SEDOGBO Sosthène Junior

Rapport Projet IA

Partie B: Réseau de neurones

On dispose d'un dataset, IRIS de taille 150, qui contient des données sur 3 catégories de fleurs:

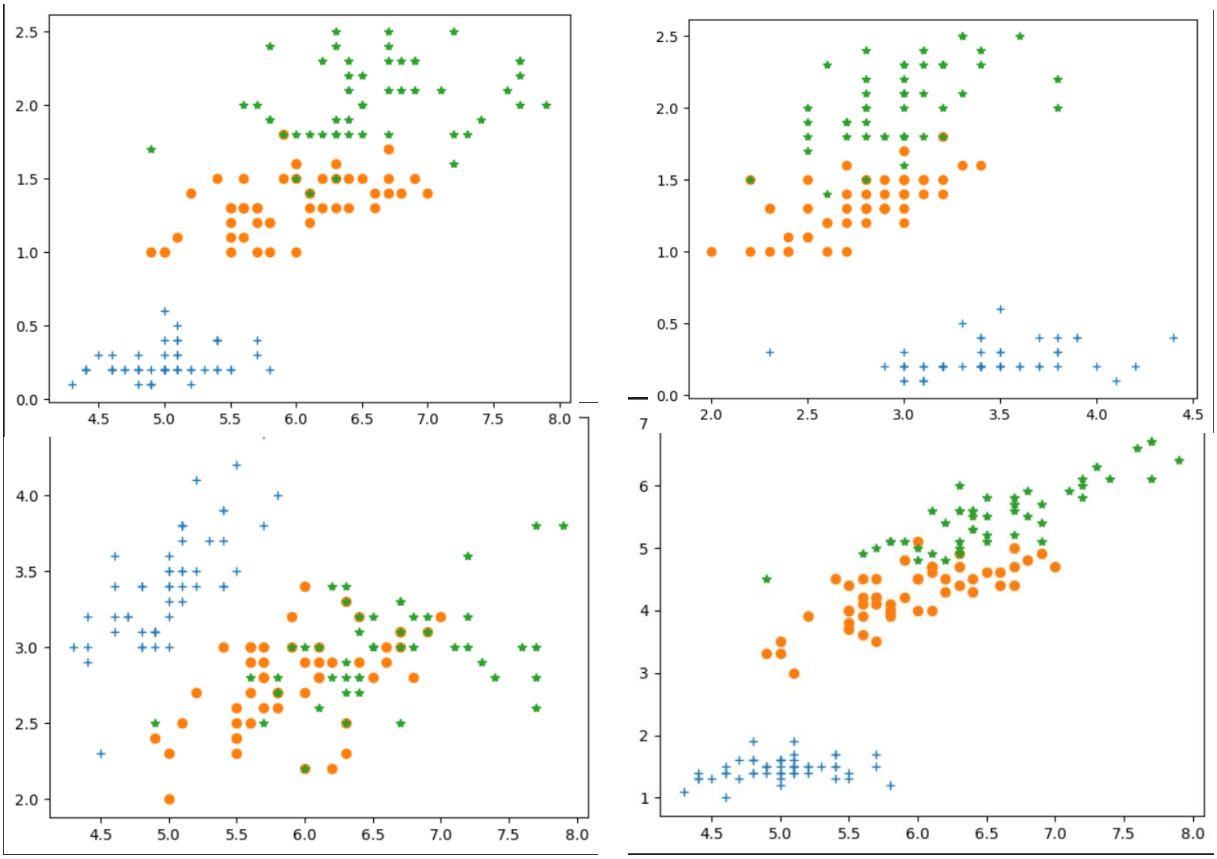
'Iris-setosa' -- 'Iris-versicolor' -- 'Iris-virginica'

Chaque fleur possède 4 caractéristiques que sont: longueur et largeur du sépale, longueur et largeur du pétale.

Le but de cette partie est de construire un réseau de neurones permettant de classifier une fleur en fonction de ces 4 caractéristiques.

Pour avoir une idée de la répartition des fleurs, j'ai pris les caractéristiques 2 par 2, et j'ai représenté les fleurs par rapport à ces caractéristiques.

Cela donne par exemple :



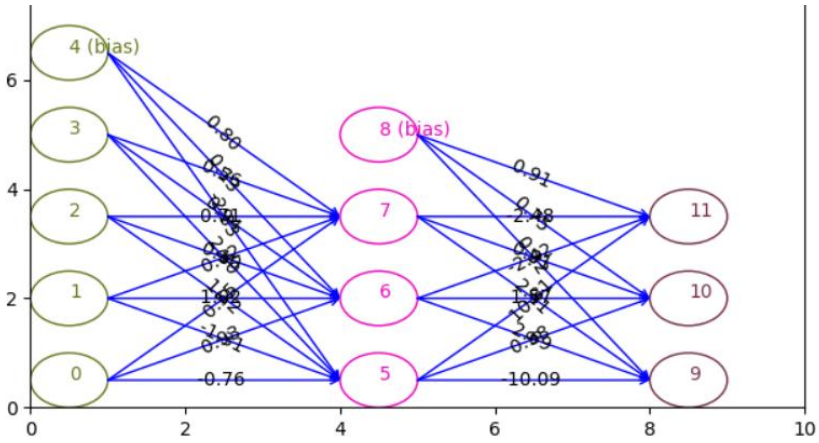
On remarque ainsi que les fleurs en bleu (Iris-setosa), sont à chaque fois, linéairement séparables des 2 autres catégories de fleurs, mais que les fleurs en orange ne sont pas linéairement séparables des fleurs entre vert.

Cela indique que le modèle de classification ne doit pas être linéaire, et qu'il faut utiliser des fonctions d'activation non linéaires.

J'ai commencé par créer une classe **Dense** qui représente une couche **entièrement connectée**, et une classe **NetModel** qui représente un **modèle** de réseau de neurones et capable d'effectuer des tâches de classification binaire ou multiple, de régression linéaire et de régression logistique.

Ma classe Dense est essentiellement constituée d'une liste de nœuds, et chaque nœud est un dictionnaire qui contient : son identifiant(id), son label éventuellement, ses nœuds successeurs, et ses nœuds prédécesseurs. Chaque couche possède aussi une fonction d'activation.

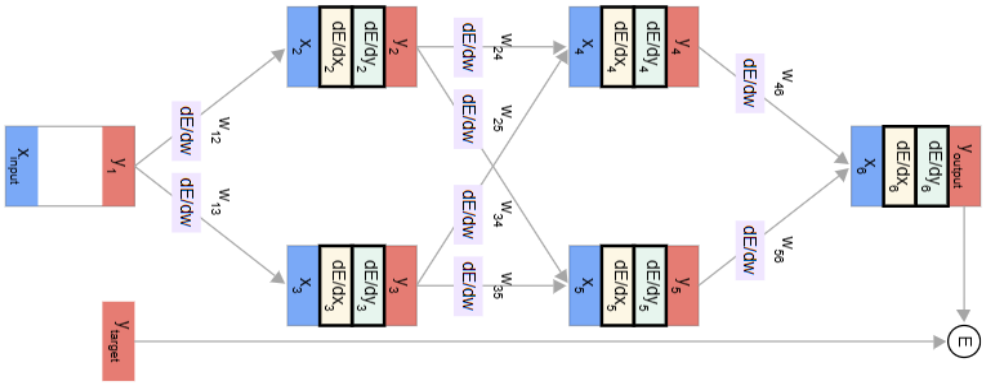
Ma classe NetModel est essentiellement constituée d'une **pile** de couches (Dense) et d'une liste de **poids** (Wij) qui sont les paramètres mis à jour lors de l'entrainement du modèle.



1Exemple de NetModel entraîné

Le fonctionnement de ma classe NetModel est simple :

- Elle effectue un **prétraitement** des données d'entrée, c'est-à-dire que pour la multi classification par exemple, elle transforme les labels en matrices selon la norme **one hot encoding**.
- Ensuite, elle crée la liste de poids ainsi que la couche de sortie avec le bon nombre de nœuds de sortie et la bonne fonction d'activation. Dans le cas du dataset **Iris**, la couche de sortie possède 3 nœuds, une par catégorie, avec la fonction **softmax**.
- Il s'entraine alors sur un ensemble d'entraînement passé par l'utilisateur, en mettant à jour les poids grâce à de la **rétropropagation**. Pour pouvoir mettre en place cette rétropropagation, je stocke dans chaque nœud, des variables supplémentaires : x pour la donnée d'entrée, y pour la donnée de sortie du nœud (transformation par la fonction d'activation), dE/dx pour la différentielle de l'erreur par rapport à x, dE/dy pour la différentielle de l'erreur par rapport à y.



SEDOGBO Sosthène Junior

Algorithme de rétropropagation

Pour mettre à jour les poids, on applique la formule $w_{ij}=w_{ij}-\alpha \times dE/dw_{ij}$

La difficulté est qu'il n'est pas possible de calculer directement dE/dwij ; c'est pourquoi on se sert des variables intermédiaires x et y comme suit pour calculer :

$\partial E/\partial y_i$
$\partial E/\partial x=dy/dx \,.\, \partial E/\partial y=d/dx\, f(x) \,.\, \partial E/\partial y$
$\partial E/\partial w_{ij}=\partial x_j/\partial w_{ij} \,.\, \partial E/\partial x_j=y_i \,.\, \partial E/\partial x_j$
$\partial E/\partial y_i=\sum_{j\in out(i)} \partial x_j/\partial y_i \,.\, \partial E/\partial x_j=\sum_{j\in out(i)} w_{ij} \,.\, \partial E/\partial x_j$

Cet algorithme est répété de manière itérative pour tous les nœuds.

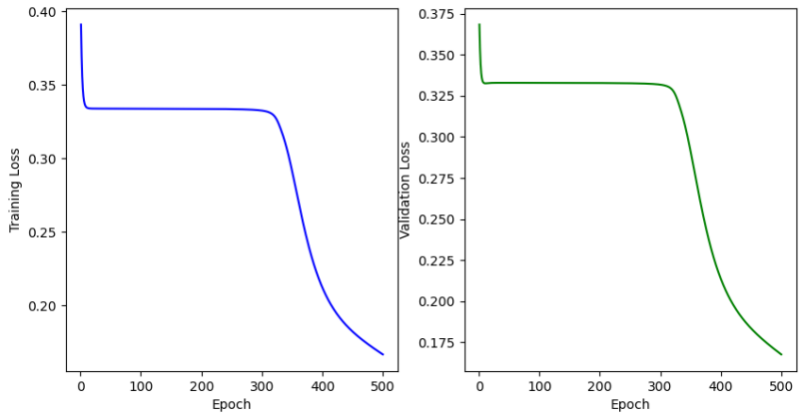
Pendant l'entraînement, je calcule également l'erreur sur l'ensemble de validation pour chaque itération (période). La fonction de calcul d'erreur utilisée est **l'erreur quadratique**.

Pour concevoir mon réseau de neurones, j'ai commencé par créer un NetModel() qui prend 4 inputs et qui fait de la multi classification. J'ai rajouté une **couche cachée** avec la fonction d'activation **sigmoïde** et constituée de **3 nœuds**.

Ce modèle a été ensuite entrainé avec les hyperparamètres suivants : Learning rate = 0.01, nombre de périodes = 500.

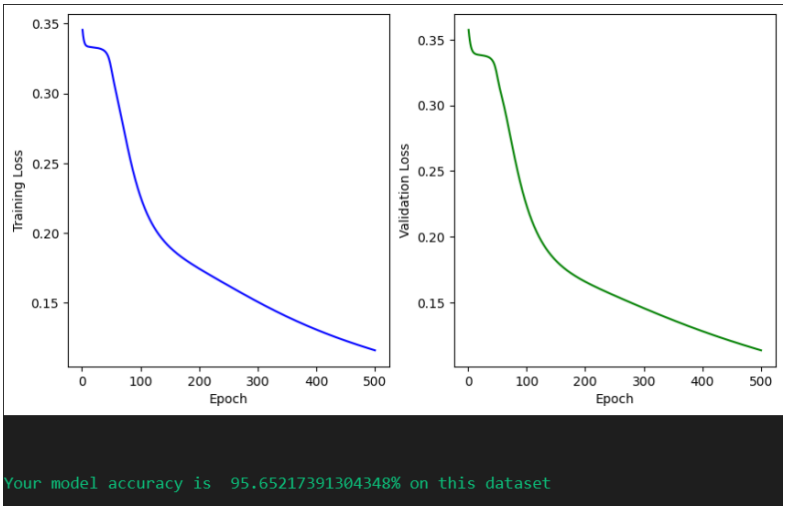
Remarque : Le dataset étant très petit, les résultats obtenus pour une même configuration varient légèrement en terme d' « accuracy ».

Les courbes d'erreurs (training et validation) obtenues sont les suivantes :



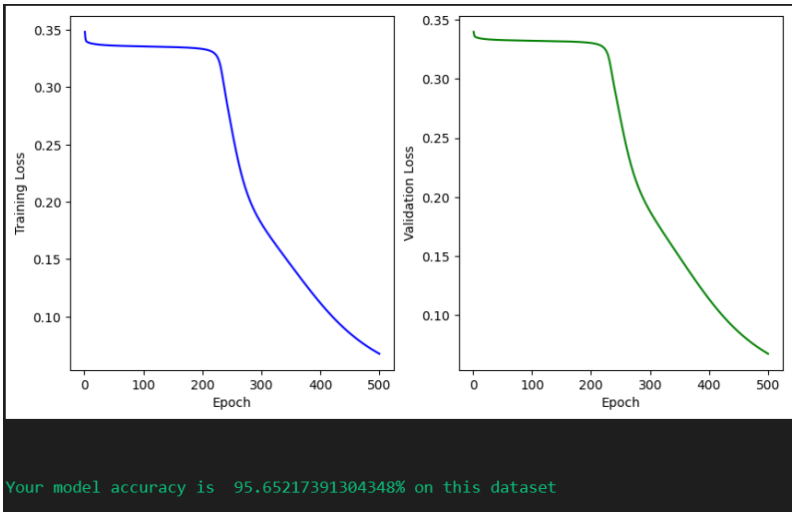
J'ai obtenu une précision de **82.61%** sur l'ensemble de test. Ce qui n'est pas très bon : (seulement **19** fleurs ont été bien classifiées sur **23**). On voit aussi que l'erreur reste très haute pendant 300 itérations, ce qui est bizarre.

J'ai donc rajouté 1 nœud à la couche cachée pour en avoir 4. Les résultats obtenus sont les suivants :



On voit ici que les courbes d'erreur descendent plus vite que lors du cas précédent. Et que la précision cette fois est plus élevée : **96%.** Ce qui est très bien : **22 fleurs sur 23** ont été bien classifiées.

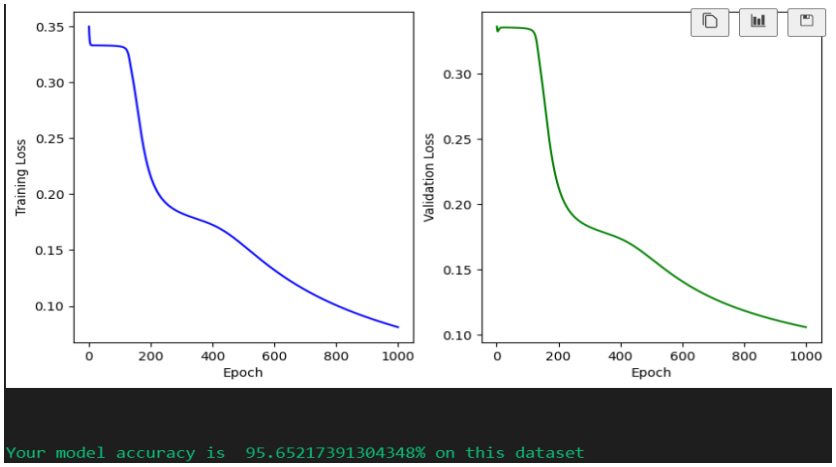
Pour essayer de voir si cette précision peut être améliorée, j'ai rajouté successivement 1 par 1, 6 nœuds dans la couche cachée, jusqu'à en avoir **10.** Les résultats obtenus sont à peu près similaires :



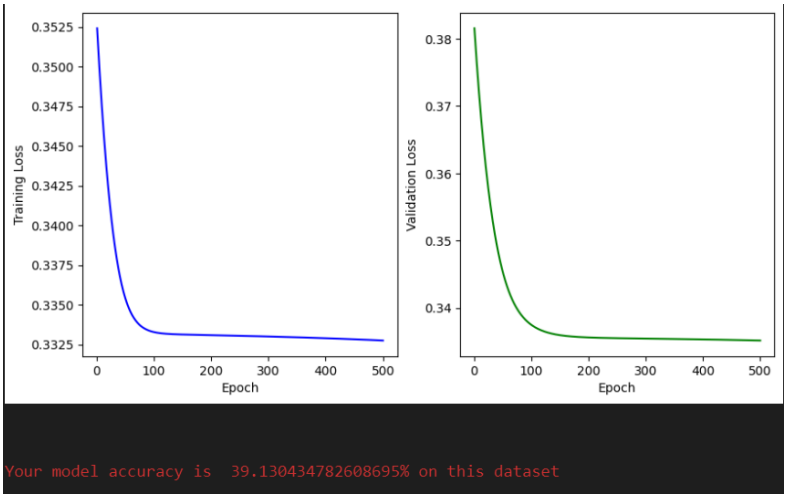
La précision ne varie pas significativement. On a toujours **1** fleur **mal classifiée** sur **23**

Donc j'ai gardé le modèle à 4 nœuds dans la couche cachée.

Ensuite, j'ai fait varier les **hyperparamètres** un par un pour voir si on avait des résultats différents. En passant de **500 périodes à 1000**, j'ai obtenu la même précision de classification, mais des courbes d'erreur légèrement différentes.

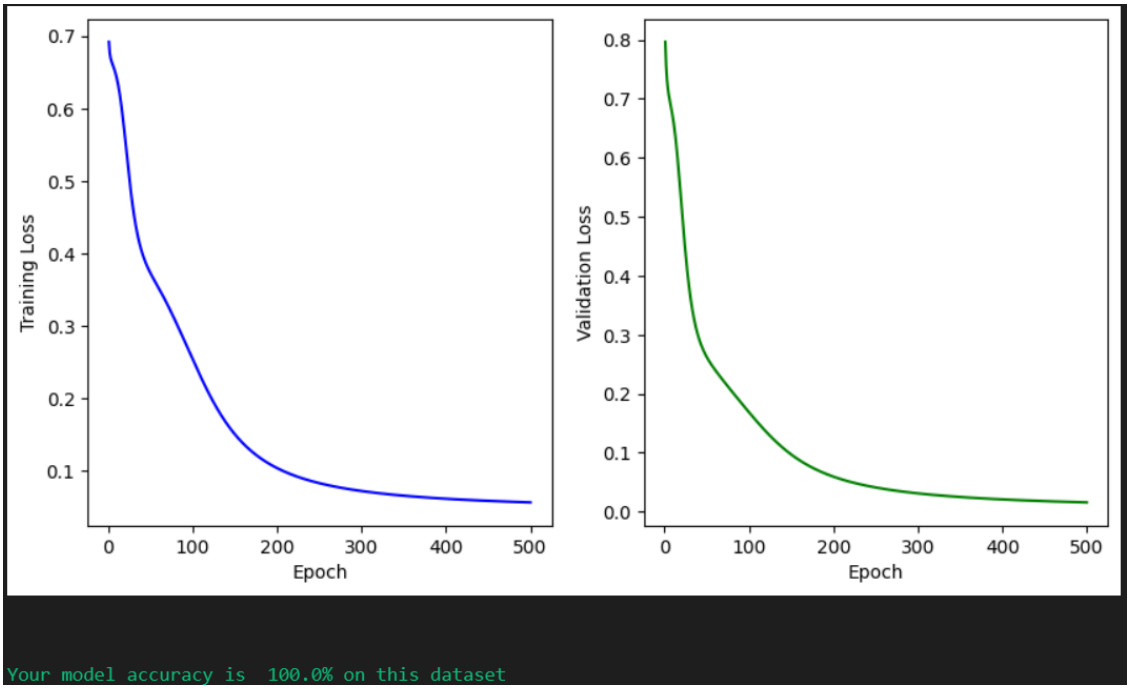


Quant au learning rate, je l'ai baissé à **0.001** et j'ai obtenu des résultats très mauvais :



Seulement **9** fleurs ont été **bien classifiées** sur les **23**. Cela s'explique par le fait que le modèle modifie les poids très faiblement.

Finalement, le meilleur modèle que j'ai pu obtenir a 1 couche cachée constituée de 4 nœuds, un Learning rate de 0.01, et un nombre d'epochs = 500. Ce modèle atteint une précision de **100%** sur l'ensemble de test très souvent.



Partie A : Implémentation d'algorithmes d'exploration dans le jeu de Pacman

Le but de cette partie était d'implémenter les algorithmes d'exploration A*, DFS, BFS, et UCS en python, en se servant de modules pré-implémentés.

J'ai juste appliqué l'algorithme vu en cours pour le DFS, en utilisant une util.Stack() pour gérer la frontière de nœuds.

Même algorithme pour le BFS, il suffit de remplacer la util.Stack par une util.Queue() ; j'ai donc modifié une seule ligne.

Pour le UCS et le A*, j'ai utilisé une util.PriorityQueue() à la place.

Les résultats obtenus sont :

- Pour le DFS :
  - python pacman.py -l tinyMaze -p SearchAgent

```
PS C:\Users\Junior\Documents\3A\AI\TDs\search> C:\Python24\python.exe .\pacman.py -l tinyMaze -p
SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
```

On obtient un coût total de 10 pour 15 nœuds visités.

  - python pacman.py -l mediumMaze -p SearchAgent

```
PS C:\Users\Junior\Documents\3A\AI\TDs\search> C:\Python24\python.exe .\pacman.py -l mediumMaze -
p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```

On obtient un coût total de 130 pour 146 nœuds visités.

  - python pacman.py -l bigMaze -z .5 -p SearchAgent

```
PS C:\Users\Junior\Documents\3A\AI\TDs\search> C:\Python24\python.exe .\pacman.py -l bigMaze -z .5
 -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Users\Junior\Documents\3A\AI\TDs\search>
```

Un coût total de 210 pour 390 nœuds visités.

- Pour le A* :
  - python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

```
PS C:\Users\Junior\Documents\3A\AI\TDs\search> C:\Python24\python.exe pacman.py -l bigMaze -z .5 -
p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.2 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

On obtient un coût de 210 pour 549 nœuds visités.