1 Introduction

Exploration de la librairie de deep learning PyTorch sous Python. Instanciation de perceptrons simples et multicouches.

Même la science n'échappe pas aux phénomènes de mode. J'entends beaucoup parler de PyTorch ces derniers temps. Un coup d'œil sur Google Trends montre une popularité grandissante auprès des internautes ces 3 dernières années.

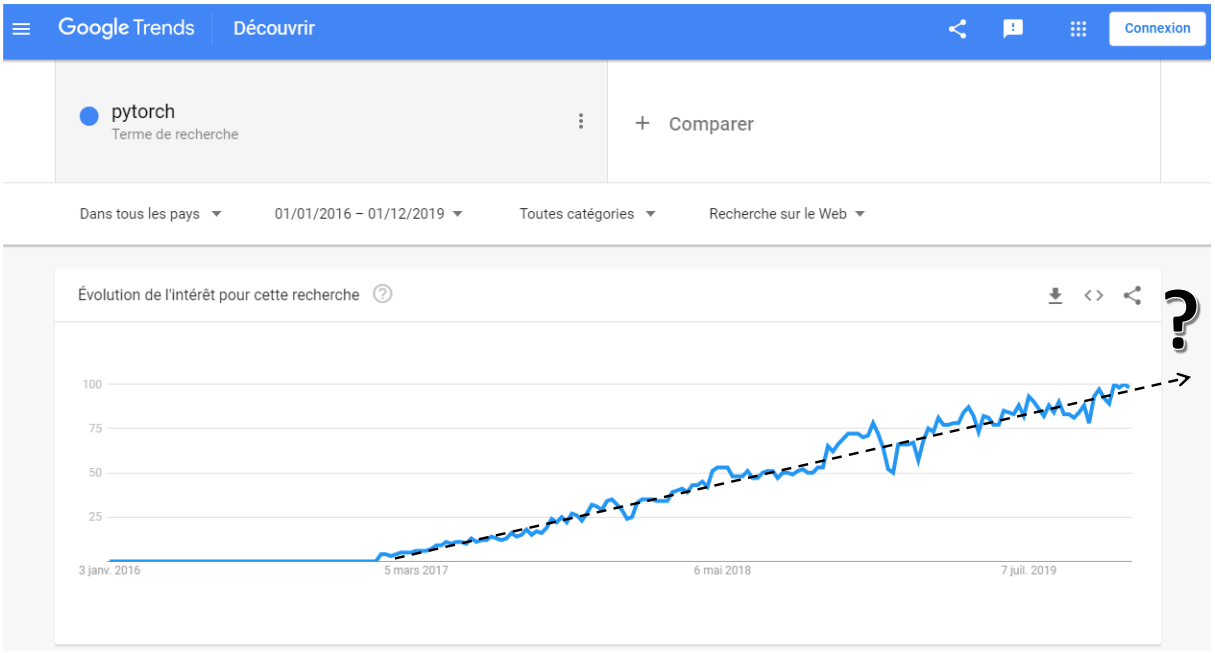

Figure 1 - Popularité des recherches sur "PyTorch" depuis le 01.01.2016 (Google Trends)

Bon, encore une librairie de deep learning pour Python, j'ai pensé dans mon for intérieur, une de plus. Je me suis dit que j'allais torcher (ok, elle est facile) vite fait un petit tutoriel de derrière les fagots et on pourra passer à autre chose. Je suis donc parti à la recherche de documentation sur le web pour éclairer ma… torche (oui, oui, je me damnerais pour un bon mot).

Et là je me suis perdu. La majorité travaillent à partir de données intégrées dans des packages (qu'est-ce qu'elle prend la brave base MNIST quand-même) ou générées artificiellement, survolant hâtivement l'étape essentielle de préparation et d'organisation de données. Or, ne perdons pas de vue que l'objectif d'un tutoriel est avant tout de donner à l'internaute la possibilité de reproduire une démarche sur ses propres données. Expliciter cette phase est très important pour bien comprendre la trame d'une analyse et le mode opératoire des fonctions de machine learning. Concernant PyTorch, c'est la nécessité de transformer les matrices et vecteurs de données (Pandas ou Numpy) en ''tensor'' reconnus par PyTorch qui m'a un peu dérouté au premier abord, d'autant plus que les tutoriels insistaient peu sur le sujet.

Dans ce document, nous nous attelons à l'instanciation, l'apprentissage et l'évaluation de perceptrons simples et multicouches avec PyTorch. Nous travaillons à partir d'un fichier de données que nous devons importer et préparer au préalable. Toutes les étapes seront détaillées. J'insisterai sur la manipulation des ''tensor'', le format de vecteurs et matrices qu'utilise la librairie pour ses manipulations internes. Je m'attarderai également sur la construction un peu particulière des réseaux qui nécessite une connaissance (très basique, n'exagérons rien) des mécanismes de classes (l'héritage et la surcharge des méthodes) sous Python.

2 Données

Nous travaillons sur une version simplifiée (des variables ont été retirées de la base initiale) des données ''breast cancer wisconsin''. Nous cherchons à prédire la ''classe'' {begnin, malignant} des cellules extraites d'une tumeur à partir de leurs caractéristiques (6 variables : ucellsize, ucellshape, mgadhesion, sepcis, normnucl, mitoses).

La base a été scindée en 2 fichiers en amont : ''**breast_train.xlsx**'' (399 observations) est dévolue à l'apprentissage des modèles ; ''**breast_test.xlsx**'' (300) à leur évaluation.

3 Librairie PyTorch

Je travaille sur **Anaconda Python 3.7.4**. L'installation de la librairie ne pose absolument aucune difficulté avec le gestionnaire de packages ''conda''. La commande à utiliser est indiquée sur le site de PyTorch en fonction de notre configuration.

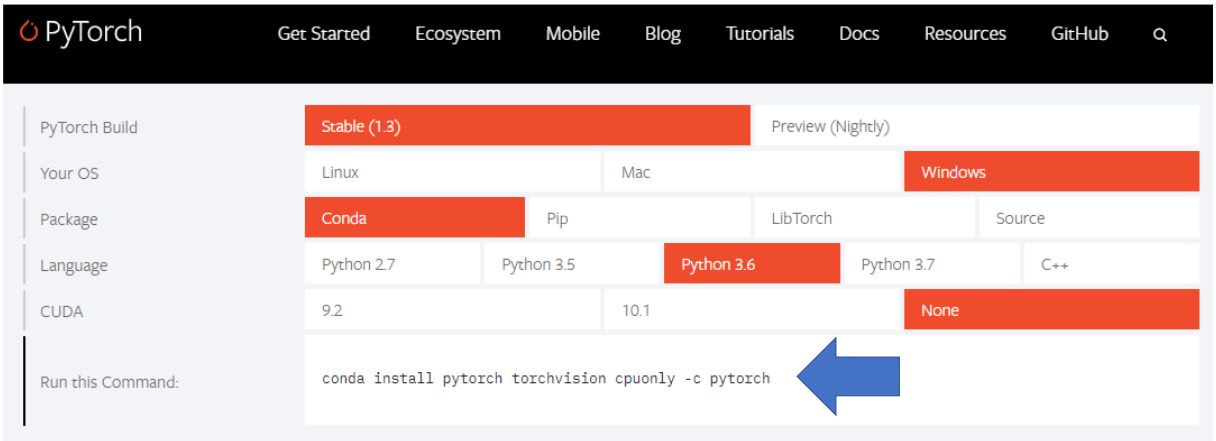

Figure 2 - Commande à utiliser pour installer la librairie (Anaconda - Conda)

En ce qui me concerne, ne disposant de matériel adapté, il ne m'a pas semblé nécessaire d'installer CUDA pour le calcul sur GPU (utilisation du processeur de la carte graphique).

4 Importation des données et préparation des tenseurs

4.1 Importation des données

La première étape passe par l'importation de la base d'apprentissage ''breast_train.xlsx''. Aucune difficulté à ce stade.

```
#modifier le dossier de travail
import os
os.chdir(" ... votre dossier ...")

#importer les données
import pandas
pdTrain = pandas.read_excel("breast_train.xlsx",header=0)
print(pdTrain.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 399 entries, 0 to 398
Data columns (total 7 columns):
ucellsize     399 non-null int64
ucellshape    399 non-null int64
mgadhesion    399 non-null int64
sepics        399 non-null int64
normnucl      399 non-null int64
mitoses       399 non-null int64
classe        399 non-null object
dtypes: int64(6), object(1)
memory usage: 21.9+ KB
```

Nous disposons d'un data frame ''pandas''.

4.2 Préparation des données

4.2.1 Standardisation des descripteurs

Il est conseillé de normaliser les descripteurs pour les méthodes de deep learning. Nous les centrons et réduisons avec la classe StandardScaler du package ''scikit-learn''.

```
#centrer et réduire les X
from sklearn.preprocessing import StandardScaler
sts = StandardScaler()

#données centrées-réduites –> toutes les colonnes sauf la dernière, la variable cible 'classe'
ZTrain = sts.fit_transform(pdTrain[pdTrain.columns[:-1]])
print(ZTrain)
```

```
[[-0.67999948 -0.73765769 -0.59835544 -0.51397882 -0.58966111 -0.31445173]
 [-0.01336608 -0.40012854 -0.59835544 -0.05313039  1.04874267 -0.31445173]
 [ 1.98653412  1.2875172   0.11644642  0.40771803  1.37642343 -0.31445173]
 ...
 [-0.67999948 -0.73765769 -0.59835544 -0.51397882 -0.58966111 -0.31445173]
 [-0.67999948 -0.73765769 -0.59835544 -0.51397882 -0.26198035 -0.31445173]
```

```
 [ 0.31995062  0.94998805 -0.59835544 -0.51397882  0.0657004  -0.31445173]]
```

La variable ZTrain est une matrice de type ''numpy'' à ce stade.

```
#type de l'objet
print(type(ZTrain))
```

```
<class 'numpy.ndarray'>
```

4.2.2 Codage de la cible

La variable cible ''classe'' doit être recodée en numérique 0/1. Le plus simple avec ''pandas'' est de la transformer avec `.astype()` en variable catégorielle (l'équivalent du type factor de R), puis de récupérer les codes.

```
#y au format Numpy, codé 0/1
yTrain = pdTrain.classe.astype('category').cat.codes.values
print(yTrain)
```

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

Nous avons aussi un objet ''numpy'', un vecteur en l'occurrence.

```
#type
print(type(yTrain))
```

```
<class 'numpy.ndarray'>
```

4.3 Création des tenseurs pour PyTorch

Le package PyTorch ne sait pas manipuler directement les structures ''pandas'' ou ''numpy''. Nous devons passer par une phase de conversion en ''tensor'' qui est le format interne des vecteurs et matrices qu'il utilise.

Nous chargeons tout d'abord la librairie PyTorch.

```
#PyTorch
import torch
print(torch.__version__)
```

```
1.3.1
```

Nous utilisons la **version 1.3.1** dans ce document (1er décembre 2019).

Descripteurs. Nous convertissons les descripteurs `ZTrain` en matrice de type ''tensor''.

```
#tranformer les variables Z en tensor
tensor_ZTrain = torch.FloatTensor(ZTrain)

#qui est d'un type particulier
print(type(tensor_ZTrain))
```

```
<class 'torch.Tensor'>
```

Les dimensions concordent…

```
#dimensions
print(tensor_ZTrain.shape)
```

```
torch.Size([399, 6])
```

… et nous avons bien les bonnes valeurs.

```
#valeurs de la matrice
print(tensor_ZTrain)
```

```
tensor([[-0.6800, -0.7377, -0.5984, -0.5140, -0.5897, -0.3145],
        [-0.0134, -0.4001, -0.5984, -0.0531,  1.0487, -0.3145],
        [ 1.9865,  1.2875,  0.1164,  0.4077,  1.3764, -0.3145],
        ...,
        [-0.6800, -0.7377, -0.5984, -0.5140, -0.5897, -0.3145],
        [-0.6800, -0.7377, -0.5984, -0.5140, -0.2620, -0.3145],
        [ 0.3200,  0.9500, -0.5984, -0.5140,  0.0657, -0.3145]])
```

Variable cible. Nous faisons de même pour la variable cible.

```
#idem pour y
tensor_yTrain = torch.FloatTensor(yTrain)
print(tensor_yTrain)
```

```
tensor([0., 0., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 1., 0.,
        0., 0., 0., 1., 1., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 1., 1., 0., 0., 0., 0., 1., 1., 0., 1., 0., 0., 0., 0., 0.,
       ...
        1., 1., 1., 1., 0., 0., 1., 0., 1., 1., 0., 1., 1., 0., 0., 0., 1., 1.,
        0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 1., 1., 1.,
        0., 0., 1.])
```

Qui est un vecteur, à une dimension donc, cette précision est importante pour la suite.

```
#dimension
print(tensor_yTrain.shape)
```

```
torch.Size([399])
```

5 Perceptron simple avec PyTorch

5.1 Définition de la structure du réseau – Perceptron simple

L'autre singularité de PyTorch est qu'il faut créer une classe héritière, en l'occurrence de ''torch.nn.Module'' pour notre exemple, pour définir notre réseau. Deux méthodes doivent être surchargées : le constructeur ''`__init__()`'', où nous définissons les outils du réseau (couches

et fonctions de transfert) ; la méthode "`forward()`" où nous décrivons la succession de calculs de l'entrée vers la sortie, en passant par les éventuelles couches intermédiaires.

Les mécanismes de classes sous Python sont relativement simples. J'ai essayé de commenter au maximum le code pour la construction d'un perceptron simple correspondant à l'architecture suivante (Figure 3).

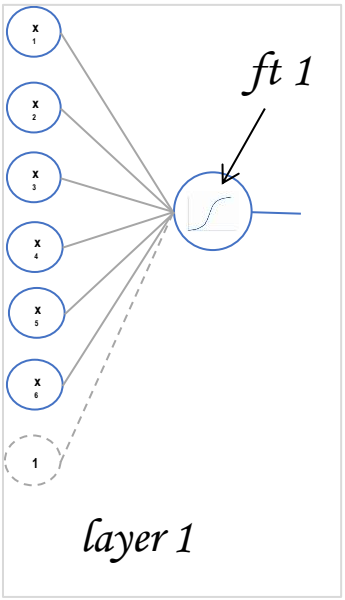

Figure 3 - Structure du perceptron simple

```
#classe de calcul - Perceptron simple
#héritier de torch.nn.Module
class MyPS(torch.nn.Module):

    #constructeur, p nombre de variables explicatives
    def __init__(self,p):
        #appel du constructeur de l'ancêtre
        super(MyPS,self).__init__()
        #couche d'entrée (p variables) vers sortie (1 neurone)
        self.layer1 = torch.nn.Linear(p,1)
        #fonction de transfert sigmoïde à la sortie
        self.ft1 = torch.nn.Sigmoid()

    #calcul de la sortie du réseau
    #à partir d'une matrice x en entrée
    def forward(self,x):
        #application de la combinaison linéaire
        comb_lin = self.layer1(x)
        #transformation avec la fonction de transfert
        proba = self.ft1(comb_lin)
        return proba
```

Notre classe ''MyPS'' hérite de ''torch.nn.Module''.

Pour le constructeur `__init__` :

- Nous faisons tout d'abord appel au constructeur de la classe ancêtre ;
- Puis nous créons la couche ''`layer1`'' faisant le pont entre l'entrée et la sortie. En entrée, il prend ''`p`'' neurones, qui correspondra au nombre de variables de la base des données. Il a un seul neurone en sortie puisque la variable cible est binaire 0/1. Notons que le biais, un neurone spécifique prenant la valeur 1 systématiquement pour matérialiser la constante dans la combinaison linéaire, est ajouté automatiquement. Nous avons donc à ce stade une équation de la forme suivante :

$$u = a_0 + a_1 z_1 + \cdots + a_p z_p$$

- A cette combinaison linéaire, nous appliquons la fonction de transfert ''`ft1`''. En sortie du réseau, nous aurons donc :

$$v = \frac{1}{1 + e^{-u}}$$

Remarque : Attention, dans le constructeur, nous avons défini les opérateurs dans l'ordre. Mais en réalité, le système dispose effectivement des outils (couche, fonction de transfert) mais n'a aucune information sur l'enchaînement des opérations à ce stade, à la différence de la définition des structures que l'on pourrait réaliser sous Keras par exemple. Intervertir les lignes de code dans le constructeur n'a aucun impact sur l'architecture du réseau.

La surcharge de la méthode `forward()` permet d'y remédier en spécifiant la succession des traitements:

- Il prend en entrée la matrice ''`x`'' des descripteurs à traiter.
- Il lui applique la combinaison linéaire en appliquant les poids synaptiques (a_j).
- Au résultat (`comb_lin`), il applique la fonction de transfert sigmoïde pour obtenir la sortie du réseau, qui correspond à une estimation de la probabilité d'affectation (`proba`) à la modalité ''1'' dans notre structure.

5.2 Outils pour l'apprentissage

Nous devons définir deux outils supplémentaires avant de pouvoir commencer le processus de modélisation :

- Le critère à optimiser, la fonction de perte. Nous faisons le choix de la MSE (mean squared error) pour notre réseau.

```
#fonction critère à optimiser
critere_ps = torch.nn.MSELoss()
```

– Et l'algorithme d'optimisation. Nous faisons le choix d'ADAM qui est une variante de la descente du gradient stochastique. Nous lui passons les paramètres du modèle, qu'il faut donc instancier au préalable, qu'il devra manipuler pour optimiser la fonction de perte.

```
#instanciation du modèle
#.shape[1] pour le nombre de descripteurs
ps = MyPS(tensor_ZTrain.shape[1])

#algorithme d'optimisation
#on lui passe les paramètres à manipuler
optimiseur_ps = torch.optim.Adam(ps.parameters())
```

5.3 Vérification de la structure – Valeur initiale de la perte (loss)

Procédons à quelques vérifications pour bien comprendre ce que nous venons de réaliser.

Les couches sont munies de poids synaptiques (coefficients). Nous affichons les poids initiaux – définis au hasard par PyTorch – avant entraînement du modèle.

```
#poids synaptiques initiaux
print(ps.layer1.weight)
```

```
Parameter containing:
tensor([[-0.3143, -0.0190, -0.2980, -0.3078,  0.3307, -0.1493]],
       requires_grad=True)
```

Les coefficients représentent les paramètres du modèle. Nous avons une matrice ligne avec 6 valeurs (a_1, a_2, …, a_6) pour 6 variables explicatives. C'est cohérent.

Voyons ce qu'il en est pour le biais (l'intercept).

```
#et l'intercept
print(ps.layer1.bias)
```

```
Parameter containing:
tensor([0.0481], requires_grad=True)
```

Il s'agit aussi d'un paramètre à estimer, et nous avons un seul coefficient a_0. Très bien.

Disposant de ce jeu de coefficients, nous pouvons calculer la sortie du modèle en l'appliquant sur les données d'apprentissage. Nous faisons appel à la fonction `forward()`.

```
#calculer la sortie du réseau
#équivalent à yPred = ps.forward(tensor_ZTrain)
yPred = ps(tensor_ZTrain)
print(yPred)
```

`forward()` est une méthode par défaut c.-à-d. que nous pouvons produire exactement le même résultat en appliquant directement l'objet sur les données d'apprentissage. D'où l'écriture équivalente ci-dessus. Ouh là là, j'avoue que ça m'a pris un peu temps avant de saisir cette subtilité, et les tutoriels trouvés ici et là sur le web sont peu diserts à ce sujet.

Nous obtenons les probabilités d'affectation correspondant aux poids initiaux du réseau.

```
tensor([[0.6140],
        [0.6566],
        [0.4357],
...
        [0.6140],
        [0.6394],
        [0.5829]], grad_fn=<SigmoidBackward>)
```

Autre bizarrerie, qui n'en est pas une après coup mais il faut s'en apercevoir. Même si nous avons une seule colonne de valeurs, nous avons bien une structure matricielle à l'issue de la prédiction, avec 399 lignes et 1 colonne.

```
#format de yPred
print(yPred.shape)
```

```
torch.Size([399, 1])
```

Il faut la transformer en vecteur pour être compatible avec le ''tensor'' représentant la variable cible. Nous le réalisons avec la méthode `.squeeze()` de l'objet :

```
#pour transformer en vecteur
print(yPred.squeeze())
```

```
tensor([0.6140, 0.6566, 0.4357, 0.0996, 0.1458, 0.6140, 0.6394, 0.6140, 0.6140,
        0.5889, 0.6140, 0.6140, 0.4244, 0.1877, 0.5117, 0.6140, 0.6631, 0.5266,
...
        0.6471, 0.1929, 0.5479, 0.2507, 0.6640, 0.6140, 0.5885, 0.3511, 0.3445,
        0.6140, 0.6394, 0.5829], grad_fn=<SqueezeBackward0>)
```

Qui est bien un vecteur cette fois-ci.

```
#en effet
print(yPred.squeeze().shape)
```

```
torch.Size([399])
```

Nous pouvons maintenant calculer la perte (MSE) initiale du réseau en opposant sa sortie avec le vecteur observé de la variable cible.

```
#mse au départ
MSE1st = critere_ps(yPred.squeeze(),tensor_yTrain)
print(MSE1st)
```

```
tensor(0.3860, grad_fn=<MseLossBackward>)
```

5.4 Boucle d'apprentissage

La boucle d'apprentissage sous PyTorch n'est absolument pas triviale, loin s'en faut. A la différence des autres librairies, nous devons la décrire nous-même, en rentrant quand-même dans beaucoup de détails. Je comprends les avantages, les initiés peuvent régler finement le processus. Mais le néophyte, lui, peut s'y perdre très facilement.

Comme nous allons l'utiliser à 2 reprises dans notre tutoriel, j'ai préféré écrire une fonction dédiée. Elle prend en entrée les données d'apprentissage (X, descripteurs ; y, cible), le modèle à apprendre (`classifier`), la fonction critère à optimiser (`optimiser`), l'algorithme d'optimisation (`optimizer`), et le nombre d'itérations à effectuer sur la base (`n_epochs`, j'ai fixé une valeur élevée par défaut, nous comprendrons pourquoi plus loin).

```
#numpy
import numpy

#fonction pour apprentissage
def train_session(X,y,classifier,criterion,optimizer,n_epochs=10000):
    #pour collecter le loss au fil des itérations
    losses = numpy.zeros(n_epochs)
    #itérer (boucle) pour optimiser
    for iter in range(n_epochs):
        #réinitialiser (ràz) le gradient
        #nécessaire à chaque passage sinon PyTorch accumule
        optimizer.zero_grad()
        #calculer la sortie du réseau
        yPred = classifier(X)
        #calculer la perte
        perte = criterion(yPred.squeeze(),y)
        #la collecter
        losses[iter] = perte.item()
        #calcul du gradient (et retropropagation)
        perte.backward()
        #màj des poids synaptiques
        optimizer.step()
    #sortie de la boucle
    return losses
```

La fonction renvoie en sortie un vecteur contenant la valeur de la perte au fil des itérations.

Notons quelques éléments :

- `.zero_grad()` est absolument nécessaire pour réinitialiser le gradient à chaque étape, sinon l'outil accumule avec les valeurs calculées dans les itérations précédentes.

– `.backward()` permet de rétro-propager la correction d'erreur dans les couches antérieures du réseau.
– `.step()` sert à corriger les poids synaptiques à partir des gradients calculés.

Et on itère ce processus `n_epochs` fois sur la base entière.

`n_epochs = 10000` par défaut peut paraître énorme. Je me suis rendu compte d'une part que la convergence avec mes paramètres était assez lente, d'autre part que l'outil s'est avéré rapide, par rapport aux autres librairies notamment, c'est ce qui doit contribuer grandement à son succès j'imagine.

Nous lançons l'apprentissage du modèle.

```
#lancer l'apprentissage
pertes = train_session(tensor_ZTrain,tensor_yTrain,ps,critere_ps,optimiseur_ps)
```

Nous affichons la courbe de décroissance de la perte au fil des itérations.

```
#courbe
import matplotlib.pyplot as plt
plt.plot(numpy.arange(0,pertes.shape[0]),pertes)
plt.title("Evolution fnct de perte")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```

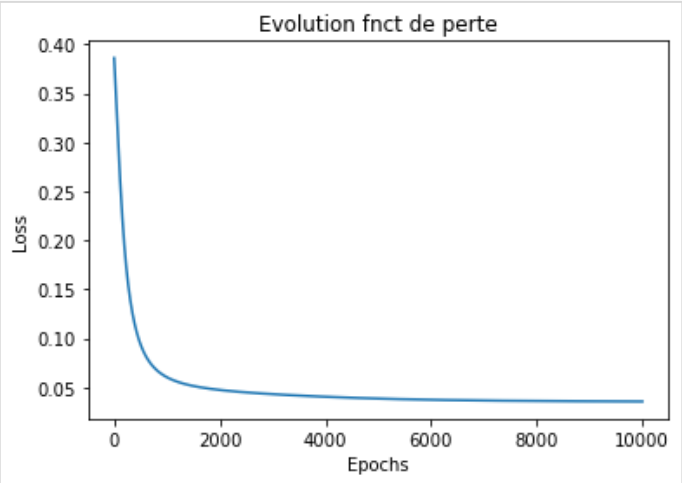

Figure 4 - Décroissance de la perte au fil des epochs - Perceptron simple

La convergence n'intervient que tardivement, nous le constatons.

Voyons maintenant si ce modèle est efficace sur l'échantillon test que nous avons réservé à part.

5.5 Evaluation sur l'échantillon test

5.5.1 Importation et préparation de l'échantillon test

Nous importons le fichier test "**breast_test.xlsx**" composé de 300 observations.

```
#chargement de l'échantillon test
pdTest = pandas.read_excel("breast_test.xlsx",header=0)
print(pdTest.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300 entries, 0 to 299
Data columns (total 7 columns):
ucellsize     300 non-null int64
ucellshape    300 non-null int64
mgadhesion    300 non-null int64
sepics        300 non-null int64
normnucl      300 non-null int64
mitoses       300 non-null int64
classe        300 non-null object
dtypes: int64(6), object(1)
memory usage: 16.5+ KB
```

Nous centrons et réduisons les descripteurs à l'aide des paramètres (moyennes, écarts-type) calculés sur l'échantillon d'apprentissage (section 4.2.1).

```
#données centrées et réduites (avec les param. de train)
#utilisation de l'objet sts créé plus haut (section 4.2.1)
ZTest = sts.transform(pdTest[pdTest.columns[:-1]])
print(ZTest)
```

```
[[-0.67999948 -0.06259939 -0.59835544 -0.97482724 -0.58966111 -0.31445173]
 [ 2.31985082  0.27492976  1.54605012 -0.05313039  2.3594657  -0.31445173]
 [-0.67999948 -0.73765769 -0.59835544 -0.51397882 -0.58966111 -0.31445173]
 ...
 [ 0.98658402 -0.06259939  1.1886492   0.40771803  1.70410419  1.57699912]
 [-0.67999948 -0.06259939 -0.59835544 -0.51397882 -0.58966111 -0.31445173]
 [-0.67999948 -0.73765769 -0.59835544 -0.51397882 -0.58966111 -0.31445173]]
```

Que nous convertissons en objet "tensor".

```
#mettre en tensor
tensor_ZTest = torch.FloatTensor(ZTest)
```

Nous faisons de même pour le vecteur cible observé.

```
#idem pour y
tensor_yTest = torch.FloatTensor(pdTest.classe.astype('category').cat.codes.values)
```

5.5.2 Fonction pour mesurer les performances

L'idée est d'opposer ce vecteur cible observé avec la prédiction du modèle sur l'échantillon test. Nous écrivons une fonction à cet effet car nous aurons à la réutiliser pour le perceptron multicouche plus loin. Nous exploitons les outils de la librairie "scikit-learn".

```
#librairie pour l'évaluation
from sklearn import metrics
#fonction pour test
def test_session(X,y,classifier):
    #appliquer le modèle sur l'échantillon test
    tensor_proba = classifier(X)
    #transformer en numpy
    proba = tensor_proba.squeeze().detach().numpy()
    #recoder la proba d'affectation en 0/1
    pred = numpy.where(proba > 0.5, 1, 0)
    #mettre en numpy y observé (qui est un tensor)
    yobs = y.detach().numpy()
    #calculer la matrice de confusion
    mc = metrics.confusion_matrix(yobs,pred)
    #et le taux de succès
    acc = metrics.accuracy_score(yobs,pred)
    #renvoyer la matrice et l'accuracy
    return mc,acc
```

Notre fonction prend en entrée les descripteurs (`X`), la variable cible (`y`) et le classifieur (`classifier`). Voici les étapes successives :

- Elle applique le classifieur sur les descripteurs.
- Nous avons une probabilité d'affectation (`tensor_proba`) qui est au format ''tensor''. Nous le convertissons en vecteur numpy (`proba`).
- Nous transformons la probabilité en prédiction d'appartenance aux classes 0/1 (`pred`).
- Nous convertissons le vecteur tensor (`y`) en vecteur numpy (`yobs`).
- La matrice de confusion (`mc`) est calculée à partir de (`yobs`) et (`pred`).
- Il en est de même pour le taux de succès (accuracy = `acc`).

La fonction renvoie ces deux dernières informations.

Voyons ce qu'il en est de notre modèle déployé sur l'échantillon test.

```
#évaluer
mc,acc = test_session(tensor_ZTest,tensor_yTest,ps)
#matrice de confusion
print(mc)
```

```
[[184   7]
 [  3 106]]
```

La matrice de confusion est pas mal, mais on sait que cette base est réputée facile. Pour ce qui est du taux d'erreur (1 – accuracy) :

```
#taux d'erreur
print(1-acc)
```

```
0.033333333333333326
```

Oui, (3 + 7) / 300 = 0.03333…

5.6 Poids synaptiques du modèle prédictif

Affichons les coefficients estimés après entraînement du modèle.

```
#récupération des poids finaux - obtenus sur les var. centrées et réduites
print(ps.layer1.weight)
```

```
Parameter containing:
tensor([[1.7848, 2.9477, 0.1646, 1.7735, 1.0819, 0.8193]], requires_grad=True)
```

Et le biais :

```
#et l'intercept
print(ps.layer1.bias)
```

```
Parameter containing:
tensor([0.2528], requires_grad=True)
```

Notre modèle, sur données centrées et réduites, s'écrit donc :

$$0.2528 + 1.7848\ z_1 + 2.9477\ z_2 + \ldots + 0.8193\ z_6$$

Où (z_1 = ucellsize) après centrage-réduction, etc.

6 Perceptron multicouche avec PyTorch

Maintenant que nous comprenons à peu près la démarche dans son ensemble, risquons-nous à créer un perceptron multicouche dans cette section.

6.1 Construction de la structure

Première étape, il faut définir l'architecture du réseau. Nous souhaitons construire un perceptron à une couche cachée comportant 2 neurones.

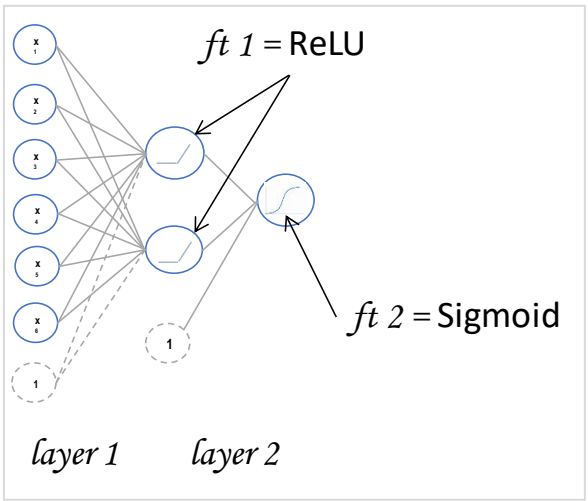

Figure 5 - Perceptron à une couche cachée

Nous utilisons une fonction de transfert ReLU dans la couche cachée.

Voici la classe correspondante dans le formalisme PyTorch.

```
#perceptron multicouche
class MyPMC(torch.nn.Module):
    #constructeur
    def __init__(self,p):
        #appel du constructeur de l'ancêtre
        super(MyPMC,self).__init__()
        #couche d'entrée (p variables) vers intermédiaire (2 neurones)
        self.layer1 = torch.nn.Linear(p,2)
        #fonction de transfert - sortie couche cachée
        self.ft1 = torch.nn.ReLU()
        #couche intermédiaire (2 neurones) vers sortie (1 neurone)
        self.layer2 = torch.nn.Linear(2,1)
        #fonction de transfert sigmoïde
        self.ft2 = torch.nn.Sigmoid()

    #calcul de la sortie du réseau
    #à partir d'une matrice x en entrée
    def forward(self,x):
        #application de la combinaison linéaire
        comb_lin_1 = self.layer1(x)
        #transformation avec la fonction de transfert
        out_1 = self.ft1(comb_lin_1)
        #puis seconde combinaison linéaire
        comb_lin_2 = self.layer2(out_1)
        #transformation
        out_2 = self.ft2(comb_lin_2)
        #return
        return out_2
```

Comme précédemment, nous définissons ensuite le critère à optimiser et l'algorithme d'optimisation.

```
#critere
critere_pmc = torch.nn.MSELoss()

#instanciation
pmc = MyPMC(tensor_ZTrain.shape[1])

#optimiseur
optimiseur_pmc = torch.optim.Adam(pmc.parameters())
```

6.2 Apprentissage et test

Nous sommes parés pour lancer l'apprentissage.

```
#lancer l'apprentissage
pertes = train_session(tensor_ZTrain,tensor_yTrain,pmc,critere_pmc,optimiseur_pmc)

#évolution
plt.plot(numpy.arange(0,pertes.shape[0]),pertes)
plt.title("Evolution fnct de perte")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```

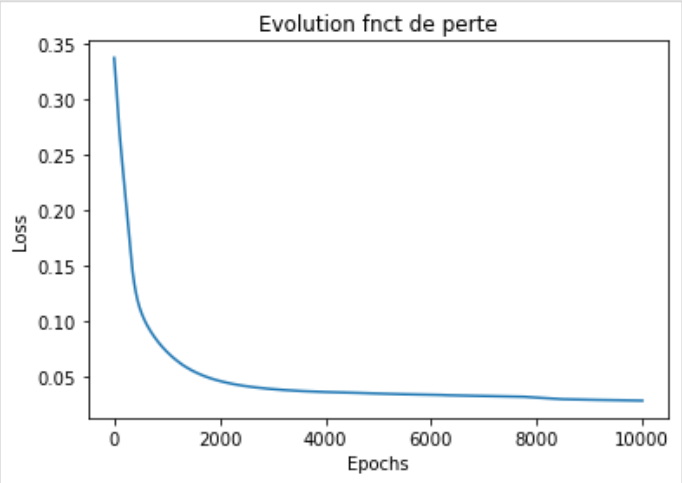

Figure 6 - Evolution de la fonction de perte - Perceptron multicouche

La convergence est toujours aussi lente, travailler sur les paramètres d'optimisation serait bénéfique ici.

Nous appliquons le modèle sur l'échantillon test, avec pour matrice de confusion…

```
#évaluation
mc,acc = test_session(tensor_ZTest,tensor_yTest,pmc)

#matrice de confusion
print(mc)
```

```
[[185   6]
 [  2 107]]
```

… soit un taux d'erreur de…

```
#taux d'erreur
print(1-acc)
```

```
0.026666666666666616
```

Le gain par rapport au perceptron simple est anecdotique. On ne va pas commencer à s'extasier sur 2 individus bien classés supplémentaires sur 300.

6.3 Poids synaptiques estimés

La structure du réseau est plus complexe. Voyons comment accéder aux coefficients estimés.

Entre l'entrée et la couche intermédiaire.

Pour les coefficients,…

```
#poids : entrée -> intermédiaire
print(pmc.layer1.weight)
```

```
Parameter containing:
tensor([[-1.2487, -0.8520, -1.6357, -1.3251, -2.1888, -0.3359],
        [-0.0715, -4.9210, -0.1725, -0.7392, -0.8271, -0.4448]],
       requires_grad=True)
```

… nous avons une matrice 2 lignes (2 neurones dans la couche intermédiaire) et 6 colonnes (6 variables prédictives).

Les coefficients du biais ont évolué également :

```
#intercept corresp.
print(pmc.layer1.bias)
```

```
Parameter containing:
tensor([-0.6574, -0.0807], requires_grad=True)
```

Et entre la couche intermédiaire et la sortie.

Les coefficients, …

```
#poids : intermédiaire -> sortie
print(pmc.layer2.weight)
```

```
Parameter containing:
tensor([[-2.7742, -3.9854]], requires_grad=True)
```

… et le biais correspondant :

```
#intercept corresp.
print(pmc.layer2.bias)
```

```
Parameter containing:
tensor([2.8080], requires_grad=True)
```

7 Conclusion

''Qu'importe le flacon pourvu qu'on ait l'ivresse'', je me répète souvent à ce sujet. Les librairies ont leur propre mode de fonctionnement mais, fondamentalement, elles s'inscrivent dans une logique globale qui s'impose à elles. Dans ce tutoriel, il s'agissait du cadre de l'analyse prédictive.

En partant de ce constat, il faut souvent peu d'efforts pour s'adapter au mode opératoire de n'importe quel package. Pour ce qui est de PyTorch, il fallait appréhender d'une part le passage obligé par les structures ''tensor'' pour les vecteurs et matrices de données, d'autre part intégrer l'idée qu'il faut créer une structure de classe pour définir un modèle. Une fois ces deux écueils assimilés, il a été facile de mettre en œuvre le dispositif sur une analyse réaliste. Ma première impression est que la librairie PyTorch *semble* assez rapide. Il faudra creuser cela prochainement.

8 Références

PyTorch, https://pytorch.org/

Jang A.E., ''PyTorch : Introduction to Neural Network – Feedformward / MLP'', février 2019.

Tutoriel Tanagra, ''Deep learning : perceptrons simples et multicouches'', novembre 2018.