Rapport Projet IA

Partie B: Réseau de neurones

On dispose d'un dataset, IRIS, 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.

Une image contenant graphique

Description générée automatiquementUne image contenant graphique

Description générée automatiquementUne image contenant graphique

Description générée automatiquementUne image contenant graphique

Description générée automatiquementCela 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.

Une image contenant diagramme, schématique

Description générée automatiquement

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**.
* Une image contenant diagramme

  Description générée automatiquementEnsuite, 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.

Algorithme de rétropropagation

|  |
| --- |
| wij=wij−α x dE/dwij |

Pour mettre à jour les poids, on applique la formule

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 :

|  |
| --- |
| ∂E/∂yi |
| ∂E/∂x=dy/dx . ∂E/∂y=d/dx f(x) . ∂E/∂y |
| ∂E/∂wij=∂xj/∂wij . ∂E/∂xj=yi . ∂E/∂xj |
| ∂E/∂yi=∑j∈out(i) ∂xj/∂yi . ∂E/∂xj=∑j∈out(i) wij . ∂E/∂xj |

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).

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

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

Une image contenant graphique

Description générée automatiquementJ’ai obtenu une précision de **85.33%** sur l’ensemble de test. Ce qui n’est pas très bon. On voit aussi que l’erreur reste très haute pendant les 200 premières 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 :

Une image contenant graphique

Description générée automatiquement

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.

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 :

Une image contenant graphique

Description générée automatiquement

La précision ne varie pas significativement.

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

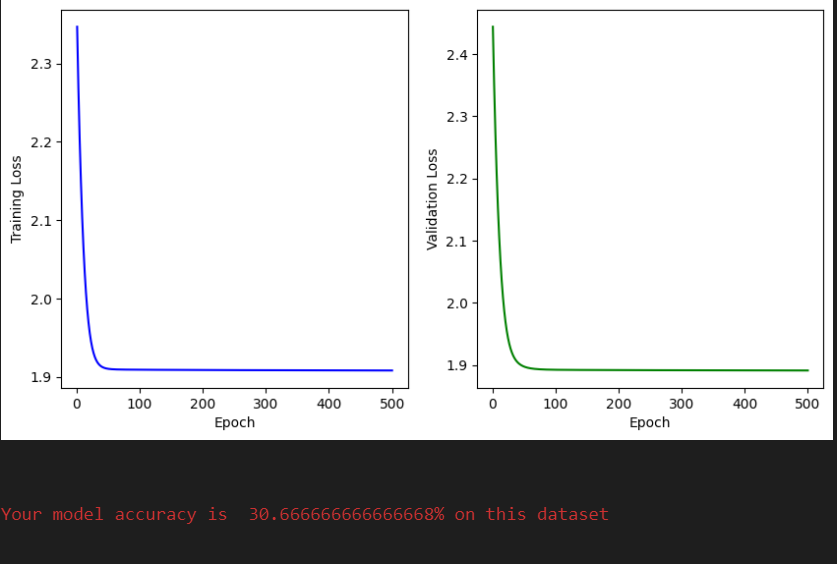
Ensuite, j’ai fait varier les hyperparamètres un par un pour voir si on avait des résultats différents.

En passant de 500 périodes à 1000, j’ai obtenu les résultats suivants :

Une image contenant graphique

Description générée automatiquement

Le constat principal est que la courbe d’erreur de validation remonte à partir de la période 500, alors que la courbe d’erreur d’entrainement descends. Ce qui veut dire que le modèle devient de plus en plus « overfitted » quand le nombre d’itérations dépasse 500. J’ai donc gardé **500** comme nombre d’itérations

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

En l’augmentant à 0.05, on obtient :

Une image contenant graphique

Description générée automatiquement

Les courbes d’erreur deviennent oscillantes, ce qui est mauvais en termes de performance.

Une image contenant graphique

Description générée automatiquement

Donc finalement, je garde mon modèle à 1 couche cachée, 4 nœuds, un learning rate de 0.01, et un nombre d’epochs = 500 qui m’a permis d’atteindre une précision de 98.66% sur l’ensemble de test.

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

Une image contenant texte

Description générée automatiquement

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

* python pacman.py -l mediumMaze -p SearchAgent

Une image contenant texte

Description générée automatiquement

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

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

Une image contenant texte

Description générée automatiquement

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

Une image contenant texte

Description générée automatiquement

On obtient un coût de 210 pour 549 nœuds visités.