

**TP – Classification d’espèces d’iris avec un réseau de neurones artificiel**  
**Partie 1**

Le but de ce TP est de travailler à la classification d’espèces d’Iris en utilisant un réseau de neurones.

**Objectifs :**

- Observer les données, comprendre leur nature et comment les adapter (si besoin) pour les utiliser avec le modèle d’apprentissage réseau de neurones
- Comprendre le fonctionnement d’un réseau de neurones pour le mettre en œuvre dans un programme

## 1 Rappels / présentation des données

*C.f.* sujet de TP 2 (Arbre de Décision)

✎ Pour ce TP, allons-nous être dans le cas d’une classification supervisée ou non-supervisée ?

## 2 Chargement et pré-traitement des données

### 2.1 Prétraitement

Comme pour les TPs précédent, commencez par charger le `csv` dans un *dataframe* :

```
import pandas as pd
iris_df = pd.read_csv("path/to/data.csv")
```

Comme nous allons mettre en œuvre notre modèle *from scratch*, cela sera plus facile pour nous de manipuler des structures de données “classiques” de Python (listes et tableaux). Pour cela, convertissez votre *dataframe* en utilisant l’instruction suivante :

```
df_columns = iris_df.columns.values.tolist()
```

Nous allons ensuite séparer les étiquettes  $y$  des attributs  $X$ . On va extraire les attributs et étiquettes puis construire des tableaux contenant les données. Avec Pandas, si on exécute l’instruction `petal_length = df_columns['petal_length']`, on va récupérer dans un tableau python la colonne correspondant à l’attribut spécifié. Comme nous avons plusieurs attributs, nous allons procéder plus généralement comme ceci :

```
features = df_columns[0:4] # 4 premières colonnes
label    = df_columns[4:]  # 4ème colonne (suppose que l’étiquette _est_ la colonne 4)
```

(NB si il y a plus d’attributs il faut bien entendu changer la plage spécifiée. Si l’étiquette –ce que l’on veut prédire– n’est pas la dernière colonne, alors on peut, avec Pandas, déplacer cette colonne à la fin)

On peut maintenant récupérer, dans des tableaux, les données :

```
X = iris_df[features]
y = iris_df[label]
```

Les étiquettes de nos données sont des catégories décrites textuellement (*Iris-setosa*, *Iris-versicolor* et *Iris-virginica*). Nous devons changer ces étiquettes de façon à pouvoir les manipuler facilement. Une première idée est de leur attribuer un identifiant numérique (*e.g.* *Iris-setosa* = 1, *Iris-versicolor* = 2 et *Iris-virginica* = 3). Le problème de cet encodage en apprentissage automatique, c'est qu'il peut entraîner un biais lors de l'apprentissage du modèle à cause de la relation d'ordre<sup>1</sup> (pas dans le cas de *tous* les modèles, mais les réseaux de neurones artificiels y sont sensibles). Une alternative intéressante est d'utiliser un encodage **one-hot**. C'est un encodage binaire dont tous les bits sont à zéro sauf celui correspondant à la classe effective :

class_Iris-setosa	class_Iris-versicolor	class_Iris-virginica
0	0	1
0	1	0
1	0	0

La bibliothèque Pandas permet de réaliser ce type d'encodage : `y = pd.get_dummies(y)`.

## 2.2 Données d'entraînement et de test

Pour pouvoir entraîner correctement notre modèle, il faut pouvoir le tester sur des données qu'il n'a jamais vu<sup>2</sup>. En général, on conseille de mettre de côté 20% du jeu de données total pour tester le modèle. Le problème, c'est que lorsque l'on a peu de données (dans notre cas par exemple), la proportion de données pour l'entraînement peut devenir ridiculement petite...

Pour mettre de côté une portion des données pour tester le modèle, on va utiliser une fonction de la bibliothèque 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=0.15, random_state=42)
```

## 3 Construction du modèle de réseau de neurones artificiel

Comme on l'a vu en cours, un réseau de neurones type "perceptron multi-couches" n'est rien d'autre qu'un ensemble de matrices stockant les paramètres du réseau sur chaque couche. On travaillera donc uniquement sur des ensembles de matrices, pour stocker les paramètres (poids et biais) d'une part, et les données d'autre part. Pour tout ça, la bibliothèque Numpy sera notre amie. Vous aurez aussi besoin de quelques fonctions d'activation et de fonctions de coûts pour mesurer la véracité de vos prédictions par rapport aux résultats attendus : pour cela, vous avez à disposition un fichier `utility.py` qui contient une classe définissant des méthodes statiques que vous pourrez utiliser.

### Questions

1. Quelles fonctions d'activation sont à disposition dans `utility.py` ?
2. Que renvoient ces fonctions ?
3. Rappelez pourquoi l'utilisation d'un **softmax** est intéressante dans le cas d'un problème de classification

---

1. Ici, on ne peut pas dire que *Iris-versicolor* < *Iris-virginica* : on ne veut pas que notre modèle infère ce type de relation. Si on reprend l'exemple de "aller jouer au golf" alors dans ce cas oui, la relation d'ordre peut être intéressante : *pluvieux* = 1 < *ensoleillé* = 2

2. Tester un modèle sur des données utilisées pour modifier les paramètres –l'entraînement– s'appelle du *data snooping*

### 3.1 Architecture du réseau de neurones

On propose de définir l'architecture illustrée sur la figure 1. Ce modèle comprend 2 couches cachées de respectivement 3 et 2 unités chacune. Les entrées et sorties du modèle sont déterminées par les données (4 attributs en entrée, one-hot de 3 bits en sortie). Les matrices de poids sont de dimensions  $n_l \times n_{l-1}$  où  $n_l$  (le nombre de lignes) est le nombre d'unités sur la couche  $l$  et  $n_{l-1}$  (le nombre de colonnes) est le nombre d'unités sur la couche  $l - 1$  (C.f. figure 2). Les biais (qui ne figurent pas sur la figure 1 pour ne pas allourdir le schéma) sont représentés par des matrices<sup>3</sup> de dimensions  $n_l \times 1$ .

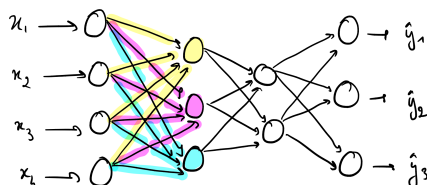



FIGURE 1 – Réseau de neurones artificiel à 2 couches cachées (les biais ne sont pas représentés, pour alléger la figure)

$$W^1 = \begin{pmatrix} w_{11} & w_{21} & w_{31} & w_{41} \\ w_{12} & w_{22} & w_{32} & w_{42} \\ w_{13} & w_{23} & w_{33} & w_{43} \end{pmatrix}$$

FIGURE 2 – Matrice de poids pour la couche  $l = 1$

 Sur papier, représentez les matrices de poids / vecteurs de biais pour les couches  $l = 2$  et  $L$  (la couche de sortie).

### 3.2 Représentation d'un modèle de réseau de neurones en Python

Les éléments de base pour représenter un réseau de neurones artificiel sont les suivants :

- un nombre de couches cachées et leur dimensions
- une liste de matrices de poids
- une liste de matrices de biais
- une liste de matrices d'entrées pondérées
- une liste de matrices d'activations
- un taux d'apprentissage ( $\eta$ )
- une fonction d'activation pour les unités des couches cachées
- un nombre d'*epoch* pendant lequel entraîner le modèle

Pour vous aider dans la suite, voici une manière de définir une liste de matrices de poids dans le constructeur de votre future classe : `self.weights = [None] * (self.n_layers + 1)` (initialise une liste de `n_layers + 1` éléments à `None`, i.e. `[None, None, ..., None]`)

**NB** : avant d'aller plus loin, on part du principe que vos données sont pré-traitées et séparées en un jeu d'entraînement et un jeu de test.

#### 3.2.1 Constructeur de la classe

Créez une classe `NeuralNet` et définissez son constructeur en utilisant les éléments ci-dessus. On propose d'utiliser le prototype suivant :

```
def __init__(self, X_train = None, y_train = None, X_test = None, y_test = None,
             hidden_layer_sizes=(4,), activation='identity', learning_rate=0.1, epoch=200):
    # À faire...
```

3. Oui, techniquement c'est un vecteur mais dans la mise en œuvre on se simplifie la vie en considérant *tout* comme des matrices

### 3.2.2 Initialization des matrices de paramètres (poids et biais)

Pour initialiser les paramètres, on va simplement assigner des valeurs aléatoires comprises dans  $[-1; 1]$ . Pour cela, on peut utiliser la fonction `numpy.random.uniform(low=0.0, high=1.0, size=None)`<sup>4</sup>. Avec le paramètre `size`, nous allons pouvoir spécifier les dimensions de nos matrices (que vous devez avoir déjà calculé lorsque l'architecture vous a été présentée!). Il faut faire cette opération d'initialisation pour chaque couche cachée puis terminer par la couche de sortie. On propose le prototype suivant : `def __weights_initialization(self, X, y):`. Vous devrez appeler cette fonction dans le constructeur de votre classe.

### 3.2.3 Propagation avant

Le modèle construit, on peut maintenant mettre en œuvre la fonction de propagation avant. Pour chaque couche  $l$  de votre réseau, vous devez calculer les entrées pondérées de la couche et les activations :  $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$  et  $A^{[l]} = g(Z^{[l]})$ . N'oubliez pas que la couche de sortie  $L$  est activée avec la fonction `softmax`!

Pour la première couche cachée,  $A^{[l-1]}$  correspond aux **entrées** du réseau. Nous allons travailler en mode *online*, c'est-à-dire utiliser les instances une par une. Les données passées en paramètres à votre réseaux (`X` et `y`) contiennent des lignes représentant les instances. Pour pouvoir les multiplier avec vos matrices de poids, il faut les transposer :

```
X_train = X_train.to_numpy().transpose()
y_train = y_train.to_numpy().transpose()
```

Une fois la propagation avant terminée (*i.e.* vous avez obtenu des prédictions en sortie), avec la fonction de coût `cross entropy`, calculez l'erreur de votre prédiction  $\hat{y}$  par rapport à la sortie attendue  $y$  et renvoyez cette erreur.

### Et la suite ??

Si vous arrivez jusqu'ici, vous avez déjà fait un gros boulot !

Il reste à mettre en œuvre la rétropropagation du gradient et à mettre à jour les paramètres, ainsi que de spécifier plus largement une époque d'entraînement (mélange des données / passe avant / passes arrière et mise à jour des paramètres / passe avant sur le jeu de test) : *cela sera l'objet du prochain TP !*

---

4. <https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.uniform.html>