



Année Universitaire 2020-2021

“MACHINE LEARNING ET OLFACTION”

Présenté par :
Assaad ASSAAD
Mike BENSIMON
Benjamin FAUSTINI
Benjamin RICHTER

**Sous la direction de M. Cédric BERNARDIN
Et le Tutorat de M. Sébastien FIORUCCI**

Projet de Fin d'Étude
Master Ingénierie Mathématique
Option Modélisation Stochastique et Statistique

SOMMAIRE

I. Introduction

II. Méthodologie

A. Jeu de données du Dream Olfaction Challenge

B. Modèle Machine Learning

1. Rappel Théorique

2. Prétraitement des Données

3. Mise en place du modèle

C. Modèle Deep Learning

1. Rappel Théorique

2. Prétraitement des Données

3. Mise en place du modèle

III. Résultats et Discussion

A. Modèle Machine Learning

B. Modèle Deep Learning

IV. Conclusion

V. Références

VI. Annexe des Codes

I-INTRODUCTION

L'Homme possède cinq sens et l'odorat est souvent considéré à tort comme le moins développé. Lorsqu'une molécule entre dans notre nez elle circule jusqu'à notre muqueuse olfactive, située au sommet de la cavité nasale, cette muqueuse est constituée de récepteur protéinique qui s'active ou non en fonction du type de la molécule odorante qui entre en contact avec eux, une molécule peut activer plusieurs récepteurs en même temps. Dès leurs activations, les neurones sensoriels olfactifs, liés à ces récepteurs, envoient un signal à notre cerveau. Et c'est ce signal que nous percevons comme une odeur. Nous disposons d'environ 400 récepteurs olfactifs permettant de différencier plus de 1000 milliards d'odeurs différentes. L'une des principales préoccupations des neurosciences est la structuration des données, par laquelle, le cerveau se représente les informations sur le monde extérieur. Certains schémas de codage expliquent comment les neurones représentent, stockent, rappellent et manipulent ces informations en se basant, pour la plupart, sur la corrélation de diverses représentations des neurones avec les caractéristiques de stimuli. Mais encore aujourd'hui, la seule façon de déterminer quelle perception olfactive va donner une structure chimique est de la sentir. En effet, la résolution de problèmes de perception olfactive est difficile car l'espace perceptif de l'olfaction est lui-même complexe et a beaucoup plus de dimensions que les autres sens. Nous observons, par exemple, en Figure 1 que des structures chimiques, très semblables, correspondent à des odeurs bien différentes.

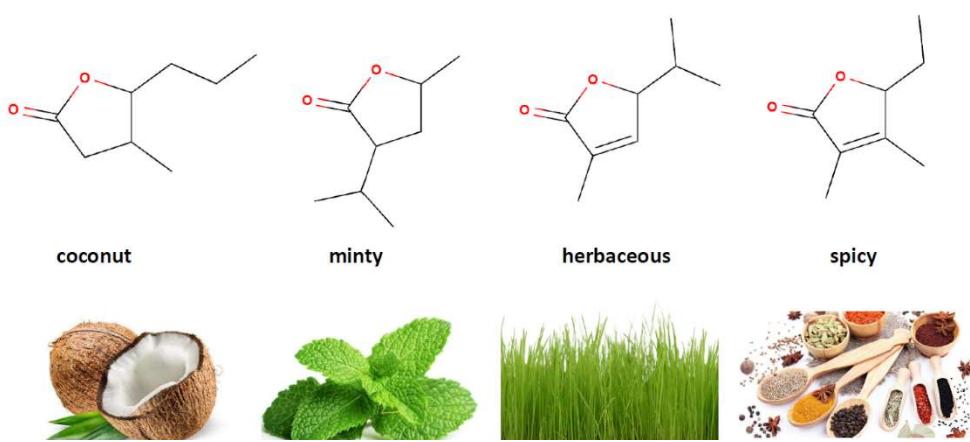


Figure 1 Exemple de relation structure-odeur

Avec l'essor de Machine Learning, des neuroscientifiques ont proposé un concours dans laquelle des data scientifiques essayeraient de prédire les odeurs ressenties à partir des données physico-chimiques d'une molécule : *Le Dream Olfaction Prediction Challenge*.

Celui-ci est découpé en 2 parties :

Le Dream Challenge 1 : qui consiste à prédire de manière individuelle 21 descripteurs olfactifs pour chaque individu testé

Le Dream Challenge 2 : qui consiste à prédire la moyenne et l'écart type des 21 descripteurs olfactifs ressentis par une population.

Predicted individual perception



Predicted population perception
(mean and standard deviation)



Figure 2 Schéma du Dream Challenge 1 et 2

Les données, pour permettre ce challenge, ont été collectées lors d'une expérience d'analyse sensorielles où un panel de 49 personnes ont évalué 476 odeurs à deux intensités différentes selon 21 attributs perceptifs que nous allons détailler par la suite.

L'étude de ce sujet permettrait de fournir des informations sur la façon dont les odeurs sont transformées en une perception d'odeur dans le cerveau. De plus, être capable de prédire comment un produit chimique serait perçu, améliorerait considérablement la conception de nouvelles molécules à utiliser en parfumerie.

Pour la réalisation de ce projet de fin d'étude, nous avons étudié les codes de Richard GERKIN et son équipe pour le Dream Challenge 2, celui-ci ayant obtenu le meilleur score de la compétition. Puis, remarquant que l'intégralité des candidats de la compétition ont utilisé des technologies de Machine Learning pour participer, nous avons décidé d'utiliser les technologies du Deep Learning afin de voir si nous pouvions améliorer les résultats obtenus lors de ce Challenge.

II. MÉTHODOLOGIE

A- Jeu de données du Dream Olfaction Challenge

Les données utilisées pour ce Challenge ont été collectées entre Février 2013 et Juillet 2014 sur une population de 49 individus (dont 28 femmes) âgés de 18 à 50 ans d'origines ethniques diverses. Il a été demandé à chacun de ces individus de sentir 476 molécules différentes à deux concentrations différentes (élévée/faible) et de décrire à quel point l'odeur ressentie était forte (*Intensity/Strength*), agréable (*Valence/Pleasantness*), et dans quelle mesure la perception de l'odeur correspond à une liste de 19 descripteurs olfactifs (*bakery, sweet, fruit, fish, garlic, spices, cold, sour, burnt, acid, warm, musky, sweaty, ammonia/ruinous, decayed, wood, grass, flower and chemical*).

De plus, parmi les molécules testées, 20 ont été reproduites, de sorte que chaque sujet ait évalué 992 stimuli (476 + 20 répliquées, à deux intensités).

Les données récoltées sont ainsi partagées en Trois parties :

-358 (338 + 20 répliquées) molécules sont utilisées pour l'entraînement.

-69 pour le Leaderboard

Ensemble, elles forment le jeu de données total fournis au participant.

Les 69 dernières forment le test set qui n'ont pas été distribué par les organisateurs pour permettre le scoring du Challenge.

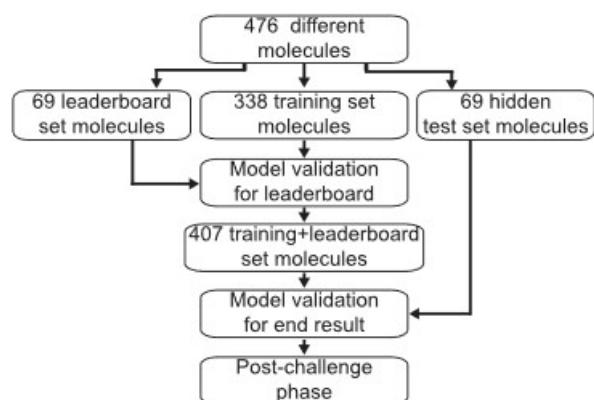


Figure 3 Répartition des Molécules

A partir du premier jeu de données, nous obtenons un fichier TrainSet contenant les données d'entraînement qui seront à prédire par nos modèles, sur lesquelles nous allons travailler, comportant 27 colonnes et 35084 lignes ((338 odeurs + 20 répliquées) * 2 intensités * 49 sujets) qui sont dans l'ordre :

- “Compound Identifier” contenant le PubChem Compound Identification (CID) qui est l’identifiant unique de la structure chimique.
- “Odor” qui comprend un nom chimique de l’odeur. Il existe, tout de même, plusieurs noms pour la même molécule.
- “Replicate” indique si le stimulus fait partie des 20 stimuli qui ont été testés deux fois. Si c'est le cas, il y aura marqué “replicate”.
- “Intensity”, nous indique si la concentration lors du test était à haute intensité (“high”) ou à faible intensité (“low”).
- “Dilution” contient la dilution de la molécule. Les molécules, dégageant une odeur faible, sont moins diluées que celle dégageant une odeur forte pour que l'on puisse avoir une intensité approximativement équivalente pour que les données ne soient pas biaisées. Afin de servir d'étalon, toutes les molécules ont aussi été testées avec la concentration 1/1000.
- “Subject#” représentant le sujet dont il s’agit, chacun des sujets correspond à un nombre entre 1 et 49.
- “Intensity/Strength” contient les données perceptuelles sur l'intensité ou la force de l'odeur perçue par le sujet. Cette valeur est définie en se basant sur une échelle de 0 à 100 où 100 désigne "extrêmement fort" et 0 "extrêmement faible".

B- Modèle Machine Learning

1-Rappel Théorique

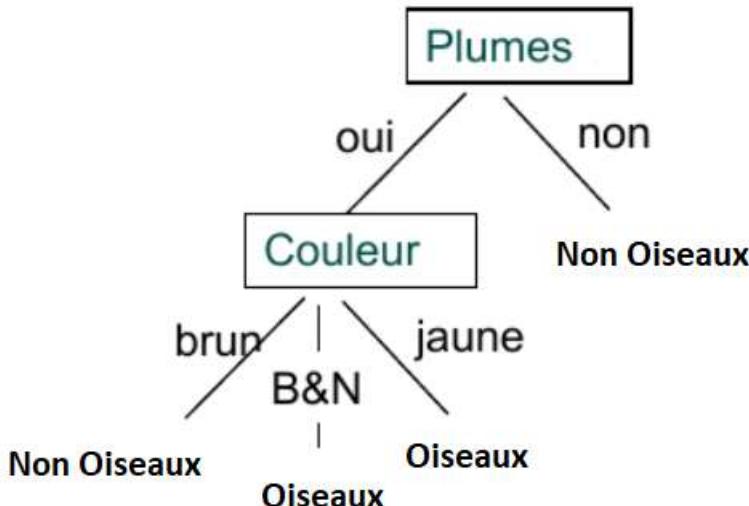


Figure 6 Arbre de décision simple

Les Arbres de Décision :

Les arbres de décision sont une technique de machine Learning permettant de décrire comment répartir une population d'individus (animaux, personne...) en groupes homogènes selon un ensemble de variables discriminantes (âge, taille, ...) et en fonction d'un objectif fixé (aussi appelé « variable d'intérêt » ou « variable de sortie »).

Chaque nœud de l'arbre est un paramètre décrivant les données et chaque liaison partant de se nœuds corresponds à certain nombre de discréétisation de ce paramètre. Enfin les feuilles sont les sorties de l'arbres et correspondent à une des classes attendues en sortie.

Exemple : Déterminer si un animal est un oiseau ou non.

L'arbre de décision ci-dessus classe les données selon la présence de plume ou non, puis en fonction de leurs couleurs. (Figure 1)

Random Forest Regressor :

Il est souvent d'usage, pour des dataset imposants, composé de nombreux paramètre, d'utiliser plusieurs arbres de prédictions. C'est ce qu'on appelle une Random Forest.

Chaque arbre prends aléatoire un nombre prédéfini de paramètre pour permettre de faire son apprentissage, puis pour la prédition, la Random Forest renvoie la valeur majoritairement prédictée par ses arbres pour le set de donnée.

Dans le cas de la Random Forest Regressor, c'est une moyenne des sorties des arbres et non la classe majoritaire, qui définit la valeur de sortie du modèle.

Extra Tree Regressor :

L'Extra Tree Regressor, est un modèle de Random Forest, à la différence que le split des nœuds des arbres le composant se fait de manière aléatoire et non optimal (comme pour une RB).

De plus, chacun de ses arbres s'entraînent sur la totalité des paramètres (et non un certain nombre comme pour les RB)

2-Pré traitement des données

Au regard du nombre considérable de données du DREAM Challenge, le prétraitement des données olfactives est une étape fondamentale pour l'optimisation du modèle de Richard Gerkin. Dans un premier temps, Gerkin crée le jeu de données X, pour cela il charge 2 fois tous les descripteurs moléculaires une fois pour chaque dilution (« high » et « low ») qui ont été faite. Il se retrouve donc avec une matrice de taille 676×4871 . Etant donné la mauvaise qualité des données (un nombre très important de NaN), il a fallu convenir d'une technique pour réviser ces données. Gerkin a décidé de nettoyer les données en plusieurs étapes.

Etape 1 :

Il a purgé les descripteurs qui contenaient plus de 25% de NaN. Mais les NaN n'étant pas dépendant du descripteur moléculaire, aucune colonne ne contenait plus de 25% de NaN.

Etape 2 :

Lorsque la colonne contenait moins de 25% de NaN, il a utilisé la méthode de l'imputer médian pour remplacer les données manquantes.

Soit (X_1, \dots, X_n) les différentes molécules et (Y_1, \dots, Y_m) les descripteurs moléculaires, on a pour $Y_{i,j}^*$, la valeur manquante dans la colonne j :

$$Y_{i,j}^* = \frac{\sum_{i=1}^{n_1} y_i}{n_1}$$

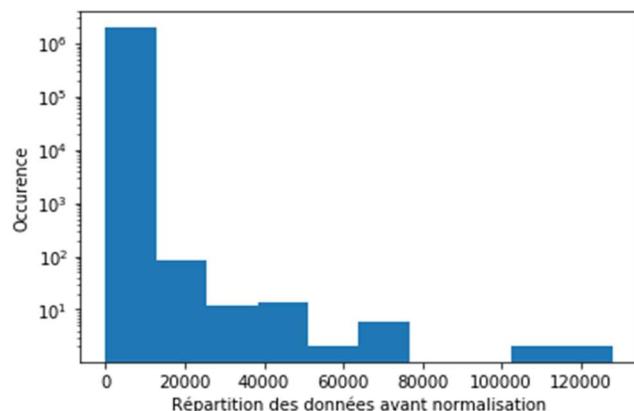
Où :

- n_1 : le nombre de valeurs sans les NaN.

Etape 3 :

Une nouvelle purge est effectuée pour déterminer les descripteurs qui ne sont pas significatifs, Gerkin récupère les colonnes où la somme des valeurs sont nulles (par exemple, le nombre d'atome de phosphore est égale à 0 pour toutes les molécules présentent dans le dataset), plus celles qui ont un écart-type nul. Gerkin supprime donc toutes ces colonnes et se retrouve avec un dataset de taille 676×3033 .

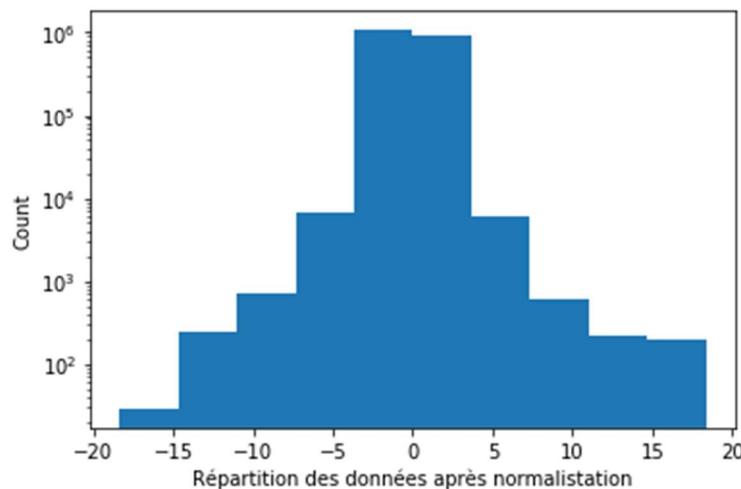
Figure 7 Répartition des données du X_training



Etape 4 :

Dans la figure 1, les données sont très étendues (supérieure à 120000). Du fait de cet écart entre les données, les nombres négatifs sont confondus avec le zéro. Après analyse de ce graphique, on remarque que les données sont biaisées à droite avec l'existence de nombres positif très grands, donc la normalisation la plus adéquate est la normalisation par la racine cubique. Voir ci-dessous la répartition des données après cette normalisation (figure 8). C'est la normalisation que Gerkin a utilisé pour ses données. Ce jeu de données correspondra à son *X_training*.

Figure 8 Répartition des données normalisée



Après quoi Gerkin crée son dataset de validation à partir de *leaderboard*, compte tenu de la répartition des données des fichiers LBS1.txt, LBS2.txt et leaderboard_set.txt.

Il sépare le dataset en 2 jeux de données différents :

D'une part, les molécules avec une concentration « high » *X_leaderboard_other* et l'autre part les concentrations égalent à 1/1000 *X_leaderboard_int*.

Pour ces deux matrices il reproduit les 4 étapes vu ci-dessus.

Pour les données du test set, Gerkin utilise la même méthode de séparation que le dataset *leaderboard* tout en réitérant les 4 étapes. Il crée donc une matrice *X_testset_other* et une matrice *X_testset_int*, toutes deux de tailles 69×3033 .

Dans la matrice *X_all*, il regroupe les données du training et du *leaderboard* brut et ensuite il renouvelle les 4 étapes pour traiter les données de la matrice. Ce qui donne une matrice de taille 814×3033 .

Pour les données d'observations, Gerkin construit 2 jeux de données, une avec l'imputer médian et une autre avec un imputer « mask », où Gerkin ne fait que masquer les NaN sans les remplacer par une valeur. Dans ces deux jeux de données il y intègre une matrice avec 49 sous matrices contenant les 21 attributs perceptuels (une pour chaque sujet en vue du Dream challenge 1) et une autre matrice avec la moyenne et l'écart-type des observations perçues par l'échantillon des attributs perceptuels des différentes molécules (en vue du Dream challenge 2). Ce qui donne pour *Y_training_imp* et *Y_training_mask* 49 matrices de taille 69×21 et une matrice *Y[‘mean_std’]* de taille 676×42 . En revanche, pour le *Y_leaderboard*, il prend les dilutions 1/1000 de chaque molécule et utilise seulement la méthode de l'imputer « mask », il obtient donc un dictionnaire *Y[‘subject’]* qui contient 49 matrices de taille 69×21 et une matrice *Y[‘mean_std’]* de taille 69×42 .

Pour le *Y_all*, il mélange les données training et leaderboard en prenant toutes les dilutions. Chacun de ces datasets est représenté 2 fois. Dans le premier il masque seulement les Nan et dans l'autre il les remplace par leur valeur médiane correspondante (grâce à l'imputer).

3-Mise en place du modèle RFC

Pour la mise en place de son modèle rfc, Gerkin a décidé de séparer les attributs perceptifs en trois ensembles, intensité, valence et les 19 autres attributs perceptifs en se basant sur les connections entre les différents descripteurs.

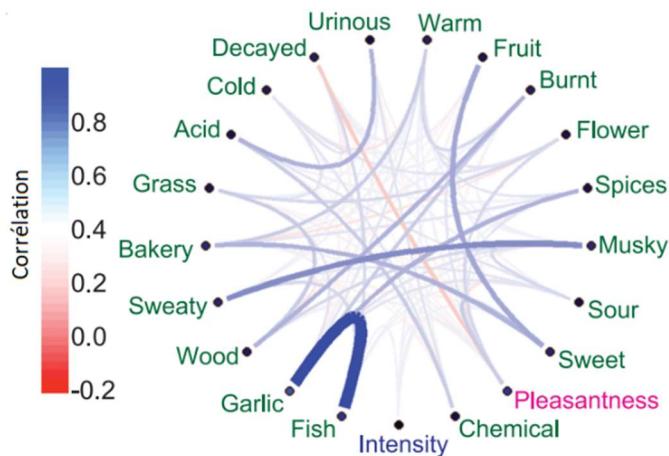


Figure 9 Cercle de Corrélation des descripteurs Olfactifs

On remarque que l'intensité et la valence ont des connections très faible avec les autres descripteurs. (Figure 9)

Le modèle rfc est donc décomposé en deux modèles de machine learning, un Extra Tree Regressor chargé de prédire uniquement la moyenne et l'écart-type de l'intensité, un modèle random forest regressor chargé de prédire séparément la valence et les 19 autres descripteurs.

Le modèle Extra Tree s'entraîne sur tout le dataset et prédit sur le dataset en entier. Tandis que le Random Forest s'entraîne aussi sur le dataset tout mais fait une prédiction via la méthode Out Of Bag.

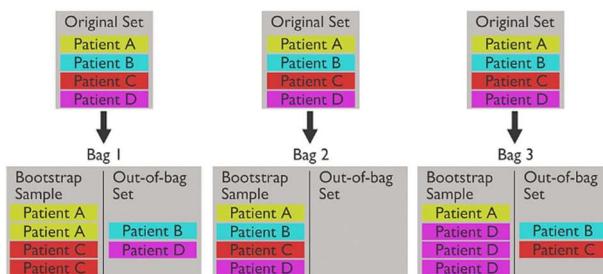


Figure 10 Exemple de Out Of Bag

La procédure out of bag consiste à entraîner les arbres sur un certain nombre d'observation de l'échantillon d'apprentissage et de prédire sur les échantillons non utilisés. En répétant ce processus sur tous les arbres tout au long de l'apprentissage, le modèle sort une prédiction du dataset entier.

Ces modèles s'entraînent deux fois avec deux jeux de données différents, le premier correspond aux observations (Y) où les NaN sont simplement masqués et la deuxième avec la méthode de l'imputer médian active.

Les prédictions de ces modèles sont comparées aux données observées via un coefficient de Pearson. Ce coefficient permettra de montrer l'intensité de la liaison entre les données. Avec ces méthodes, on obtient donc 6 résultats. Dans un premier temps, ils sont divisés chacun par sa valeur de référence correspondante, donné à tous les candidats. Elles proviennent d'un modèle basique que les organisateurs ont codées.

Explications des calculs pour obtenir le score :

Calcul de l'intensité moyenne :

$$\overline{int} = \frac{1}{49} \sum_{i=1}^{49} \left(\frac{1}{69} \sum_{j=1}^{69} int_j \right)_i$$

Calcul de la moyenne de la *Valence* :

$$\overline{val} = \frac{1}{49} \sum_{i=1}^{49} \left(\frac{1}{69} \sum_{j=1}^{69} val_j \right)_i$$

Calcul de la moyenne des descripteurs :

$$dec = \frac{1}{19} \sum_{i=1}^{19} desc_i$$

Où :

- *desc* correspond aux différents attributs perceptifs de la molécule

$$\overline{dec} = \frac{1}{49} \sum_{i=1}^{49} \left(\frac{1}{69} \sum_{j=1}^{69} dec_j \right)_i$$

Où les int, val, dec sont les coefficients de Pearson calculé en comparant les relations entre les données prédites et les données observées. On effectue alors le Z-score de chacune de ces moyennes de coefficient par rapport à la valeur de référence correspondante. On obtient le score final en faisant la moyenne de ces 6 z-scores.

C- Modèle Deep Learning

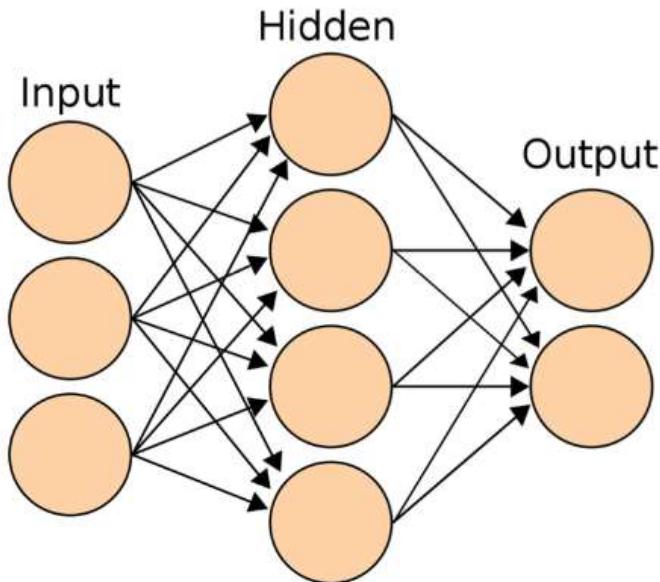


Figure 11 Schéma d'un réseau de neuronne

Nous allons présenter, dans un premier temps le fonctionnement d'un réseau de neurones. Puis nous présenterons la préparation les données dans le but de les utiliser dans nos réseaux de neurones. Enfin nous présenterons les 2 modèles de neural network que nous avons utilisé ainsi que leur mise en œuvre.

1- Rappel Théorique

Un réseau de neurones (neural networks en anglais), est un système informatique matériel dont le fonctionnement est calqué sur celui du cerveau humain. Il s'agit d'une variété de technologie Deep Learning (apprentissage profond), qui fait partie du Machine Learning. Chaque neurone est considéré comme un point du réseau. Il reçoit une ou plusieurs informations en entrée, et renvoie une information en sortie. À la différence d'un réseau de serveur internet, les informations échangées dans un réseau de neurones sont simples : elles correspondent à seulement une intensité de signal, à l'image des échanges électrochimiques de notre cerveau.

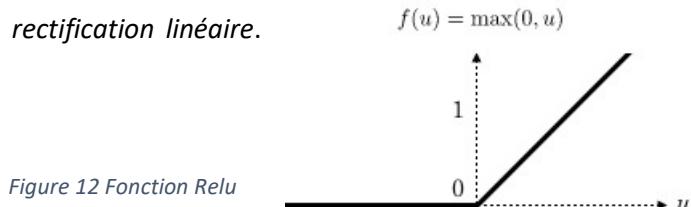
Un réseau repose généralement sur plusieurs couches de neurones, s'activant les unes après les autres. La première couche, dite couche d'entrée ou input layer, reçoit le signal brut (les données). Puis viennent les couches centrales du réseau. Comme nous ne voyons pas les échanges d'informations entrant et sortant de leurs neurones, elles ont été baptisées couches cachées (hidden layers). La dernière couche émet le signal final, d'où son appellation d'output layers. De manière générale, le nombre de neurones en entrée est défini en fonction de l'information traitée, celui des outputs layers, dépend lui du nombre de réponses possibles que le réseau peut renvoyer.

Par exemple, si à partir d'une photo, notre réseau doit faire la différence entre un chat et un chien, les données pourront être les valeurs RGB des pixels de la photo et les sorties seront : "chat" ou "chien". Une fois que l'on connaît le nombre d'input et d'output, on peut définir le nombre de couches cachées ainsi que le nombre de neurones associés. Cependant, obtenir un réseau optimisé répondant à l'objectif visé est encore bien souvent compliqué, les problèmes de sur ou de sous-apprentissage sont fréquents. Dans le cadre d'un Full-Connected Network, c'est-à-dire lorsque chaque neurone est connecté à l'ensemble des neurones de la couche précédente, le neurone reçoit plusieurs valeurs d'entrée, chacune possédant un poids.

Il calcule ce que l'on appelle la valeur d'agrégation de la couche. Cette valeur est ensuite transmise à la fonction d'activation. Elle comporte un biais, qui peut être considéré comme le seuil d'activation de notre neurone. Celui-ci permet de palier à une activation du neurone due à un bruit résiduel. L'image de cette valeur d'agrégation par la fonction d'activation nous donne la sortie du neurone. Elle est comprise entre 0 et 1, où 0 signifie que le neurone est totalement inactif, et 1, que le neurone est totalement actif. Il existe plusieurs types de fonctions d'activation comme la tangente hyperbolique ou la sigmoïde.

Pour nos réseaux, nous utiliserons la *rectification linéaire*.

Celle-ci est définie comme :



Nous utiliserons également la fonction d'activation LogSoftmax, qui est interprétée par la formule pour tout : $z = (z_1, \dots, z_k) \in \mathbb{R}^k$

$$\text{LogSoftmax}(z_j) = \log\left(\frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}\right), \forall j \in \{1, \dots, K\}$$

L'ensemble de ce processus est nommé feed-forward. Le but du réseau lors de l'apprentissage est de trouver les valeurs adéquates des poids de connexions entrantes ainsi que des biais pour chacun des neurones du réseau. Mathématiquement, on peut exprimer le calcul de la valeur d'activation entre chaque couche par l'équation matricielle suivante :

$$a^1 = \sigma(Wa^0 + b)$$

$$\begin{pmatrix} a_0^1 \\ \vdots \\ a_k^1 \end{pmatrix} = \sigma\left(\begin{pmatrix} W_{00} & \dots & W_{0n} \\ \vdots & \dots & \vdots \\ W_{k0} & \dots & W_{kn} \end{pmatrix} \begin{pmatrix} a_0^0 \\ \vdots \\ a_n^0 \end{pmatrix} + \begin{pmatrix} b_0 \\ \vdots \\ b_k \end{pmatrix}\right)$$

Où :

- a^1 correspond à la valeur d'activation de la couche calculée.
- a^0 correspond à la valeur d'activation de la couche précédente.
- $W \in M_{k,n}(\mathbb{R})$ représente l'ensemble des poids de connexions entre deux couches de neurones.
- b est le biais de chaque neurone.
- σ est la fonction d'activation.

Par la suite, afin d'indiquer au réseau dans quelle mesure il est éloigné du résultat souhaité, nous utiliserons une fonction coût, C, définie comme suit :

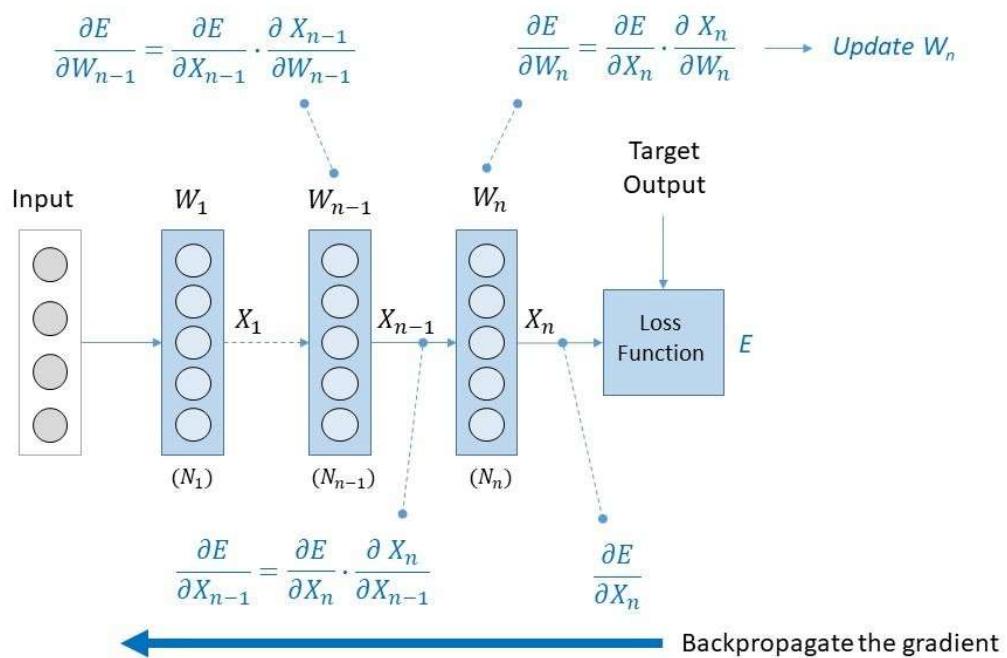
$$C = \sum_{i=0}^n (t_i - z_i)^2$$

Où :

- t_i représentent les valeurs de sorties des neurones
- z_i les valeurs attendues

La moyenne de cette fonction sur un grand nombre d'images, retiendra donc notre attention. Au lancement de l'apprentissage, on attribue des valeurs aléatoires pour chaque paramètre du réseau. Ensuite, ce dernier utilisera l'algorithme de descente du gradient en N dimensions (avec N, nombre total de poids et de biais modifiables) afin d'optimiser le choix des poids et des biais. Ce processus de modification en partant des outputs layer et remontant couches par couches, est intitulé **backpropagation**.

Figure 13 Schéma de Backpropagation



Où :

- $W \in M_{k,n}(\mathbb{R})$ représente l'ensemble des poids de connexions entre deux couches de neurones.
- $\frac{\delta E}{\delta W}$ est la variation de la fonction de loss au regard de W_k .
- μ le taux d'apprentissage (Learning rate).
- X_n ensemble des valeurs d'activation des neurones de la couche n.

D'autre part, afin d'éviter les problèmes liés à une stabilisation dans un minimum local, on utilise un momentum (terme d'inertie). Celui-ci permet de sortir des minimums locaux, dans la mesure du possible, et de poursuivre la descente du gradient. À chaque itération, le changement de poids conserve les informations des changements précédents. Cet effet de mémoire permet d'éviter les oscillations et accélère l'optimisation du réseau. La formule de Backpropagation devient :

$$\Delta W_k = -\alpha * \frac{\delta E}{\delta W_k} + \mu * \Delta W_{k-1}$$

Où :

- $W \in M_{k,n}(\mathbb{R})$ représente l'ensemble des poids de connexions entre deux couches de neurones.

- ΔW_k Gradient de W_k .

- $\frac{\delta E}{\delta W}$ est la variation de la fonction de loss au regard de W_k .

- μ le taux d'apprentissage (Learning rate).

- α le momentum, compris entre 0 et 1.

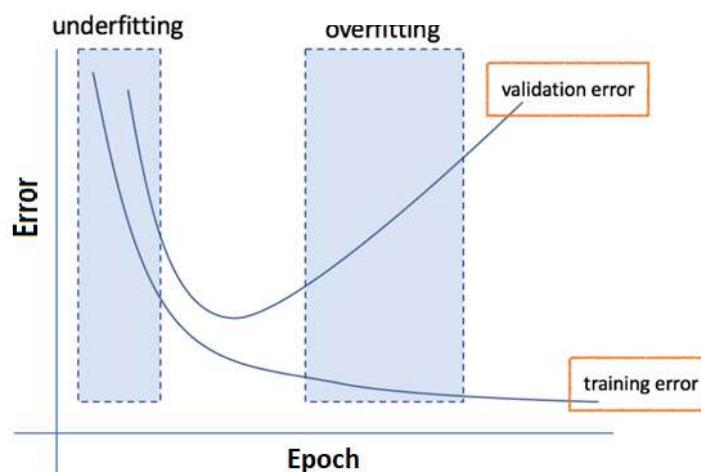
C'est donc grâce à la répétition de feed-forward et de backpropagation durant l'apprentissage que l'on obtient un réseau de neurone optimal pour le problème rencontré. Par ailleurs, plutôt que de répéter ce processus pour chaque input (un bloc de donnée correspondant à une image par exemple), ceux qui pourrait conduire à de nombreuses erreurs dans l'algorithme de descente du gradient, il est préférable de procéder par batch (lot) d'input.

Une Epoch est une unité de mesure de temps de l'apprentissage. C'est la période pendant laquelle le réseau répète les processus de feed-forward et de backpropagation une fois par batch présent dans le dataset.

Enfin on crée un couple de jeu de donnée Train/Validation. Ce couple permet une régularisation du réseau pour détecter l'underfitting (Le modèle manque d'apprentissage) ou l'overfitting (le modèle est trop spécialisé sur les données reçues et ne parvient pas à généraliser) (Figure 14).

De plus, il est souvent judicieux d'ajouter des couches dropout entre chaque couche de neurone. Le but est de faire oublier un petit pourcentage d'information à chaque Epoch, ce qui aidera à empêcher l'overfitting du model.

Figure 14 Evolution du loss



2-Prétraitement des Données

Notre objectif lors de notre préparation des données, a été de créer deux jeux de données distincts prêt à être utilisés, par nos réseaux de neurones.

Le premier permettant de reproduire le *Dream challenge 1*, qui consiste à savoir prédire chacun des descripteurs olfactifs perçus par un des 49 sujets en fonction de la molécule présenté.

Le second, permettant de reproduire le *Dream challenge 2*, qui consiste à savoir prédire la moyenne de chacun des descripteurs olfactifs perçus par les 49 sujets par molécules.

Pour ces 2 jeux de données, nous avons dans un premier temps créé un *Senteur Dataset*, un dataset regroupant les 21 descripteurs olfactifs que nous allons apprendre à prédire.

Puis, nous avons créé un *Molecular Dataset*, un dataset regroupant les données physico-chimiques qui vont nous permettre l'apprentissage, en fonction du *Senteur Dataset* associé.

Pour cela, nous partirons des .CSV de Gerkins, construit à partir des fichiers Txt des données mise en ligne pour la compétition.

Senteur Dataset Dream 1 :

Etape 0 :

Suivant les conseils de Monsieur FIORUCCI sur le fait que le train set et le LeaderBoard set, servant de validation test, ne couvraient pas les mêmes espaces physico-chimiques, ce qui a pour conséquence d'entrainer un sur-apprentissage sur ses régions ciblés dans le train, nous avons décidé de les regrouper en un seul dataset.

Etape 1 :

Sur les 39886 lignes de données que nous disposons, 10329 sont des ‘Nan’.

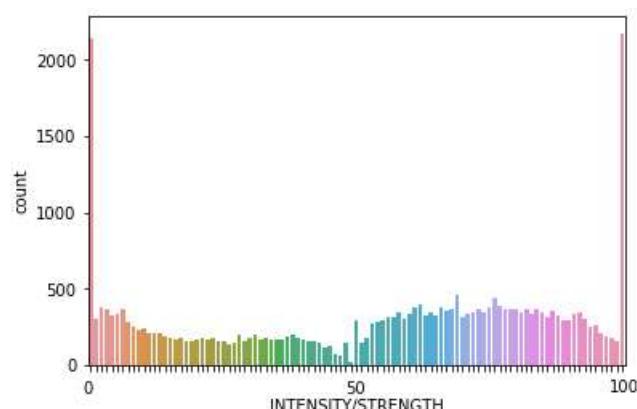
Nous avons décidé de supprimer ces lignes plutôt que de créer des données pour les remplacer.

Etape 2 :

Nous avons numérisé les colonnes Intensity (composé de string : ‘high’ et ‘low’) et Dilution (caractérisé par des string allant de « 1/10 » à « 1/10.000.000 ») afin de permettre par la suite, leur intégrations au *Molecular Dataset*.

Etape 3 :

Nous avons enfin labelisé les différents descripteurs olfactifs. En effet, leurs valeurs vont de 0 à 100, mais certaines valeurs ne sont pas assez représentées dans le dataset pour permettre un bon apprentissage. (Figure 1) Nous les avons donc regroupés de la façon suivante :



- La valeur 0 est identique à l'ancienne et marque l'absence totale de ressenti du descripteur.
- Les valeurs de 1 à 9 regroupent les anciennes valeurs par dizaine (de 70 à 79 =>8).
- Les valeurs de 90 à 99 sont regroupées avec la valeur 100 pour former la classe 10.

Figure 15 Répartition des données avant discréétisation

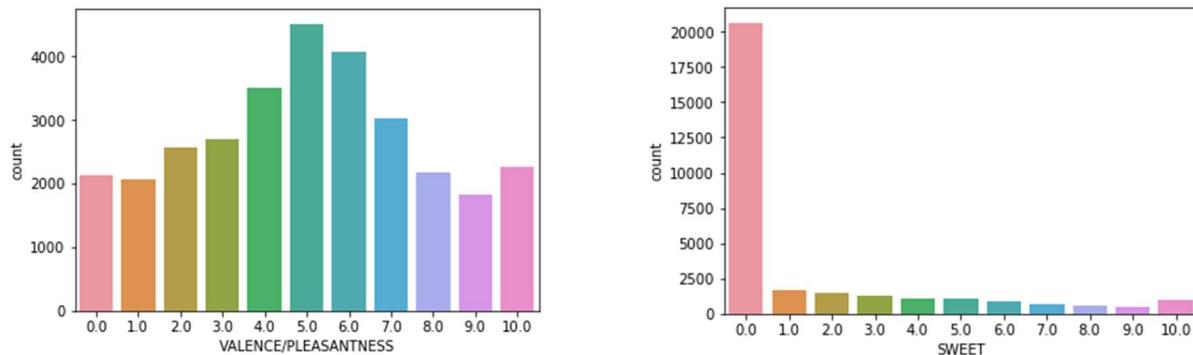
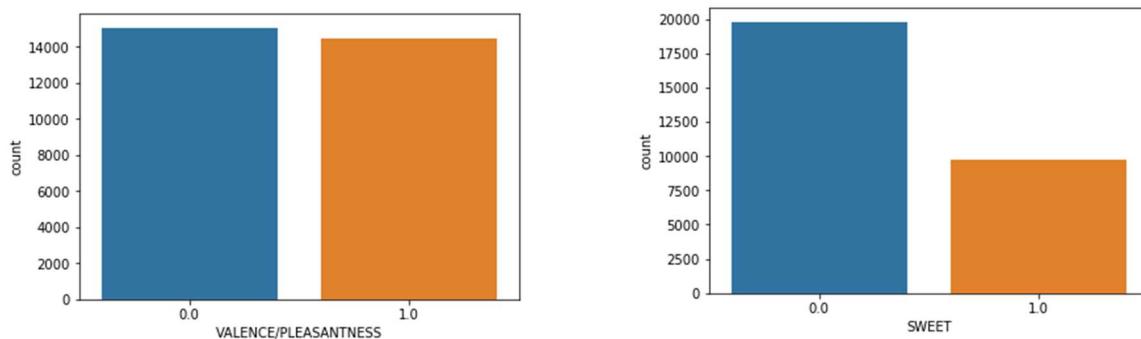


Figure 16 Répartition des données après discréétisation

Nous obtenons finalement 11 différentes classes à prédire pour chacun des 21 descripteurs olfactifs (Figure 16). Nous remarquons que les labels sont bien répartis pour *Intensity/Strength* et pour *Valence/Pleasantness*. Les 19 autres descripteurs comportent une majorité de 0 (Figure 16).

Senteur Dataset Binaire Dream 1 :

Figure 17 Répartition des données après une discréétisation binaire



Etape 0,1,2 :

Sa construction suit les mêmes étapes 0 et 1 de *Senteur Dataset Dream 1*

Etape 3 :

La labélisation se fait de la manière suivante (Figure 3) :

- La valeur 0 est identique à l'ancienne et marque l'absence totale de ressenti du descripteur.
- La valeur 1 pour toutes les autres valeurs.

Molecular Dataset Dream 1 :

Etape 0 :

Après analyse des données pour détecter la position des données manquantes ('NaN'), nous avons remarqué que l'intégralité de celle-ci était concentrée sur 5 molécules (807, 887, 962, 6505, et 8122). Nous les avons retirés du dataset. Les données liées à ces molécules dans le *Senteur Dataset Dream 1* ont aussi été supprimé.

Etape 1 :

Nous avons ensuite supprimé toutes les colonnes ne comportant que des 0, elles n'apportent aucune information intéressante pour l'apprentissage et alourdissent notre dataset, elles sont au nombre de 1790, soit près d'un tiers des données brutes.

Etape 2 :

Nous remplaçons ensuite chaque valeur par sa racine cubique, afin de permettre une meilleure normalisation. (Figure 18)

Etape 3 :

Molecular Dataset Dream 1 est alors composé de chacune des 407 molécules que contient *Senteur Dataset Dream 1* en autant d'occurrence que celles-ci apparaissent dans celui-ci.

Etape 4 :

On transfert les colonnes '*Dilution*', '*Intensity*' et '*Subject*' de *Senteur Dataset Dream 1* vers *Molecular Dataset Dream 1*.

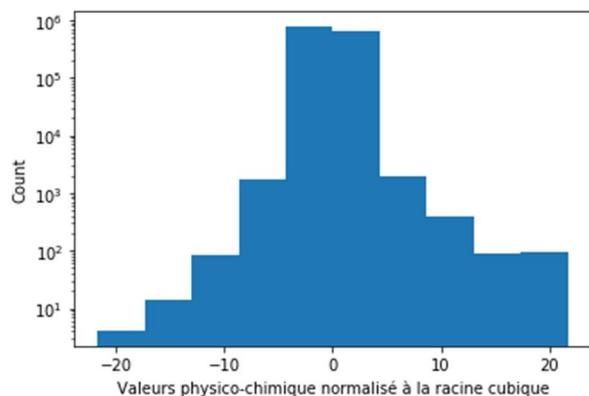


Figure 18 Décompte global des valeurs de Molecular Dataset Dream 1

Nous obtenons alors un dataset de 29339 lignes de 3083 colonnes chacune.

Pour prédire un dataset de 29339 lignes de 21 colonnes chacune.

Senteur Dataset Dream 2 :**Etape 0,1,2 :**

Pour construire ce dataset, nous suivons les mêmes étapes 0 et 1 que pour *Senteur Dataset Dream 1*.

Etape 3 :

-Nous avons ensuite calculé la moyenne de descripteurs olfactifs pour chaque molécule en fonction de la dilution. Chaque molécule possède alors deux valeurs par descripteur olfactif :

Sa moyenne pour la dilution 'High', et sa moyenne pour la dilution 'Low'.

Etape 4 :

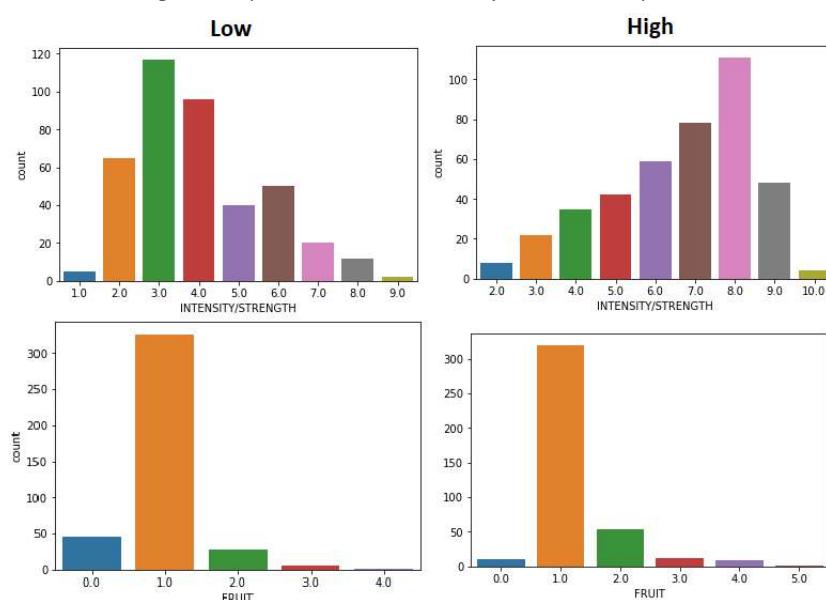
-Chaque moyenne étant une valeur unique, elles ne permettent pas un bon apprentissage.

Nous les avons labélisés selon les principes de l'étape 3 (Figure 19).

Etape 5 :

Les colonnes '*Intensity*' et '*Subject*' sont supprimées.

Figure 19 Répartition des classes des moyennes des descripteurs



Molecular Dataset Dream 2 :

Pour construire ce dataset, nous reproduisons les étapes 0,1,2 et 3 faites pour *Molecular Dataset Dream 1* en se basant sur *Senteur Dataset Dream 2* pour l'étape 3.

Etape 4 :

On transfert la colonne ‘*Dilution*’, de *Senteur Dataset Dream 1* vers *Molecular Dataset Dream 1*.

On obtient alors un dataset de 814 lignes de 3081 colonnes chacune.

Par la suite, à chaque chargement de nos dataset pour la mise en œuvre de nos réseaux, nous créons un *Training* et un *Validation Dataset* à partir de ces dernières. Pour cela, on utilise la fonction `train_test_split` de la librairie Sklearn, ce qui nous donne deux couples de dataset (*Data_train*, *Labels_train*) et (*Data_Validation*, *Labels_Validation*). Ensuite, nous normalisons les données moléculaires ceci permettant à notre model une bien meilleure flexibilité lors de la backpropagation. Pour se faire, on utilise la fonction `StandardScaler` de Sklearn, qui fait un calcul de normalisation adapté à des données matricielles.

Nous obtenons alors un dataset de 806 lignes de 3083 colonnes chacune.

Pour prédire un dataset de 806 lignes de 21 colonnes chacune.

3-Mise en place du modèle :

Nous décidons d'utiliser ici deux Framework différents pour la création de model :

- Pytorch : Prisé par le domaine académique, marche avec des Tensors personnalisés. Permet une bonne optimisation des expressions

- Keras : Plus utilisé en entreprise, marche en utilisant Tensorflow et avec bien plus utilité. Très pratique et facile d'utilisation et ayant un développement fiable et rapide.

Nos 2 modèles sont des réseaux de neurones *Fully-connected* comprenant :

- Une couche d'entrée composé de 3083/3081 neurones (En fonction du *Dream 1* ou *2*).

- Trois couches cachées de 2400,1200 et 600 neurones chacune.

 - On applique la fonction `RELU` et un `dropout` à 0.3 sur chacune de ces couches.

 - Une couche de sortie avec 11 neurones, un pour chaque classe à prédire.

 - On applique la fonction `LogSoftmax` à la couche de sortie.

Pour entraîner ces modèles, nous utilisons un batch à 300 ou 1000 pour le *Dream 1* et à 40 pour *Dream 2*. Les modèles sont entraînés sur 30 epoch.

Nous utilisons un dataset de validation contenant 20% des données.

Pour le calcul du loss du Pytorch, nous utilisons la fonction de *Negative Log Likelihood loss*.

Elle est définie comme : $l(x, y) = L = \{l_1, \dots, l_n\}^T, l_n = -w_{y_n} * x_{n,y_n}, w_c = weight[c]$

Où X sont les inputs, Y les output, w les poids et N la taille du batch.

Pour le calcul du gradient, nous utilisons la fonction `optim.SGD` de torch avec :

Un *Learning rate* est à 0.003 et un *Momentum* est à 0.9.

Pour le Keras, nous utilisons la fonction `Adam`.

Un *Learning rate* est à 0.001 et un *Momentum* est à 0.9.

Mise en œuvre des modèles NN :

Dream Challenge 1 :

En vue de ce challenge nous avons appliqué nos modèles dans différents notebooks :

Dream 1 NN : A partir des datasets *Dream 1*, on prédit le descripteur *Intensity/Strength* et nous étudions en profondeur les résultats.

NN Label : A partir des datasets *Dream 1*, on prédit chacun des descripteurs olfactifs, puis, nous calculons l'accuracy et le loss à 15 et 30 epoch.

NN Subject : A partir des datasets *Dream 1*, on prédit les descripteurs *Intensity/Strength*, *Valence/Pleasantness* et *SWEET* pour chaque sujet de manière indépendante. Puis nous calculons l'accuracy et le loss à 15 et 30 epoch.

Dream 1 Binaire NN : A partir des datasets Binaire *Dream 1*, on prédit le descripteur *Intensity/Strength* et nous étudions en profondeur les résultats.

NN Binaire Label : A partir des datasets Binaire *Dream 1*, on prédit chacun des descripteurs olfactifs, puis nous calculons l'accuracy et le loss à 15 et 30 epoch.

Dream Challenge 2 :

En vue de ce challenge nous avons appliqué nos modèles dans différents notebooks :

Dream 2 NN : A partir des datasets *Dream 2*, on prédit le descripteur *Intensity/Strength* et nous étudions en profondeur les résultats.

NN Label : A partir des datasets *Dream 2*, on prédit chacun des descripteurs olfactifs, puis nous calculons l'accuracy et le loss à 15 et 30 epoch.

III-Résultats et Discussion

A- Modèle Machine Learning

Dans un premier temps, nous allons discuter des résultats de Gerkin avec ses paramètres optimaux. Puis, nous discuterons de nos résultats obtenus en ayant fait varier certains paramètres. Avec les paramètres de Gerkin nous avons obtenu un score de 8.4 (sachant que son score maximal était de 8.5). Les résultats estampillés «mean» sont obtenus en comparant les prédictions avec les données imputées (NaN remplacé par la médiane), et celle nommé « sigma » sont les résultats où les prédictions sont comparées aux données masquées (donnée brute avec les NaN).

Les résultats sont les suivants : (Figure 20 Output du rfc.final)

```
Entrainement avec n_estimators = 900 sur train
For subchallenge 2:
```

```
Score = 8.24
int_mean = 1.000
int_sigma = 0.154
ple_mean = 0.655
ple_sigma = 0.282
dec_mean = 0.487
dec_sigma = 0.386
```

```
Training time: 1090.040082454681 seconds
```

```
Sur All
```

```
For subchallenge 2:
```

```
Score = 8.40
int_mean = 1.000
int_sigma = 0.145
ple_mean = 0.666
ple_sigma = 0.270
dec_mean = 0.496
dec_sigma = 0.406
```

```
Trainning on All time: 1164.442169189453 seconds
```

Le résultat de la corrélation entre la moyenne de l'*Intensité* des données prédite et la moyenne de l'*Intensité* des données imputées est biaisé car il utilise exactement le même dataset pour l'entraînement et la prédiction, Le modèle se contente de répéter ce qu'il a appris sur le dataset. Cette partie du modèle étant biaisé et ne nous apprend rien sur la capacité de prédiction du modèle de Gerkin concernant l'*Intensité*.

L'intensité de la liaison entre les données est dite forte si le coefficient de corrélation de Pearson se situe autour de 0.8, moyenne s'il est autour de 0.5 et faible, s'il est autour de 0.2. On remarque pour la *Valence* une corrélation de 0.6 pour les données imputées ce qui montre une relation modérée entre les données. En revanche, pour les données masquées, on se retrouve avec une corrélation de 0.25 ce qui est assez faible. On peut donc en déduire qu'on a une certaine difficulté à prédire l'attribut *Valence* des molécules.

Les résultats des 19 autres descripteurs sont similaires à ceux de la *Valence*. On obtient 0.48 pour la corrélation entre les données imputées et 0.39 pour les données masquées, ce qui implique une intensité de liaison modérée entre les données prédites et observées.

En gardant les mêmes paramètres que Gerkins mais en faisant varier le nombre d'arbres, nous remarquons que nous atteignons un plateau autour du score de 8.4. (Figure 21).
 Mais si nous faisons varier la profondeur de chaque arbre, nous remarquons que le plateau atteint est de plus en plus haut. (Figure 22)
 Nous atteignons un score de maximum de 9.20

Figure 21

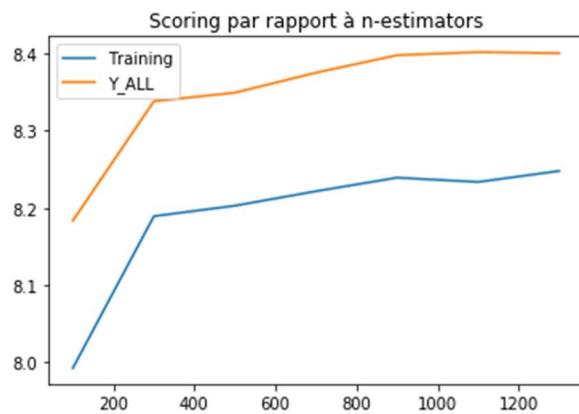
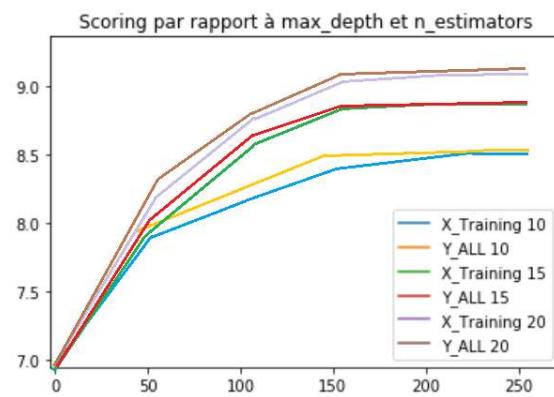


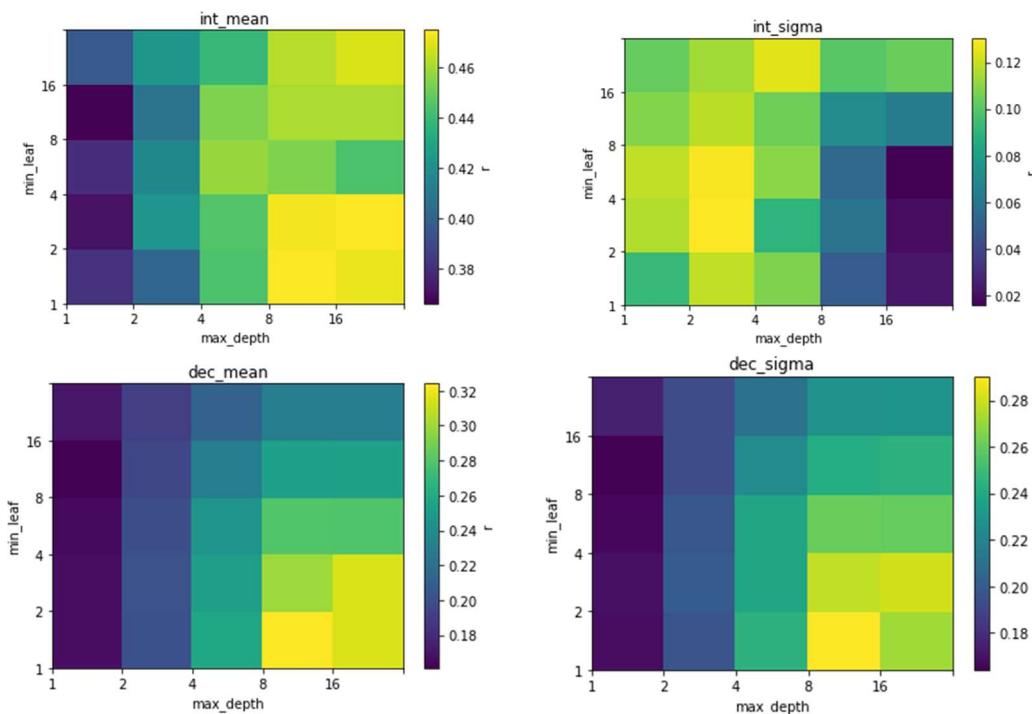
Figure 22



Conclusion:

En conclusion sur la partie de Gerkins, partant du principe que Gerkins utilisait ce qu'il définissait comme les paramètres optimum lors de ces recherches (Figure 23). Notre capacité d'augmenté le score vient alors de la puissance de calculs de nos ordinateurs (superieur à celle de 2015).

Figure 23 Discretisation du coef de Person en fonction du min leaf



B- Modèle Deep Learning

Challenge Dream 2 :

Commençons tout d'abord par les résultats pour le Dream challenge 2. On choisit dans un premier temps d'étudier l'un des deux descripteurs majeurs, l'Intensité (qui a des résultats similaires avec le descripteur Valence).

Voici ici les courbes d'accuracy et de loss pour le model Pytorch pour 100 epoch :

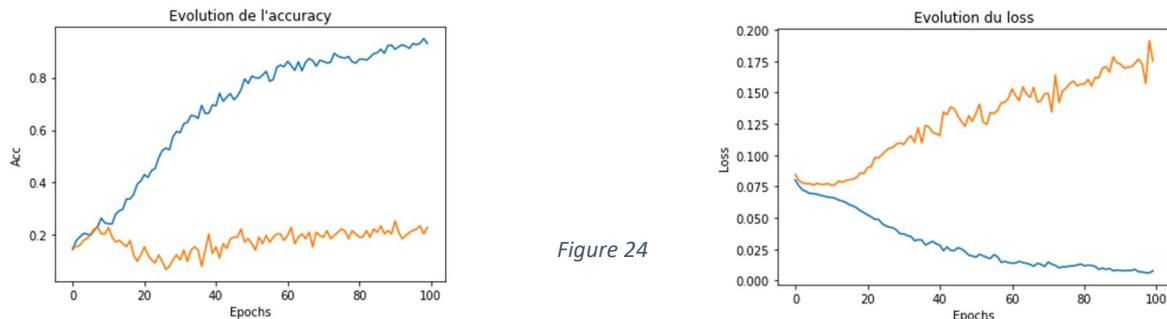


Figure 24

Le cas d'overfitting est flagrant, la valeur de validation reste très faible alors que sa loss continue de croître.

Regardons maintenant ces mêmes graphiques issus du modèle Keras pour 100 epoch et avec l'optimiseur SGD, celui-ci permet de lutter contre l'overfitting :

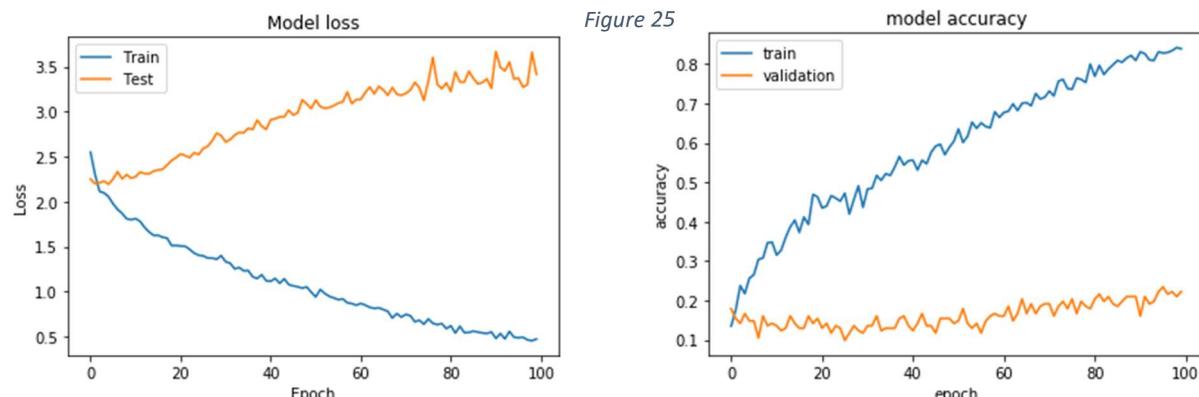
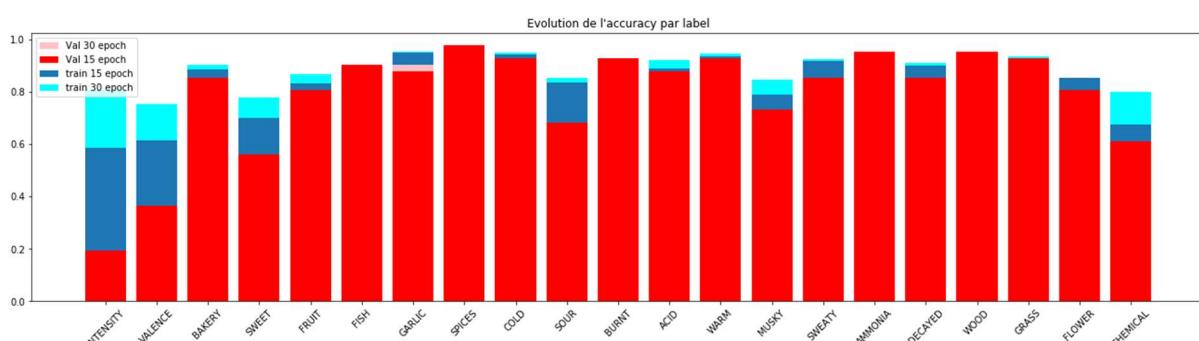


Figure 25

On obtient à peu près les mêmes résultats qu'avec Pytorch. On pose l'hypothèse que ces mauvais résultats sont causés principalement par une mauvaise représentation dans les données.

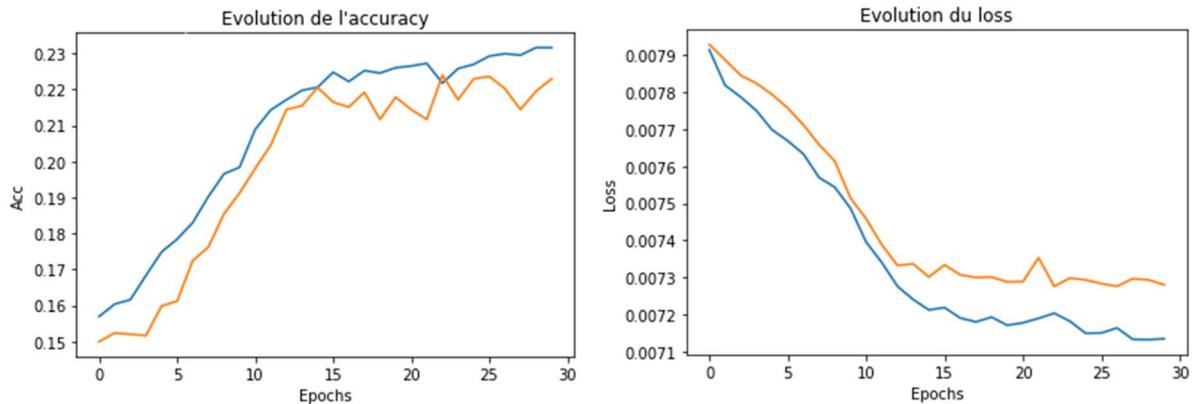
L'étude sur l'évolution de l'accuracy pour chacun des descripteurs olfactifs montre un overfitting. Sur Intensity et Valence. Les autres, ayant une majorité de 1, sont mieux prédit.

Figure 26

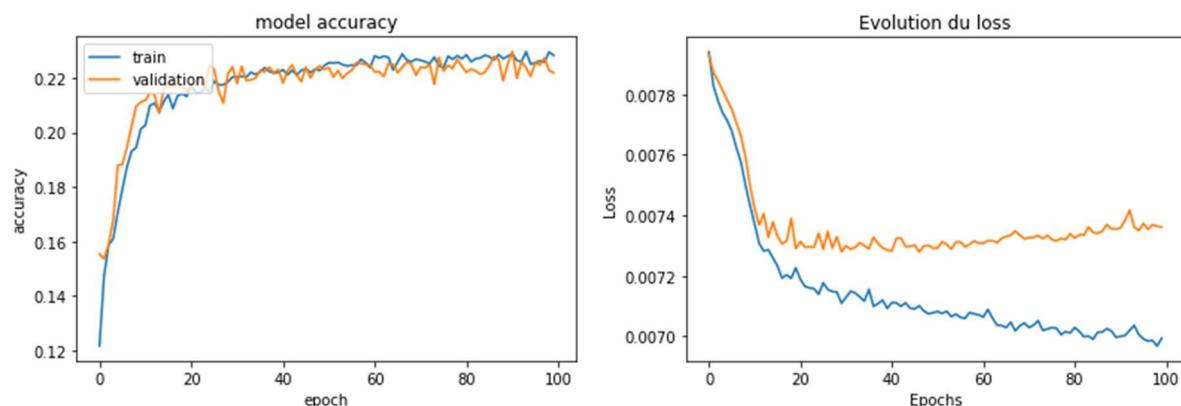


Passons maintenant aux résultats concernant le Dream Challenge 1. Encore une fois on choisit en un premier temps d'étudier l'un des deux descripteurs majeurs, *l'Intensité*.

Figure 27 courbes d'accuracy et de loss pour le model Pytorch pour 30 epoch et 100 epoch



On tend ici vers 0.22 de validation et il nous semble apercevoir un début d'overfitting. Ainsi pour s'en assurer, nous refaisons l'apprentissage de notre réseau sur 100 epoch :



Ici l'overfitting est flagrant ainsi que le maximum de validation majorer par 0.22.

Etudions maintenant les résultats avec le modèle Keras pour 100 epoch :

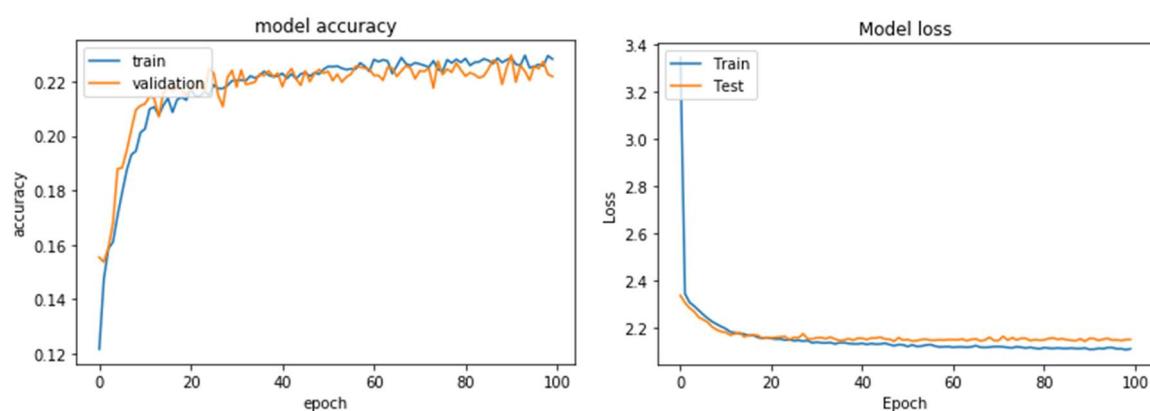
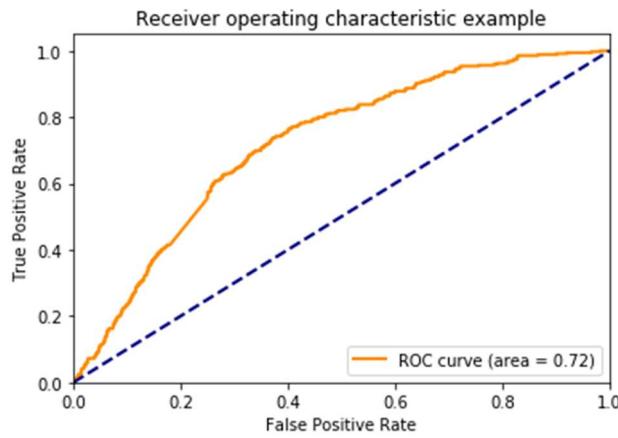


Figure 28

Ici pas d'overfitting mais globalement la même performance. Le modèle Keras prend plus de temps de processing mais permet de prendre un batch beaucoup plus grand (ici batch = 1000 vs 300 pour Pytorch), ce qui égalise leurs temps d'entraînement.

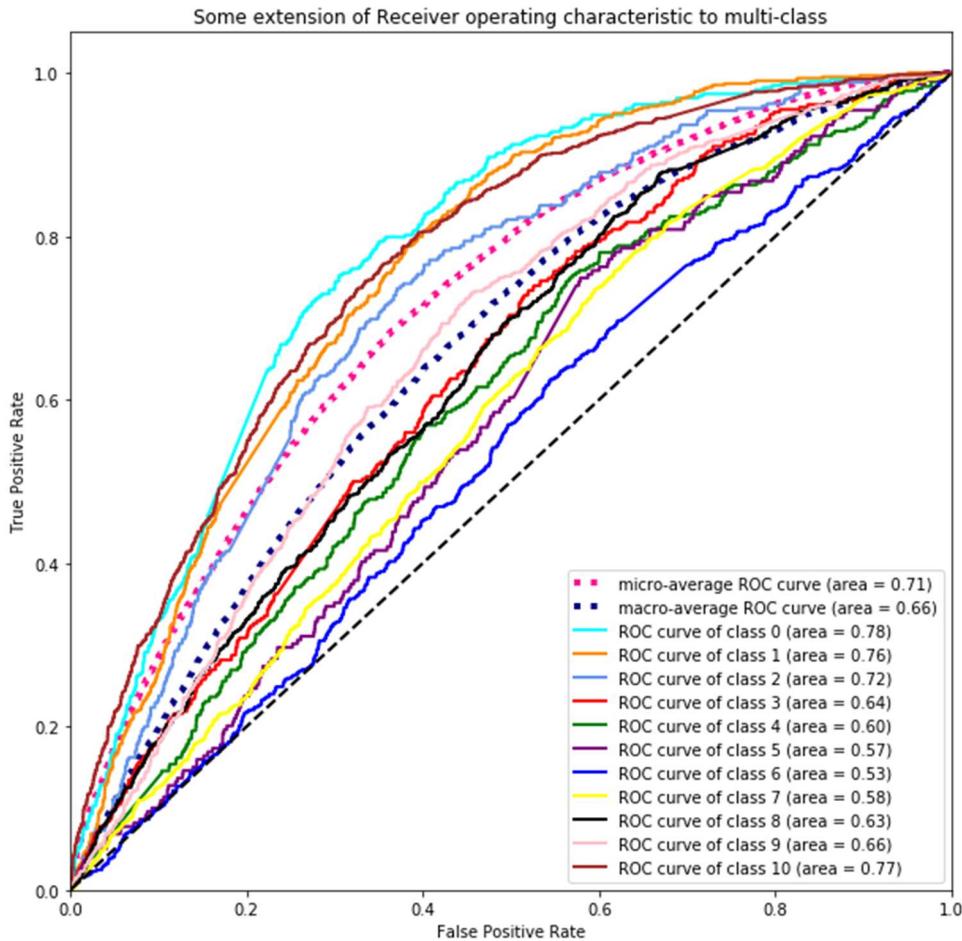
Nous affichons maintenant la courbe ROC, d'abord en moyenne sur toutes les classes sans tenir compte d'éventuelle surreprésentation de l'une d'elle :

Figure 29

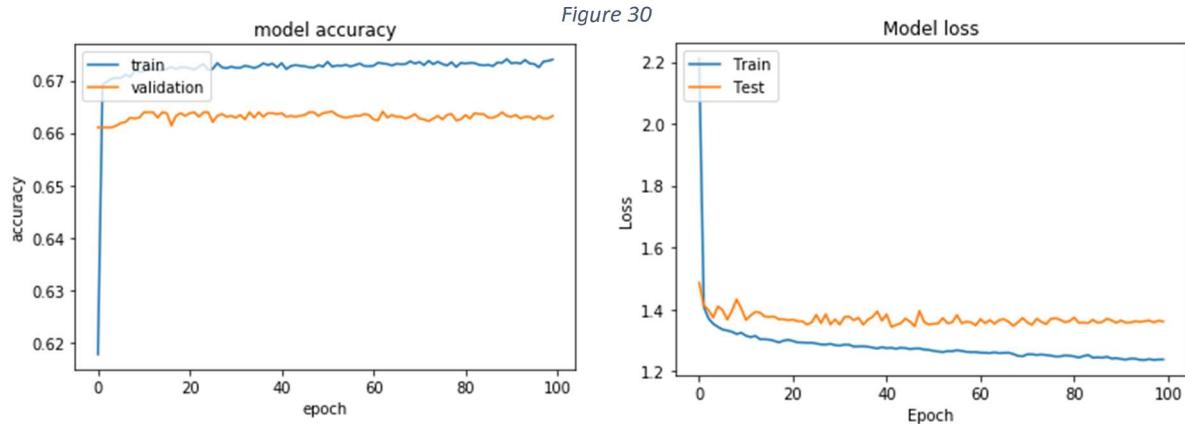


Ensuite, nous utilisons une extrapolation sur les données pour calculer une courbe « macro », mieux représentatrice des classes cumulées :

Figure 30



En comparaisons on regarde les résultats pour un descripteur mineur ici Sweet avec le modèle Keras avec 100 epochs:



Nous obtenons ici une accuracy bien plus importante, entre 0.66 et 0.67 pour la validation, et pas ou peu d'overfitting. Ce résultat, (similaire avec tous les descripteurs mineurs), est assez intrigant.

Nous décidons donc de regarder et comparer les courbes ROC :

Voici la courbe brute sans extrapolation données-classes et la courbe multi-classe (Figure 31 et 32).

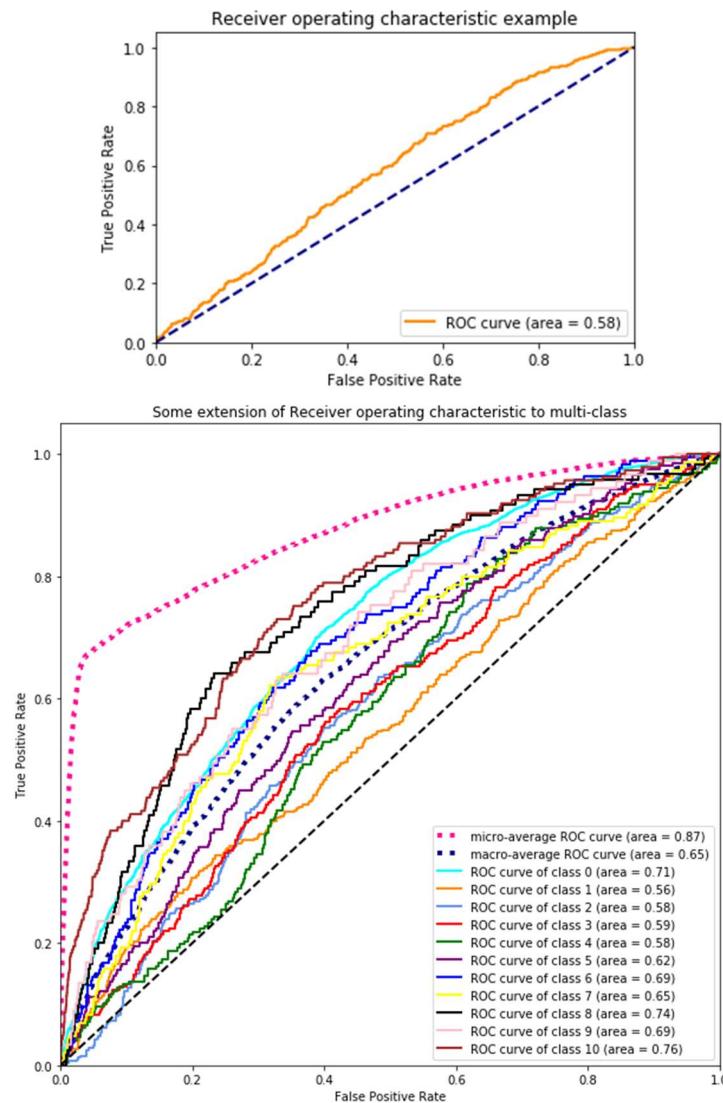


Figure 31

On voit ici que malgré un résultat bien plus satisfaisant au niveau de l'accuracy pour le descripteur Sweet, les courbes ROC montre un résultat bien plus intéressant pour l'Intensité.

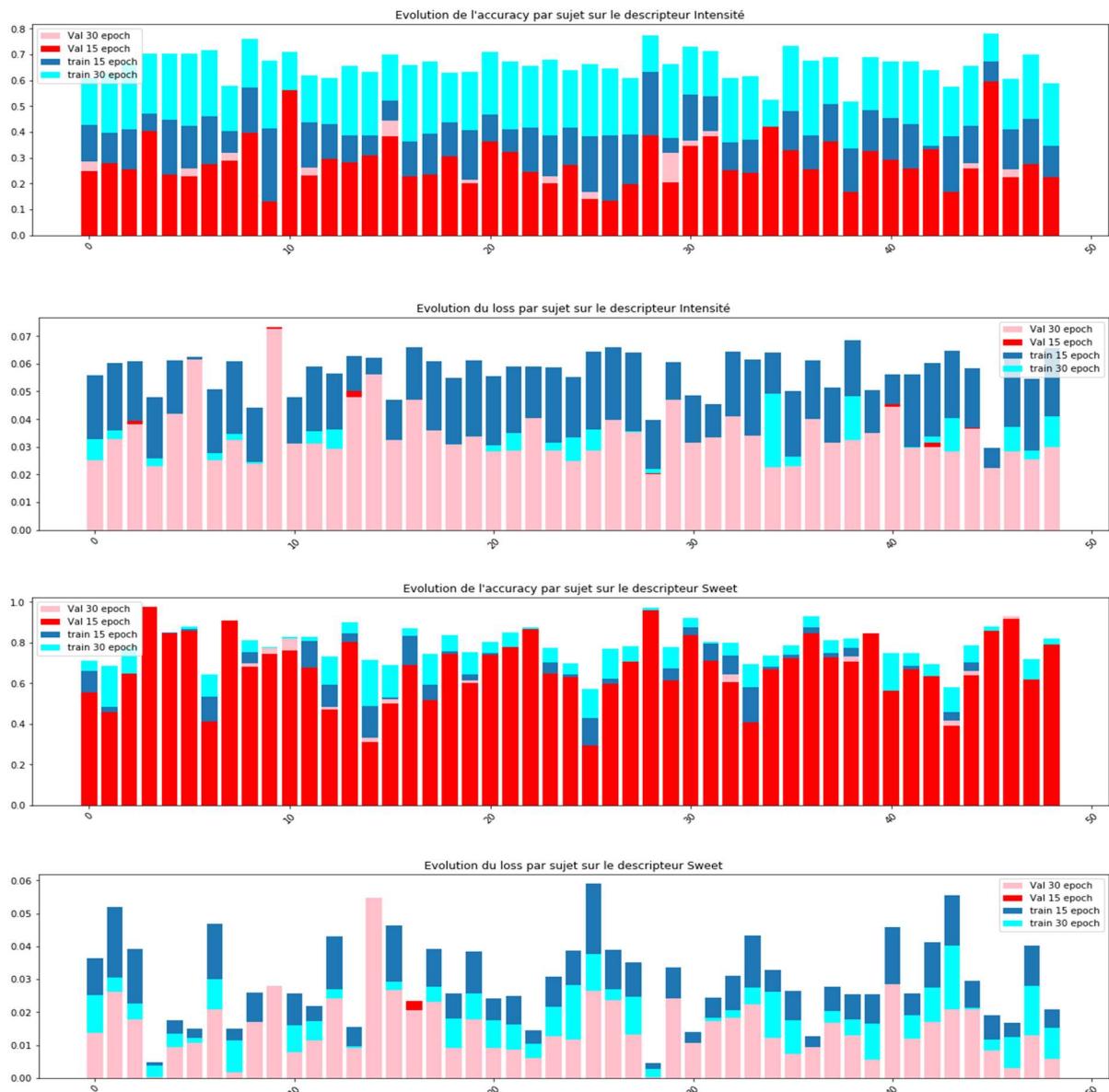
Encore une fois une étude sur les données est nécessaire pour expliquer ce phénomène.

En étudiant la capacité de nos modèles à prédire les descripteurs de manière indépendante pour chaque sujet, nous remarquons que l'apprentissage n'est pas homogène. En effet, certains sujets sont plus faciles à prédire que d'autre, que ce soit sur les descripteurs Intensity et Valence (Figure) ou sur les 19 autres descripteurs (Figure).

En faisant apparaître la meilleure accuracy sur 15 et 30 epoch, ainsi que la meilleure loss, nous remarquons que le modèle est en overfitting sur 30 epoch voir 15 pour un certain nombre de sujets sur les descripteurs Intensity et Valence. En effet, la différence entre la meilleure loss sur 15 epoch et celle sur 30 est nulle car la loss s'est mise à augmenter à partir de la 15epoch. Cela est dû au faible nombre de données par sujet (entre 500 et 814).

En revanche, l'overfitting se fait sentir qu'à partir de 30epoch sur les autres descripteurs.

Figure 31 Evolution de l'accuracy et du loss sur 15 et 30 epoch par sujet pour Intensity et Sweet



Etudions maintenant la capacité de nos modèles à prédire chacun des descripteurs de manière indépendante. Nous remarquons également que l'apprentissage n'est également pas homogène. Les descripteurs Intensity et Valence (Figure) sont bien plus durs à prédire que les 19 autres descripteurs. Nous remarquons également que le modèle n'est pas en overfitting au bout de 30 epoch. Cependant, par la présence majoritaire de la classe 0 dans ces 19 descripteurs, on peut donc considérer que nos modèles sont capables de détecter, non pas la force de la présence de ces descripteurs mais plutôt, l'absence de celui-ci.

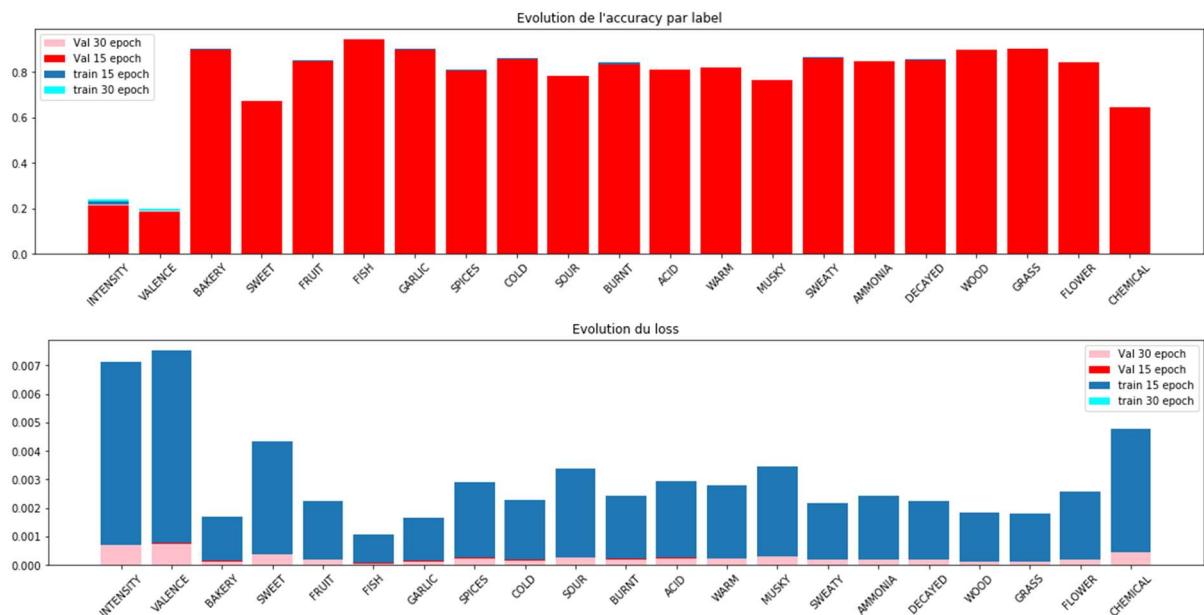
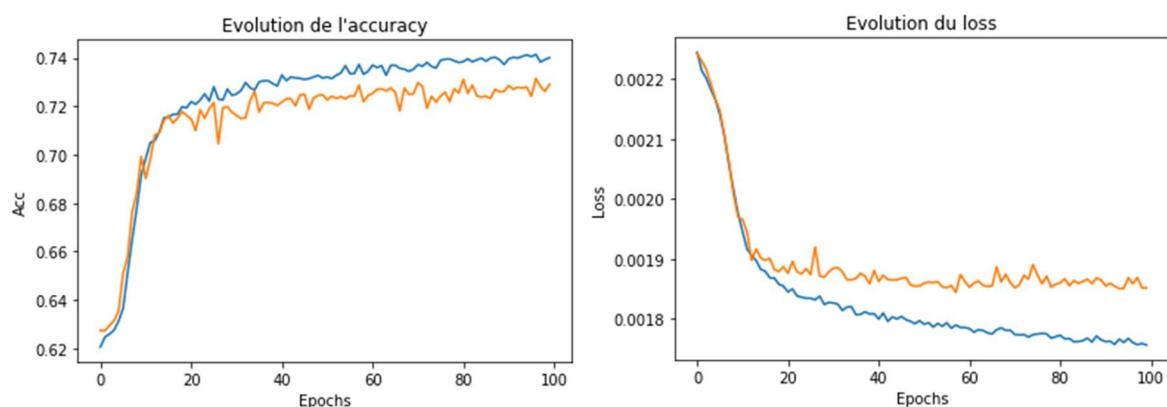


Figure 32 Barplot de l'accuracy et du loss par descripteurs

En ce sens, nous allons étudier le *Senteur Dataset Binaire Dream 1*.

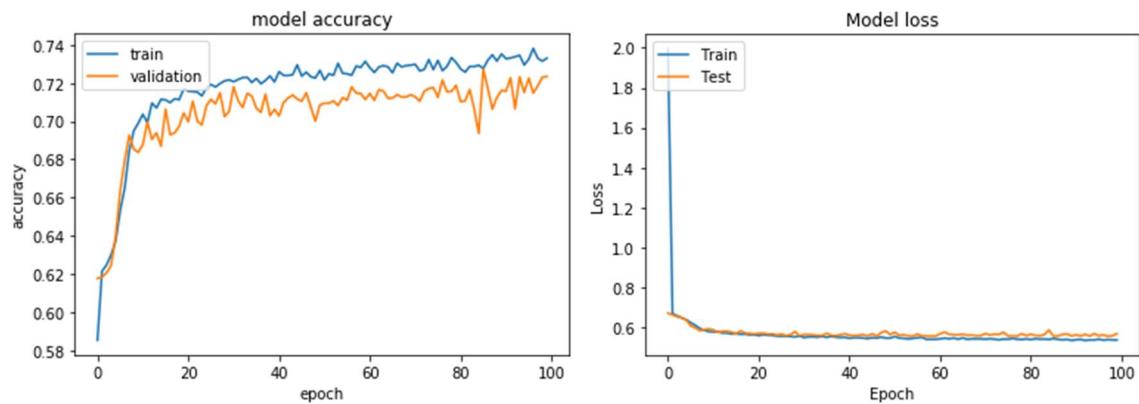
Etudions donc le cas de classification binaire. Nous commençons encore une fois avec le descripteur Intensité.

Regardons l'accuracy et la loss du modèle Pytorch pour 100 epoch :



Nous obtenons ici de bien meilleurs résultats avec pas (ou presque pas) d'overfitting ce qui nous convient parfaitement.

Maintenant avec le modèle Keras toujours sur 100 epoch :



Les résultats sont similaires, nous arrivons à un maximum d'environ 0.72 pour la validation ce qui est un résultat honorable avec très peu de loss.

Ce type de classification nous permet certainement de compenser partiellement les problèmes issus des données fournies telle que l'imprécision des sujets et la surreprésentation de certaine composante.

IV- Conclusion

Dans un premier temps, nous avons bien réussi à comprendre l'ensemble du code de Gerkins, composé de nombreux notebooks, nous avons réussi à les mettre à jour, en effet de nombreuses fonctions utilisées, ont été déprécié au fil des mise à jour.

Puis nous avons réussi à améliorer leur score.

Dans un second temps, nous avons mis en place des modèles de réseaux neuronaux en toute somme basique mais correctes pour le travail de classification de données linéaires mise notre dispositions, seulement nos résultats n'ont pas été à la hauteur de nos attentes. Le plus gros problème affectant nos prédictions vient des datasets fournie par le challenge où les données sont beaucoup trop imprécises pour être bien digérées par nos modèles et permettre un apprentissage optimal.

D'autre parts, nous n'avons pas intégré la solution évidente de la Data Augmentation qui consiste à utiliser diverses techniques pour créer de nouvelles données viables à partir de celles que nous possédons car même si cela aurait sûrement pu améliorer nos performances, cette technique est trop approximative d'un point de vu physico-chimique (comme l'imputer de Gerkins).

En outre le passage en classification binaire nous permet d'avoir de bien meilleurs résultats même si la prédiction en elle-même est moins intéressante.

Un autre type de réseau neuronal que l'on aurait pu tester pour ce genre de classification sur des données moléculaires est le *Graph Neural Network GNN*. Ce modèle prend en entré des données graphiques tel que la représentation de Cram de molécules par exemple, il effectue ensuite des transformations sur ces données similairement au techniques CNN de manière à prédire des représentations graphiques. Or, notre manque d'expérience avec ce genre de modèle et le fait que nous n'ayons pas de données moléculaires sous forme graphique nous a détourné de cette voie. Il nous semble judicieux d'étudier cette voie afin de continuer nos travaux.

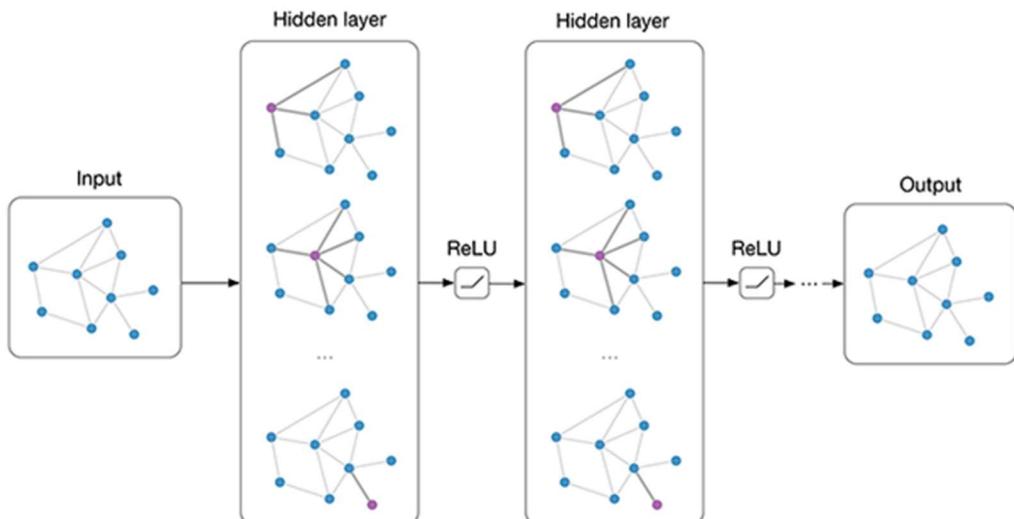


Figure 35 GNN

Nos remerciements à Dr. S Fiorucci pour son aide et son soutien durant toute la durée de ce projet.

V- Références

Présentation de Monsieur Fiorucci

https://science.sciencemag.org/content/sci/suppl/2017/02/17/science.aal2014.DC1/Keller_SM.pdf

<https://www.synapse.org/#!Synapse:syn2811262/wiki/78368>

<https://www.synapse.org/#!Synapse:syn2811262/wiki/78387>

<https://keras.io/>

<https://pytorch.org/docs/stable/index.html>

<https://www.quora.com/What-are-Graph-Neural-Networks-GNN-in-AI-and-Machine-Learning-ML>

<https://stackoverflow.com/>

<https://www.dlogy.com/blog/simple-guide-on-how-to-generate-roc-plot-for-keras-classifier/>

<https://scikit-learn.org/stable/index.html>

<https://tel.archives-ouvertes.fr/tel-02010618>

Preparation des datasets Dream 1

February 12, 2021

```
[1]: # Import numerical libraries.  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sn  
from sklearn.preprocessing import StandardScaler  
  
#Librairies  
import time  
import torch  
import random  
  
#Raccourcis utilisé  
import torch.nn as nn  
import torch.utils.data as data_utils  
import pandas as pd
```

0.1 Senteur Dataset Dream 1

```
[2]: df1 = pd.read_csv('LeaderboardSet2.csv',sep=';')  
df1=df1.drop('0dor',axis=1)  
List=[]  
for i in range(len(df1)):  
    if df1.iat[i,1] == 'replicate ':  
        List.append(i)  
for i in range(len(List)):  
    df1 = df1.drop(List[i])  
print("Nombre de ligne repliqué que l'on supprime:", len(List))  
del List  
df1=df1.drop('Replicate',axis=1)  
df1 = df1.dropna()
```

Nombre de ligne repliqué que l'on supprime: 0

```
[3]: dft = pd.read_csv('TrainSet.csv',sep=';')  
dft=dft.drop('0dor',axis=1)  
List=[]  
for i in range(len(dft)):  
    if dft.iat[i,1] == 'replicate ':
```

```

        List.append(i)
for i in range(len(List)):
    dft = dft.drop(List[i])
print("Nombre de ligne repliqué que l'on supprime:", len(List))
del List
dft=dft.drop('Replicate',axis=1)
dft = dft.dropna()

```

Nombre de ligne repliqué que l'on supprime: 1960

[4]:

```

df= pd.concat([dft, dfl])
df = df.sort_values(by = ['Compound Identifier','subject'])
del dft
del dfl

```

[5]:

```

# Numérisation des colonne Intensity et Dilution
#Colonne Intenisty
for i in range(len(df)):
    if df.iat[i,1] == 'low ':
        df.iat[i,1] = 0
    if df.iat[i,1] == 'high ':
        df.iat[i,1] = 1
    if df.iat[i,1] == 'low':
        df.iat[i,1] = 0
    if df.iat[i,1] == 'high':
        df.iat[i,1] = 1

```

[6]:

```

#Colonne Dilution
for i in range(len(df)):
    if df.iat[i,2] == '1/10':
        df.iat[i,2] = 0
    if df.iat[i,2] == '1/1000':
        df.iat[i,2] = 1
    if df.iat[i,2] == '1/1,000':
        df.iat[i,2] = 1
    if df.iat[i,2] == '1/100000':
        df.iat[i,2] = 2
    if df.iat[i,2] == '1/100,000':
        df.iat[i,2] = 2
    if df.iat[i,2] == '1/10,000,000':
        df.iat[i,2] = 3

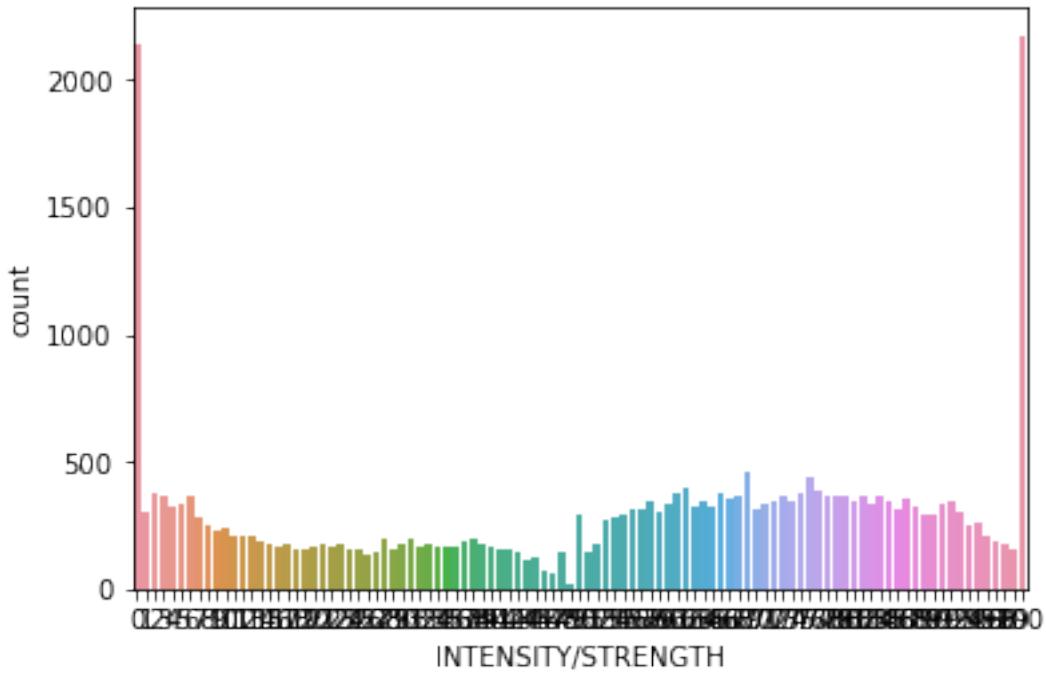
```

[7]:

```
sn.countplot(df['INTENSITY/STRENGTH'])
```

[7]:

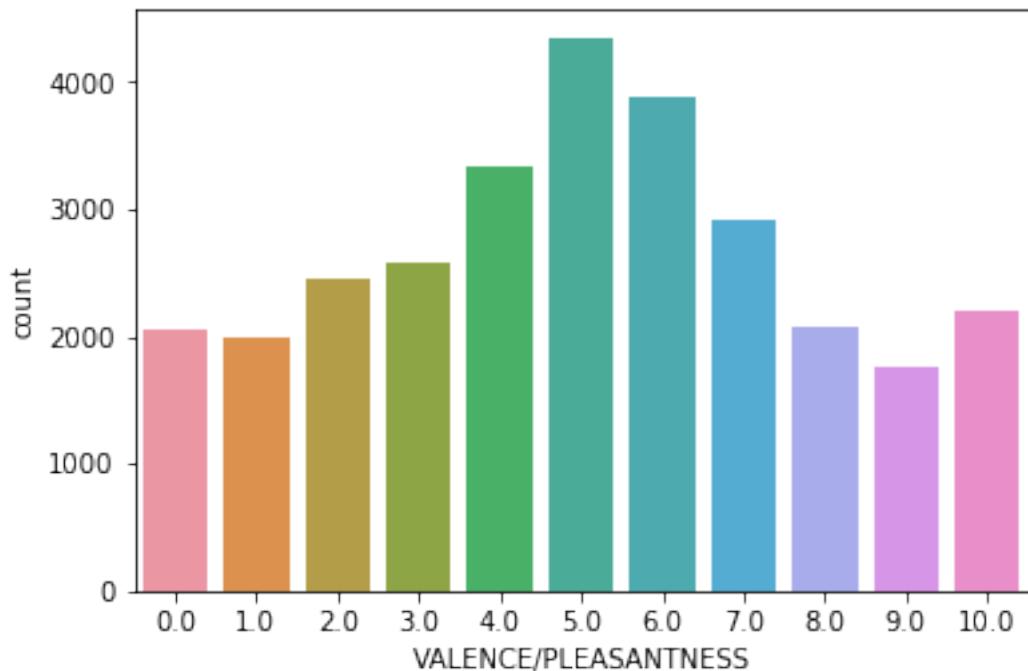
```
<matplotlib.axes._subplots.AxesSubplot at 0x1d81e769308>
```



```
[8]: # Labelisation des Valeurs à prédire
#Colonnes sentation Olfactive
for i in range(len(df)):
    for j in range (21):
        if df.iat[i,j+4] == 0:
            df.iat[i,j+4] = 0
        elif df.iat[i,j+4] == 100:
            df.iat[i,j+4] = 10
        else :
            df.iat[i,j+4] = (df.iat[i,j+4] // 10) +1
```

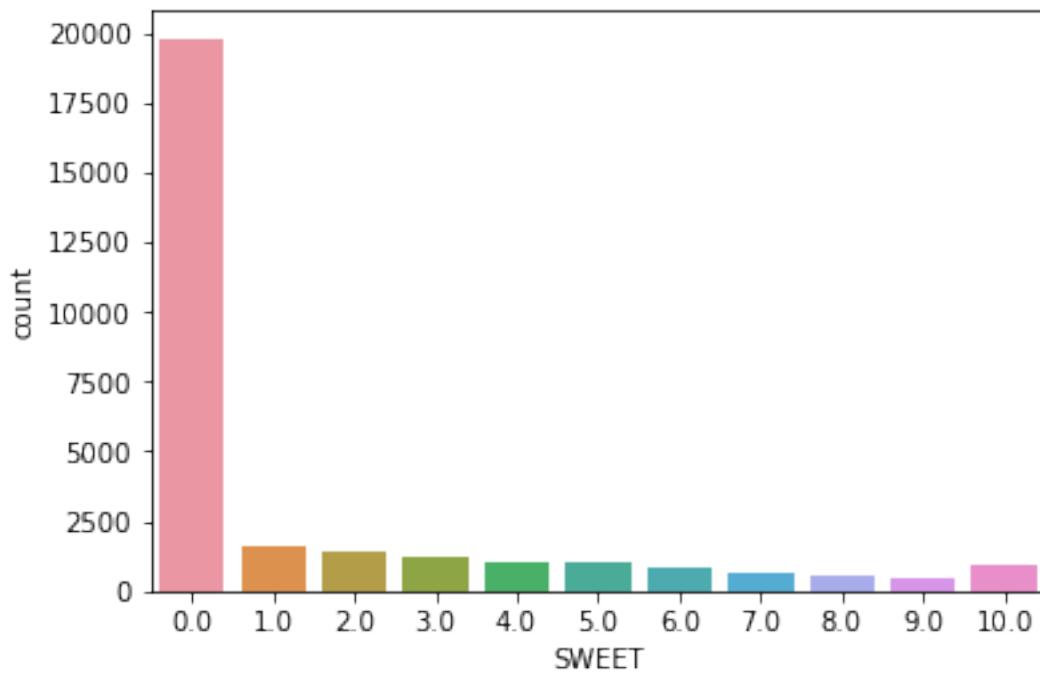
```
[9]: sn.countplot(df['VALENCE/PLEASANTNESS'])
```

```
[9]: <matplotlib.axes._subplots.AxesSubplot at 0x1d8218482c8>
```



```
[10]: sn.countplot(df['SWEET'])
```

```
[10]: <matplotlib.axes._subplots.AxesSubplot at 0x1d821911788>
```



```
[12]: df.to_csv('Senteur Dataset Binaire Dream 1.csv',sep=';')
df = pd.read_csv('Senteur Dataset Binaire Dream 1.csv',sep=';')
df = df.drop('Unnamed: 0',axis=1)
df.to_csv('Senteur Dataset Binaire Dream 1.csv',sep=';')
```

```
[13]: df.to_csv('Senteur Dataset Dream 1.csv',sep=';')
df = pd.read_csv('Senteur Dataset Dream 1.csv',sep=';')
df = df.drop('Unnamed: 0',axis=1)
df.to_csv('Senteur Dataset Dream 1.csv',sep=';')
```

0.2 Molecular Dataset Dream 1

```
[14]: mdf = pd.read_csv('molecular_descriptors_data2.csv',sep=';')
mdf = mdf.dropna()
print(" On drop les Molécules 807, 887, 962, 6505, et 8122.",
      "La 962 est présente dans le dataframe test.")
print("Ces molécules présente de nombreux NaN dans leurs données moléculaires.")
```

On drop les Molécules 807, 887, 962, 6505, et 8122. La 962 est présente dans le dataframe test.

Ces molécules présente de nombreux NaN dans leurs données moléculaires.

```
[15]: mdf.to_csv('Mol_descriptor.csv',sep=';')
mdf = pd.read_csv('Mol_descriptor.csv',sep=';')
mdf = mdf.drop('Unnamed: 0',axis=1)
mdf.to_csv('Mol_descriptor.csv',sep=';')
```

```
[17]: print("On supprime les données olfactives liées à ces molécules:")
#Cmdf = mdf[['CID']]
Cmdf = pd.DataFrame(mdf[['CID']],columns = ['CID'])
List= []
for i in range (len(df)):
    a=0
    for j in range (len(Cmdf)):
        if Cmdf.iat[j,0] == df.iat[i,0]:
            a+=1
        if a == 0:
            List.append(i)
print('Nombre de ligne concernée:', len(List))
for i in range(len(List)):
    df = df.drop(List[i])
```

On supprime les données olfactives liées à ces molécules:

Nombre de ligne concernée: 218

```
[18]: Liste=[]
Xcolumns= mdf.columns
for j in range (len(Xcolumns)):
    a=0
    for i in range (len(mdf)):
        if mdf.iat[i,j] != 0 :
            a+=1
    if a == 0:
        Liste.append(j)
print('Nombre de colonne sans importance:',len(Liste))
print('On les supprime.')
for j in range(len(Liste)):
    mdf.drop([Xcolumns[Liste[j]]], axis = 1, inplace = True)
```

Nombre de colonne sans importance: 1790

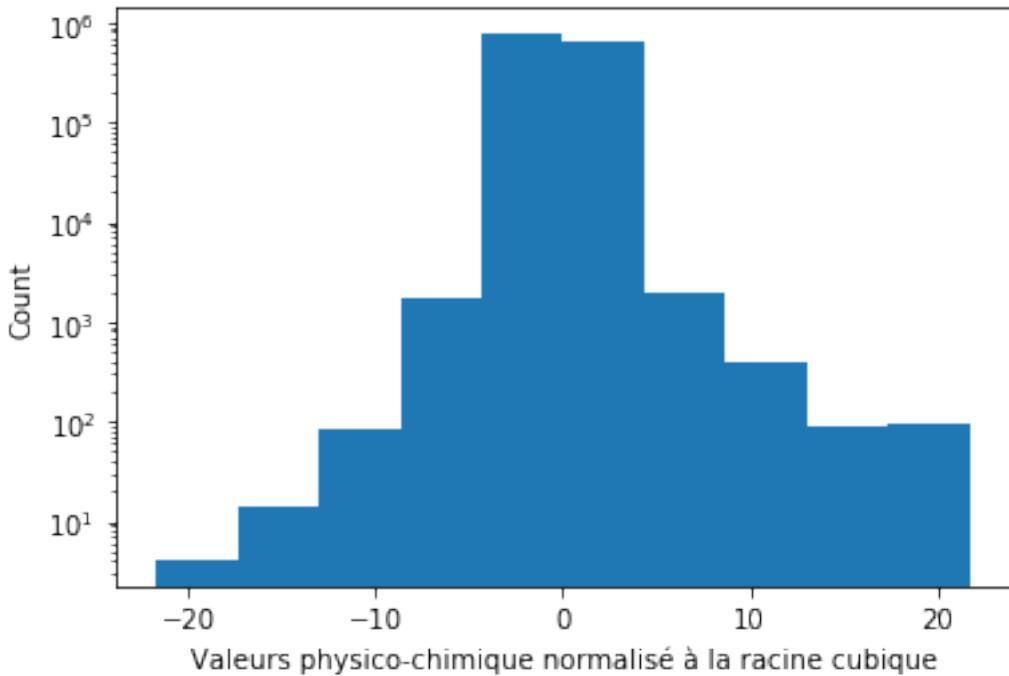
On les supprime.

```
[19]: #Pour ne pas modifier la colonne CID, on navigue sur les colonnes de 1 à 3117
#en ajoutant +1 à j qui va de 0 à 3116.
#Code pour la racine cubique, permet une meilleure normalisation
for j in range (3079):
    for i in range(len(mdf)):
        if mdf.iat[i,j+1] < 0:
            mdf.iat[i,j+1] = ((mdf.iat[i,j+1]**(2))**(1/2))**(1/3)
            mdf.iat[i,j+1] = (mdf.iat[i,j+1])*1000//1/1000
            mdf.iat[i,j+1] *= (-1)
        else :
            mdf.iat[i,j+1] = ((mdf.iat[i,j+1]**(2))**(1/2))**(1/3)
            mdf.iat[i,j+1] = (mdf.iat[i,j+1])*1000//1/1000
```

```
[20]: dataX = np.float32(mdf.values)
Xcolumns = mdf.columns
sc = StandardScaler()
dataX = sc.fit_transform(dataX)
```

```
[21]: plt.hist(dataX.ravel())
plt.yscale('log')
plt.ylabel('Count')
plt.xlabel('Valeurs physico-chimique normalisé à la racine cubique')
plt.figure()
```

[21]: <Figure size 432x288 with 0 Axes>



<Figure size 432x288 with 0 Axes>

```
[22]: # Alignement des index des deux dataframes
df.to_csv('Senteur Dataset Dream 1.csv',sep=';')
df = pd.read_csv('Senteur Dataset Dream 1.csv',sep=';')
df = df.drop('Unnamed: 0',axis=1)
df.to_csv('Senteur Dataset Dream 1.csv',sep=';')

[23]: mdf.to_csv('Mol_descriptor.csv',sep=';')
mdf = pd.read_csv('Mol_descriptor.csv',sep=';')
mdf = mdf.drop('Unnamed: 0',axis=1)
mdf.to_csv('Mol_descriptor.csv',sep=';')
```

Le but de ce dataset est d'aligner autant de lignes de descripteurs moléculaires qu'il y a de ligne de sensation olfactive perçus. Soit 30812.

On traite la création de ces données molécules par molécules.

```
[ ]: Cdf = df[['Compound Identifier']]
colo = mdf.columns
tdf= pd.DataFrame(columns = colo)
for j in range(len(mdf)):
    inter = pd.DataFrame(columns = colo)
    for i in range(len(Cdf)):
        if mdf.iat[j,0] == Cdf.iat[i,0] :
```

```

b= mdf.iloc[j:j+1]
inter = pd.concat([inter, b])
tdf = pd.concat([tdf, inter])
print('Chargement: ',j+1,'/471;           Molécule:',mdf.iat[j,0],'
↪   Nombre de ligne:',len(inter))

```

Enregistrement des Dataset créés

[26]:

```

df.to_csv('Senteur Dataset Dream 1.csv',sep=';')
mdf.to_csv('Mol_descriptor.csv',sep=';')
tdf.to_csv('brouillon.csv',sep=';')

```

[27]:

```

tdf = pd.read_csv('brouillon.csv',sep=';')
tdf = tdf.drop('Unnamed: 0',axis=1)
tdf.to_csv('Molecular Dataset Dream 1.csv',sep=';')

```

DREAM_2_CLEAN

February 12, 2021

1 Préparation des données

Faire un tableau de best prediction de tous les descripteurs par rapport a celle bien predite par le random

```
[1]: # Import numerical libraries.
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import seaborn as sns
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder, MultiLabelBinarizer
from sklearn.metrics import accuracy_score, roc_curve, auc, roc_auc_score

#Librairies
import time
import torch
import random

#Raccourcis utilisé
from tqdm import tqdm_notebook
import torch.nn as nn
import torch.utils.data as data_utils
import pandas as pd

import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D
from tensorflow.keras.optimizers import SGD, RMSprop, Adam, Adadelta, Adamax, Nadam
from sklearn.model_selection import train_test_split
from math import exp
from itertools import cycle
```

Using TensorFlow backend.

```
[2]: dfX = pd.read_csv('Molecular Dataset Dream 2.csv',sep=';')
dfY = pd.read_csv('Senteur Dataset Dream 2.csv',sep=';')
```

```
[3]: df = dfX.merge(dfY)
df
```

```
[3]:      Unnamed: 0      CID  complexity from pubmed      MW      AMW      Sv      Se \
0            0        126          4.532  4.961  2.011  2.155  2.482
1            1        126          4.532  4.961  2.011  2.155  2.482
2            2        176          3.141  3.916  1.958  1.648  2.034
3            3        176          3.141  3.916  1.958  1.648  2.034
4            4        177          2.175  3.531  1.846  1.556  1.921
...
801         801    6429333          5.336  5.268  1.909  2.372  2.747
802         802    6999977          4.657  5.268  1.826  2.341  2.891
803         803    6999977          4.657  5.268  1.826  2.341  2.891
804         804  16220109          5.867  5.670  1.804  2.579  3.128
805         805  16220109          5.867  5.670  1.804  2.579  3.128

      Sp      Si      Mv ...   ACID   WARM   MUSKY   SWEATY  AMMONIA/URINOUS \
0   2.168   2.554  0.873 ...   1.0    1.0    1.0     1.0           1.0
1   2.168   2.554  0.873 ...   1.0    1.0    1.0     1.0           1.0
2   1.642   2.099  0.824 ...   1.0    1.0    1.0     1.0           1.0
3   1.642   2.099  0.824 ...   1.0    1.0    1.0     1.0           1.0
4   1.584   2.003  0.813 ...   1.0    1.0    1.0     1.0           1.0
...
801   2.425   2.855  0.859 ...   0.0    1.0    1.0     1.0           1.0
802   2.392   3.019  0.811 ...   1.0    1.0    1.0     1.0           1.0
803   2.392   3.019  0.811 ...   1.0    2.0    2.0     1.0           1.0
804   2.657   3.275  0.821 ...   1.0    1.0    1.0     1.0           1.0
805   2.657   3.275  0.821 ...   1.0    1.0    1.0     1.0           1.0

      DECAYED   WOOD   GRASS  FLOWER  CHEMICAL
0            1.0    1.0    1.0    1.0     2.0
1            1.0    1.0    1.0    1.0     2.0
2            1.0    1.0    1.0    1.0     2.0
3            1.0    1.0    1.0    1.0     1.0
4            1.0    1.0    1.0    1.0     1.0
...
801           1.0    1.0    1.0    1.0     2.0
802           1.0    1.0    1.0    1.0     2.0
803           2.0    1.0    1.0    1.0     2.0
804           1.0    1.0    1.0    1.0     2.0
805           1.0    1.0    3.0    2.0     1.0
```

[806 rows x 3104 columns]

On remarque que la colonne ‘Unnamed 0’ qui sert à marier les 2 dataframe est complètement

fauisser. On se retrouve donc à prédire la classe ‘INTENSITY/STRENGTH’ avec les mauvais descripteurs.

```
[4]: list_labelY = ['INTENSITY/STRENGTH']
list_label = ['INTENSITY/STRENGTH', 'VALENCE/PLEASANTNESS', 'BAKERY', 'SWEET',
    'FRUIT', 'FISH', 'GARLIC', 'SPICES', 'COLD', 'SOUR', 'BURNT', 'ACID',
    'WARM', 'MUSKY', 'SWEATY', 'AMMONIA/URINOUS', 'DECAYED', 'WOOD',
    'GRASS', 'FLOWER', 'CHEMICAL']
```

```
[5]: dfY = df[list_labelY]
dfX = df
dfX = dfX.drop(list_label, axis=1)
dfX = dfX.drop('Unnamed: 0', axis=1)
dfX = dfX.drop('CID', axis=1)
```

```
[6]: dfX
```

	complexity from pubmed	MW	AMW	Sv	Se	Sp	Si	Mv	\
0	4.532	4.961	2.011	2.155	2.482	2.168	2.554	0.873	
1	4.532	4.961	2.011	2.155	2.482	2.168	2.554	0.873	
2	3.141	3.916	1.958	1.648	2.034	1.642	2.099	0.824	
3	3.141	3.916	1.958	1.648	2.034	1.642	2.099	0.824	
4	2.175	3.531	1.846	1.556	1.921	1.584	2.003	0.813	
..	
801	5.336	5.268	1.909	2.372	2.747	2.425	2.855	0.859	
802	4.657	5.268	1.826	2.341	2.891	2.392	3.019	0.811	
803	4.657	5.268	1.826	2.341	2.891	2.392	3.019	0.811	
804	5.867	5.670	1.804	2.579	3.128	2.657	3.275	0.821	
805	5.867	5.670	1.804	2.579	3.128	2.657	3.275	0.821	
	Me	Mp	...	Hypertens-80	Hypertens-50	Hypnotic-80	Hypnotic-50	\	
0	1.006	0.879	...	0	0	0	0		
1	1.006	0.879	...	0	0	0	0		
2	1.017	0.821	...	0	0	0	0		
3	1.017	0.821	...	0	0	0	0		
4	1.004	0.828	...	0	0	0	0		
..		
801	0.995	0.878	...	0	0	0	0		
802	1.002	0.829	...	0	0	0	0		
803	1.002	0.829	...	0	0	0	0		
804	0.995	0.845	...	0	0	1	0		
805	0.995	0.845	...	0	0	1	0		
	Neoplastic-80	Neoplastic-50		Infective-80	Infective-50	\			
0	0	0		0	0				
1	0	0		0	0				
2	0	0		0	0				
3	0	0		0	0				

```

4          0          0          0          0
..
801        0          0          1          ...
802        0          0          0          0
803        0          0          0          0
804        1          0          1          0
805        1          0          1          0

      Compound Identifier  Intensity
0                  126          0
1                  126          1
2                  176          0
3                  176          1
4                  177          0
..
801        ...          ...
802        6429333         0
803        6999977         1
804        6999977         0
805        16220109        0
805        16220109        1

[806 rows x 3081 columns]

```

1.1 Test sur le réseau neuronal

```
[7]: def to_class(y_):
    res = np.zeros((len(y_), 11), dtype='q')
    for i in range(len(y_)):
        res[i][int(y_[i])] = 1
    return res
```

```
[8]: dataX = np.float32(dfX.values)
dataY = np.longlong(dfY.values)
dataY = to_class(dataY)
```

```
[9]: # Normalisation
sc = StandardScaler()
dataX = sc.fit_transform(dataX)
```

```
[10]: DATA_Train, DATA_Test, TARGET_Train, TARGET_Test = train_test_split(dataX, ↴
    dataY ,test_size=0.2)
```

```
[11]: ## PyTorch is used to working with batches.
Batch=30

X_train_tensor = torch.from_numpy(DATA_Train)
Y_train_tensor = torch.from_numpy(TARGET_Train)
```

```

X_test_tensor = torch.from_numpy(DATA_Test)
Y_test_tensor = torch.from_numpy(TARGET_Test)

train = data_utils.TensorDataset(X_train_tensor, Y_train_tensor)
train_loader = data_utils.DataLoader(train, batch_size=Batch, shuffle=True)

test = data_utils.TensorDataset(X_test_tensor, Y_test_tensor)
test_loader = data_utils.DataLoader(test, batch_size=Batch, shuffle=True)

```

```

[12]: class MyNetwork(nn.Module):

    def __init__(self):
        super(MyNetwork, self).__init__()

        ## Activation layer
        self.relu = nn.ReLU()

        self.fc1 = nn.Linear(in_features = 3081, out_features = 2400)
        self.fc2 = nn.Linear(2400, 1200)
        self.fc3 = nn.Linear(1200, 600)
        self.output = nn.Linear(600, 11)
        self.softmax = nn.LogSoftmax(dim=1)
        self.dr = nn.Dropout(0.3)

    def forward(self, x):

        ## First full connection
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dr(x)

        ## Second full connection
        x = self.fc2(x)
        x = self.relu(x)
        x = self.dr(x)

        ## Third full connection
        x = self.fc3(x)
        x = self.relu(x)
        x = self.dr(x)

        ## Output layer
        x = self.output(x)
        y = self.softmax(x)

    return y

```

```
[13]: X_train_tensor.shape
```

```
[13]: torch.Size([644, 3081])
```

```
[14]: ## Create an instance of our network  
net = MyNetwork()
```

```
[15]: net = MyNetwork()  
net = net.cuda()
```

```
LEARNING_RATE = 0.003
```

```
MOMENTUM = 0.9
```

```
criterion = nn.NLLLoss()
```

```
#criterion = nn.CrossEntropyLoss
```

```
# Méthode stochastique de descente du gradient
```

```
optimizer = torch.optim.SGD(net.parameters(), lr=LEARNING_RATE, momentum=MOMENTUM)
```

```
[16]: ## Nombre d'époque d'apprentissage
```

```
N_EPOCHS = 100
```

```
epoch_loss, epoch_acc, epoch_val_loss, epoch_val_acc = [], [], [], []  
start_time = time.time()
```

```
for e in range(N_EPOCHS):
```

```
    print("EPOCH:", e)
```

```
    ### boucle d'entraînement
```

```
    running_loss = 0
```

```
    running_accuracy = 0
```

```
    running_acc=0
```

```
    start_epoch_time=time.time()
```

```
    ## Le réseau est mis en mode "entraînement"  
    net.train()
```

```
    for i, batch in enumerate(tqdm_notebook(train_loader)):
```

```
        # Obtenir batch du dataloader
```

```
        x = batch[0]
```

```
        labels = batch[1]
```

```
        # déplacer le batch sur le GPU
```

```
        x = x.cuda()
```

```

labels = labels.cuda()

# Calcul de l'output et les loss
output = net(x)
y = output

loss = criterion(y, torch.max(labels, 1)[1])

# Réinitialisation du gradients
optimizer.zero_grad()

# Calculs du gradients
loss.backward()

# Appliquecation d'une étape d'optimisation de l'algorithme de descente
pour mettre à jour les poids
optimizer.step()

with torch.no_grad():
    running_loss += loss.item()
    running_accuracy += (y.max(1)[1] == torch.max(labels, 1)[1]).sum().
item()

print("Training accuracy:", running_accuracy/float(len(train)),
      "Training loss:", running_loss/float(len(train)))

epoch_loss.append(running_loss/len(train))
epoch_acc.append(running_accuracy/len(train))

### Boucle de validation
## Le réseau est mis en mode validation
net.eval()

running_val_loss = 0
running_val_accuracy = 0

for i, batch in enumerate(tqdm_notebook(test_loader)):

    with torch.no_grad():

        x = batch[0]
        labels = batch[1]

        x = x.cuda()
        labels = labels.cuda()

        output = net(x)

```

```

y = output

loss = criterion(y, torch.max(labels, 1)[1])

running_val_loss += loss.item()
running_val_accuracy += (y.max(1)[1] == torch.max(labels, 1)[1]).  

↪sum().item()

print("Validation accuracy:", running_val_accuracy/float(len(test)),  

      "Validation loss:", running_val_loss/float(len(test)))

epoch_val_loss.append(running_val_loss/len(test))
epoch_val_acc.append(running_val_accuracy/len(test))

inter = time.time() - start_epoch_time
print ('Temps de 1 Epoch ',e,' en secondes:', inter )

interval = time.time() - start_time
print ('Temps total en secondes:', interval )

```

EPOCH: 0

```

C:\Users\benja\anaconda3\lib\site-packages\ipykernel_launcher.py:20:
TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.14440993788819875 Training loss: 0.08028652838298253

```

C:\Users\benja\anaconda3\lib\site-packages\ipykernel_launcher.py:62:
TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`

HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.15432098765432098 Validation loss: 0.08447247670020586

Temps de 1 Epoch 0 en secondes: 0.8953862190246582

EPOCH: 1

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.17857142857142858 Training loss: 0.07557500630432035

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.15432098765432098 Validation loss: 0.07970355616675483
Temps de l Epoch 1 en secondes: 0.1906599998474121
EPOCH: 2
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.19254658385093168 Training loss: 0.07237764119361498
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.16049382716049382 Validation loss: 0.07817324738443633
Temps de l Epoch 2 en secondes: 0.19002962112426758
EPOCH: 3
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.20496894409937888 Training loss: 0.07103519113907902
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.17901234567901234 Validation loss: 0.07730475619987205
Temps de l Epoch 3 en secondes: 0.18261003494262695
EPOCH: 4
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.20341614906832298 Training loss: 0.06955264703087184
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.18518518518518517 Validation loss: 0.07740315095877942
Temps de l Epoch 4 en secondes: 0.18449687957763672
EPOCH: 5
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.19875776397515527 Training loss: 0.06936420009743353
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.2037037037037 Validation loss: 0.0761353932780984
```

```
Temps de 1 Epoch 5 en secondes: 0.1914825439453125
```

```
EPOCH: 6
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.21739130434782608 Training loss: 0.06889997024713836
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.2222222222222222 Validation loss: 0.07762155047169438
```

```
Temps de 1 Epoch 6 en secondes: 0.20445680618286133
```

```
EPOCH: 7
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.2329192546583851 Training loss: 0.06793796960611521
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.22839506172839505 Validation loss: 0.07664171560311023
```

```
Temps de 1 Epoch 7 en secondes: 0.19346332550048828
```

```
EPOCH: 8
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.2639751552795031 Training loss: 0.06740961552406691
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.2037037037037 Validation loss: 0.07668631385873866
```

```
Temps de 1 Epoch 8 en secondes: 0.19646048545837402
```

```
EPOCH: 9
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.2453416149068323 Training loss: 0.06655264437568854
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.2037037037037 Validation loss: 0.07748384828920718
```

```
Temps de l Epoch 9 en secondes: 0.1850736141204834
```

```
EPOCH: 10
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.2422360248447205 Training loss: 0.06633467130039049
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.22839506172839505 Validation loss: 0.07587774079522969
```

```
Temps de l Epoch 10 en secondes: 0.19743776321411133
```

```
EPOCH: 11
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.2422360248447205 Training loss: 0.06574801628634055
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.19135802469135801 Validation loss: 0.07655269054718959
```

```
Temps de l Epoch 11 en secondes: 0.19289875030517578
```

```
EPOCH: 12
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.2795031055900621 Training loss: 0.06424796174031606
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.1728395061728395 Validation loss: 0.07949617394694576
```

```
Temps de l Epoch 12 en secondes: 0.1952824592590332
```

```
EPOCH: 13
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.2919254658385093 Training loss: 0.06344143759389842
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.17901234567901234 Validation loss: 0.07843730331938943

Temps de l Epoch 13 en secondes: 0.1918783187866211

EPOCH: 14

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.2981366459627329 Training loss: 0.06237299064671771

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.1666666666666666 Validation loss: 0.07963046615506396

Temps de l Epoch 14 en secondes: 0.19245505332946777

EPOCH: 15

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.33695652173913043 Training loss: 0.06043868349946063

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.15432098765432098 Validation loss: 0.08051541558018437

Temps de l Epoch 15 en secondes: 0.19140267372131348

EPOCH: 16

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.33695652173913043 Training loss: 0.05958941560354292

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.17901234567901234 Validation loss: 0.08088621166017321

Temps de l Epoch 16 en secondes: 0.1874713897705078

EPOCH: 17

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.35403726708074534 Training loss: 0.057930728293353724

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.12345679012345678 Validation loss: 0.08231356482446929
```

```
Temps de l Epoch 17 en secondes: 0.20000362396240234
```

```
EPOCH: 18
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.39285714285714285 Training loss: 0.055713740379913994
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.09876543209876543 Validation loss: 0.08597808414035374
```

```
Temps de l Epoch 18 en secondes: 0.20245075225830078
```

```
EPOCH: 19
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.4052795031055901 Training loss: 0.054189244227379745
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.12345679012345678 Validation loss: 0.08536184128419852
```

```
Temps de l Epoch 19 en secondes: 0.1880178451538086
```

```
EPOCH: 20
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.4301242236024845 Training loss: 0.05239875401769366
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.15432098765432098 Validation loss: 0.0902779985357214
```

```
Temps de l Epoch 20 en secondes: 0.19402050971984863
```

```
EPOCH: 21
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.4192546583850932 Training loss: 0.05092754848995564
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.12345679012345678 Validation loss: 0.09063500975385125
```

```
Temps de 1 Epoch 21 en secondes: 0.20448040962219238
```

```
EPOCH: 22
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.44565217391304346 Training loss: 0.048846815498719304
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.10493827160493827 Validation loss: 0.09807062149047852
```

```
Temps de 1 Epoch 22 en secondes: 0.1904618740081787
```

```
EPOCH: 23
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.453416149068323 Training loss: 0.048912881509117455
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.09259259259259259 Validation loss: 0.09782316655288507
```

```
Temps de 1 Epoch 23 en secondes: 0.20445632934570312
```

```
EPOCH: 24
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.4937888198757764 Training loss: 0.0455918628606737
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.12345679012345678 Validation loss: 0.10028687082690957
```

```
Temps de 1 Epoch 24 en secondes: 0.19846773147583008
```

```
EPOCH: 25
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.5217391304347826 Training loss: 0.043723444575848786
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.09876543209876543 Validation loss: 0.10343518669222608

Temps de 1 Epoch 25 en secondes: 0.2025127410888672

EPOCH: 26

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.531055900621118 Training loss: 0.04267998946749646

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.06790123456790123 Validation loss: 0.10565804846492814

Temps de 1 Epoch 26 en secondes: 0.19544696807861328

EPOCH: 27

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.5248447204968945 Training loss: 0.04215720305161447

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.08024691358024691 Validation loss: 0.10612001683976915

Temps de 1 Epoch 27 en secondes: 0.2014293670654297

EPOCH: 28

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.5745341614906833 Training loss: 0.040325227638949517

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.10493827160493827 Validation loss: 0.1089421069180524

Temps de 1 Epoch 28 en secondes: 0.19472575187683105

EPOCH: 29

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.59472049689441 Training loss: 0.03732438496551158

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.12345679012345678 Validation loss: 0.10991624108067265
```

```
Temps de 1 Epoch 29 en secondes: 0.2050025463104248
```

```
EPOCH: 30
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.5900621118012422 Training loss: 0.037271205127609446
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.10493827160493827 Validation loss: 0.10851413085136885
```

```
Temps de 1 Epoch 30 en secondes: 0.19451141357421875
```

```
EPOCH: 31
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.6242236024844721 Training loss: 0.03595300268682634
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.1419753086419753 Validation loss: 0.11265228707113384
```

```
Temps de 1 Epoch 31 en secondes: 0.18946146965026855
```

```
EPOCH: 32
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.6288819875776398 Training loss: 0.035160943680668466
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.09876543209876543 Validation loss: 0.11556092603707019
```

```
Temps de 1 Epoch 32 en secondes: 0.1984710693359375
```

```
EPOCH: 33
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.65527950310559 Training loss: 0.03179180890506839
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.1419753086419753 Validation loss: 0.11026973636062057
```

```
Temps de 1 Epoch 33 en secondes: 0.19006752967834473
```

```
EPOCH: 34
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.6537267080745341 Training loss: 0.03255070384985172
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.15432098765432098 Validation loss: 0.12199688840795446
```

```
Temps de 1 Epoch 34 en secondes: 0.1925210952758789
```

```
EPOCH: 35
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.6444099378881988 Training loss: 0.032406038193969255
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.1419753086419753 Validation loss: 0.10981476012571359
```

```
Temps de 1 Epoch 35 en secondes: 0.20180487632751465
```

```
EPOCH: 36
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.6940993788819876 Training loss: 0.02843040366720709
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.08024691358024691 Validation loss: 0.12365668202623908
```

```
Temps de 1 Epoch 36 en secondes: 0.19103407859802246
```

```
EPOCH: 37
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.6630434782608695 Training loss: 0.029795445177866066
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.1419753086419753 Validation loss: 0.12266900068447914

Temps de 1 Epoch 37 en secondes: 0.18711638450622559

EPOCH: 38

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.6645962732919255 Training loss: 0.031335382265333805

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.2037037037037037 Validation loss: 0.11832580595840643

Temps de 1 Epoch 38 en secondes: 0.19553542137145996

EPOCH: 39

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.6956521739130435 Training loss: 0.029337982166998136

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.12962962962962962 Validation loss: 0.11721043969378059

Temps de 1 Epoch 39 en secondes: 0.19750070571899414

EPOCH: 40

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.6925465838509317 Training loss: 0.028280006293554483

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.15432098765432098 Validation loss: 0.11561626563837499

Temps de 1 Epoch 40 en secondes: 0.2028200626373291

EPOCH: 41

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.7406832298136646 Training loss: 0.023843361816791274

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.1111111111111111 Validation loss: 0.13453474162537374
Temps de l Epoch 41 en secondes: 0.19727420806884766
EPOCH: 42

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.7096273291925466 Training loss: 0.026755002714832377
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.1666666666666666 Validation loss: 0.13249038031071791
Temps de l Epoch 42 en secondes: 0.19889068603515625
EPOCH: 43

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.7251552795031055 Training loss: 0.02412442844476759
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.14814814814814814 Validation loss: 0.13844051331649593
Temps de l Epoch 43 en secondes: 0.20046281814575195
EPOCH: 44

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.7391304347826086 Training loss: 0.023830439687145422
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.18518518518518517 Validation loss: 0.13718114075837312
Temps de l Epoch 44 en secondes: 0.18804502487182617
EPOCH: 45

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.7158385093167702 Training loss: 0.026174186734679323
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.19135802469135801 Validation loss: 0.1312598372683113

Temps de 1 Epoch 45 en secondes: 0.1964871883392334

EPOCH: 46

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.7298136645962733 Training loss: 0.025201937704352858

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.19135802469135801 Validation loss: 0.12692332414933194

Temps de 1 Epoch 46 en secondes: 0.2109518051147461

EPOCH: 47

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.7546583850931677 Training loss: 0.02324649774880143

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.2222222222222222 Validation loss: 0.12297458413206501

Temps de 1 Epoch 47 en secondes: 0.1978287696838379

EPOCH: 48

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.7950310559006211 Training loss: 0.020282256168238123

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.1728395061728395 Validation loss: 0.13153922116314923

Temps de 1 Epoch 48 en secondes: 0.19643592834472656

EPOCH: 49

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.7779503105590062 Training loss: 0.01960645167168623

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.18518518518518517 Validation loss: 0.12701175389466463
```

```
Temps de l Epoch 49 en secondes: 0.20046353340148926
```

```
EPOCH: 50
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8059006211180124 Training loss: 0.01862410992754172
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.1666666666666666 Validation loss: 0.13351998211425029
```

```
Temps de l Epoch 50 en secondes: 0.19060611724853516
```

```
EPOCH: 51
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.7996894409937888 Training loss: 0.020955175439023085
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.1419753086419753 Validation loss: 0.14065601649107756
```

```
Temps de l Epoch 51 en secondes: 0.19850397109985352
```

```
EPOCH: 52
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.7981366459627329 Training loss: 0.019716602582368792
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.19135802469135801 Validation loss: 0.12669167989566002
```

```
Temps de l Epoch 52 en secondes: 0.20967364311218262
```

```
EPOCH: 53
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8090062111801242 Training loss: 0.018489426082890968
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.1666666666666666 Validation loss: 0.12445140179292655

Temps de 1 Epoch 53 en secondes: 0.19159793853759766

EPOCH: 54

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8245341614906833 Training loss: 0.01755136938661522

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.19753086419753085 Validation loss: 0.13405622008406085

Temps de 1 Epoch 54 en secondes: 0.19464635848999023

EPOCH: 55

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.7857142857142857 Training loss: 0.02071948631764939

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.1728395061728395 Validation loss: 0.13334953784942627

Temps de 1 Epoch 55 en secondes: 0.19947290420532227

EPOCH: 56

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.7934782608695652 Training loss: 0.018962885235777553

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.19135802469135801 Validation loss: 0.13564471992445581

Temps de 1 Epoch 56 en secondes: 0.187530517578125

EPOCH: 57

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8400621118012422 Training loss: 0.01452237342177711

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.2037037037037 Validation loss: 0.14183754096796483
Temps de l Epoch  57  en secondes: 0.2050461769104004
EPOCH: 58

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.8478260869565217 Training loss: 0.015334055383012902
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.2037037037037 Validation loss: 0.14223678906758627
Temps de l Epoch  58  en secondes: 0.1965475082397461
EPOCH: 59

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.8416149068322981 Training loss: 0.014291697222253551
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.17901234567901234 Validation loss: 0.14560096940876524
Temps de l Epoch  59  en secondes: 0.19323062896728516
EPOCH: 60

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.8618012422360248 Training loss: 0.013865636372418137
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.19753086419753085 Validation loss: 0.15299532001401173
Temps de l Epoch  60  en secondes: 0.18949246406555176
EPOCH: 61

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.8462732919254659 Training loss: 0.013991605721830582
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.22839506172839505 Validation loss: 0.14739914882330246
```

```
Temps de 1 Epoch 61 en secondes: 0.19975566864013672
```

```
EPOCH: 62
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.827639751552795 Training loss: 0.015349949738994149
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.1666666666666666 Validation loss: 0.14369533680103444
```

```
Temps de 1 Epoch 62 en secondes: 0.198469877243042
```

```
EPOCH: 63
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.860248447204969 Training loss: 0.014304631291339116
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.19135802469135801 Validation loss: 0.15475538189028515
```

```
Temps de 1 Epoch 63 en secondes: 0.21638178825378418
```

```
EPOCH: 64
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8260869565217391 Training loss: 0.013798056066221332
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.2037037037037 Validation loss: 0.14885980111581307
```

```
Temps de 1 Epoch 64 en secondes: 0.19550275802612305
```

```
EPOCH: 65
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8586956521739131 Training loss: 0.012897805143032015
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.18518518518518517 Validation loss: 0.14617134759455552

Temps de 1 Epoch 65 en secondes: 0.1877453327178955

EPOCH: 66

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8726708074534162 Training loss: 0.011432118713855743

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.20987654320987653 Validation loss: 0.15414795463467823

Temps de 1 Epoch 66 en secondes: 0.19042253494262695

EPOCH: 67

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8649068322981367 Training loss: 0.013662058235325429

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.15432098765432098 Validation loss: 0.14244388945308734

Temps de 1 Epoch 67 en secondes: 0.18705153465270996

EPOCH: 68

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.843167701863354 Training loss: 0.013013356614001789

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.20987654320987653 Validation loss: 0.14353945667361034

Temps de 1 Epoch 68 en secondes: 0.1894543170928955

EPOCH: 69

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8664596273291926 Training loss: 0.01126237972552732

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.19753086419753085 Validation loss: 0.14899948938393298

Temps de l Epoch 69 en secondes: 0.1984691619873047

EPOCH: 70

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8618012422360248 Training loss: 0.014751481123685097

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.19135802469135801 Validation loss: 0.14943603968914645

Temps de l Epoch 70 en secondes: 0.19550657272338867

EPOCH: 71

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8555900621118012 Training loss: 0.01310462666594464

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.21604938271604937 Validation loss: 0.13454539393201287

Temps de l Epoch 71 en secondes: 0.19228243827819824

EPOCH: 72

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8586956521739131 Training loss: 0.012074208801022227

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.18518518518518517 Validation loss: 0.16391037128589772

Temps de l Epoch 72 en secondes: 0.18447613716125488

EPOCH: 73

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8928571428571429 Training loss: 0.010151254302411346

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.19753086419753085 Validation loss: 0.14207562841015098

Temps de l Epoch 73 en secondes: 0.1855332851409912

EPOCH: 74

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8819875776397516 Training loss: 0.01101420597630258

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.20987654320987653 Validation loss: 0.15148183445871613

Temps de l Epoch 74 en secondes: 0.2014312744140625

EPOCH: 75

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8773291925465838 Training loss: 0.010907675631298042

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.2222222222222222 Validation loss: 0.15351712115016986

Temps de l Epoch 75 en secondes: 0.20349669456481934

EPOCH: 76

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8742236024844721 Training loss: 0.011608652053227336

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.21604938271604937 Validation loss: 0.1572703226113025

Temps de l Epoch 76 en secondes: 0.18810510635375977

EPOCH: 77

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.8804347826086957 Training loss: 0.011760558030620125

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.18518518518518517 Validation loss: 0.15923120063028218
```

```
Temps de l Epoch 77 en secondes: 0.1900310516357422
```

```
EPOCH: 78
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.860248447204969 Training loss: 0.012413456388141798
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.21604938271604937 Validation loss: 0.15534710148234426
```

```
Temps de l Epoch 78 en secondes: 0.18793678283691406
```

```
EPOCH: 79
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8555900621118012 Training loss: 0.013226491462740098
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.2037037037037 Validation loss: 0.15683655385617856
```

```
Temps de l Epoch 79 en secondes: 0.1870126724243164
```

```
EPOCH: 80
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8695652173913043 Training loss: 0.011765544713469026
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.19135802469135801 Validation loss: 0.1566677741062494
```

```
Temps de l Epoch 80 en secondes: 0.19048857688903809
```

```
EPOCH: 81
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8711180124223602 Training loss: 0.012252428817637958
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.19135802469135801 Validation loss: 0.16054925506497608
```

```
Temps de 1 Epoch 81 en secondes: 0.20245671272277832
```

```
EPOCH: 82
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8664596273291926 Training loss: 0.012031409496105976
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.21604938271604937 Validation loss: 0.1551705863740709
```

```
Temps de 1 Epoch 82 en secondes: 0.18450713157653809
```

```
EPOCH: 83
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8773291925465838 Training loss: 0.011302122256215314
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.19135802469135801 Validation loss: 0.16211817588335203
```

```
Temps de 1 Epoch 83 en secondes: 0.19050073623657227
```

```
EPOCH: 84
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8913043478260869 Training loss: 0.008850543001979034
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.2222222222222222 Validation loss: 0.16169184961436708
```

```
Temps de 1 Epoch 84 en secondes: 0.19152092933654785
```

```
EPOCH: 85
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.8944099378881988 Training loss: 0.009887426883352469
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.20987654320987653 Validation loss: 0.16934547012234913
Temps de l Epoch 85 en secondes: 0.192457914352417
EPOCH: 86

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.9083850931677019 Training loss: 0.008965793603695697
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.2345679012345679 Validation loss: 0.17069128118915322
Temps de l Epoch 86 en secondes: 0.19499802589416504
EPOCH: 87

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.8928571428571429 Training loss: 0.009591969539937765
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.2037037037037037 Validation loss: 0.16632597240400904
Temps de l Epoch 87 en secondes: 0.19667387008666992
EPOCH: 88

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.922360248447205 Training loss: 0.007810923045281298
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.21604938271604937 Validation loss: 0.17846229782810918
Temps de l Epoch 88 en secondes: 0.19906234741210938
EPOCH: 89

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.9239130434782609 Training loss: 0.008309805418671289
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.2037037037037 Validation loss: 0.1737961033244192
```

```
Temps de l Epoch 89 en secondes: 0.18851280212402344
```

```
EPOCH: 90
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.9083850931677019 Training loss: 0.00811811621005861
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.25308641975308643 Validation loss: 0.1728303211706656
```

```
Temps de l Epoch 90 en secondes: 0.1914987564086914
```

```
EPOCH: 91
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.9177018633540373 Training loss: 0.007794554339525122
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.20987654320987653 Validation loss: 0.16948223997045447
```

```
Temps de l Epoch 91 en secondes: 0.18761301040649414
```

```
EPOCH: 92
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.9254658385093167 Training loss: 0.008176892619834553
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

```
Validation accuracy: 0.18518518518518517 Validation loss: 0.17025840135268222
```

```
Temps de l Epoch 92 en secondes: 0.2044532299041748
```

```
EPOCH: 93
```

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

```
Training accuracy: 0.9192546583850931 Training loss: 0.007992859756261666
```

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.19753086419753085 Validation loss: 0.17048692114559222

Temps de 1 Epoch 93 en secondes: 0.18658757209777832

EPOCH: 94

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.9114906832298136 Training loss: 0.00893434242385885

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.20987654320987653 Validation loss: 0.17316449718710816

Temps de 1 Epoch 94 en secondes: 0.19068670272827148

EPOCH: 95

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.9301242236024845 Training loss: 0.0070518076547091794

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.21604938271604937 Validation loss: 0.17673125679110302

Temps de 1 Epoch 95 en secondes: 0.19037461280822754

EPOCH: 96

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.9254658385093167 Training loss: 0.0068679823198859

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))
```

Validation accuracy: 0.2222222222222222 Validation loss: 0.17290929511741357

Temps de 1 Epoch 96 en secondes: 0.18838119506835938

EPOCH: 97

```
HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))
```

Training accuracy: 0.9316770186335404 Training loss: 0.006209660506581668

```
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.2345679012345679 Validation loss: 0.15718597541620702
Temps de l Epoch 97 en secondes: 0.18853259086608887
EPOCH: 98

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.9487577639751553 Training loss: 0.006104377957396441
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

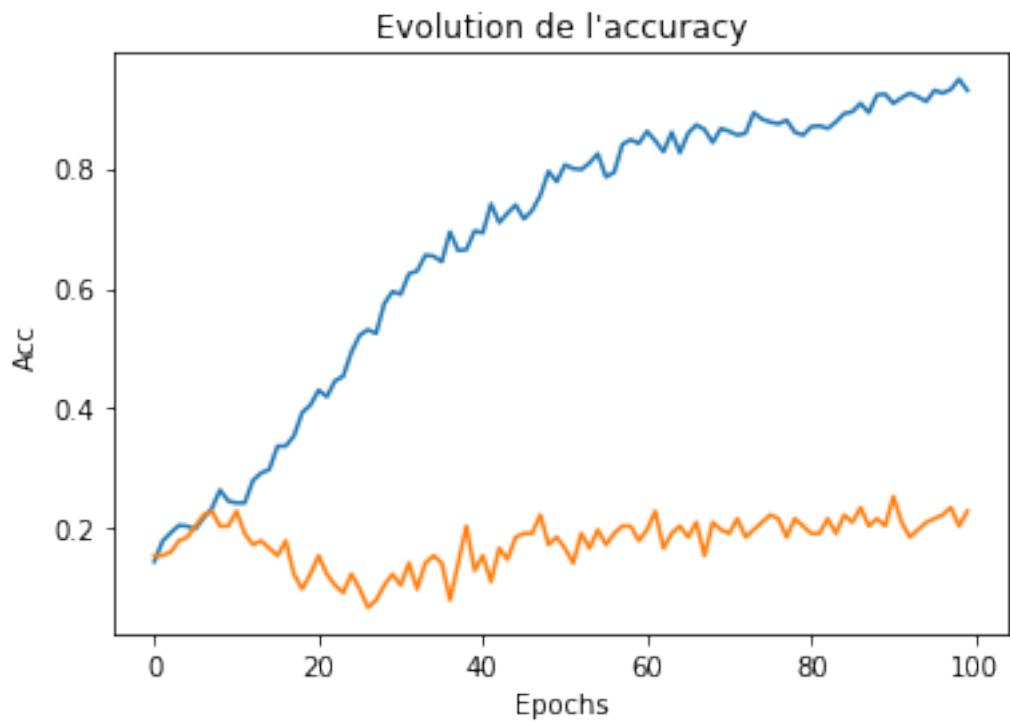
Validation accuracy: 0.2037037037037037 Validation loss: 0.19146656107019494
Temps de l Epoch 98 en secondes: 0.19045686721801758
EPOCH: 99

HBox(children=(FloatProgress(value=0.0, max=22.0), HTML(value='')))

Training accuracy: 0.9301242236024845 Training loss: 0.007766451455236222
HBox(children=(FloatProgress(value=0.0, max=6.0), HTML(value='')))

Validation accuracy: 0.22839506172839505 Validation loss: 0.17512791245071976
Temps de l Epoch 99 en secondes: 0.19051671028137207
Temps total en secondes: 20.15778398513794
```

```
[17]: plt.plot(np.arange(0,N_EPOCHS),epoch_acc)
plt.plot(np.arange(0,N_EPOCHS),epoch_val_acc)
plt.title("Evolution de l'accuracy")
plt.xlabel("Epochs")
plt.ylabel("Acc")
plt.show()
```



```
[18]: plt.plot(np.arange(0,N_EPOCHS),epoch_loss)
plt.plot(np.arange(0,N_EPOCHS),epoch_val_loss)
plt.title("Evolution du loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```



2 Keras

```
[19]: X_train = dataX #features
Y_train = dataY #labels
```

```
[20]: np.shape(X_train)
```

```
[20]: (806, 3081)
```

```
[21]: x_train,x_test,y_train,y_test = train_test_split(X_train,Y_train,test_size=0.2)
#y_train = to_class(y_train)
#y_test = to_class(y_test)
```

```
[39]: model = Sequential()
model.add(Dense(2400, activation='relu', input_dim=3081))
model.add(Dropout(0.3))
model.add(Dense(1200, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(600, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(11, activation='softmax'))
```

```

[33]: model1 = Sequential()
model1.add(Dense(800, activation='relu', input_dim=3081))
model1.add(Dropout(0.3))
model1.add(Dense(11, activation='softmax'))

[34]: adam = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07, ↴
    ↵amsgrad=False)

[35]: model.compile(loss='categorical_crossentropy', optimizer=adam, ↴
    ↵metrics=['accuracy'])

[40]: epochs = 100
learning_rate = 0.1
decay_rate = learning_rate / epochs
momentum = 0.8
sgd = SGD(lr=learning_rate, momentum=momentum, decay=decay_rate, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer='sgd', ↴
    ↵metrics=['accuracy'])

[41]: history = model.fit(x_train, y_train, epochs=100, batch_size=30, ↴
    ↵validation_data=(x_test,y_test), shuffle=True)
model.test_on_batch(x_test, y_test)
model.metrics_names

Train on 644 samples, validate on 162 samples
Epoch 1/100
644/644 [=====] - 1s 1ms/sample - loss: 2.5481 - acc: 0.1351 - val_loss: 2.2513 - val_acc: 0.1790
Epoch 2/100
644/644 [=====] - 1s 954us/sample - loss: 2.3004 - acc: 0.1755 - val_loss: 2.1985 - val_acc: 0.1543
Epoch 3/100
644/644 [=====] - 1s 986us/sample - loss: 2.1126 - acc: 0.2376 - val_loss: 2.2096 - val_acc: 0.1420
Epoch 4/100
644/644 [=====] - 1s 991us/sample - loss: 2.0967 - acc: 0.2174 - val_loss: 2.2267 - val_acc: 0.1667
Epoch 5/100
644/644 [=====] - 1s 1ms/sample - loss: 2.0534 - acc: 0.2562 - val_loss: 2.1927 - val_acc: 0.1481
Epoch 6/100
644/644 [=====] - 1s 939us/sample - loss: 1.9749 - acc: 0.2655 - val_loss: 2.2506 - val_acc: 0.1481
Epoch 7/100
644/644 [=====] - 1s 1ms/sample - loss: 1.9129 - acc: 0.3043 - val_loss: 2.3324 - val_acc: 0.1049
Epoch 8/100
644/644 [=====] - 1s 938us/sample - loss: 1.8690 - acc:

```

```
0.3075 - val_loss: 2.2535 - val_acc: 0.1605
Epoch 9/100
644/644 [=====] - 1s 974us/sample - loss: 1.8089 - acc:
0.3463 - val_loss: 2.2992 - val_acc: 0.1358
Epoch 10/100
644/644 [=====] - 1s 1ms/sample - loss: 1.7986 - acc:
0.3478 - val_loss: 2.2606 - val_acc: 0.1420
Epoch 11/100
644/644 [=====] - 1s 1ms/sample - loss: 1.8106 - acc:
0.3152 - val_loss: 2.2759 - val_acc: 0.1358
Epoch 12/100
644/644 [=====] - 1s 1ms/sample - loss: 1.7794 - acc:
0.3276 - val_loss: 2.3270 - val_acc: 0.1235
Epoch 13/100
644/644 [=====] - 1s 975us/sample - loss: 1.7129 - acc:
0.3602 - val_loss: 2.3113 - val_acc: 0.1296
Epoch 14/100
644/644 [=====] - 1s 921us/sample - loss: 1.6612 - acc:
0.3866 - val_loss: 2.3086 - val_acc: 0.1605
Epoch 15/100
644/644 [=====] - 1s 962us/sample - loss: 1.6239 - acc:
0.4037 - val_loss: 2.3387 - val_acc: 0.1420
Epoch 16/100
644/644 [=====] - 1s 923us/sample - loss: 1.6243 - acc:
0.3727 - val_loss: 2.3514 - val_acc: 0.1296
Epoch 17/100
644/644 [=====] - 1s 973us/sample - loss: 1.6034 - acc:
0.4115 - val_loss: 2.3592 - val_acc: 0.1296
Epoch 18/100
644/644 [=====] - 1s 933us/sample - loss: 1.5931 - acc:
0.3929 - val_loss: 2.4073 - val_acc: 0.1605
Epoch 19/100
644/644 [=====] - 1s 1ms/sample - loss: 1.5099 - acc:
0.4689 - val_loss: 2.4574 - val_acc: 0.1420
Epoch 20/100
644/644 [=====] - 1s 954us/sample - loss: 1.5116 - acc:
0.4627 - val_loss: 2.4889 - val_acc: 0.1543
Epoch 21/100
644/644 [=====] - 1s 1ms/sample - loss: 1.5048 - acc:
0.4348 - val_loss: 2.5300 - val_acc: 0.1296
Epoch 22/100
644/644 [=====] - 1s 937us/sample - loss: 1.5026 - acc:
0.4394 - val_loss: 2.5115 - val_acc: 0.1420
Epoch 23/100
644/644 [=====] - 1s 979us/sample - loss: 1.4683 - acc:
0.4658 - val_loss: 2.4887 - val_acc: 0.1173
Epoch 24/100
644/644 [=====] - 1s 985us/sample - loss: 1.4263 - acc:
```

```
0.4596 - val_loss: 2.5437 - val_acc: 0.1358
Epoch 25/100
644/644 [=====] - 1s 1ms/sample - loss: 1.4028 - acc:
0.4519 - val_loss: 2.5222 - val_acc: 0.1296
Epoch 26/100
644/644 [=====] - 1s 1ms/sample - loss: 1.3963 - acc:
0.4720 - val_loss: 2.5905 - val_acc: 0.0988
Epoch 27/100
644/644 [=====] - 1s 1ms/sample - loss: 1.3718 - acc:
0.4193 - val_loss: 2.6147 - val_acc: 0.1173
Epoch 28/100
644/644 [=====] - 1s 1ms/sample - loss: 1.3714 - acc:
0.4565 - val_loss: 2.6759 - val_acc: 0.1358
Epoch 29/100
644/644 [=====] - 1s 1ms/sample - loss: 1.3578 - acc:
0.4907 - val_loss: 2.7614 - val_acc: 0.1235
Epoch 30/100
644/644 [=====] - 1s 1ms/sample - loss: 1.3983 - acc:
0.4363 - val_loss: 2.7340 - val_acc: 0.1173
Epoch 31/100
644/644 [=====] - 1s 1ms/sample - loss: 1.3308 - acc:
0.4829 - val_loss: 2.6627 - val_acc: 0.1358
Epoch 32/100
644/644 [=====] - 1s 1ms/sample - loss: 1.3140 - acc:
0.4845 - val_loss: 2.6896 - val_acc: 0.1358
Epoch 33/100
644/644 [=====] - 1s 994us/sample - loss: 1.2509 - acc:
0.5171 - val_loss: 2.7391 - val_acc: 0.1605
Epoch 34/100
644/644 [=====] - 1s 1ms/sample - loss: 1.2660 - acc:
0.5047 - val_loss: 2.7702 - val_acc: 0.1235
Epoch 35/100
644/644 [=====] - 1s 1ms/sample - loss: 1.2295 - acc:
0.5217 - val_loss: 2.7645 - val_acc: 0.1296
Epoch 36/100
644/644 [=====] - 1s 1ms/sample - loss: 1.2325 - acc:
0.5171 - val_loss: 2.8147 - val_acc: 0.1296
Epoch 37/100
644/644 [=====] - 1s 1ms/sample - loss: 1.1627 - acc:
0.5388 - val_loss: 2.8020 - val_acc: 0.1296
Epoch 38/100
644/644 [=====] - 1s 1ms/sample - loss: 1.1435 - acc:
0.5652 - val_loss: 2.9066 - val_acc: 0.1543
Epoch 39/100
644/644 [=====] - 1s 1ms/sample - loss: 1.1856 - acc:
0.5435 - val_loss: 2.8381 - val_acc: 0.1605
Epoch 40/100
644/644 [=====] - 1s 1ms/sample - loss: 1.1183 - acc:
```

```
0.5543 - val_loss: 2.8038 - val_acc: 0.1358
Epoch 41/100
644/644 [=====] - 1s 1ms/sample - loss: 1.1136 - acc:
0.5559 - val_loss: 2.9114 - val_acc: 0.1235
Epoch 42/100
644/644 [=====] - 1s 949us/sample - loss: 1.1452 - acc:
0.5311 - val_loss: 2.9211 - val_acc: 0.1420
Epoch 43/100
644/644 [=====] - 1s 1ms/sample - loss: 1.0908 - acc:
0.5559 - val_loss: 2.9425 - val_acc: 0.1667
Epoch 44/100
644/644 [=====] - 1s 997us/sample - loss: 1.1398 - acc:
0.5466 - val_loss: 2.9422 - val_acc: 0.1358
Epoch 45/100
644/644 [=====] - 1s 1ms/sample - loss: 1.0763 - acc:
0.5761 - val_loss: 3.0173 - val_acc: 0.1358
Epoch 46/100
644/644 [=====] - 1s 1ms/sample - loss: 1.0620 - acc:
0.5916 - val_loss: 2.9618 - val_acc: 0.1173
Epoch 47/100
644/644 [=====] - 1s 1ms/sample - loss: 1.0513 - acc:
0.5963 - val_loss: 2.9879 - val_acc: 0.1543
Epoch 48/100
644/644 [=====] - 1s 967us/sample - loss: 1.0367 - acc:
0.5699 - val_loss: 3.1323 - val_acc: 0.1543
Epoch 49/100
644/644 [=====] - 1s 1ms/sample - loss: 1.0498 - acc:
0.5885 - val_loss: 3.0855 - val_acc: 0.1543
Epoch 50/100
644/644 [=====] - 1s 947us/sample - loss: 0.9914 - acc:
0.6040 - val_loss: 3.0355 - val_acc: 0.1420
Epoch 51/100
644/644 [=====] - 1s 985us/sample - loss: 0.9384 - acc:
0.6351 - val_loss: 3.1278 - val_acc: 0.1481
Epoch 52/100
644/644 [=====] - 1s 963us/sample - loss: 1.0218 - acc:
0.6009 - val_loss: 3.0574 - val_acc: 0.1790
Epoch 53/100
644/644 [=====] - 1s 1ms/sample - loss: 0.9775 - acc:
0.6165 - val_loss: 3.0369 - val_acc: 0.1420
Epoch 54/100
644/644 [=====] - 1s 977us/sample - loss: 0.9455 - acc:
0.6522 - val_loss: 3.0465 - val_acc: 0.1296
Epoch 55/100
644/644 [=====] - 1s 1ms/sample - loss: 0.9281 - acc:
0.6366 - val_loss: 3.0675 - val_acc: 0.1420
Epoch 56/100
644/644 [=====] - 1s 947us/sample - loss: 0.9085 - acc:
```

```
0.6506 - val_loss: 3.0961 - val_acc: 0.1173
Epoch 57/100
644/644 [=====] - 1s 985us/sample - loss: 0.9168 - acc:
0.6413 - val_loss: 3.1085 - val_acc: 0.1481
Epoch 58/100
644/644 [=====] - 1s 975us/sample - loss: 0.8739 - acc:
0.6382 - val_loss: 3.2180 - val_acc: 0.1605
Epoch 59/100
644/644 [=====] - 1s 1ms/sample - loss: 0.8651 - acc:
0.6786 - val_loss: 3.0888 - val_acc: 0.1667
Epoch 60/100
644/644 [=====] - 1s 981us/sample - loss: 0.8462 - acc:
0.6646 - val_loss: 3.1350 - val_acc: 0.1605
Epoch 61/100
644/644 [=====] - 1s 1ms/sample - loss: 0.8669 - acc:
0.6770 - val_loss: 3.1364 - val_acc: 0.1605
Epoch 62/100
644/644 [=====] - 1s 933us/sample - loss: 0.8473 - acc:
0.6801 - val_loss: 3.2137 - val_acc: 0.1852
Epoch 63/100
644/644 [=====] - 1s 974us/sample - loss: 0.8228 - acc:
0.6988 - val_loss: 3.2742 - val_acc: 0.1481
Epoch 64/100
644/644 [=====] - 1s 923us/sample - loss: 0.8118 - acc:
0.6817 - val_loss: 3.1975 - val_acc: 0.1667
Epoch 65/100
644/644 [=====] - 1s 1ms/sample - loss: 0.8171 - acc:
0.7003 - val_loss: 3.2827 - val_acc: 0.2037
Epoch 66/100
644/644 [=====] - 1s 923us/sample - loss: 0.7987 - acc:
0.7019 - val_loss: 3.2419 - val_acc: 0.1667
Epoch 67/100
644/644 [=====] - 1s 942us/sample - loss: 0.7786 - acc:
0.6941 - val_loss: 3.1821 - val_acc: 0.1914
Epoch 68/100
644/644 [=====] - 1s 945us/sample - loss: 0.7048 - acc:
0.7252 - val_loss: 3.2699 - val_acc: 0.1667
Epoch 69/100
644/644 [=====] - 1s 966us/sample - loss: 0.7535 - acc:
0.7112 - val_loss: 3.1949 - val_acc: 0.1852
Epoch 70/100
644/644 [=====] - 1s 932us/sample - loss: 0.7143 - acc:
0.7158 - val_loss: 3.1819 - val_acc: 0.1914
Epoch 71/100
644/644 [=====] - 1s 1ms/sample - loss: 0.7444 - acc:
0.7314 - val_loss: 3.2027 - val_acc: 0.1914
Epoch 72/100
644/644 [=====] - 1s 945us/sample - loss: 0.7301 - acc:
```

```
0.7189 - val_loss: 3.2405 - val_acc: 0.1605
Epoch 73/100
644/644 [=====] - 1s 994us/sample - loss: 0.6628 - acc:
0.7562 - val_loss: 3.3267 - val_acc: 0.1852
Epoch 74/100
644/644 [=====] - 1s 951us/sample - loss: 0.6798 - acc:
0.7609 - val_loss: 3.2679 - val_acc: 0.1975
Epoch 75/100
644/644 [=====] - 1s 988us/sample - loss: 0.6351 - acc:
0.7376 - val_loss: 3.1256 - val_acc: 0.1790
Epoch 76/100
644/644 [=====] - 1s 936us/sample - loss: 0.6940 - acc:
0.7360 - val_loss: 3.3409 - val_acc: 0.2037
Epoch 77/100
644/644 [=====] - 1s 980us/sample - loss: 0.6453 - acc:
0.7640 - val_loss: 3.6049 - val_acc: 0.1667
Epoch 78/100
644/644 [=====] - 1s 974us/sample - loss: 0.6314 - acc:
0.7609 - val_loss: 3.3046 - val_acc: 0.1975
Epoch 79/100
644/644 [=====] - 1s 979us/sample - loss: 0.6421 - acc:
0.7531 - val_loss: 3.2548 - val_acc: 0.1852
Epoch 80/100
644/644 [=====] - 1s 969us/sample - loss: 0.5860 - acc:
0.7997 - val_loss: 3.3208 - val_acc: 0.1790
Epoch 81/100
644/644 [=====] - 1s 964us/sample - loss: 0.6178 - acc:
0.7686 - val_loss: 3.2249 - val_acc: 0.2037
Epoch 82/100
644/644 [=====] - 1s 906us/sample - loss: 0.5366 - acc:
0.7966 - val_loss: 3.4431 - val_acc: 0.2160
Epoch 83/100
644/644 [=====] - 1s 976us/sample - loss: 0.6127 - acc:
0.7733 - val_loss: 3.3309 - val_acc: 0.1975
Epoch 84/100
644/644 [=====] - 1s 950us/sample - loss: 0.5409 - acc:
0.7873 - val_loss: 3.3302 - val_acc: 0.2099
Epoch 85/100
644/644 [=====] - 1s 989us/sample - loss: 0.5447 - acc:
0.7981 - val_loss: 3.4499 - val_acc: 0.1914
Epoch 86/100
644/644 [=====] - 1s 946us/sample - loss: 0.5577 - acc:
0.8090 - val_loss: 3.3700 - val_acc: 0.1852
Epoch 87/100
644/644 [=====] - 1s 1ms/sample - loss: 0.5478 - acc:
0.8043 - val_loss: 3.3089 - val_acc: 0.1975
Epoch 88/100
644/644 [=====] - 1s 917us/sample - loss: 0.5383 - acc:
```

```

0.8152 - val_loss: 3.3246 - val_acc: 0.2099
Epoch 89/100
644/644 [=====] - 1s 949us/sample - loss: 0.5318 - acc:
0.8214 - val_loss: 3.3655 - val_acc: 0.2099
Epoch 90/100
644/644 [=====] - 1s 920us/sample - loss: 0.5487 - acc:
0.8090 - val_loss: 3.2609 - val_acc: 0.2099
Epoch 91/100
644/644 [=====] - 1s 973us/sample - loss: 0.4775 - acc:
0.8307 - val_loss: 3.6695 - val_acc: 0.1605
Epoch 92/100
644/644 [=====] - 1s 962us/sample - loss: 0.5316 - acc:
0.8261 - val_loss: 3.4917 - val_acc: 0.2099
Epoch 93/100
644/644 [=====] - 1s 1ms/sample - loss: 0.4720 - acc:
0.8106 - val_loss: 3.4547 - val_acc: 0.1914
Epoch 94/100
644/644 [=====] - 1s 933us/sample - loss: 0.5547 - acc:
0.8090 - val_loss: 3.5521 - val_acc: 0.1975
Epoch 95/100
644/644 [=====] - 1s 968us/sample - loss: 0.4903 - acc:
0.8307 - val_loss: 3.3650 - val_acc: 0.2222
Epoch 96/100
644/644 [=====] - 1s 932us/sample - loss: 0.4857 - acc:
0.8276 - val_loss: 3.3738 - val_acc: 0.2346
Epoch 97/100
644/644 [=====] - 1s 999us/sample - loss: 0.4905 - acc:
0.8292 - val_loss: 3.2733 - val_acc: 0.2160
Epoch 98/100
644/644 [=====] - 1s 959us/sample - loss: 0.4614 - acc:
0.8339 - val_loss: 3.3017 - val_acc: 0.2222
Epoch 99/100
644/644 [=====] - 1s 959us/sample - loss: 0.4516 - acc:
0.8416 - val_loss: 3.6604 - val_acc: 0.2099
Epoch 100/100
644/644 [=====] - 1s 949us/sample - loss: 0.4700 - acc:
0.8385 - val_loss: 3.4158 - val_acc: 0.2222

```

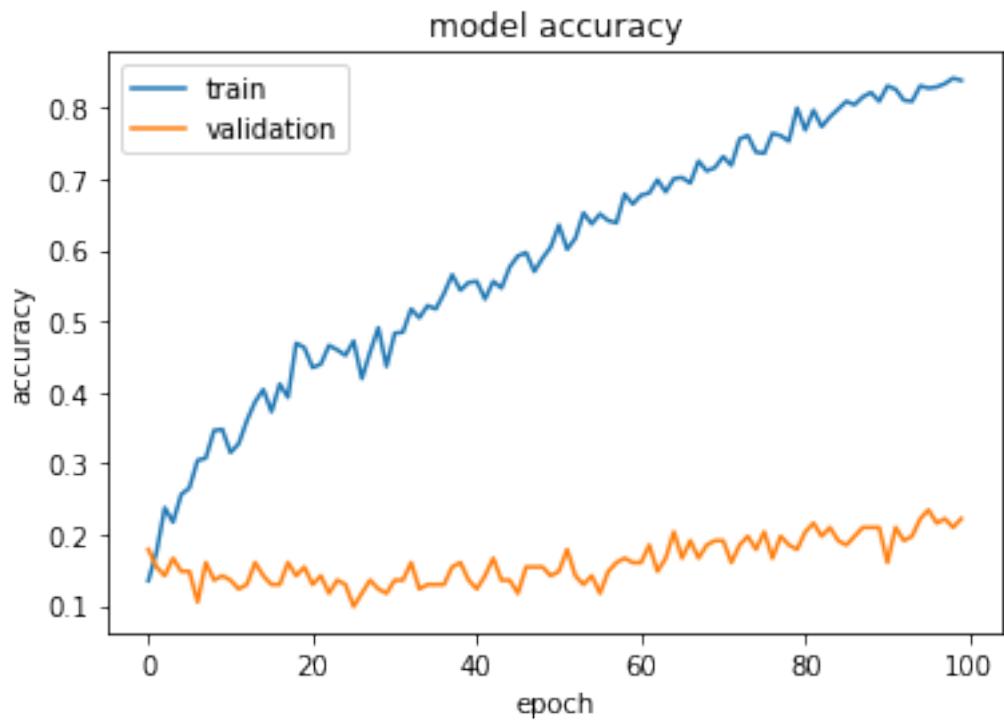
[41]: ['loss', 'acc']

```

[42]: # "Accuracy"
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')

```

```
plt.show()
```



```
[43]: plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



[]:

DREAM 1 NN subject

February 12, 2021

```
[2]: def to_class(y_):
    res = np.zeros((len(y_), 11), dtype='q')
    for i in range(len(y_)):
        res[i][int(y_[i])] = 1
    return res
def maximum(liste):
    maxi = liste[0]
    for i in liste:
        if i >= maxi:
            maxi = i
    return maxi
def minimum(liste):
    mini = liste[0]
    for i in liste:
        if i <= mini:
            mini = i
    return mini
```

```
[3]: dfX = pd.read_csv('Molecular Dataset Dream 1.csv',sep=';')
dfY = pd.read_csv('Senteur Dataset Dream 1.csv',sep=';')
df = dfX.merge(dfY)
```

```
[4]: list_label = ['INTENSITY/STRENGTH', 'VALENCE/PLEASANTNESS', 'BAKERY', 'SWEET',
                 'FRUIT', 'FISH', 'GARLIC', 'SPICES', 'COLD', 'SOUR', 'BURNT', 'ACID',
                 'WARM', 'MUSKY', 'SWEATY', 'AMMONIA/URINOUS', 'DECAYED', 'WOOD',
                 'GRASS', 'FLOWER', 'CHEMICAL']

#Pour les barplot
list_label_bar = ['INTENSITY', 'VALENCE', 'BAKERY', 'SWEET',
                  'FRUIT', 'FISH', 'GARLIC', 'SPICES', 'COLD', 'SOUR', 'BURNT', 'ACID',
                  'WARM', 'MUSKY', 'SWEATY', 'AMMONIA', 'DECAYED', 'WOOD',
                  'GRASS', 'FLOWER', 'CHEMICAL']

Dico_labels = {}
for i in range (21):
    Dico_labels[i] = list_label[i]
```

```
[5]: class MyNetwork(nn.Module):
    def __init__(self):
        super(MyNetwork, self).__init__()
        ## Activation layer
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(in_features = 3082, out_features = 2400)
        self.fc2 = nn.Linear(2400, 1200)
        self.fc3 = nn.Linear(1200, 600)
        self.output = nn.Linear(600, 11)
        self.softmax = nn.LogSoftmax(dim=1)
    def forward(self, x):
        ## First full connection
        x = self.fc1(x)
        x = self.relu(x)
        ## Second full connection
        x = self.fc2(x)
        x = self.relu(x)
        ## Third full connection
        x = self.fc3(x)
        x = self.relu(x)
        ## Output layer
        x = self.output(x)
        y = self.softmax(x)
        return y
```

```
[6]: #Meta parametre
LEARNING_RATE = 0.003
MOMENTUM = 0.9
N_EPOCHS = 30
Batch = 30
# Normalisation
sc = StandardScaler()
```

```
[7]: Subject_acc_Train_15, Subject_acc_Train_30 = [], []
Subject_acc_Val_15, Subject_acc_Val_30 = [], []
Subject_loss_Train_15, Subject_loss_Train_30 = [], []
Subject_loss_Val_15, Subject_loss_Val_30 = [], []

for i in range(49):
    print('Prédiction du sujet:', i+1)
    dfY = df[['INTENSITY/STRENGTH', 'subject']]
    dfY = dfY[df['subject'] == i+1]
    dfY = dfY.drop('subject', axis=1)
    dfX = df[df['subject'] == i+1]
    dfX = dfX.drop(list_label, axis=1)
    dfX = dfX.drop('Unnamed: 0', axis=1)
    dfX = dfX.drop('CID', axis=1)
```

```

dfX = dfX.drop('subject',axis=1)
dataX = np.float32(dfX.values)
dataY = np.longlong(dfY.values)
dataY = to_class(dataY)
dataX = sc.fit_transform(dataX)

DATA_Train, DATA_Test, TARGET_Train, TARGET_Test = train_test_split(dataX, dataY ,test_size=0.1)
X_train_tensor = torch.from_numpy(DATA_Train)
Y_train_tensor = torch.from_numpy(TARGET_Train)
X_test_tensor = torch.from_numpy(DATA_Test)
Y_test_tensor = torch.from_numpy(TARGET_Test)
train = data_utils.TensorDataset(X_train_tensor, Y_train_tensor)
train_loader = data_utils.DataLoader(train, batch_size=Batch, shuffle=True)
test = data_utils.TensorDataset(X_test_tensor, Y_test_tensor)
test_loader = data_utils.DataLoader(test, batch_size=Batch, shuffle=True)

net = MyNetwork()
net = net.cuda()
criterion = nn.NLLLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=LEARNING_RATE, momentum=MOMENTUM)
epoch_loss, epoch_acc, epoch_val_loss, epoch_val_acc = [], [], [], []
start_time = time.time()

for e in range(N_EPOCHS):
    running_loss = 0
    running_accuracy = 0
    running_acc=0
    start_epoch_time=time.time()

    net.train()
    for i, batch in enumerate(train_loader):
        x = batch[0]
        labels = batch[1]
        x = x.cuda()
        labels = labels.cuda()
        output = net(x)
        y = output
        loss = criterion(y, torch.max(labels, 1)[1])
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        with torch.no_grad():
            running_loss += loss.item()
            running_accuracy += (y.max(1)[1] == torch.max(labels, 1)[1]).sum().item()

```

```

epoch_loss.append(running_loss/len(train))
epoch_acc.append(running_accuracy/len(train))

net.eval()
running_val_loss = 0
running_val_accuracy = 0
for i, batch in enumerate(test_loader):
    with torch.no_grad():
        x = batch[0]
        labels = batch[1]
        x = x.cuda()
        labels = labels.cuda()
        output = net(x)
        y = output
        loss = criterion(y, torch.max(labels, 1)[1])
        running_val_loss += loss.item()
        running_val_accuracy += (y.max(1)[1] == torch.max(labels, 1)[1]).sum().item()

    epoch_val_loss.append(running_val_loss/len(test))
    epoch_val_acc.append(running_val_accuracy/len(test))

if e == 14 :
    Subject_acc_Train_15.append(maximum(epoch_acc))
    Subject_acc_Val_15.append(maximum(epoch_val_acc))
    Subject_loss_Train_15.append(minimum(epoch_loss))
    Subject_loss_Val_15.append(minimum(epoch_val_loss))
if e == 29 :
    Subject_acc_Train_30.append(maximum(epoch_acc))
    Subject_acc_Val_30.append(maximum(epoch_val_acc))
    Subject_loss_Train_30.append(minimum(epoch_loss))
    Subject_loss_Val_30.append(minimum(epoch_val_loss))
print('Meilleur: Acc:',maximum(epoch_acc))
interval = time.time() - start_time
print ('Temps total en secondes:', interval)
del net
print( )

```

Prédiction du sujet: 1
Meilleur: Acc: 0.6496551724137931
Temps total en secondes: 8.900108098983765

Prédiction du sujet: 2
Meilleur: Acc: 0.6153846153846154
Temps total en secondes: 4.480771064758301

Prédiction du sujet: 3
Meilleur: Acc: 0.6739606126914661
Temps total en secondes: 3.726060390472412

Prédiction du sujet: 4
Meilleur: Acc: 0.710948905109489
Temps total en secondes: 5.395589828491211

Prédiction du sujet: 5
Meilleur: Acc: 0.6717724288840262
Temps total en secondes: 3.7651989459991455

Prédiction du sujet: 6
Meilleur: Acc: 0.7170418006430869
Temps total en secondes: 2.63811993598938

Prédiction du sujet: 7
Meilleur: Acc: 0.7208588957055214
Temps total en secondes: 5.137445688247681

Prédiction du sujet: 8
Meilleur: Acc: 0.61082910321489
Temps total en secondes: 4.6874823570251465

Prédiction du sujet: 9
Meilleur: Acc: 0.7553191489361702
Temps total en secondes: 3.7611331939697266

Prédiction du sujet: 10
Meilleur: Acc: 0.6559139784946236
Temps total en secondes: 2.374671459197998

Prédiction du sujet: 11
Meilleur: Acc: 0.7444444444444444
Temps total en secondes: 3.52140736579895

Prédiction du sujet: 12
Meilleur: Acc: 0.6179001721170396
Temps total en secondes: 4.667260646820068

Prédiction du sujet: 13
Meilleur: Acc: 0.6178010471204188
Temps total en secondes: 4.661982536315918

Prédiction du sujet: 14
Meilleur: Acc: 0.6446078431372549
Temps total en secondes: 3.302344560623169

Prédiction du sujet: 15
Meilleur: Acc: 0.6558265582655827
Temps total en secondes: 3.063683032989502

Prédiction du sujet: 16
Meilleur: Acc: 0.7112068965517241
Temps total en secondes: 3.925522565841675

Prédiction du sujet: 17
Meilleur: Acc: 0.6352941176470588
Temps total en secondes: 3.5310447216033936

Prédiction du sujet: 18
Meilleur: Acc: 0.6878504672897197
Temps total en secondes: 4.194800615310669

Prédiction du sujet: 19
Meilleur: Acc: 0.6609195402298851
Temps total en secondes: 4.16489839553833

Prédiction du sujet: 20
Meilleur: Acc: 0.6533795493934142
Temps total en secondes: 4.649585723876953

Prédiction du sujet: 21
Meilleur: Acc: 0.7058823529411765
Temps total en secondes: 4.9179017543792725

Prédiction du sujet: 22
Meilleur: Acc: 0.6252045826513911
Temps total en secondes: 4.895959377288818

Prédiction du sujet: 23
Meilleur: Acc: 0.6778947368421052
Temps total en secondes: 3.7378644943237305

Prédiction du sujet: 24
Meilleur: Acc: 0.6600910470409712
Temps total en secondes: 5.146965980529785

Prédiction du sujet: 25
Meilleur: Acc: 0.6347222222222222
Temps total en secondes: 5.5621442794799805

Prédiction du sujet: 26
Meilleur: Acc: 0.6210826210826211
Temps total en secondes: 5.5741119384765625

Prédiction du sujet: 27
Meilleur: Acc: 0.6056644880174292
Temps total en secondes: 3.7382404804229736

Prédiction du sujet: 28
Meilleur: Acc: 0.6190476190476191
Temps total en secondes: 4.4603471755981445

Prédiction du sujet: 29
Meilleur: Acc: 0.7747603833865815
Temps total en secondes: 4.906871795654297

Prédiction du sujet: 30
Meilleur: Acc: 0.6820512820512821
Temps total en secondes: 3.0558528900146484

Prédiction du sujet: 31
Meilleur: Acc: 0.7274826789838337
Temps total en secondes: 3.527992010116577

Prédiction du sujet: 32
Meilleur: Acc: 0.7124463519313304
Temps total en secondes: 3.7260470390319824

Prédiction du sujet: 33
Meilleur: Acc: 0.6129032258064516
Temps total en secondes: 3.954813003540039

Prédiction du sujet: 34
Meilleur: Acc: 0.6559322033898305
Temps total en secondes: 4.690351247787476

Prédiction du sujet: 35
Meilleur: Acc: 0.4882108183079057
Temps total en secondes: 5.822922468185425

Prédiction du sujet: 36
Meilleur: Acc: 0.7341772151898734
Temps total en secondes: 5.6443932056427

Prédiction du sujet: 37
Meilleur: Acc: 0.644880174291939
Temps total en secondes: 3.87461256980896

Prédiction du sujet: 38
Meilleur: Acc: 0.6741344195519349
Temps total en secondes: 4.205828666687012

Prédiction du sujet: 39
Meilleur: Acc: 0.5039619651347068
Temps total en secondes: 5.11771821975708

Prédiction du sujet: 40
Meilleur: Acc: 0.6816239316239316
Temps total en secondes: 3.738997459411621

Prédiction du sujet: 41
Meilleur: Acc: 0.6937354988399071
Temps total en secondes: 3.523041009902954

Prédiction du sujet: 42
Meilleur: Acc: 0.6937394247038917
Temps total en secondes: 4.983323097229004

Prédiction du sujet: 43
Meilleur: Acc: 0.6661016949152543
Temps total en secondes: 4.79532527923584

Prédiction du sujet: 44
Meilleur: Acc: 0.5947901591895803
Temps total en secondes: 5.607699632644653

Prédiction du sujet: 45
Meilleur: Acc: 0.6263982102908278
Temps total en secondes: 3.518599510192871

Prédiction du sujet: 46
Meilleur: Acc: 0.7712765957446809
Temps total en secondes: 3.034904956817627

Prédiction du sujet: 47
Meilleur: Acc: 0.613564668769716
Temps total en secondes: 5.378824234008789

Prédiction du sujet: 48
Meilleur: Acc: 0.7052469135802469
Temps total en secondes: 5.171278238296509

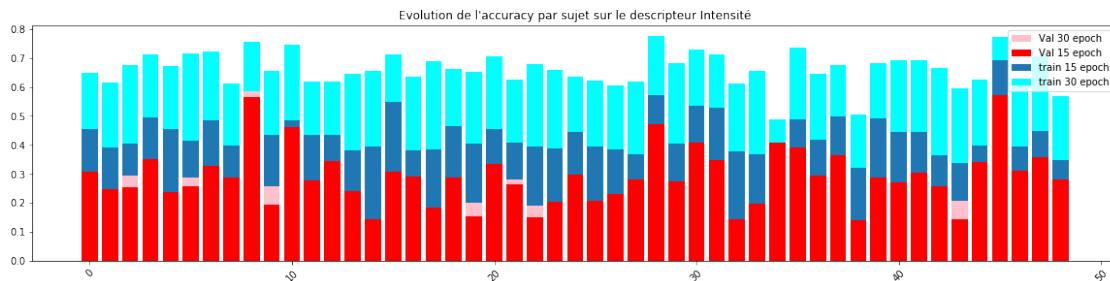
Prédiction du sujet: 49
Meilleur: Acc: 0.5680379746835443
Temps total en secondes: 5.153305768966675

```
[8]: plt.subplots(1,figsize=(16,4),constrained_layout=True)
d = plt.bar(np.arange(49),Subject_acc_Train_30,color=['cyan'])
c = plt.bar(np.arange(49),Subject_acc_Train_15)
a = plt.bar(np.arange(49),Subject_acc_Val_30,color=['pink'])
b = plt.bar(np.arange(49),Subject_acc_Val_15,color=['red'])
```

```

plt.title("Evolution de l'accuracy par sujet sur le descripteur Intensité")
plt.legend((a,b,c,d),('Val 30 epoch','Val 15 epoch','train 15 epoch','train 30epoch'))
plt.xticks(rotation=45)
plt.show()

```



```
[10]: plt.subplots(1,figsize=(16,4),constrained_layout=True)
```

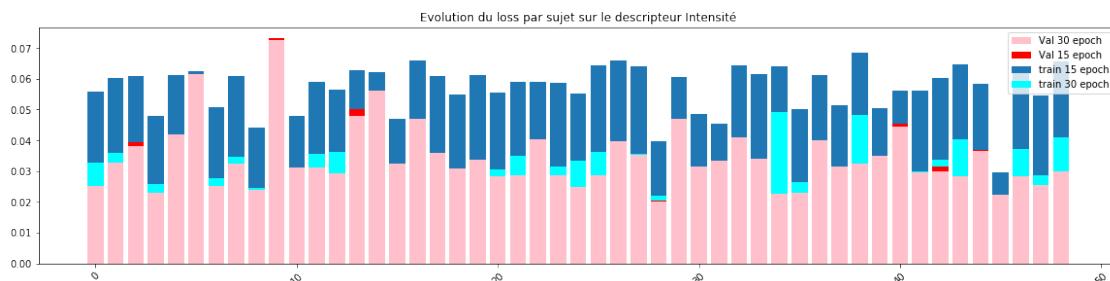
```

c = plt.bar(np.arange(49),Subject_loss_Train_15)
d = plt.bar(np.arange(49),Subject_loss_Train_30,color=['cyan'])
b = plt.bar(np.arange(49),Subject_loss_Val_15,color=['red'])
a = plt.bar(np.arange(49),Subject_loss_Val_30,color=['pink'])

plt.title("Evolution du loss par sujet sur le descripteur Intensité")
plt.legend((a,b,c,d),('Val 30 epoch','Val 15 epoch','train 15 epoch','train 30epoch'))
plt.xticks(rotation=45)

```

```
[10]: (array([-10., 0., 10., 20., 30., 40., 50., 60.]),  
<a list of 8 Text xticklabel objects>)
```



Les codes permettant de prédire le descripteur Valence et Sweet par sujet sont identiques à ceux pour intensité.

DREAM 1 NN Label

February 12, 2021

```
[7]: dfX = pd.read_csv('Molecular Dataset Dream 1.csv',sep=';')
dfY = pd.read_csv('Senteur Dataset Dream 1.csv',sep=';')

[8]: df = dfX.merge(dfY)

[9]: list_label = ['INTENSITY/STRENGTH', 'VALENCE/PLEASANTNESS', 'BAKERY', 'SWEET',
   'FRUIT', 'FISH', 'GARLIC', 'SPICES', 'COLD', 'SOUR', 'BURNT', 'ACID',
   'WARM', 'MUSKY', 'SWEATY', 'AMMONIA/URINOUS', 'DECAYED', 'WOOD',
   'GRASS', 'FLOWER', 'CHEMICAL']

#Pour les barplot
list_label_bar = ['INTENSITY', 'VALENCE', 'BAKERY', 'SWEET',
   'FRUIT', 'FISH', 'GARLIC', 'SPICES', 'COLD', 'SOUR', 'BURNT', 'ACID',
   'WARM', 'MUSKY', 'SWEATY', 'AMMONIA', 'DECAYED', 'WOOD',
   'GRASS', 'FLOWER', 'CHEMICAL']

Dico_labels = {}
for i in range (21):
    Dico_labels[i] = list_label[i]

[10]: class MyNetwork(nn.Module):

    def __init__(self):
        super(MyNetwork, self).__init__()

        ## Activation layer
        self.relu = nn.ReLU()

        self.fc1 = nn.Linear(in_features = 3083, out_features = 2400)
        self.fc2 = nn.Linear(2400, 1200)
        self.fc3 = nn.Linear(1200, 600)
        self.output = nn.Linear(600, 11)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):

        ## First full connection
        x = self.fc1(x)
```

```

x = self.relu(x)

## Second full connection
x = self.fc2(x)
x = self.relu(x)

## Third full connection
x = self.fc3(x)
x = self.relu(x)

## Output layer
x = self.output(x)
y = self.softmax(x)

return y

```

```

[ ]: Label_acc_Train_15, Label_acc_Train_30 = [], []
Label_acc_Val_15, Label_acc_Val_30 = [], []
Label_loss_Train_15, Label_loss_Train_30 = [], []
Label_loss_Val_15, Label_loss_Val_30 = [], []

for j in range(len(list_label)):

    print('Prédiction du label:', Dico_labels[j])
    dfY = df[[Dico_labels[j]]]
    dfX = df
    dfX = dfX.drop(list_label, axis=1)
    dfX = dfX.drop('Unnamed: 0', axis=1)
    dfX = dfX.drop('CID', axis=1)

    dataX = np.float32(dfX.values)
    dataY = np.longlong(dfY.values)
    dataY = to_class(dataY)
    dataX = sc.fit_transform(dataX)

    DATA_Train, DATA_Test, TARGET_Train, TARGET_Test = train_test_split(dataX, ↴
    ↴dataY , test_size=0.1)

    Batch=300
    X_train_tensor = torch.from_numpy(DATA_Train)
    Y_train_tensor = torch.from_numpy(TARGET_Train)

    X_test_tensor = torch.from_numpy(DATA_Test)
    Y_test_tensor = torch.from_numpy(TARGET_Test)

    train = data_utils.TensorDataset(X_train_tensor, Y_train_tensor)
    train_loader = data_utils.DataLoader(train, batch_size=Batch, shuffle=True)

```

```

test = data_utils.TensorDataset(X_test_tensor, Y_test_tensor)
test_loader = data_utils.DataLoader(test, batch_size=Batch, shuffle=True)

net = MyNetwork()
net = net.cuda()

LEARNING_RATE = 0.003
MOMENTUM = 0.9

criterion = nn.NLLLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=LEARNING_RATE, momentum=MOMENTUM)

N_EPOCHS = 30
#print('Début de l'entraînement sur', N_EPOCHS, 'Epoch')

epoch_loss, epoch_acc, epoch_val_loss, epoch_val_acc = [], [], [], []
start_time = time.time()

for e in range(N_EPOCHS):

    ### boucle d'entraînement
    running_loss = 0
    running_accuracy = 0
    running_acc=0
    start_epoch_time=time.time()

    ## Le réseau est mis en mode "entraînement"
    net.train()

    for i, batch in enumerate(train_loader):

        # Obtenir batch du dataloader
        x = batch[0]
        labels = batch[1]

        # déplacer le batch sur le GPU
        x = x.cuda()
        labels = labels.cuda()

        # Calcul de l'output et les loss
        output = net(x)
        y = output

        loss = criterion(y, torch.max(labels, 1)[1])

```

```

# Réinitialisation du gradients
optimizer.zero_grad()

# Calculs du gradients
loss.backward()

# Appliquecation d'une étape d'optimisation de l'algorithme de
# descente pour mettre à jour les poids
optimizer.step()

with torch.no_grad():
    running_loss += loss.item()
    running_accuracy += (y.max(1)[1] == torch.max(labels, 1)[1]).sum().item()

epoch_loss.append(running_loss/len(train))
epoch_acc.append(running_accuracy/len(train))

### Boucle de validation
## Le réseau est mis en mode validation
net.eval()
running_val_loss = 0
running_val_accuracy = 0

for i, batch in enumerate(test_loader):
    with torch.no_grad():
        x = batch[0]
        labels = batch[1]
        x = x.cuda()
        labels = labels.cuda()
        output = net(x)
        y = output

        loss = criterion(y, torch.max(labels, 1)[1])
        running_val_loss += loss.item()
        running_val_accuracy += (y.max(1)[1] == torch.max(labels, 1)[1]).sum().item()

    epoch_val_loss.append(running_val_loss/len(test))
    epoch_val_acc.append(running_val_accuracy/len(test))

if e == 14 :
    Label_acc_Train_15.append(maximum(epoch_acc))
    Label_acc_Val_15.append(maximum(epoch_val_acc))
    Label_loss_Train_15.append(minimum(epoch_loss))
    Label_loss_Val_15.append(minimum(epoch_val_loss))

```

```

if e == 29 :
    Label_acc_Train_30.append(maximum(epoch_acc))
    Label_acc_Val_30.append(maximum(epoch_val_acc))
    Label_loss_Train_30.append(minimum(epoch_loss))
    Label_loss_Val_30.append(minimum(epoch_val_loss))

#print('Dernière epoch: Acc:',running_accuracy/len(train))
interval = time.time() - start_time
print ('Best accuracy d entrainement',maximum(epoch_acc))
print ('Temps total en secondes:', interval)

def net
print( )

```

[17]:

```

plt.subplots(1,figsize=(16,4),constrained_layout=True)

d = plt.bar(list_label_bar,Label_acc_Train_30,color=['cyan'])
c = plt.bar(list_label_bar,Label_acc_Train_15)
a = plt.bar(list_label_bar,Label_acc_Val_30,color=['pink'])
b = plt.bar(list_label_bar,Label_acc_Val_15,color=['red'])

plt.title("Evolution de l'accuracy par label")
plt.legend((a,b,c,d),('Val 30 epoch','Val 15 epoch','train 15 epoch','train 30 epoch'))
plt.xticks(rotation=45)
plt.show()

```



[18]:

```

plt.subplots(1,figsize=(16,4),constrained_layout=True)

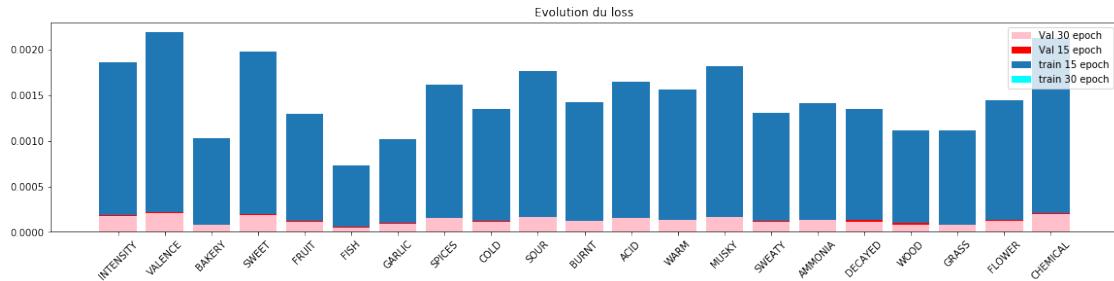
d = plt.bar(list_label_bar,Label_loss_Train_30,color=['cyan'])
c = plt.bar(list_label_bar,Label_loss_Train_15)
b = plt.bar(list_label_bar,Label_loss_Val_15,color=['red'])
a = plt.bar(list_label_bar,Label_loss_Val_30,color=['pink'])

plt.title("Evolution du loss")

```

```
plt.legend((a,b,c,d),('Val 30 epoch','Val 15 epoch','train 15 epoch','train 30 epoch'))
plt.xticks(rotation=45)
```

[18]: ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
<a list of 21 Text xticklabel objects>)



[]:

DREAM_1_CLEAN_BINAIRE

February 12, 2021

1 Préparation des données

Faire un tableau de best prediction de tous les descripteurs par rapport a celle bien predite par le random

```
[1]: # Import numerical libraries.
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import seaborn as sns
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder, MultiLabelBinarizer
from sklearn.metrics import accuracy_score, roc_curve, auc, roc_auc_score

#Librairies
import time
import torch
import random

#Raccourcis utilisé
from tqdm import tqdm_notebook
import torch.nn as nn
import torch.utils.data as data_utils
import pandas as pd

import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D
from tensorflow.keras.optimizers import SGD, RMSprop, Adam, Adadelta, Adamax, Nadam
from sklearn.model_selection import train_test_split
from math import exp
from itertools import cycle
```

Using TensorFlow backend.

```
[2]: dfX = pd.read_csv('Molecular Dataset Dream 1.csv',sep=';')
dfY = pd.read_csv('Senteur_Dataset_Binaire_Dream_1.csv',sep=';')
```

```
[3]: df = dfX.merge(dfY)
df
```

```
[3]:      Unnamed: 0      CID  complexity from pubmed      MW      AMW      Sv  \
0          0        126          4.532  4.961  2.011  2.155
1          1        126          4.532  4.961  2.011  2.155
2          2        126          4.532  4.961  2.011  2.155
3          3        126          4.532  4.961  2.011  2.155
4          4        126          4.532  4.961  2.011  2.155
...
29334      29334    16220109          ...  ...  ...  ...
29335      29335    16220109          5.867  5.670  1.804  2.579
29336      29336    16220109          5.867  5.670  1.804  2.579
29337      29337    16220109          5.867  5.670  1.804  2.579
29338      29338    16220109          5.867  5.670  1.804  2.579

      Se      Sp      Si      Mv  ...  ACID  WARM  MUSKY  SWEATY  \
0    2.482  2.168  2.554  0.873  ...  0.0  0.0  0.0  1.0
1    2.482  2.168  2.554  0.873  ...  0.0  0.0  0.0  0.0
2    2.482  2.168  2.554  0.873  ...  0.0  0.0  0.0  0.0
3    2.482  2.168  2.554  0.873  ...  0.0  0.0  0.0  0.0
4    2.482  2.168  2.554  0.873  ...  0.0  1.0  0.0  0.0
...
29334  3.128  2.657  3.275  0.821  ...  0.0  0.0  0.0  0.0
29335  3.128  2.657  3.275  0.821  ...  0.0  0.0  0.0  0.0
29336  3.128  2.657  3.275  0.821  ...  0.0  0.0  0.0  0.0
29337  3.128  2.657  3.275  0.821  ...  1.0  0.0  1.0  0.0
29338  3.128  2.657  3.275  0.821  ...  1.0  1.0  0.0  0.0

      AMMONIA/URINOUS  DECAYED  WOOD  GRASS  FLOWER  CHEMICAL
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
2            0.0       0.0     0.0     0.0     0.0       1.0
3            0.0       0.0     0.0     0.0     0.0       1.0
4            0.0       0.0     0.0     0.0     0.0       0.0
...
29334        0.0       0.0     1.0     0.0     0.0       0.0
29335        0.0       0.0     0.0     1.0     0.0       0.0
29336        0.0       0.0     0.0     0.0     0.0       0.0
29337        0.0       0.0     0.0     0.0     0.0       1.0
29338        0.0       0.0     0.0     0.0     0.0       1.0
```

[29339 rows x 3106 columns]

On remarque que la colonne ‘Unnamed 0’ qui sert à marier les 2 dataframe est complètement

fauisser. On se retrouve donc à prédire la classe ‘INTENSITY/STRENGTH’ avec les mauvais descripteurs.

```
[4]: list_labelY = ['subject','INTENSITY/STRENGTH']
list_label = ['INTENSITY/STRENGTH', 'VALENCE/PLEASANTNESS', 'BAKERY', 'SWEET',
    'FRUIT', 'FISH', 'GARLIC', 'SPICES', 'COLD', 'SOUR', 'BURNT', 'ACID',
    'WARM', 'MUSKY', 'SWEATY', 'AMMONIA/URINOUS', 'DECAYED', 'WOOD',
    'GRASS', 'FLOWER', 'CHEMICAL']
```

```
[5]: dfY = df[list_labelY]
dfX = df
dfX = dfX.drop(list_label,axis=1)
dfX = dfX.drop('Unnamed: 0',axis=1)
dfX = dfX.drop('CID',axis=1)
```

```
[6]: dfX
```

	complexity from pubmed	MW	AMW	Sv	Se	Sp	Si	\
0		4.532	4.961	2.011	2.155	2.482	2.168	2.554
1		4.532	4.961	2.011	2.155	2.482	2.168	2.554
2		4.532	4.961	2.011	2.155	2.482	2.168	2.554
3		4.532	4.961	2.011	2.155	2.482	2.168	2.554
4		4.532	4.961	2.011	2.155	2.482	2.168	2.554
...	
29334		5.867	5.670	1.804	2.579	3.128	2.657	3.275
29335		5.867	5.670	1.804	2.579	3.128	2.657	3.275
29336		5.867	5.670	1.804	2.579	3.128	2.657	3.275
29337		5.867	5.670	1.804	2.579	3.128	2.657	3.275
29338		5.867	5.670	1.804	2.579	3.128	2.657	3.275
	Mv	Me	Mp	...	Hypnotic-80	Hypnotic-50	Neoplastic-80	\
0	0.873	1.006	0.879	...	0	0	0	
1	0.873	1.006	0.879	...	0	0	0	
2	0.873	1.006	0.879	...	0	0	0	
3	0.873	1.006	0.879	...	0	0	0	
4	0.873	1.006	0.879	...	0	0	0	
...	
29334	0.821	0.995	0.845	...	1	0	1	
29335	0.821	0.995	0.845	...	1	0	1	
29336	0.821	0.995	0.845	...	1	0	1	
29337	0.821	0.995	0.845	...	1	0	1	
29338	0.821	0.995	0.845	...	1	0	1	
	Neoplastic-50	Infective-80	Infective-50	Compound Identifier	\			
0	0	0	0	126				
1	0	0	0	126				
2	0	0	0	126				
3	0	0	0	126				

```

4          0          0          0          126
...
...      ...
29334      0          1          0          6114390
29335      0          1          0          6114390
29336      0          1          0          6114390
29337      0          1          0          6114390
29338      0          1          0          6114390

    Intensity  Dilution  subject
0            0          1         1
1            1          0         1
2            0          1         2
3            1          0         2
4            0          1         3
...
...      ...
29334      1          1        38
29335      0          2        39
29336      1          1        39
29337      0          2        40
29338      1          1        40

[29339 rows x 3083 columns]

```

1.1 Test sur le réseau neuronal

```
[7]: dfY=dfY.drop('subject',axis=1)
```

```
[8]: def to_class(y_):
    res = np.zeros((len(y_), 2), dtype='q')
    for i in range(len(y_)):
        res[i][int(y_[i])] = 1
    return res
```

```
[9]: dataX = np.float32(dfX.values)
dataY = np.longlong(dfY.values)
dataY = to_class(dataY)
```

```
[10]: # Normalisation
sc = StandardScaler()
dataX = sc.fit_transform(dataX)
```

```
[11]: DATA_Train, DATA_Test, TARGET_Train, TARGET_Test = train_test_split(dataX, ↴
dataY ,test_size=0.2)
```

```
[14]: ## PyTorch is used to working with batches.
Batch=300
```

```

X_train_tensor = torch.from_numpy(DATA_Train)
Y_train_tensor = torch.from_numpy(TARGET_Train)

X_test_tensor = torch.from_numpy(DATA_Test)
Y_test_tensor = torch.from_numpy(TARGET_Test)

train = data_utils.TensorDataset(X_train_tensor, Y_train_tensor)
train_loader = data_utils.DataLoader(train, batch_size=Batch, shuffle=True)

test = data_utils.TensorDataset(X_test_tensor, Y_test_tensor)
test_loader = data_utils.DataLoader(test, batch_size=Batch, shuffle=True)

```

[15]: `class MyNetwork(nn.Module):`

```

def __init__(self):
    super(MyNetwork, self).__init__()

    ## Activation layer
    self.relu = nn.ReLU()

    self.fc1 = nn.Linear(in_features = 3083, out_features = 2400)
    self.fc2 = nn.Linear(2400, 1200)
    self.fc3 = nn.Linear(1200, 600)
    self.output = nn.Linear(600, 2)
    self.softmax = nn.LogSoftmax(dim=1)
    self.dr = nn.Dropout(0.3)

def forward(self, x):

    ## First full connection
    x = self.fc1(x)
    x = self.relu(x)
    x = self.dr(x)

    ## Second full connection
    x = self.fc2(x)
    x = self.relu(x)
    x = self.dr(x)

    ## Third full connection
    x = self.fc3(x)
    x = self.relu(x)
    x = self.dr(x)

    ## Output layer
    x = self.output(x)
    y = self.softmax(x)

```

```

    return y

[16]: X_train_tensor.shape

[16]: torch.Size([23471, 3083])

[17]: ## Create an instance of our network
net = MyNetwork()

[18]: net = MyNetwork()
net = net.cuda()

LEARNING_RATE = 0.003
MOMENTUM = 0.9

criterion = nn.BCELoss()
#criterion = nn.CrossEntropyLoss

# Méthode stochastique de descente du gradient
optimizer = torch.optim.SGD(net.parameters(), lr=LEARNING_RATE, momentum=MOMENTUM)

[19]: ## Nombre d'époque d'apprentissage
N_EPOCHS = 100

epoch_loss, epoch_acc, epoch_val_loss, epoch_val_acc = [], [], [], []
start_time = time.time()

for e in range(N_EPOCHS):

    print("EPOCH:", e)

    ### boucle d'entraînement
    running_loss = 0
    running_accuracy = 0
    running_acc=0
    start_epoch_time=time.time()

    ## Le réseau est mis en mode "entraînement"
    net.train()

    for i, batch in enumerate(tqdm_notebook(train_loader)):

        # Obtenir batch du dataloader
        x = batch[0]
        labels = batch[1]

```

```

# déplacer le batch sur le GPU
x = x.cuda()
labels = labels.cuda()

# Calcul de l'output et les loss
output = net(x)
y = output

loss = criterion(y, torch.max(labels, 1)[1])

# Réinitialisation du gradients
optimizer.zero_grad()

# Calculs du gradients
loss.backward()

# Appliquecation d'une étape d'optimisation de l'algorithme de descente
pour mettre à jour les poids
optimizer.step()

with torch.no_grad():
    running_loss += loss.item()
    running_accuracy += (y.max(1)[1] == torch.max(labels, 1)[1]).sum().
item()

print("Training accuracy:", running_accuracy/float(len(train)),
      "Training loss:", running_loss/float(len(train)))

epoch_loss.append(running_loss/len(train))
epoch_acc.append(running_accuracy/len(train))

### Boucle de valisation
## Le réseau est mis en mode validation
net.eval()

running_val_loss = 0
running_val_accuracy = 0

for i, batch in enumerate(tqdm_notebook(test_loader)):

    with torch.no_grad():

        x = batch[0]
        labels = batch[1]

        x = x.cuda()

```

```

    labels = labels.cuda()

    output = net(x)
    y = output

    loss = criterion(y, torch.max(labels, 1)[1])

    running_val_loss += loss.item()
    running_val_accuracy += (y.max(1)[1] == torch.max(labels, 1)[1]).sum().item()

    print("Validation accuracy:", running_val_accuracy/float(len(test)),
          "Validation loss:", running_val_loss/float(len(test)))

    epoch_val_loss.append(running_val_loss/len(test))
    epoch_val_acc.append(running_val_accuracy/len(test))

    inter = time.time() - start_epoch_time
    print ('Temps de 1 Epoch ',e,' en secondes:', inter )

interval = time.time() - start_time
print ('Temps total en secondes:', interval )

```

EPOCH: 0

```

C:\Users\benja\anaconda3\lib\site-packages\ipykernel_launcher.py:20:
TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))


```

Training accuracy: 0.6206382344169401 Training loss: 0.002245001146655318

```

C:\Users\benja\anaconda3\lib\site-packages\ipykernel_launcher.py:62:
TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))


```

```

Validation accuracy: 0.6274710293115201 Validation loss: 0.0022428325023794143
Temps de 1 Epoch 0 en secondes: 2.121151924133301
EPOCH: 1

```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

Training accuracy: 0.6247283882237655 Training loss: 0.002214275796657925

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.6273006134969326 Validation loss: 0.002230662225295511

Temps de l Epoch 1 en secondes: 1.4414172172546387

EPOCH: 2

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.6259639555195774 Training loss: 0.0022015850685476865

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.6295160190865713 Validation loss: 0.002216515162402298

Temps de l Epoch 2 en secondes: 1.4423062801361084

EPOCH: 3

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.627540368965958 Training loss: 0.002183052219279375

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.6313905930470347 Validation loss: 0.002192921992378651

Temps de l Epoch 3 en secondes: 1.449186086654663

EPOCH: 4

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.6312470708533936 Training loss: 0.002165773195653784

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.635480572597137 Validation loss: 0.0021673561402056275

Temps de l Epoch 4 en secondes: 1.4446170330047607

EPOCH: 5

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.6365301861872098 Training loss: 0.0021407213083475003

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.6508179959100204 Validation loss: 0.002147652518611094

Temps de l Epoch 5 en secondes: 1.4062042236328125

EPOCH: 6

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.6511865706616676 Training loss: 0.002105077547927693

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.6579754601226994 Validation loss: 0.002105835726168502

Temps de l Epoch 6 en secondes: 1.7054831981658936

EPOCH: 7

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.6649056282220612 Training loss: 0.0020572064625917737

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.676209952283572 Validation loss: 0.002061545554276787

Temps de l Epoch 7 en secondes: 1.4047327041625977

EPOCH: 8

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.6776447530995697 Training loss: 0.0020159364869102823

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.6840490797546013 Validation loss: 0.002007847904508912

Temps de l Epoch 8 en secondes: 1.3899998664855957

EPOCH: 9

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

```
Training accuracy: 0.69161944527289 Training loss: 0.001975926878627933
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.6992160872528971 Validation loss: 0.00196958095058748
Temps de l Epoch 9 en secondes: 1.400327205657959
EPOCH: 10
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.6984789740530868 Training loss: 0.0019452731171839127
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.6901840490797546 Validation loss: 0.001965822414549013
Temps de l Epoch 10 en secondes: 1.412438154220581
EPOCH: 11
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7049976566827149 Training loss: 0.0019160192027312816
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.6981935923653715 Validation loss: 0.0019441745824852847
Temps de l Epoch 11 en secondes: 2.117582321166992
EPOCH: 12
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7058071662903157 Training loss: 0.0019062264168462086
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7080777096114519 Validation loss: 0.0018972465901774618
Temps de l Epoch 12 en secondes: 1.3845481872558594
EPOCH: 13
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7093860508712879 Training loss: 0.0018968913228159022
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7091002044989775 Validation loss: 0.0019164561676832796
Temps de l Epoch 13 en secondes: 1.4017579555511475
EPOCH: 14
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7152230411997784 Training loss: 0.001882474398888258
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7142126789366053 Validation loss: 0.0019013116462505344
Temps de l Epoch 14 en secondes: 1.3817174434661865
EPOCH: 15
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7154786758127051 Training loss: 0.0018791939772306479
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7160872528970689 Validation loss: 0.0018977585014390067
Temps de l Epoch 15 en secondes: 1.3961775302886963
EPOCH: 16
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7166290315708747 Training loss: 0.0018682781665479871
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7130197682344922 Validation loss: 0.0019006673360894794
Temps de l Epoch 16 en secondes: 1.4007220268249512
EPOCH: 17
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.716714243108517 Training loss: 0.0018678075486294864
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7150647580095433 Validation loss: 0.0018822112880601473
```

```
Temps de l Epoch 17 en secondes: 1.388683795928955
```

```
EPOCH: 18
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7197392526948149 Training loss: 0.001857537143419682
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7177914110429447 Validation loss: 0.0018786207684547461
```

```
Temps de l Epoch 18 en secondes: 1.3874430656433105
```

```
EPOCH: 19
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.719355800775425 Training loss: 0.0018545708949471113
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7162576687116564 Validation loss: 0.0018868860474393412
```

```
Temps de l Epoch 19 en secondes: 1.409400224685669
```

```
EPOCH: 20
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7218695411358698 Training loss: 0.001844796693135596
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7147239263803681 Validation loss: 0.0018759318452833782
```

```
Temps de l Epoch 20 en secondes: 1.3867595195770264
```

```
EPOCH: 21
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7208470026841635 Training loss: 0.001849653311132506
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7099522835719154 Validation loss: 0.0018956620695644605
```

```
Temps de l Epoch 21 en secondes: 1.4164228439331055
```

```
EPOCH: 22
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7225512334370073 Training loss: 0.0018380105452281352
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7184730743012951 Validation loss: 0.0018784833364772731
```

```
Temps de l Epoch 22 en secondes: 1.403334617614746
```

```
EPOCH: 23
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7251075795662733 Training loss: 0.001835968724900459
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7150647580095433 Validation loss: 0.0018748908259873498
```

```
Temps de l Epoch 23 en secondes: 1.4415860176086426
```

```
EPOCH: 24
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7220825699799753 Training loss: 0.0018342342617352167
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7189843217450579 Validation loss: 0.0018836796872861222
```

```
Temps de l Epoch 24 en secondes: 1.5164988040924072
```

```
EPOCH: 25
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7280473776149291 Training loss: 0.0018341221935244912
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7215405589638718 Validation loss: 0.0018736371527989055
Temps de l Epoch 25 en secondes: 1.4300765991210938
EPOCH: 26
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7228920795875762 Training loss: 0.0018315303209343898
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7044989775051125 Validation loss: 0.0019194530989738573
Temps de l Epoch 26 en secondes: 1.4887645244598389
EPOCH: 27
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7225938392058284 Training loss: 0.001837543337860439
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7193251533742331 Validation loss: 0.0018711890606304864
Temps de l Epoch 27 en secondes: 1.493562936782837
EPOCH: 28
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7270674449320438 Training loss: 0.0018237252970039655
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7198364008179959 Validation loss: 0.001869460546247148
Temps de l Epoch 28 en secondes: 1.4294779300689697
EPOCH: 29
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7244684930339568 Training loss: 0.0018270552313265317
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7174505794137696 Validation loss: 0.0018782901800483282
Temps de l Epoch 29 en secondes: 1.3687186241149902
EPOCH: 30
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7250649737974522 Training loss: 0.001826217550020753
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7162576687116564 Validation loss: 0.0018842708073104411
Temps de l Epoch 30 en secondes: 1.4252333641052246
EPOCH: 31
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7264709641685484 Training loss: 0.0018242351375264692
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7148943421949557 Validation loss: 0.0018852922631187672
Temps de l Epoch 31 en secondes: 1.4003427028656006
EPOCH: 32
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7297942141365941 Training loss: 0.0018137306410484872
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7152351738241309 Validation loss: 0.0018782781585593897
Temps de l Epoch 32 en secondes: 1.3885445594787598
EPOCH: 33
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7276213199267181 Training loss: 0.001819447819991886
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7222222222222222 Validation loss: 0.0018650516816849764
Temps de l Epoch 33 en secondes: 1.383819341659546
EPOCH: 34

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.726726598781475 Training loss: 0.0018199684102251535
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7261417859577369 Validation loss: 0.001865021356247523
Temps de l Epoch 34 en secondes: 1.4579157829284668
EPOCH: 35

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7300924545183418 Training loss: 0.0018065464427538102
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7176209952283572 Validation loss: 0.0018669802698052164
Temps de l Epoch 35 en secondes: 1.4316346645355225
EPOCH: 36

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7304759064377316 Training loss: 0.001807156554231818
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7213701431492843 Validation loss: 0.001875855470483776
Temps de l Epoch 36 en secondes: 1.446256160736084
EPOCH: 37

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7305185122065527 Training loss: 0.0018115310541767066
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7215405589638718 Validation loss: 0.0018707472467877544
Temps de l Epoch 37 en secondes: 1.416534185409546
EPOCH: 38
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7293255506795621 Training loss: 0.0018078521935881177
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7211997273346966 Validation loss: 0.0018585129617507065
Temps de l Epoch 38 en secondes: 1.4036545753479004
EPOCH: 39
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7283456179966767 Training loss: 0.0018083078538415115
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7203476482617587 Validation loss: 0.001880481971228456
Temps de l Epoch 39 en secondes: 1.3872103691101074
EPOCH: 40
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7328618294917132 Training loss: 0.001799423847691759
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7217109747784595 Validation loss: 0.0018620989358417014
Temps de l Epoch 40 en secondes: 1.3625411987304688
EPOCH: 41
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7305611179753738 Training loss: 0.001809517098531609
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7230743012951601 Validation loss: 0.0018723610008989943
Temps de l Epoch 41 en secondes: 1.4027984142303467
EPOCH: 42
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7320949256529334 Training loss: 0.0017954593033181311
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7230743012951601 Validation loss: 0.00186539519052564
Temps de l Epoch 42 en secondes: 1.4023191928863525
EPOCH: 43
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7316688679647224 Training loss: 0.0018030128416633745
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7201772324471711 Validation loss: 0.0018648642388914586
Temps de l Epoch 43 en secondes: 1.401014804840088
EPOCH: 44
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.731498444889438 Training loss: 0.0017996242321536608
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7246080436264485 Validation loss: 0.0018648911107537205
Temps de l Epoch 44 en secondes: 1.4142086505889893
EPOCH: 45
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.731072387201227 Training loss: 0.0018036470885656906
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7249488752556237 Validation loss: 0.0018674146279885643
```

```
Temps de l Epoch 45 en secondes: 1.4508943557739258
```

```
EPOCH: 46
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7314132333517959 Training loss: 0.0017983142390817367
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7188139059304703 Validation loss: 0.0018688594819577925
```

```
Temps de l Epoch 46 en secondes: 1.4510784149169922
```

```
EPOCH: 47
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7320523198841123 Training loss: 0.001795657047967999
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7234151329243353 Validation loss: 0.0018564538183306542
```

```
Temps de l Epoch 47 en secondes: 1.380979299545288
```

```
EPOCH: 48
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7327340121852499 Training loss: 0.0017916047346058146
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7244376278118609 Validation loss: 0.001854245947710738
```

```
Temps de l Epoch 48 en secondes: 1.3957939147949219
```

```
EPOCH: 49
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7316262621959013 Training loss: 0.0017967830731868643
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7247784594410361 Validation loss: 0.0018548781602010487
Temps de l Epoch 49 en secondes: 1.538508415222168
EPOCH: 50
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7320949256529334 Training loss: 0.0017907464960336451
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7225630538513974 Validation loss: 0.0018591260424502137
Temps de l Epoch 50 en secondes: 1.3635728359222412
EPOCH: 51
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7314132333517959 Training loss: 0.0017931912032295555
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7240967961826857 Validation loss: 0.0018611935506325315
Temps de l Epoch 51 en secondes: 1.417417287826538
EPOCH: 52
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.732776617954071 Training loss: 0.0017861177906972375
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7232447171097478 Validation loss: 0.001860205411667229
Temps de l Epoch 52 en secondes: 1.33388352394104
EPOCH: 53
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.733671339099314 Training loss: 0.0017920644237786807
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7239263803680982 Validation loss: 0.0018617505259715381
```

```
Temps de l Epoch 53 en secondes: 1.3783760070800781
```

```
EPOCH: 54
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7367389544544332 Training loss: 0.001787436228882118
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7230743012951601 Validation loss: 0.0018535250189683026
```

```
Temps de l Epoch 54 en secondes: 1.356717824935913
```

```
EPOCH: 55
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7334583102552086 Training loss: 0.0017929806023522508
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7242672119972734 Validation loss: 0.0018516788189315277
```

```
Temps de l Epoch 55 en secondes: 1.3590855598449707
```

```
EPOCH: 56
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7335009160240297 Training loss: 0.001784124058374055
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7240967961826857 Validation loss: 0.001855503196365263
```

```
Temps de l Epoch 56 en secondes: 1.3596625328063965
```

```
EPOCH: 57
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

Training accuracy: 0.7372502236802864 Training loss: 0.0017893659327485362

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7286980231765507 Validation loss: 0.0018439764777819316

Temps de l Epoch 57 en secondes: 1.3764417171478271

EPOCH: 58

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7332026756422819 Training loss: 0.0017853063056214099

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.721881390593047 Validation loss: 0.0018736603679344942

Temps de l Epoch 58 en secondes: 1.3622000217437744

EPOCH: 59

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7344382429380938 Training loss: 0.0017853087359245522

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7244376278118609 Validation loss: 0.0018622539556587157

Temps de l Epoch 59 en secondes: 1.362657070159912

EPOCH: 60

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7369519832985386 Training loss: 0.0017834148326835747

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7252897068847989 Validation loss: 0.0018524568586342986

Temps de l Epoch 60 en secondes: 1.3578298091888428

EPOCH: 61

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

```
Training accuracy: 0.7359294448468323 Training loss: 0.0017775709249860442
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7269938650306749 Validation loss: 0.0018596327377974622
Temps de l Epoch 61 en secondes: 1.4757647514343262
EPOCH: 62
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7367389544544332 Training loss: 0.0017816555382705538
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7271642808452624 Validation loss: 0.0018627833227113388
Temps de l Epoch 62 en secondes: 1.4333465099334717
EPOCH: 63
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.732776617954071 Training loss: 0.0017802805339547012
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7263122017723245 Validation loss: 0.0018556795115363621
Temps de l Epoch 63 en secondes: 1.4192688465118408
EPOCH: 64
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7368667717608964 Training loss: 0.0017762940101937214
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7275051124744376 Validation loss: 0.001853962393951156
Temps de l Epoch 64 en secondes: 1.3935267925262451
EPOCH: 65
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

Training accuracy: 0.7369093775297175 Training loss: 0.0017744126571702244

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7258009543285617 Validation loss: 0.0018546560343707917

Temps de l Epoch 65 en secondes: 1.423992395401001

EPOCH: 66

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7356312044650846 Training loss: 0.001781689048265447

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.71813224267212 Validation loss: 0.0018865570232993337

Temps de l Epoch 66 en secondes: 1.4373629093170166

EPOCH: 67

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7352903583145158 Training loss: 0.0017845693460326897

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7275051124744376 Validation loss: 0.0018615679283408625

Temps de l Epoch 67 en secondes: 1.4327378273010254

EPOCH: 68

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7344382429380938 Training loss: 0.0017805364662073868

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7249488752556237 Validation loss: 0.00187403900166637

Temps de l Epoch 68 en secondes: 1.4123950004577637

EPOCH: 69

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

```
Training accuracy: 0.7352477525456946 Training loss: 0.0017807574384096998
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7251192910702113 Validation loss: 0.0018600728553253205
Temps de l Epoch 69 en secondes: 1.427295446395874
EPOCH: 70
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7371650121426441 Training loss: 0.0017736383910315458
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7297205180640763 Validation loss: 0.0018518202326780447
Temps de l Epoch 70 en secondes: 1.3450593948364258
EPOCH: 71
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7364833198415065 Training loss: 0.0017729405169137362
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.728186775732788 Validation loss: 0.001856544175030995
Temps de l Epoch 71 en secondes: 1.3788464069366455
EPOCH: 72
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7380597332878872 Training loss: 0.0017733445046559642
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7191547375596455 Validation loss: 0.0018725628217386404
Temps de l Epoch 72 en secondes: 1.3550746440887451
EPOCH: 73
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7363981083038643 Training loss: 0.0017694070364496317
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7240967961826857 Validation loss: 0.0018661080998633292
```

```
Temps de l Epoch 73 en secondes: 1.3848681449890137
```

```
EPOCH: 74
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7357590217715478 Training loss: 0.001773329887284087
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7217109747784595 Validation loss: 0.001890319891543151
```

```
Temps de l Epoch 74 en secondes: 1.397242546081543
```

```
EPOCH: 75
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7389970602019513 Training loss: 0.0017744827982076193
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7242672119972734 Validation loss: 0.0018726687753940002
```

```
Temps de l Epoch 75 en secondes: 1.3101725578308105
```

```
EPOCH: 76
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7393805121213413 Training loss: 0.0017754680017075801
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7258009543285617 Validation loss: 0.0018581990368445878
```

```
Temps de l Epoch 76 en secondes: 1.3390696048736572
```

```
EPOCH: 77
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7395083294278045 Training loss: 0.0017747035723287984
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.721881390593047 Validation loss: 0.0018712721597441378
Temps de l Epoch 77 en secondes: 1.3164772987365723
EPOCH: 78
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7387414255890248 Training loss: 0.0017674769580067107
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7271642808452624 Validation loss: 0.0018527540590946741
Temps de l Epoch 78 en secondes: 1.3133065700531006
EPOCH: 79
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7380597332878872 Training loss: 0.0017699958783310174
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7254601226993865 Validation loss: 0.001857427838267968
Temps de l Epoch 79 en secondes: 1.3382885456085205
EPOCH: 80
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7384005794384559 Training loss: 0.001772343481330064
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7310838445807771 Validation loss: 0.0018598088041079962
Temps de l Epoch 80 en secondes: 1.3315205574035645
EPOCH: 81
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7395509351966256 Training loss: 0.0017664236615774888
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7252897068847989 Validation loss: 0.0018667121657158293
```

```
Temps de l Epoch 81 en secondes: 1.3385100364685059
```

```
EPOCH: 82
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7383579736696348 Training loss: 0.001766924728121564
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7286980231765507 Validation loss: 0.001855106695460558
```

```
Temps de l Epoch 82 en secondes: 1.3425648212432861
```

```
EPOCH: 83
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7391248775084146 Training loss: 0.0017614411401645047
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7242672119972734 Validation loss: 0.0018635509001127945
```

```
Temps de l Epoch 83 en secondes: 1.2994840145111084
```

```
EPOCH: 84
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7398491755783733 Training loss: 0.0017618707260840295
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7237559645535105 Validation loss: 0.0018590141161412873
```

```
Temps de l Epoch 84 en secondes: 1.3572347164154053
```

```
EPOCH: 85
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7386136082825615 Training loss: 0.0017633410112345145
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7240967961826857 Validation loss: 0.0018622424623636299
```

```
Temps de l Epoch 85 en secondes: 1.3115851879119873
```

```
EPOCH: 86
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7398917813471944 Training loss: 0.0017673513529820881
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7232447171097478 Validation loss: 0.001866218736160346
```

```
Temps de l Epoch 86 en secondes: 1.317847490310669
```

```
EPOCH: 87
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7401900217289421 Training loss: 0.0017610817117891355
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7269938650306749 Validation loss: 0.001858431619897373
```

```
Temps de l Epoch 87 en secondes: 1.3829319477081299
```

```
EPOCH: 88
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7391674832772357 Training loss: 0.001771031581092351
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7259713701431493 Validation loss: 0.0018679626087957121
```

```
Temps de l Epoch 88 en secondes: 1.3763251304626465
```

```
EPOCH: 89
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

Training accuracy: 0.7372076179114652 Training loss: 0.0017646015906820493

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7261417859577369 Validation loss: 0.0018647313625859661

Temps de l Epoch 89 en secondes: 1.404975175857544

EPOCH: 90

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7395083294278045 Training loss: 0.0017617268760093288

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7285276073619632 Validation loss: 0.0018547773462781795

Temps de l Epoch 90 en secondes: 1.3551506996154785

EPOCH: 91

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7401900217289421 Training loss: 0.001762461815424434

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7269938650306749 Validation loss: 0.00185906003853376

Temps de l Epoch 91 en secondes: 1.3290550708770752

EPOCH: 92

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

Training accuracy: 0.7398917813471944 Training loss: 0.0017570878983884802

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Validation accuracy: 0.7278459441036128 Validation loss: 0.001854386995784588

Temps de l Epoch 92 en secondes: 1.3457293510437012

EPOCH: 93

HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))

```
Training accuracy: 0.740573473648332 Training loss: 0.0017658624240797557
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7275051124744376 Validation loss: 0.0018503646878620843
```

```
Temps de l Epoch 93 en secondes: 1.373091220855713
```

```
EPOCH: 94
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7411699544118273 Training loss: 0.001760415119253463
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7278459441036128 Validation loss: 0.0018506511568454308
```

```
Temps de l Epoch 94 en secondes: 1.4790213108062744
```

```
EPOCH: 95
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7404882621106897 Training loss: 0.0017670154695968788
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7240967961826857 Validation loss: 0.0018685272124728623
```

```
Temps de l Epoch 95 en secondes: 1.3655962944030762
```

```
EPOCH: 96
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7413829832559329 Training loss: 0.0017601600542406133
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7314246762099523 Validation loss: 0.0018585720686743515
```

```
Temps de l Epoch 96 en secondes: 1.4011344909667969
```

```
EPOCH: 97
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7382727621319927 Training loss: 0.0017565814734957844
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.728186775732788 Validation loss: 0.0018686125614891754
```

```
Temps de l Epoch 97 en secondes: 1.4343667030334473
```

```
EPOCH: 98
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.739252694814878 Training loss: 0.0017585183946260794
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7261417859577369 Validation loss: 0.0018514858554964904
```

```
Temps de l Epoch 98 en secondes: 1.3651933670043945
```

```
EPOCH: 99
```

```
HBox(children=(FloatProgress(value=0.0, max=79.0), HTML(value='')))
```

```
Training accuracy: 0.7400195986536577 Training loss: 0.0017561738364895905
```

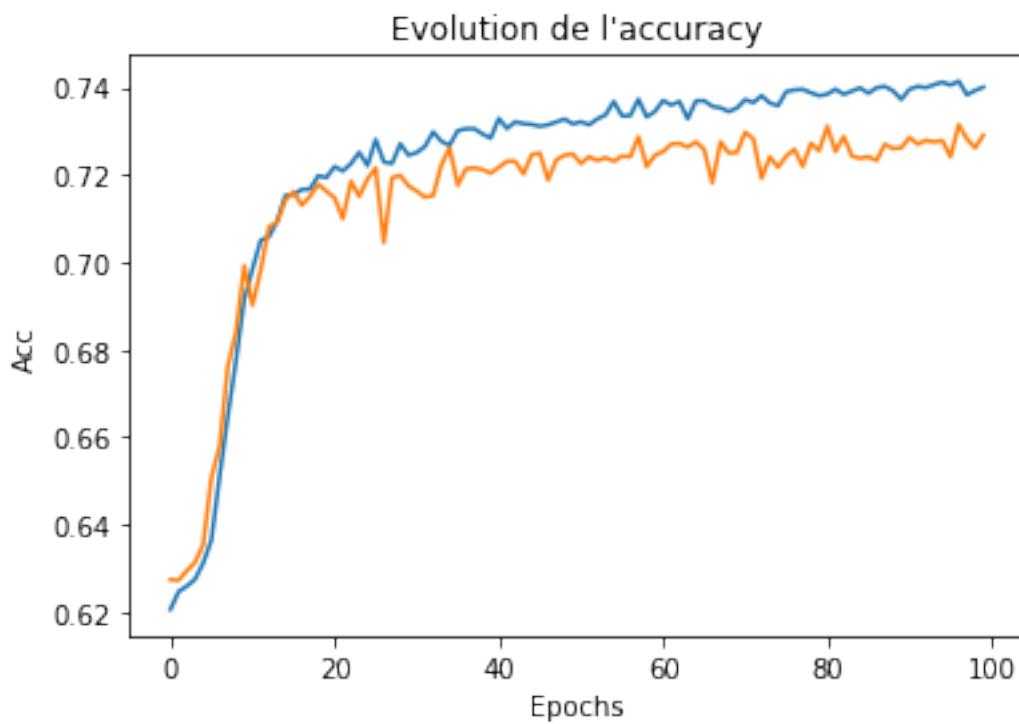
```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))
```

```
Validation accuracy: 0.7290388548057259 Validation loss: 0.001851585633346802
```

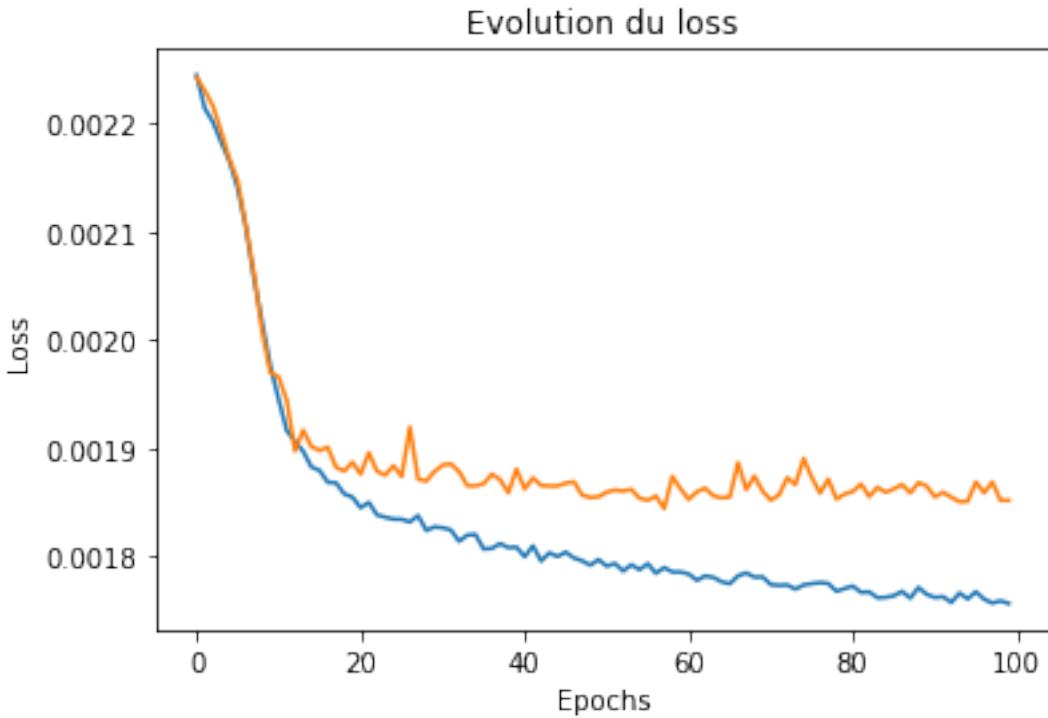
```
Temps de l Epoch 99 en secondes: 1.3897426128387451
```

```
Temps total en secondes: 141.2752423286438
```

```
[20]: plt.plot(np.arange(0,N_EPOCHS),epoch_acc)
plt.plot(np.arange(0,N_EPOCHS),epoch_val_acc)
plt.title("Evolution de l'accuracy")
plt.xlabel("Epochs")
plt.ylabel("Acc")
plt.show()
```



```
[21]: plt.plot(np.arange(0,N_EPOCHS),epoch_loss)
plt.plot(np.arange(0,N_EPOCHS),epoch_val_loss)
plt.title("Evolution du loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```



2 Keras

```
[12]: X_train = dataX #features
Y_train = dataY #labels
```

```
[13]: x_train,x_test,y_train,y_test = train_test_split(X_train,Y_train,test_size=0.2)
#y_train = to_class(y_train)
#y_test = to_class(y_test)
```

```
[14]: model = Sequential()
model.add(Dense(2400, activation='relu', input_dim=3083))
model.add(Dropout(0.3))
model.add(Dense(1200, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(600, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(2, activation='softmax'))
```

WARNING:tensorflow:From C:\Users\benja\anaconda3\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.
Instructions for updating:

If using Keras pass *_constraint arguments to layers.

```
[15]: adam = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07, amsgrad=False)
```

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

```
[ ]: #epochs = 30
      learning_rate = 0.1
      decay_rate = learning_rate / epochs
      momentum = 0.8
      sgd = SGD(lr=learning_rate, momentum=momentum, decay=decay_rate, nesterov=False)
      model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

```
[17]: history = model.fit(x_train, y_train, epochs=100, batch_size=300, validation_data=(x_test,y_test), shuffle=True)
      model.test_on_batch(x_test, y_test)
      model.metrics_names
```

WARNING:tensorflow:From C:\Users\benja\anaconda3\lib\site-packages\tensorflow_core\python\ops\math_grad.py:1424: where (from tensorflow.python.ops.array_ops) is deprecated and will be removed in a future version.

Instructions for updating:

Use tf.where in 2.0, which has the same broadcast rule as np.where

Train on 23471 samples, validate on 5868 samples

Epoch 1/100

23471/23471 [=====] - 7s 279us/sample - loss: 1.9945 - acc: 0.5854 - val_loss: 0.6695 - val_acc: 0.6178

Epoch 2/100

23471/23471 [=====] - 6s 263us/sample - loss: 0.6673 - acc: 0.6215 - val_loss: 0.6613 - val_acc: 0.6184

Epoch 3/100

23471/23471 [=====] - 6s 276us/sample - loss: 0.6563 - acc: 0.6248 - val_loss: 0.6519 - val_acc: 0.6207

Epoch 4/100

23471/23471 [=====] - 6s 255us/sample - loss: 0.6464 - acc: 0.6301 - val_loss: 0.6469 - val_acc: 0.6244

Epoch 5/100

23471/23471 [=====] - 7s 282us/sample - loss: 0.6371 - acc: 0.6372 - val_loss: 0.6363 - val_acc: 0.6396

Epoch 6/100

23471/23471 [=====] - 7s 281us/sample - loss: 0.6218 - acc: 0.6531 - val_loss: 0.6066 - val_acc: 0.6634

Epoch 7/100

23471/23471 [=====] - 6s 275us/sample - loss: 0.6091 -

```
acc: 0.6650 - val_loss: 0.5959 - val_acc: 0.6794
Epoch 8/100
23471/23471 [=====] - 6s 270us/sample - loss: 0.5931 -
acc: 0.6845 - val_loss: 0.5824 - val_acc: 0.6927
Epoch 9/100
23471/23471 [=====] - 6s 269us/sample - loss: 0.5842 -
acc: 0.6950 - val_loss: 0.5864 - val_acc: 0.6854
Epoch 10/100
23471/23471 [=====] - 6s 276us/sample - loss: 0.5772 -
acc: 0.6990 - val_loss: 0.5908 - val_acc: 0.6837
Epoch 11/100
23471/23471 [=====] - 7s 281us/sample - loss: 0.5760 -
acc: 0.7036 - val_loss: 0.5859 - val_acc: 0.6878
Epoch 12/100
23471/23471 [=====] - 7s 279us/sample - loss: 0.5764 -
acc: 0.6992 - val_loss: 0.5729 - val_acc: 0.6996
Epoch 13/100
23471/23471 [=====] - 6s 268us/sample - loss: 0.5701 -
acc: 0.7096 - val_loss: 0.5782 - val_acc: 0.6905
Epoch 14/100
23471/23471 [=====] - 6s 269us/sample - loss: 0.5703 -
acc: 0.7070 - val_loss: 0.5793 - val_acc: 0.6939
Epoch 15/100
23471/23471 [=====] - 6s 266us/sample - loss: 0.5656 -
acc: 0.7115 - val_loss: 0.5744 - val_acc: 0.6869
Epoch 16/100
23471/23471 [=====] - 6s 273us/sample - loss: 0.5665 -
acc: 0.7112 - val_loss: 0.5659 - val_acc: 0.7064
Epoch 17/100
23471/23471 [=====] - 6s 271us/sample - loss: 0.5642 -
acc: 0.7098 - val_loss: 0.5813 - val_acc: 0.6929
Epoch 18/100
23471/23471 [=====] - 7s 279us/sample - loss: 0.5645 -
acc: 0.7118 - val_loss: 0.5700 - val_acc: 0.6939
Epoch 19/100
23471/23471 [=====] - 6s 274us/sample - loss: 0.5610 -
acc: 0.7113 - val_loss: 0.5681 - val_acc: 0.6973
Epoch 20/100
23471/23471 [=====] - 7s 278us/sample - loss: 0.5615 -
acc: 0.7170 - val_loss: 0.5645 - val_acc: 0.7043
Epoch 21/100
23471/23471 [=====] - 7s 279us/sample - loss: 0.5575 -
acc: 0.7158 - val_loss: 0.5685 - val_acc: 0.6997
Epoch 22/100
23471/23471 [=====] - 6s 275us/sample - loss: 0.5619 -
acc: 0.7155 - val_loss: 0.5694 - val_acc: 0.7105
Epoch 23/100
23471/23471 [=====] - 7s 281us/sample - loss: 0.5591 -
```

```
acc: 0.7153 - val_loss: 0.5642 - val_acc: 0.7001
Epoch 24/100
23471/23471 [=====] - 6s 267us/sample - loss: 0.5582 -
acc: 0.7133 - val_loss: 0.5657 - val_acc: 0.6980
Epoch 25/100
23471/23471 [=====] - 6s 273us/sample - loss: 0.5548 -
acc: 0.7175 - val_loss: 0.5605 - val_acc: 0.7081
Epoch 26/100
23471/23471 [=====] - 6s 275us/sample - loss: 0.5540 -
acc: 0.7192 - val_loss: 0.5643 - val_acc: 0.7115
Epoch 27/100
23471/23471 [=====] - 6s 275us/sample - loss: 0.5547 -
acc: 0.7179 - val_loss: 0.5576 - val_acc: 0.7091
Epoch 28/100
23471/23471 [=====] - 6s 272us/sample - loss: 0.5531 -
acc: 0.7197 - val_loss: 0.5612 - val_acc: 0.7149
Epoch 29/100
23471/23471 [=====] - 6s 269us/sample - loss: 0.5509 -
acc: 0.7211 - val_loss: 0.5771 - val_acc: 0.7025
Epoch 30/100
23471/23471 [=====] - 6s 267us/sample - loss: 0.5549 -
acc: 0.7217 - val_loss: 0.5570 - val_acc: 0.7052
Epoch 31/100
23471/23471 [=====] - 6s 271us/sample - loss: 0.5474 -
acc: 0.7207 - val_loss: 0.5626 - val_acc: 0.7180
Epoch 32/100
23471/23471 [=====] - 6s 267us/sample - loss: 0.5510 -
acc: 0.7219 - val_loss: 0.5625 - val_acc: 0.7106
Epoch 33/100
23471/23471 [=====] - 6s 272us/sample - loss: 0.5512 -
acc: 0.7228 - val_loss: 0.5602 - val_acc: 0.7072
Epoch 34/100
23471/23471 [=====] - 6s 271us/sample - loss: 0.5494 -
acc: 0.7229 - val_loss: 0.5576 - val_acc: 0.7146
Epoch 35/100
23471/23471 [=====] - 6s 266us/sample - loss: 0.5534 -
acc: 0.7203 - val_loss: 0.5579 - val_acc: 0.7132
Epoch 36/100
23471/23471 [=====] - 6s 275us/sample - loss: 0.5489 -
acc: 0.7223 - val_loss: 0.5682 - val_acc: 0.7069
Epoch 37/100
23471/23471 [=====] - 6s 271us/sample - loss: 0.5532 -
acc: 0.7197 - val_loss: 0.5586 - val_acc: 0.7047
Epoch 38/100
23471/23471 [=====] - 7s 277us/sample - loss: 0.5498 -
acc: 0.7214 - val_loss: 0.5563 - val_acc: 0.7142
Epoch 39/100
23471/23471 [=====] - 6s 267us/sample - loss: 0.5475 -
```

```
acc: 0.7237 - val_loss: 0.5610 - val_acc: 0.7030
Epoch 40/100
23471/23471 [=====] - 6s 270us/sample - loss: 0.5485 -
acc: 0.7205 - val_loss: 0.5665 - val_acc: 0.7062
Epoch 41/100
23471/23471 [=====] - 6s 272us/sample - loss: 0.5437 -
acc: 0.7260 - val_loss: 0.5589 - val_acc: 0.7028
Epoch 42/100
23471/23471 [=====] - 6s 272us/sample - loss: 0.5467 -
acc: 0.7240 - val_loss: 0.5570 - val_acc: 0.7098
Epoch 43/100
23471/23471 [=====] - 6s 270us/sample - loss: 0.5464 -
acc: 0.7240 - val_loss: 0.5589 - val_acc: 0.7122
Epoch 44/100
23471/23471 [=====] - 6s 268us/sample - loss: 0.5452 -
acc: 0.7243 - val_loss: 0.5622 - val_acc: 0.7137
Epoch 45/100
23471/23471 [=====] - 6s 269us/sample - loss: 0.5419 -
acc: 0.7295 - val_loss: 0.5544 - val_acc: 0.7117
Epoch 46/100
23471/23471 [=====] - 6s 270us/sample - loss: 0.5468 -
acc: 0.7236 - val_loss: 0.5663 - val_acc: 0.7105
Epoch 47/100
23471/23471 [=====] - 6s 273us/sample - loss: 0.5448 -
acc: 0.7257 - val_loss: 0.5545 - val_acc: 0.7149
Epoch 48/100
23471/23471 [=====] - 6s 269us/sample - loss: 0.5499 -
acc: 0.7233 - val_loss: 0.5715 - val_acc: 0.7074
Epoch 49/100
23471/23471 [=====] - 6s 271us/sample - loss: 0.5446 -
acc: 0.7226 - val_loss: 0.5801 - val_acc: 0.7001
Epoch 50/100
23471/23471 [=====] - 6s 266us/sample - loss: 0.5447 -
acc: 0.7266 - val_loss: 0.5625 - val_acc: 0.7086
Epoch 51/100
23471/23471 [=====] - 6s 274us/sample - loss: 0.5526 -
acc: 0.7216 - val_loss: 0.5739 - val_acc: 0.7094
Epoch 52/100
23471/23471 [=====] - 6s 277us/sample - loss: 0.5448 -
acc: 0.7249 - val_loss: 0.5583 - val_acc: 0.7094
Epoch 53/100
23471/23471 [=====] - 7s 278us/sample - loss: 0.5434 -
acc: 0.7240 - val_loss: 0.5611 - val_acc: 0.7106
Epoch 54/100
23471/23471 [=====] - 7s 286us/sample - loss: 0.5405 -
acc: 0.7303 - val_loss: 0.5560 - val_acc: 0.7082
Epoch 55/100
23471/23471 [=====] - 7s 289us/sample - loss: 0.5435 -
```

```
acc: 0.7252 - val_loss: 0.5548 - val_acc: 0.7123
Epoch 56/100
23471/23471 [=====] - 7s 313us/sample - loss: 0.5460 -
acc: 0.7248 - val_loss: 0.5645 - val_acc: 0.7110
Epoch 57/100
23471/23471 [=====] - 7s 311us/sample - loss: 0.5479 -
acc: 0.7241 - val_loss: 0.5555 - val_acc: 0.7180
Epoch 58/100
23471/23471 [=====] - 8s 326us/sample - loss: 0.5386 -
acc: 0.7289 - val_loss: 0.5557 - val_acc: 0.7147
Epoch 59/100
23471/23471 [=====] - 8s 321us/sample - loss: 0.5387 -
acc: 0.7282 - val_loss: 0.5548 - val_acc: 0.7146
Epoch 60/100
23471/23471 [=====] - 7s 313us/sample - loss: 0.5386 -
acc: 0.7314 - val_loss: 0.5546 - val_acc: 0.7113
Epoch 61/100
23471/23471 [=====] - 7s 309us/sample - loss: 0.5411 -
acc: 0.7280 - val_loss: 0.5660 - val_acc: 0.7146
Epoch 62/100
23471/23471 [=====] - 7s 314us/sample - loss: 0.5439 -
acc: 0.7255 - val_loss: 0.5744 - val_acc: 0.7110
Epoch 63/100
23471/23471 [=====] - 7s 307us/sample - loss: 0.5413 -
acc: 0.7284 - val_loss: 0.5637 - val_acc: 0.7166
Epoch 64/100
23471/23471 [=====] - 8s 322us/sample - loss: 0.5445 -
acc: 0.7291 - val_loss: 0.5611 - val_acc: 0.7157
Epoch 65/100
23471/23471 [=====] - 8s 323us/sample - loss: 0.5396 -
acc: 0.7285 - val_loss: 0.5610 - val_acc: 0.7120
Epoch 66/100
23471/23471 [=====] - 7s 317us/sample - loss: 0.5452 -
acc: 0.7254 - val_loss: 0.5635 - val_acc: 0.7122
Epoch 67/100
23471/23471 [=====] - 7s 310us/sample - loss: 0.5387 -
acc: 0.7304 - val_loss: 0.5591 - val_acc: 0.7139
Epoch 68/100
23471/23471 [=====] - 7s 300us/sample - loss: 0.5412 -
acc: 0.7284 - val_loss: 0.5598 - val_acc: 0.7127
Epoch 69/100
23471/23471 [=====] - 7s 301us/sample - loss: 0.5410 -
acc: 0.7291 - val_loss: 0.5561 - val_acc: 0.7127
Epoch 70/100
23471/23471 [=====] - 7s 300us/sample - loss: 0.5407 -
acc: 0.7294 - val_loss: 0.5570 - val_acc: 0.7139
Epoch 71/100
23471/23471 [=====] - 7s 301us/sample - loss: 0.5393 -
```

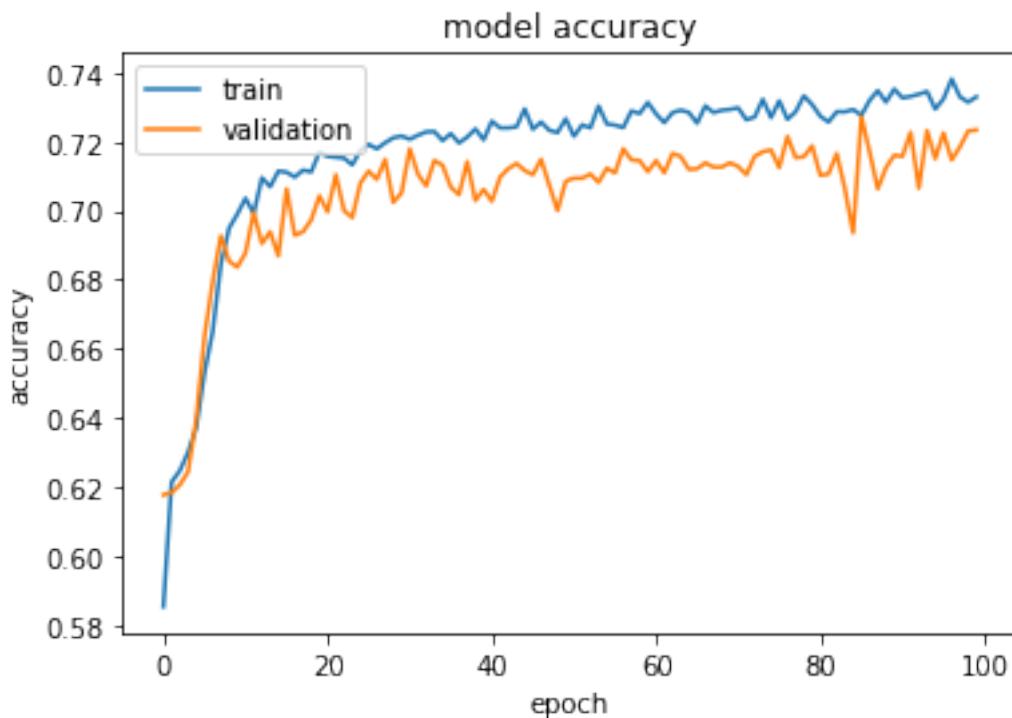
```
acc: 0.7298 - val_loss: 0.5675 - val_acc: 0.7128
Epoch 72/100
23471/23471 [=====] - 7s 303us/sample - loss: 0.5418 -
acc: 0.7263 - val_loss: 0.5601 - val_acc: 0.7105
Epoch 73/100
23471/23471 [=====] - 7s 310us/sample - loss: 0.5393 -
acc: 0.7272 - val_loss: 0.5639 - val_acc: 0.7157
Epoch 74/100
23471/23471 [=====] - 7s 318us/sample - loss: 0.5362 -
acc: 0.7323 - val_loss: 0.5632 - val_acc: 0.7171
Epoch 75/100
23471/23471 [=====] - 8s 324us/sample - loss: 0.5397 -
acc: 0.7270 - val_loss: 0.5737 - val_acc: 0.7176
Epoch 76/100
23471/23471 [=====] - 7s 309us/sample - loss: 0.5389 -
acc: 0.7318 - val_loss: 0.5548 - val_acc: 0.7125
Epoch 77/100
23471/23471 [=====] - 7s 296us/sample - loss: 0.5430 -
acc: 0.7265 - val_loss: 0.5556 - val_acc: 0.7215
Epoch 78/100
23471/23471 [=====] - 7s 298us/sample - loss: 0.5391 -
acc: 0.7287 - val_loss: 0.5545 - val_acc: 0.7154
Epoch 79/100
23471/23471 [=====] - 7s 307us/sample - loss: 0.5377 -
acc: 0.7333 - val_loss: 0.5663 - val_acc: 0.7157
Epoch 80/100
23471/23471 [=====] - 7s 309us/sample - loss: 0.5403 -
acc: 0.7308 - val_loss: 0.5568 - val_acc: 0.7188
Epoch 81/100
23471/23471 [=====] - 7s 305us/sample - loss: 0.5375 -
acc: 0.7274 - val_loss: 0.5635 - val_acc: 0.7103
Epoch 82/100
23471/23471 [=====] - 7s 309us/sample - loss: 0.5415 -
acc: 0.7257 - val_loss: 0.5579 - val_acc: 0.7106
Epoch 83/100
23471/23471 [=====] - 7s 314us/sample - loss: 0.5395 -
acc: 0.7287 - val_loss: 0.5563 - val_acc: 0.7166
Epoch 84/100
23471/23471 [=====] - 7s 315us/sample - loss: 0.5397 -
acc: 0.7286 - val_loss: 0.5618 - val_acc: 0.7064
Epoch 85/100
23471/23471 [=====] - 7s 312us/sample - loss: 0.5378 -
acc: 0.7293 - val_loss: 0.5850 - val_acc: 0.6936
Epoch 86/100
23471/23471 [=====] - 7s 310us/sample - loss: 0.5425 -
acc: 0.7277 - val_loss: 0.5542 - val_acc: 0.7273
Epoch 87/100
23471/23471 [=====] - 7s 306us/sample - loss: 0.5380 -
```

```
acc: 0.7320 - val_loss: 0.5559 - val_acc: 0.7171
Epoch 88/100
23471/23471 [=====] - 7s 302us/sample - loss: 0.5359 -
acc: 0.7348 - val_loss: 0.5628 - val_acc: 0.7064
Epoch 89/100
23471/23471 [=====] - 7s 304us/sample - loss: 0.5378 -
acc: 0.7315 - val_loss: 0.5650 - val_acc: 0.7125
Epoch 90/100
23471/23471 [=====] - 7s 305us/sample - loss: 0.5332 -
acc: 0.7352 - val_loss: 0.5556 - val_acc: 0.7161
Epoch 91/100
23471/23471 [=====] - 7s 311us/sample - loss: 0.5366 -
acc: 0.7327 - val_loss: 0.5609 - val_acc: 0.7157
Epoch 92/100
23471/23471 [=====] - 7s 304us/sample - loss: 0.5377 -
acc: 0.7331 - val_loss: 0.5558 - val_acc: 0.7227
Epoch 93/100
23471/23471 [=====] - 7s 307us/sample - loss: 0.5333 -
acc: 0.7338 - val_loss: 0.5630 - val_acc: 0.7065
Epoch 94/100
23471/23471 [=====] - 7s 312us/sample - loss: 0.5354 -
acc: 0.7345 - val_loss: 0.5552 - val_acc: 0.7232
Epoch 95/100
23471/23471 [=====] - 7s 307us/sample - loss: 0.5353 -
acc: 0.7295 - val_loss: 0.5668 - val_acc: 0.7151
Epoch 96/100
23471/23471 [=====] - 7s 303us/sample - loss: 0.5387 -
acc: 0.7324 - val_loss: 0.5564 - val_acc: 0.7226
Epoch 97/100
23471/23471 [=====] - 7s 305us/sample - loss: 0.5349 -
acc: 0.7381 - val_loss: 0.5607 - val_acc: 0.7147
Epoch 98/100
23471/23471 [=====] - 7s 303us/sample - loss: 0.5367 -
acc: 0.7330 - val_loss: 0.5538 - val_acc: 0.7185
Epoch 99/100
23471/23471 [=====] - 7s 301us/sample - loss: 0.5355 -
acc: 0.7315 - val_loss: 0.5566 - val_acc: 0.7231
Epoch 100/100
23471/23471 [=====] - 7s 307us/sample - loss: 0.5352 -
acc: 0.7330 - val_loss: 0.5668 - val_acc: 0.7234
```

```
[17]: ['loss', 'acc']
```

```
[18]: # "Accuracy"
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
```

```
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```



```
[19]: plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

