Preminilaires de l'apprentissage profond

1. Librairie Scikit-learn

Créé en 2007, Scikit-Learn est l'une des librairies open-source les plus populaires pour le Machine Learning en Python. La librairie est construite à partir de numpy, matplotlib et scipy, et s'interface aussi avec des dataframes pandas.
En plus de contenir et de rendre accessible facilement tous les principaux algorithmes de Machine Learning (**classification**, **clustering**, **régression**), elle contient aussi de nombreux modules pour la réduction de dimension, le processing des données et l'évaluation des modèles.

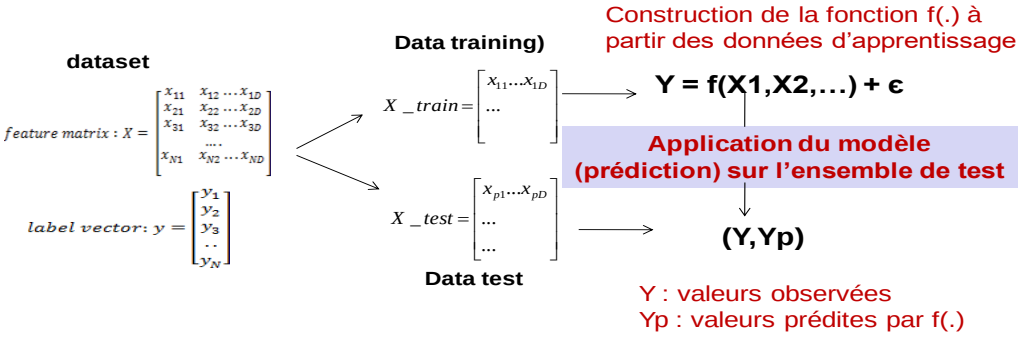
Les jeux de données disponibles dans scikit-learn sont : **iris, breast-cancer, boston, diabetes, digits, linnerud, sample images, 20newsgroups, ...**
Les jeux de données comprennent un certain nombres d'attributs parmi (tous ne sont pas toujours définis) : **data, target, target_names, feature_names**, **DESCR** :

1. **data** est un tableau de dimensions n × m ou n est le nombre d'instances, et m le nombre d'attributs;
2. **target** stocke les classes (étiquettes) de chaque instance (dans le cas supervisé)
3. **target_names** contient le nom des classes
4. **feature_names** contient le nom des attributs
5. **DESCR** est un texte décrivant le jeu de données

La plupart des algorithmes de **scikit-learn** utilisent des jeux de données sous forme de tableaux (ou de matrices) à deux dimensions. La dimension de cette matrice sera
[n_samples, n_features], avec:

- **n_samples**: Le nombre d'échantillons de données ou d'observations. Un échantillon peut représenter un document, une image, un son, une vidéo, un objet, une ligne dans une base de données ou un fichier csv
- **n_features**: Le nombre de caractéristiques qui décrivent une observation de manière quantitative. Ces données doivent être des nombres (réels ou entiers)

Dans le cadre d'un **apprentissage supervisé**, on représente les données sous la forme d'une **feature matrix** et d'un **label vector**:



Afin de tester la performance de l'algorithme en prédiction, il faut séparer le **dataset en deux** : Un premier set que l'on va appeler **train set**, qui va constituer la base d'apprentissage (les données dont on connaît la classe), et un deuxième appelé **test set**, sur lequel on va tester les performances de l'algorithme (on fait donc comme si on ne connaissait pas la classe, puis on peut comparer la prédiction à la classe réelle).
La séparation peut être faite de manière très simple en scikit-learn :

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2, random_state=99)
print (X_train.shape, y_train.shape)
print X_test.shape, y_test.shape
```

Ici, nous avons séparé le dataset en deux: le premier set que l'on va appeler **train set représente 80%**, qui va constituer la base d'apprentissage dont on connaît la classe, et le deuxième appelé **test set (20%).** Nous pouvons maintenant construire notre modèle sur le train set. Le paramètre **random_state** détermine le nombre de génération aléatoire pour la création du jeu de données. Transmettre en entier pour une sortie reproductible sur plusieurs appels de l'algo.

2. Entrainer un modele de ML

La plupart des opérations vont être réalisées à l'aide d'objets de type **estimator**. Un algorithme de Machine Learning correspond à une classe de type estimator :

```
from sklearn import tree
model = tree.DecisionTreeClassifier()
model.fit(X, y)
```

scikit-learn s'efforce d'avoir une interface uniforme pour tous les algorithmes, ce qui est une de ses plus grandes forces. Étant donné un objet estimator model, on peut utiliser les méthodes suivantes :

a. Pour tous les estimators
- **model.fit()** : lance le mécanisme d'apprentissage.
  - model.fit(X, y) pour les algorithmes supervisés (X les observations d'entraînement, y leurs labels)
  - model.fit(X) pour les algorithmes non-supervisés

b. **Les estimators supervisés**
- model.predict(X_new) : prédit et retourne les labels d'un nouveau jeu de données X_new
- model.predict_proba(X_new) : pour les algorithmes de classification, prédit et retourne les probabilités d'appartenance à une classe d'un nouveau jeu de données X_new
- model.score(X_test, y_test) : Retourne un score de performance du modèle entre 0 (mauvais) et 1 (idéal mais suspicieux)

c. **Les estimators non-supervisés**
- model.transform(X_new) : transforme un jeu de données selon le modèle
- model.fit_transform(X) : apprend ses paramètres grâce à un jeu de données et transforme celui-ci.

3. Normalisation et standardisation de données

La normalisation ou la standardisation se sont les parties les plus importantes du prétraitement des données.

Si nous voyons notre ensemble de données, certains attributs contiennent des informations en valeur numérique, certaines valeurs sont très élevées et d'autres sont très faibles. Cela entraînera des problèmes dans notre modèle de machine learning.

La plupart des algorithmes d'apprentissage automatique utilisent la distance euclédienne entre deux points de données dans leurs calculs. Nous devons amener toutes les fonctionnalités au même niveau de grandeurs. Ceci peut être réalisé par mise à l'échelle. Il existe deux méthodes pour résoudre ce problème. La première est la normalisation et la seconde est la standardisation.

Standardisation	Normalisation
$x_{stand} = \frac{x - \text{mean}(x)}{\text{standard deviation } (x)}$	$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}$

```
From sklearn.preprocessing import MinMaxScaler
Sc=MinMaxScaler()
X=sc.fit_transform(X)
```

```
# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

4. Outils pour l'apprentissage profond

TensorFlow (Google) : https://www.tensorflow.org/

- Librairié open source d'apprentissage automatique créée par Google Brain Team, une équipe de l'organisation de recherche de Google
- Permet de former et d'exécuter efficacement de très grands réseaux de neurones en répartissant les calculs sur potentiellement des milliers de serveurs multi-GPU.
- Lancée en novembre 2015, adoption massive par la communaute
- Codée en C++, avec interface d'utilisation en Python
- Entierement organisée autour de graphes computationnels

Theano (Universite de Montreal) : http://deeplearning.net/software/theano/

- Librairié Python pour calcul matriciel efficace, incluant gradient automatique de graphes computationnel
- Optimise traitement par compilation dynamique de code C et exploitation de GPU
- Performant mais assez bas niveau, librairies disponibles offrant abstractions de plus haut niveau pour faire de l'apprentissage profond (Lasagne, Keras)

Torch (Collobert et collaborateurs) : http://torch.ch/

- Programmée en C++
- Plus ancien que Theano et TensorFlow, ajout recent de gradient automatique

Keras

- Implémenter un réseau de neurones avec Keras revient à créer un modèle Sequential et à l'enrichir avec les couches correspondantes dans le bon ordre.
- L'étape la plus difficile est de définir correctement les paramètres de chacune des couches – d'où l'importance de bien comprendre l'architecture du réseau !

```
from keras.models import Sequential
model = Sequential()  # Création d'un réseau de neurones vide
```

- Les couches s'ajoutent soit en tant que paramètres d'entrée du constructeur Sequential() , soit une par une avec la méthode model.add() .

Installer les deux outils :

```
Conda –pip install tensorflow
Conda –pip install keras
```

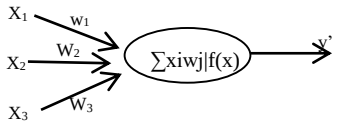
TP n1 en Apprentissage profond

Exercice 1. Percptron simple

Algorithme d'apprentissage pour des réseaux sans couche cachée

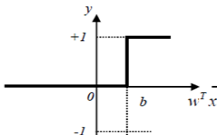
1. Initialiser les poids avec une valeur arbitraire
2. Répéter : Choisir un exemple dans la base d'entrainement $<x, \check{o}>$ (x est l'entrée, est la sortie désirée
- Calculer la somme de chaque neurone : $S_j = \sum_{i=1}^{N} w_{ij} x_i$
- Calculer les sorties () du réseau : $o_j = f(S)$
- Pour chaque sortie, calculer l'erreur : $\delta_j = \left(\check{o}_j - o_j\right)$
- Actualiser les poids synaptiques : $w_{ij} = w_{ij} + \eta . \delta_j . x_i . \frac{df(S)}{dS}$

On veut construire par apprentissage un perceptron qui calcule le « ET » logique en utilisant l'algorithme « par correction d'erreur ».



On cherche à créer un modèle qui effectue une classification binaire (prédiction une probabilité que le résultat d'une entrée soit 0 ou 1)

1. Quels sont les critères d'arrêt possibles de cet algorithme :
2. En choisissant :



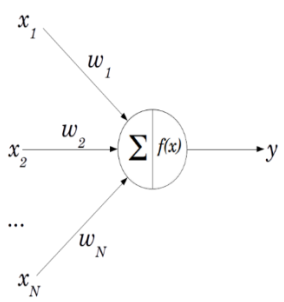
- comme critère d'arrêt la stabilité des poids,
- comme fonction d'activation linéaire (fonction identity),
- comme valeurs initiales des poids : (-1, 1, 1)
- Introduire les exemples de l'échantillon complet dans l'ordre lexicographique, c'est à dire : 100, 101, 110, 111.

Reproduire les premieres étapes d'exécution de l'algorithme dans le tableau suivant :

Etape	Entrée	sortieD	Poids intial			Σxi*wi	sortieC	Erreur	Poids final		
	X_1 X_2 X_3		w_1	w_2	w_3				w_1	w_2	w_3
1	100	0	-1	1	1						
2	101	0	-1	1	1						
3	110	0	-1	1	1						
4	111	1	-1	1	1						

Exercice 2: Perceptron Multicouche

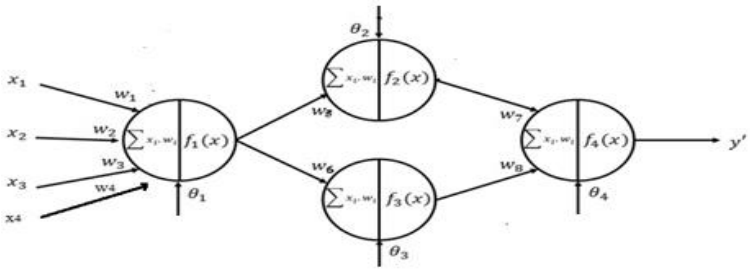
Algorithme d'apprentissage pour des réseaux avec couches cachées :



1. Initialiser les poids avec des valeurs aleatoires
2. Répéter
   - Choisir un exemple dans la base d'apprentissage $<x, \check{o}>$ (x est l'entrée, o est la sortie désirée)
   - Calculer la somme de chaque neurone : $S_j = \sum_{i=1}^{N} w_{ij} x_i$
   - Calculer les sorties () du réseau : $y' = f(S)$
   - Pour chaque sortie, calculer l'erreur. En pratique et pour simplifier la suite des équations, l'erreur est divisée par 2 : $E = \frac{1}{2}(y' - y)^2$
   - Actualiser les poids synaptiques avec descente de gradient :

$$w_i' = w_i - \eta . \frac{dE}{dw_i} = w_i - \eta . x_i . (y' - y) . \frac{df(S)}{dS}$$

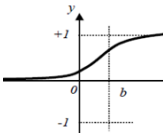
On considère un réseau multicouche avec quatre entrées comme est indiqué dans la figure suivante.



Architecture de réseau

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
0,5	-0,5	0,2	-0,5	0,2	0,5	-0,5	0,2

θ_1	θ_2	θ_3	θ_4
1	1	1	1



1. Donner la formule qui permet de calculer la sortie y', avec $f_1(x) = f_2(x) = f_3(x) = f_4(x)$ = sigmoide. La fonction sigmoïde permettra de traduire facilement un résultat entre 0 et 1.
2. Vérifier que y' = 0.7767 pour $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, et $x_4=0$,
3. Donner l'équation de réajustement (d'apprentissage) de w_8 qui permet de minimiser l'erreur quadratique par l'algorithme de la rétro-propagation de gradient.
4. Donner la formule du gradient partiel de $\frac{\partial E}{\partial w_i}$

$$\Delta w_{ji} = w_{ji}^{t+1} - w_{ji}^{t} = \alpha (y_j^0 - y_j) x_i$$

5. Sachant que le taux d'apprentissage α=0.05, et la sortie désirie y=1, calculer les valeurs de w_i

	w_8	w_7	w_6	w_5	w_4	w_3	w_2	w_1
rétropropagation								

Exercice 3:

On considère un réseau multicouche avec deux entrées. On s'intéresse à l'apprentissage des fonctions booléennes AND, OR et XOR par un MLP.

Entrée	Sortie (classe)		
X1 X2	X1 AND X2	X1 OR X2	X1 XOR X2
0 0	0	0	0
0 1	0	1	1
1 0	0	1	1
1 1	1	1	0

1. Copier les observations dans varaible X, et les classes dnas variable y
2. Définir un classifieur MLP pour apprendre l'opérateur AND, sans couche cachée (hidden_layer_sizes = ()), avec une activation linéaire (fonction identity), et un solver de type lbfgs. Vérifier que les résultats prédits par le classifieur sont corrects, en appelant classifier.predict(x_test).

MLPClassifier (activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08, hidden_layer_sizes=(4, 2), learning_rate='constant', learning_rate_init=0.001, max_iter=200, momentum=0.9, n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5, random_state=None, shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1, verbose=False, warm_start=False)

Chacun des paramètres apparaissant dans le retour de l'appel de la fonction **mlp.fit** peut être précisé lors de l'appel de la fonction MLPClassifier.

solver : ce paramètre représente l'algorithme à utiliser pour le problème d'optimisation. Il peut prendre l'une des options suivantes :

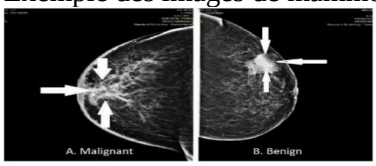
- ***linlinear :*** le meilleur choix pour les petits ensembles de données. Il gère la pénalité L1.
- ***newton-cg :*** gère que la pénalité L2.
- ***lbfgs :*** gère la perte multinomiale pour les problèmes multi classes. Il ne gère également que la pénalité L2.

- ***saga*** *:* représente un bon choix pour les grands ensembles de données. Pour les problèmes multi classes, il gère les pertes multinomiales. Il prend en charge la pénalité L1 et la pénalité « *elasticnet*».
- ***sag*** *:* cette option est aussi idéale pour les grands ensembles de données et gère la perte multinomiale pour les problèmes multi classes.

3. Définir un classifieur MLP pour apprendre l'opérateur OR, sans couche cachée (hidden_layer_sizes = ()), avec une activation linéaire (fonction identity), et un solver de type lbfgs. Vérifier que les résultats prédits par le classifieur sont corrects.
4. Définir un classifieur MLP pour apprendre l'opérateur XOR,
   (a) sans couche cachée (hidden_layer_sizes = ()), avec une activation linéaire (fonction identity) et un solver de type lbfgs. Est-ce que les résultats prédits par le classifieur sont corrects ?
   (b) utiliser une seule couche cachée composée de 4 neurones, des activations linéaires (fonction identity) et un solver de type lbfgs. Est-ce que les résultats prédits par le classifieur sont corrects ?
   (c) utiliser deux couches cachées composées de 4 neurones (première couche) et 2 neurones (deuxième couche), des activations linéaires (fonction identity) et un solver de type lbfgs. Est-ce que les résultats prédits par le classifieur sont corrects ?
   (d) utiliser deux couches cachées composées de 4 neurones (première couche) et 2 neurones (deuxième couche), des activations non-linéaires de type tangente hyperbolique (fonction tanh) et un solver de type lbfgs. Est-ce que les résultats prédits par le classifieur sont corrects ?
5. Refaire l'apprentissage plusieurs fois après avoir constaté les résultats prédits par le classifieur. Est-ce que les résultats prédits par le classifieur sont corrects ?

Exercice 4 PMC avec Keras

Dateset : Breast cancer La base de données Breast Cancer comporte des descriptions de tumeurs avec une étiquette précisant si la tumeur est maligne ou bénigne. La base de données contient 569 exemples décrits à l'aide de 30 descripteurs.
L'analyse de cette base de données, vise à identifier les caractéristiques les plus utiles pour prédire le cancer malin ou bénin et à identifier les tendances générales susceptibles de nous aider dans la sélection du modèle et dans la sélection des hyper paramètres. L'objectif est de déterminer si le cancer du sein est bénin ou malin.
Exemple des images de mammographie pour les types de cancer du sein



Informations sur les attributs : La moyenne, l'erreur standard et la "worst" ou la plus grande (moyenne des trois plus grandes valeurs) de ces caractéristiques ont été calculées pour chaque image, ce qui a donné 30 caractéristiques. Par exemple, le champ 3 est le rayon moyen, le champ 13 est le rayon SE, le champ 23 est le pire rayon.
La base de données est complète et fournit en sortie une valeur binaire (2 classes) : -
0 : tumeur bénigne, 1 : tumeur maligne

1. Charger la base de données Breast Cancer
2. Copier les observations dans variable X, et les classes dans variable y
3. Diviser les données de la base en un ensemble d'apprentissage et un autre ensemble de test
4. Standariser les données d'apprentissage et de test
5. Entrainer le modele sans couche cachée

```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
couche_sortie = Dense(output_dim=1, activation='sigmoid', input_dim=D)
model.add(couche_sortie)
```

input_couche : pour definir la couche d'entrée et la premiere couche cachée
couche_cachee1 : pour definir la seconde couche cachée
…
couche_sortie : pour definir la couche de sortie

- **input_dim** : nombre de colonnes de dataset ((ici D=30)
- **output_dim** : nombre de neurones de la couche suivante (ici une seule sortie (0, ou 1))
- **activation** : fonction d'activation. La fonction ReLU est f(x)=max(0,x), le calcul du gradient est très simple (soit 0 soit 1 selon le signe de x).

6. Compiler le modele, avec l'optimiseur « adam », et la fonction de cout (lost function) « Binary_crossentropy »

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

- **optimizer** : c'est un algorithme qui va dicter comment mettre à jour le modele, pour diminuer le loss, et avoir donc de meilleures prédictions. Ici on s'appuiera sur « adam » (adaptive moment estimation), très souvent utilisé.
- **metrics**=['accuracy'] est le taux de bonne classifications
- **Loss** : La perte d'entropie croisée, ou perte logarithmique, mesure les performances d'un modèle de classification dont la sortie est une valeur de probabilité comprise entre 0 et 1. La perte d'entropie croisée augmente à mesure que la probabilité prédite s'écarte de l'étiquette réelle.

Fonctions de perte classiques : "mse" en régression, "categorical_crossentropy" en classification multi-classes, et "binary_crossentropy" en classification binaire.

7. Lancer l'apprentissage

```
r=model.fit(X_train, y_train, batch_size=100, nb_epoch=150, validation_data=(X_test, y_test))
```

- **batch size** correspond au nombre d'échantillons traité avant de mettre à jour le modèle.
- **epochs** correspond au nombre de fois que l'on traite le jeu d'entrainement.
- **validation_data** permet de fixer, lors de l'appel à la méthode fit(), une fraction du jeu d'apprentissage à utiliser pour la validation, pour se rendre compte de l'ampleur du phénomène de sur-apprentissage.

  Répéter les comparaisons de modèles en focalisant sur le taux de bonnes classifications obtenu sur le jeu de validation.

8. Predire le modele sur les données de test

```
y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5)                    # conversion en binaire (True ou Fasle)
print("Train score:", model.evaluate(X_train, y_train)) # return loss and accuracy
print("Test score:", model.evaluate(X_test, y_test)) # return val_loss and val_accuracy
```

9. Afficher graphiquement la matrice de confusion, en utilisant la librarie seaborn.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
…
```

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

```
import matplotlib.pyplot as plt
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
plt.show()
```

11. Comparer les performances de ce premier modèle avec celle de modèle avec respectivement **2 couches cachées de 16 neurones chacune**. Utilisere la fonction ReLU ("relu") comme fonction d'activation pour les neurones des couches cachées. Mettre en place une stratégie de *Drop-Out* pour aider à la régularisation du réseau.