du jeu de données d'apprentissage le score sur l'apprentissage ne change pas. De la même manière pour le jeu de validation un palier est obtenu. C'est un cas typique d'**underfitting**. Le résulat était aussi un peu attendu.

**Cas du Naive Bayes**

```
cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
estimator=GaussianNB()
train_sizes, train_scores, test_scores = learning_curve(estimator=estimator, X=X, y=y,
                                                        cv=cv, train_sizes=np.linspace(0.1, 1.0, 40),
                                                       n_jobs=1)

plot_learningcurve(train_scores,test_scores)
```

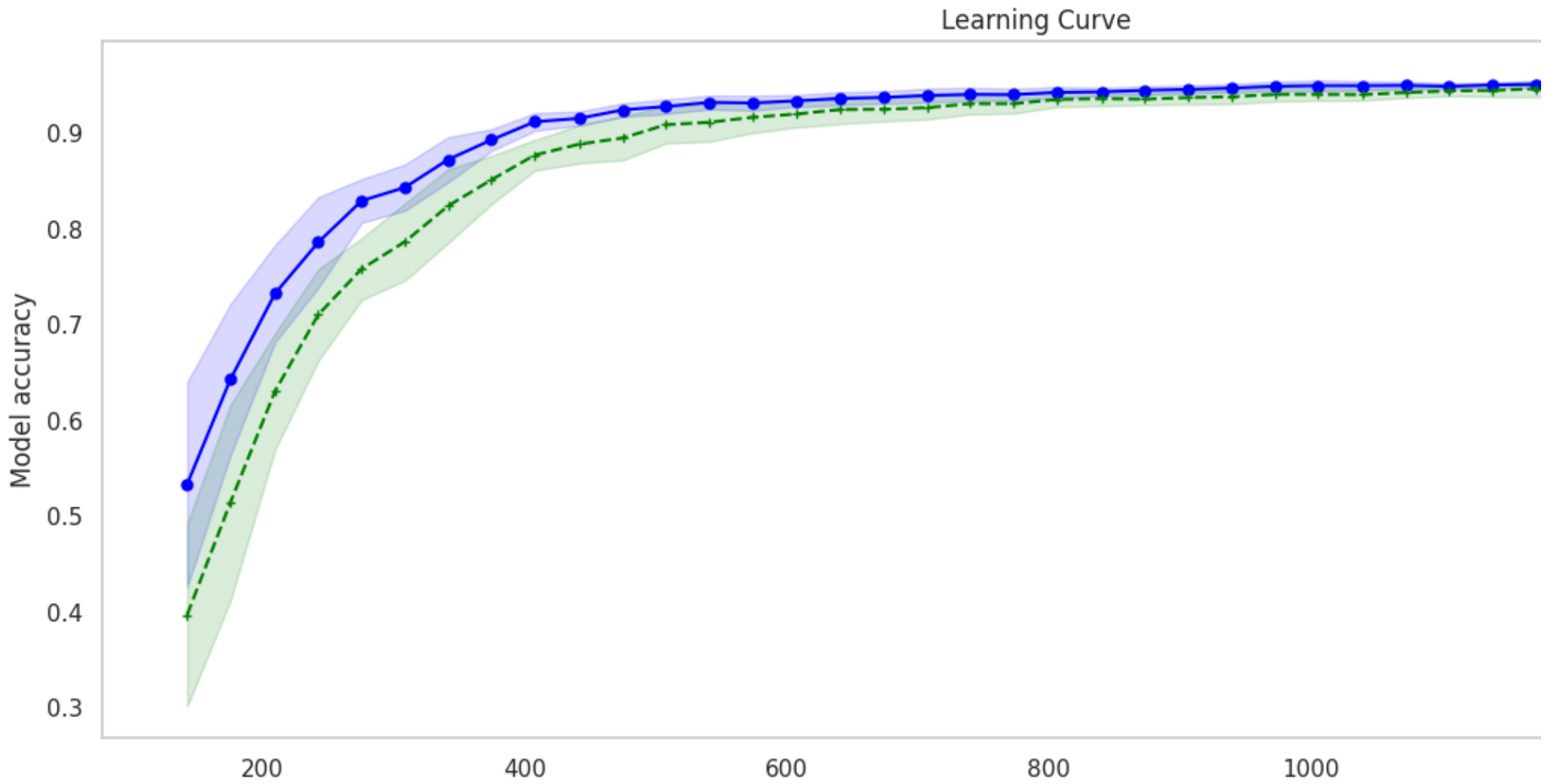



Pour Naïve Bayes, le score de validation et le score d'apprentissage convergent vers une valeur assez faible avec l'augmentation de la taille de l'ensemble d'apprentissage. Cela illustre que le fait d'avoir plus de données d'entrainement ne serait pas utile. Par contre, cette forme de courbe peut se retrouver souvent dans des jeux de données plus complexes : le score d'entraînement est très élevé au début et diminue et le score de validation croisée est très faible au début et augmente.

**Cas du SVM**

```
cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
estimator = SVC(gamma=0.001)
train_sizes, train_scores, test_scores = learning_curve(estimator=estimator, X=X, y=y,
                                                        cv=cv, train_sizes=np.linspace(0.1, 1.0, 40),
                                                       n_jobs=1)
```

```
plot_learningcurve(train_scores,test_scores)
```




À partir de la courbe, nous pouvons voir que plus la taille de l'ensemble d'apprentissage augmente, plus la courbe de score d'apprentissage et la courbe de score de validation croisée convergent. La précision de la validation croisée augmente à mesure que nous ajoutons plus de données d'apprentissage. L'ajout de plus de données d'apprentisage augmente sans doute la généralisation.

# Apprentissage profond

Pour illustrer la partie apprentissage profond, nous utilisons le jeu de données classiques pima-indians-diabetes qui permet de prédire si des femmes au Perou sont sujet au diabète. Vous pouvez récupérer le fichier en décommantant la ligne suivante :

```
!wget http://www.lirmm.fr/~poncelet/Ressources/pima-indians-diabetes.csv
```

```
--2023-09-07 17:29:26--  http://www.lirmm.fr/~poncelet/Ressources/pima-indians-diabetes.csv
Resolving www.lirmm.fr (www.lirmm.fr)... 193.49.104.251
Connecting to www.lirmm.fr (www.lirmm.fr)|193.49.104.251|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://www.lirmm.fr/~poncelet/Ressources/pima-indians-diabetes.csv [following]
--2023-09-07 17:29:27--  https://www.lirmm.fr/~poncelet/Ressources/pima-indians-diabetes.csv
Connecting to www.lirmm.fr (www.lirmm.fr)|193.49.104.251|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 24146 (24K) [text/csv]
Saving to: ‘pima-indians-diabetes.csv’

pima-indians-diabet 100%[===================>]  23.58K  --.-KB/s    in 0.1s

2023-09-07 17:29:28 (194 KB/s) - ‘pima-indians-diabetes.csv’ saved [24146/24146]
```

```
# # creation d'un dataframe pour récupérer les données
df = pd.read_csv('pima-indians-diabetes.csv', delimiter=';')

# Note: 0 — Non Diabetic Patient and 1 — Diabetic Patient

# affichage des 5 premières lignes du jeu de données
display(df.head())


print ("Repartition des données",df['Outcome'].value_counts())
```

```
print ("Matrice de corrélation")
sns.heatmap(df.corr().round(decimals=2), annot = True, linewidths=.2);

class_names=['non diab','diabete']
```

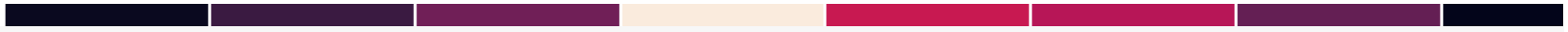
| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 6 | 148 | 72 | 35 | 0 | 33.6 | 0.627 | 50 | 1 |
| **1** | 1 | 85 | 66 | 29 | 0 | 26.6 | 0.351 | 31 | 0 |
| **2** | 8 | 183 | 64 | 0 | 0 | 23.3 | 0.672 | 32 | 1 |
| **3** | 1 | 89 | 66 | 23 | 94 | 28.1 | 0.167 | 21 | 0 |
| **4** | 0 | 137 | 40 | 35 | 168 | 43.1 | 2.288 | 33 | 1 |

```
Repartition des données 0    500
1    268
Name: Outcome, dtype: int64
Matrice de corrélation
```


Dans cette section nous analysons différentes architectures de réseaux de neurones.

## Cas d'underfitting

En analysant la matrice de corrélation, nous constatons que *SkinThickness* n'est vraiment pas corrélé à la classe (0.07). L'objectif ici est de montrer qu'avec une telle donnée nous ne sommes pas capable de réaliser un bon modèle de prédiction. Il s'agit juste d'une illustration bien sûr !. Nous résumons donc les variables prédictives à ce seul attribut.



```
X = df[['SkinThickness']]
# Il n'est pas nécessaire de normaliser ici car il n'y a qu'un attribut
y = df.Outcome

# création d'un jeu de test et d'apprentissage
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)
```

Nous allons créer tout d'abord un modèle simple composé de 2 neurones dans la première couche. La couche de sortie (avec 1 neurone) contient bien évidemment une *sigmoid* car il s'agit ici d'une classification binaire.

Paramètres (ils ne sont pas forcément optimisés ici) :

- Nombre d'epochs : 20
- Batchsize : 32
- Optimizer : Adam
- Loss : binary_crossentropy (classification binaire)

```
# definition du modele keras très simple
model = Sequential()
model.add(Dense(2, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# compilation du modèle
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit le modèle sur le jeu de données. Nous mettons un batchsize de 32
history=model.fit(X_train, y_train, validation_data=(X_test,y_test),epochs=20, batch_size=32)

y_pred=model.predict(X_test)
y_pred[y_pred <= 0.5] = 0.
y_pred[y_pred > 0.5] = 1.

print ("\nRappel du modèle testé")
print (model.summary(),'\n')
conf=confusion_matrix(y_test,y_pred)
plot_curves_confusion (history,conf,class_names)
```

```
Epoch 1/20
17/17 [==============================] - 2s 31ms/step - loss: 4.8920 - accuracy: 0.5681 - val_loss: 3.9485 - val_accu
Epoch 2/20
17/17 [==============================] - 0s 13ms/step - loss: 4.6875 - accuracy: 0.5642 - val_loss: 3.7845 - val_accu
Epoch 3/20
17/17 [==============================] - 0s 7ms/step - loss: 4.4892 - accuracy: 0.5642 - val_loss: 3.6294 - val_accur
Epoch 4/20
17/17 [==============================] - 0s 7ms/step - loss: 4.2934 - accuracy: 0.5642 - val_loss: 3.4717 - val_accur
Epoch 5/20
17/17 [==============================] - 0s 9ms/step - loss: 4.1151 - accuracy: 0.5642 - val_loss: 3.3319 - val_accur
Epoch 6/20
17/17 [==============================] - 0s 7ms/step - loss: 3.9489 - accuracy: 0.5642 - val_loss: 3.2059 - val_accur
Epoch 7/20
17/17 [==============================] - 0s 9ms/step - loss: 3.7893 - accuracy: 0.5642 - val_loss: 3.0727 - val_accur
Epoch 8/20
17/17 [==============================] - 0s 11ms/step - loss: 3.6310 - accuracy: 0.5642 - val_loss: 2.9539 - val_accu
Epoch 9/20
17/17 [==============================] - 0s 7ms/step - loss: 3.4875 - accuracy: 0.5642 - val_loss: 2.8318 - val_accur
Epoch 10/20
17/17 [==============================] - 0s 15ms/step - loss: 3.3289 - accuracy: 0.5642 - val_loss: 2.7017 - val_accu
Epoch 11/20
17/17 [==============================] - 0s 15ms/step - loss: 3.1786 - accuracy: 0.5642 - val_loss: 2.5778 - val_accu
Epoch 12/20
17/17 [==============================] - 0s 7ms/step - loss: 3.0335 - accuracy: 0.5642 - val_loss: 2.4724 - val_accur
Epoch 13/20
17/17 [==============================] - 0s 10ms/step - loss: 2.8989 - accuracy: 0.5642 - val_loss: 2.3632 - val_accu
Epoch 14/20
17/17 [==============================] - 0s 14ms/step - loss: 2.7657 - accuracy: 0.5642 - val_loss: 2.2541 - val_accu
Epoch 15/20
17/17 [==============================] - 0s 8ms/step - loss: 2.6235 - accuracy: 0.5642 - val_loss: 2.1457 - val_accur
Epoch 16/20
17/17 [==============================] - 0s 6ms/step - loss: 2.5031 - accuracy: 0.5642 - val_loss: 2.0515 - val_accur
Epoch 17/20
17/17 [==============================] - 0s 5ms/step - loss: 2.3843 - accuracy: 0.5642 - val_loss: 1.9583 - val_accur
Epoch 18/20
17/17 [==============================] - 0s 5ms/step - loss: 2.2753 - accuracy: 0.5642 - val_loss: 1.8730 - val_accur
Epoch 19/20
17/17 [==============================] - 0s 5ms/step - loss: 2.1691 - accuracy: 0.5642 - val_loss: 1.7837 - val_accur
Epoch 20/20
17/17 [==============================] - 0s 4ms/step - loss: 2.0660 - accuracy: 0.5642 - val_loss: 1.7057 - val_accur
8/8 [==============================] - 0s 2ms/step
```

```
Rappel du modèle testé
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (None, 2)                 4         

 dense_1 (Dense)             (None, 1)                 3         
```

Les courbes indiquent clairement un **underfitting**. Au niveau du *loss*, nous constatons que la courbe pour la validation est en dessous de celle du training. Elles ne convergent jamais. Nous voyons également que l'*accuracy* ne change pas et que la validation est au dessus. Généralement pour la *loss* lorsque la courbe de validation reste en dessous il s'agit d'un **underfitting**.

Ajouter des epochs à ce modèle ne changera rien comme l'illustre l'expérience suivante avec :

- epochs : 40

```
# definition du modele keras très simple
model = Sequential()
model.add(Dense(2, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# compilation du modèle
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit le modèle sur le jeu de données. Nous mettons un batchsize de 32
history=model.fit(X_train, y_train, validation_data=(X_test,y_test),epochs=40, batch_size=32)

y_pred=model.predict(X_test)
y_pred[y_pred <= 0.5] = 0.
y_pred[y_pred > 0.5] = 1.

print ("\nRappel du modèle testé")
print (model.summary(),'\n')

conf=confusion_matrix(y_test,y_pred)
plot_curves_confusion (history,conf,class_names)
```

Nous sommes toujours en **underfitting**, il faut soit essayer d'avoir un modèle plus complexe, soit ajouter des attributs (ici nous avons triché ... donc on peut mettre d'autres features).

### Essai avec un modèle plus complexe

Dans un premier temps nous essayons de rendre le modèle plus complexe. La première couche cachée contient maintenant 8 neurones et nous ajoutons également une couche dense composée de 16 neurones.

```
# definition du modele keras plus complexe
model = Sequential()
model.add(Dense(8, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

```
# compilation du modèle
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit le modèle sur le jeu de données. Nous mettons un batchsize de 32
history=model.fit(X_train, y_train, validation_data=(X_test,y_test),epochs=40, batch_size=32)

y_pred=model.predict(X_test)
y_pred[y_pred <= 0.5] = 0.
y_pred[y_pred > 0.5] = 1.

print ("\nRappel du modèle testé")
print (model.summary(),'\n')
conf=confusion_matrix(y_test,y_pred)
plot_curves_confusion (history,conf,class_names)
```

Comme attendu, nous voyons qu'il y a toujours un **underfitting**. En effet les données seules, SkinThickness avec une mauvaise corrélation, ne sont pas suffisante pour faire une bonne prédiction.

### Ajout d'autres attributs

Nous pouvons constater sur la matrice de corrélation que les attributs *Age*, *BMI*,*Glucose* semblent être plus en corrélation avec la variable prédictive (respectivement 0.24, 0.29 et 0.47). Ici il faut faire attention car les valeurs sont sur des domaines très différents. Il faut penser à normaliser.

```
X = df[['Age','BMI','Glucose']]
y = df.Outcome
# normalisation du jeu de données
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)

#y = df["Label"].values
# création d'un jeu de test et d'apprentissage
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)
```

Avec le modèle simple :

```
# definition du modele keras très simple
model = Sequential()
model.add(Dense(2, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# compilation du modèle
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit le modèle sur le jeu de données. Nous mettons un batchsize de 32
history=model.fit(X_train, y_train, validation_data=(X_test,y_test),epochs=40, batch_size=32)

y_pred=model.predict(X_test)
y_pred[y_pred <= 0.5] = 0.
y_pred[y_pred > 0.5] = 1.
print ("\nRappel du modèle testé")
print (model.summary(),'\n')
conf=confusion_matrix(y_test,y_pred)
plot_curves_confusion (history,conf,class_names)
```

```
Epoch 1/40
17/17 [==============================] - 1s 14ms/step - loss: 0.7375 - accuracy: 0.3599 - val_loss: 0.7243 - val_accu
Epoch 2/40
17/17 [==============================] - 0s 5ms/step - loss: 0.7296 - accuracy: 0.3774 - val_loss: 0.7175 - val_accur
Epoch 3/40
17/17 [==============================] - 0s 4ms/step - loss: 0.7230 - accuracy: 0.4066 - val_loss: 0.7116 - val_accur
Epoch 4/40
17/17 [==============================] - 0s 4ms/step - loss: 0.7166 - accuracy: 0.4591 - val_loss: 0.7061 - val_accur
Epoch 5/40
17/17 [==============================] - 0s 5ms/step - loss: 0.7110 - accuracy: 0.5097 - val_loss: 0.7020 - val_accur
Epoch 6/40
17/17 [==============================] - 0s 5ms/step - loss: 0.7061 - accuracy: 0.5623 - val_loss: 0.6977 - val_accur
Epoch 7/40
17/17 [==============================] - 0s 5ms/step - loss: 0.7016 - accuracy: 0.6089 - val_loss: 0.6939 - val_accur
Epoch 8/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6975 - accuracy: 0.6226 - val_loss: 0.6903 - val_accur
Epoch 9/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6935 - accuracy: 0.6420 - val_loss: 0.6868 - val_accur
Epoch 10/40
17/17 [==============================] - 0s 4ms/step - loss: 0.6898 - accuracy: 0.6556 - val_loss: 0.6837 - val_accur
Epoch 11/40
17/17 [==============================] - 0s 4ms/step - loss: 0.6866 - accuracy: 0.6615 - val_loss: 0.6810 - val_accur
Epoch 12/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6834 - accuracy: 0.6634 - val_loss: 0.6782 - val_accur
Epoch 13/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6803 - accuracy: 0.6654 - val_loss: 0.6754 - val_accur
Epoch 14/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6773 - accuracy: 0.6673 - val_loss: 0.6732 - val_accur
Epoch 15/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6745 - accuracy: 0.6634 - val_loss: 0.6711 - val_accur
Epoch 16/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6721 - accuracy: 0.6654 - val_loss: 0.6688 - val_accur
Epoch 17/40
17/17 [==============================] - 0s 4ms/step - loss: 0.6695 - accuracy: 0.6673 - val_loss: 0.6666 - val_accur
Epoch 18/40
17/17 [==============================] - 0s 4ms/step - loss: 0.6669 - accuracy: 0.6673 - val_loss: 0.6640 - val_accur
Epoch 19/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6643 - accuracy: 0.6673 - val_loss: 0.6619 - val_accur
Epoch 20/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6616 - accuracy: 0.6673 - val_loss: 0.6594 - val_accur
```

Nous constatons qu'il n'y a pas d'amélioration. Le modèle est vraiment trop simple.

Avec le modèle complexe nous avons :

```
17/17 [==============================] - 0s 5ms/step - loss: 0.6533 - accuracy: 0.6790 - val_loss: 0.6519 - val_accur
```

```
# definition du modele keras plus complexe
model = Sequential()
model.add(Dense(8, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# compilation du modèle
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit le modèle sur le jeu de données. Nous mettons un batchsize de 32


history=model.fit(X_train, y_train, validation_data=(X_test,y_test),epochs=40, batch_size=32)

y_pred=model.predict(X_test)
y_pred[y_pred <= 0.5] = 0.
y_pred[y_pred > 0.5] = 1.

print ("\nRappel du modèle testé")
print (model.summary(),'\n')
conf=confusion_matrix(y_test,y_pred)
plot_curves_confusion (history,conf,class_names)
```

```
Epoch 1/40
17/17 [==============================] - 1s 15ms/step - loss: 0.7136 - accuracy: 0.4747 - val_loss: 0.7009 - val_accu
Epoch 2/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6799 - accuracy: 0.5642 - val_loss: 0.6718 - val_accur
Epoch 3/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6512 - accuracy: 0.6751 - val_loss: 0.6472 - val_accur
Epoch 4/40
17/17 [==============================] - 0s 4ms/step - loss: 0.6277 - accuracy: 0.7062 - val_loss: 0.6280 - val_accur
Epoch 5/40
17/17 [==============================] - 0s 4ms/step - loss: 0.6057 - accuracy: 0.7276 - val_loss: 0.6066 - val_accur
Epoch 6/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5823 - accuracy: 0.7510 - val_loss: 0.5872 - val_accur
Epoch 7/40
17/17 [==============================] - 0s 6ms/step - loss: 0.5620 - accuracy: 0.7588 - val_loss: 0.5701 - val_accur
Epoch 8/40
17/17 [==============================] - 0s 4ms/step - loss: 0.5440 - accuracy: 0.7568 - val_loss: 0.5569 - val_accur
Epoch 9/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5286 - accuracy: 0.7626 - val_loss: 0.5443 - val_accur
Epoch 10/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5145 - accuracy: 0.7588 - val_loss: 0.5349 - val_accur
Epoch 11/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5038 - accuracy: 0.7665 - val_loss: 0.5276 - val_accur
Epoch 12/40
17/17 [==============================] - 0s 4ms/step - loss: 0.4952 - accuracy: 0.7646 - val_loss: 0.5225 - val_accur
Epoch 13/40
17/17 [==============================] - 0s 4ms/step - loss: 0.4884 - accuracy: 0.7704 - val_loss: 0.5191 - val_accur
Epoch 14/40
17/17 [==============================] - 0s 5ms/step - loss: 0.4832 - accuracy: 0.7763 - val_loss: 0.5157 - val_accur
Epoch 15/40
17/17 [==============================] - 0s 5ms/step - loss: 0.4788 - accuracy: 0.7743 - val_loss: 0.5140 - val_accur
Epoch 16/40
17/17 [==============================] - 0s 5ms/step - loss: 0.4760 - accuracy: 0.7763 - val_loss: 0.5130 - val_accur
Epoch 17/40
17/17 [==============================] - 0s 5ms/step - loss: 0.4741 - accuracy: 0.7724 - val_loss: 0.5123 - val_accur
Epoch 18/40
17/17 [==============================] - 0s 4ms/step - loss: 0.4726 - accuracy: 0.7763 - val_loss: 0.5130 - val_accur
Epoch 19/40
17/17 [==============================] - 0s 5ms/step - loss: 0.4719 - accuracy: 0.7821 - val_loss: 0.5132 - val_accur
Epoch 20/40
17/17 [==============================] - 0s 5ms/step - loss: 0.4708 - accuracy: 0.7802 - val_loss: 0.5117 - val_accur
Epoch 21/40
```

```
17/17 [==============================] - 0s 5ms/step - loss: 0.4707 - accuracy: 0.7763 - val_loss: 0.5103 - val_accur
Epoch 22/40
17/17 [==============================] - 0s 4ms/step - loss: 0.4698 - accuracy: 0.7763 - val_loss: 0.5100 - val_accur
Epoch 23/40
17/17 [==============================] - 0s 4ms/step - loss: 0.4692 - accuracy: 0.7763 - val_loss: 0.5099 - val_accur
```

Nous pouvons constater que le résultat est nettement meilleur. Par contre vers l'epoch 10, le score loss pour la validation diminue, cela montre qu'il commence à y avoir de l'**overfitting** : le modèle apprend "par coeur" les données d'apprentissage. Nous pouvons aussi le constater sur l'accuracy.

```
17/17 [==============================] - 0s 5ms/step - loss: 0.4676 - accuracy: 0.7763 - val_loss: 0.5083 - val_accur
```

# Cas d'overfitting

Lors de la dernière expérimentation nous avons constaté qu'il y avait de l'**overfitting**, i.e. le modèle commence à trop suivre les données et ne généralise pas. Nous allons donc rajouter de la régularisation. Une manière simple consiste à rajouter des *dropout* qui ont pour but de ne pas activer un pourcentage de neurones lors de l'apprentissage.

**Ajout de Dropout 0.2**

```
# definition du modele keras plus complexe
model = Sequential()
model.add(Dense(8, input_dim=X_train.shape[1], activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

# compilation du modèle
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit le modèle sur le jeu de données. Nous mettons un batchsize de 32
history=model.fit(X_train, y_train, validation_data=(X_test,y_test),epochs=40, batch_size=32)

y_pred=model.predict(X_test)
y_pred[y_pred <= 0.5] = 0.
y_pred[y_pred > 0.5] = 1.
```

```
print ("\nRappel du modèle testé")
print (model.summary(),'\n')
conf=confusion_matrix(y_test,y_pred)
plot_curves_confusion (history,conf,class_names)
```

```
Epoch 1/40
17/17 [==============================] - 1s 15ms/step - loss: 0.6903 - accuracy: 0.6167 - val_loss: 0.6672 - val_accu
Epoch 2/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6698 - accuracy: 0.6537 - val_loss: 0.6461 - val_accur
Epoch 3/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6425 - accuracy: 0.6848 - val_loss: 0.6280 - val_accur
Epoch 4/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6355 - accuracy: 0.6790 - val_loss: 0.6123 - val_accur
Epoch 5/40
17/17 [==============================] - 0s 4ms/step - loss: 0.6149 - accuracy: 0.7004 - val_loss: 0.5994 - val_accur
Epoch 6/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6069 - accuracy: 0.6829 - val_loss: 0.5860 - val_accur
Epoch 7/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5840 - accuracy: 0.7121 - val_loss: 0.5755 - val_accur
Epoch 8/40
17/17 [==============================] - 0s 4ms/step - loss: 0.5998 - accuracy: 0.6926 - val_loss: 0.5645 - val_accur
Epoch 9/40
17/17 [==============================] - 0s 6ms/step - loss: 0.5790 - accuracy: 0.7218 - val_loss: 0.5541 - val_accur
Epoch 10/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5735 - accuracy: 0.7315 - val_loss: 0.5426 - val_accur
Epoch 11/40
```

Nous voyons que le modèle est nettement meilleur.

**Remarque :** il se peut pour l'accuracy que pour la validation le score soit meilleur que pour l'entrainement. Cela est lié au fait que le modèle, lorsqu'il apprend, applique les dropout alors que dans la validation ces dropout n'apparaissent plus.

Une autre manière de faire de la régularisation est de faire du **EarlyStopping**. Le principe consiste à suivre une mesure (e.g. *val_score*) et si elle n'évolue pas pendant une période donnée, i.e. *patience*, d'arrêter l'apprentissage (C.f. https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping).

**Remarque :** Dans notre contexte particulier, petit jeu de données et avec les variations liées au batchsize, il est difficile de "tuner" la patience.

```
Epoch 18/40
```

```
early_stop = keras.callbacks.EarlyStopping(
  monitor='val_accuracy',
  patience=7
)
```

```
# definition du modele keras plus complexe
model = Sequential()
model.add(Dense(8, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# compilation du modèle
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit le modèle sur le jeu de données. Nous mettons un batchsize de 32
history=model.fit(X_train, y_train, validation_data=(X_test,y_test),epochs=40, batch_size=32, callbacks=[early_stop])

y_pred=model.predict(X_test)
y_pred[y_pred <= 0.5] = 0.
y_pred[y_pred > 0.5] = 1.

print ("\nRappel du modèle testé")
print (model.summary(),'\n')
conf=confusion_matrix(y_test,y_pred)
plot_curves_confusion (history,conf,class_names)
```

```
Epoch 1/40
17/17 [==============================] - 1s 15ms/step - loss: 0.6478 - accuracy: 0.6284 - val_loss: 0.6337 - val_accu
Epoch 2/40
17/17 [==============================] - 0s 7ms/step - loss: 0.6226 - accuracy: 0.6848 - val_loss: 0.6113 - val_accur
Epoch 3/40
17/17 [==============================] - 0s 4ms/step - loss: 0.6010 - accuracy: 0.7023 - val_loss: 0.5914 - val_accur
Epoch 4/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5810 - accuracy: 0.7062 - val_loss: 0.5755 - val_accur
Epoch 5/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5655 - accuracy: 0.7140 - val_loss: 0.5638 - val_accur
Epoch 6/40
17/17 [==============================] - 0s 6ms/step - loss: 0.5527 - accuracy: 0.7276 - val_loss: 0.5535 - val_accur
Epoch 7/40
17/17 [==============================] - 0s 4ms/step - loss: 0.5406 - accuracy: 0.7393 - val_loss: 0.5440 - val_accur
Epoch 8/40
17/17 [==============================] - 0s 4ms/step - loss: 0.5278 - accuracy: 0.7374 - val_loss: 0.5336 - val_accur
Epoch 9/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5141 - accuracy: 0.7432 - val_loss: 0.5230 - val_accur
Epoch 10/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5016 - accuracy: 0.7588 - val_loss: 0.5144 - val_accur
Epoch 11/40
17/17 [==============================] - 0s 5ms/step - loss: 0.4927 - accuracy: 0.7626 - val_loss: 0.5076 - val_accur
Epoch 12/40
17/17 [==============================] - 0s 12ms/step - loss: 0.4837 - accuracy: 0.7763 - val_loss: 0.5032 - val_accu
Epoch 13/40
17/17 [==============================] - 0s 15ms/step - loss: 0.4793 - accuracy: 0.7763 - val_loss: 0.5004 - val_accu
Epoch 14/40
17/17 [==============================] - 0s 14ms/step - loss: 0.4756 - accuracy: 0.7763 - val_loss: 0.4989 - val_accu
Epoch 15/40
17/17 [==============================] - 0s 11ms/step - loss: 0.4737 - accuracy: 0.7743 - val_loss: 0.4966 - val_accu
Epoch 16/40
17/17 [==============================] - 0s 11ms/step - loss: 0.4701 - accuracy: 0.7704 - val_loss: 0.4966 - val_accu
Epoch 17/40
17/17 [==============================] - 0s 18ms/step - loss: 0.4684 - accuracy: 0.7763 - val_loss: 0.4961 - val_accu
Epoch 18/40
17/17 [==============================] - 1s 34ms/step - loss: 0.4667 - accuracy: 0.7743 - val_loss: 0.4968 - val_accu
Epoch 19/40
17/17 [==============================] - 0s 23ms/step - loss: 0.4659 - accuracy: 0.7782 - val_loss: 0.4975 - val_accu
```

Maintenant que nous avons appris un modèle qui se comporte bien nous l'appliquons sur toutes les données pour voir s'il est adapté. Attention, généralement outre le fait de faire de la *feature selection* sur votre jeu de données, vous commencez par l'ensemble des features. Ici nous avons juste décrit quelques exemples d'overfitting et d'underfitting.

*Remarque : * il faut faire attention car le jeu de données est très déséquilibré (500 vs. 268). Par conséquent le classifieur ne pourra pas être très performant. Dans les expériences précédentes, volontairement, lors du train_test_split, nous n'avons pas sélectionné l'option *stratify=y* cela veut dire qu'avec le déséquilibre la classe moins représenté peut ne pas apparaître suffisamment dans le jeu d'apprentissage. Le fait de faire un échantillonage stratifié oblige à créer un échantillon représentarif en terme de distribution des classes.

**Remarque :** lors des expérimentations et dans la suivante, nous effectuons par simplification une seule fois un *train_test_split*. Bien entendu, il faudrait faire les expérimentations avec un K-fold pour que les résultats et donc le modèle soit vraiment représentatif. Dans notre cas il faudrait même utiliser un *StratifiedKfold*.

```
# # creation d'un dataframe pour récupérer les données
df = pd.read_csv('pima-indians-diabetes.csv', delimiter=';')
print (df.shape)
# Note: 0 — Non Diabetic Patient and 1 — Diabetic Patient

class_names=['non diab','diabete']

array = df.values
X = array[:,0:8]
y = array[:,8]

# normalisation du jeu de données
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)

# création d'un jeu de test et d'apprentissage
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, stratify=y,random_state=0)

# definition du modele keras plus complexe
model = Sequential()
```

```
model.add(Dense(8, input_dim=X_train.shape[1], activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

# compilation du modèle
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit le modèle sur le jeu de données. Nous mettons un batchsize de 32
history=model.fit(X_train, y_train, validation_data=(X_test,y_test),epochs=40, batch_size=32)

y_pred=model.predict(X_test)
y_pred[y_pred <= 0.5] = 0.
y_pred[y_pred > 0.5] = 1.

print ("\nRappel du modèle testé")
print (model.summary(),'\n')
conf=confusion_matrix(y_test,y_pred)
plot_curves_confusion (history,conf,class_names)
```

```
(768, 9)
Epoch 1/40
17/17 [==============================] - 5s 42ms/step - loss: 0.7062 - accuracy: 0.5214 - val_loss: 0.6427 - val_accu
Epoch 2/40
17/17 [==============================] - 0s 8ms/step - loss: 0.6708 - accuracy: 0.5856 - val_loss: 0.6131 - val_accur
Epoch 3/40
17/17 [==============================] - 0s 4ms/step - loss: 0.6481 - accuracy: 0.6245 - val_loss: 0.5921 - val_accur
Epoch 4/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6269 - accuracy: 0.6556 - val_loss: 0.5768 - val_accur
Epoch 5/40
17/17 [==============================] - 0s 6ms/step - loss: 0.6204 - accuracy: 0.6459 - val_loss: 0.5642 - val_accur
Epoch 6/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5994 - accuracy: 0.6790 - val_loss: 0.5534 - val_accur
Epoch 7/40
17/17 [==============================] - 0s 5ms/step - loss: 0.6065 - accuracy: 0.6770 - val_loss: 0.5437 - val_accur
Epoch 8/40
17/17 [==============================] - 0s 4ms/step - loss: 0.5955 - accuracy: 0.6732 - val_loss: 0.5356 - val_accur
Epoch 9/40
17/17 [==============================] - 0s 7ms/step - loss: 0.5734 - accuracy: 0.6809 - val_loss: 0.5289 - val_accur
Epoch 10/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5834 - accuracy: 0.6926 - val_loss: 0.5239 - val_accur
Epoch 11/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5545 - accuracy: 0.7237 - val_loss: 0.5194 - val_accur
Epoch 12/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5595 - accuracy: 0.7179 - val_loss: 0.5147 - val_accur
Epoch 13/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5654 - accuracy: 0.7101 - val_loss: 0.5114 - val_accur
Epoch 14/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5708 - accuracy: 0.6926 - val_loss: 0.5095 - val_accur
Epoch 15/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5321 - accuracy: 0.7257 - val_loss: 0.5068 - val_accur
Epoch 16/40
17/17 [==============================] - 0s 4ms/step - loss: 0.5316 - accuracy: 0.7354 - val_loss: 0.5037 - val_accur
Epoch 17/40
17/17 [==============================] - 0s 5ms/step - loss: 0.5256 - accuracy: 0.7393 - val_loss: 0.5018 - val_accur
Epoch 18/40
17/17 [==============================] - 0s 4ms/step - loss: 0.5383 - accuracy: 0.7315 - val_loss: 0.4991 - val_accur
Epoch 19/40
```

# Utilisation de tensorboard

Tensorflow propose tensorboard pour pouvoir notamment visualiser les courbes et les manipuler.

Il est tout à fait possible de l'utiliser dans un notebook. Il faut au préalable préciser que l'on utilise l'extension tensorboard puis sauvegarder les modèles appris. Enfin via tensorboard on peut analyser les différentes courbes. L'exemple ci-dessous illustre une utilisation de tensorflow sur le jeu de données fashion mnist. L'objectif ici n'est pas trouver un très bon modèle mais plutôt de voir comment utiliser tensorboard.

```
import tensorflow as tf
%load_ext tensorboard
import datetime, os

#effacer les fichiers logs existant
!rm -rf ./logs/
fashion_mnist = tf.keras.datasets.fashion_mnist

(x_train, y_train),(x_test, y_test) = fashion_mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

def create_model():
  return tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
  ])

def train_model():

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

```
  logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
  tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)

  model.fit(x=x_train,
            y=y_train,
            epochs=5,
            validation_data=(x_test, y_test),
            callbacks=[tensorboard_callback])

train_model()

%tensorboard --logdir logs

from tensorboard import notebook
notebook.list()

notebook.display(port=6006, height=1000)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
Epoch 1/5
1875/1875 [==============================] - 19s 10ms/step - loss: 0.4959 - accuracy: 0.8234 - val_loss: 0.4187 - val
Epoch 2/5
1875/1875 [==============================] - 18s 9ms/step - loss: 0.3816 - accuracy: 0.8612 - val_loss: 0.3749 - val_
Epoch 3/5
1875/1875 [==============================] - 17s 9ms/step - loss: 0.3517 - accuracy: 0.8698 - val_loss: 0.3733 - val_
Epoch 4/5
1875/1875 [==============================] - 19s 10ms/step - loss: 0.3293 - accuracy: 0.8773 - val_loss: 0.3673 - val
Epoch 5/5
1875/1875 [==============================] - 18s 9ms/step - loss: 0.3153 - accuracy: 0.8824 - val_loss: 0.3398 - val_
Known TensorBoard instances:
  - port 6006: logdir logs (started 0:00:00 ago; pid 6732)
Selecting TensorBoard with logdir logs (started 0:00:00 ago; port 6006, pid 6732).
```