Rapport de Projet: Etape intermédiaire 2

Projet Annuel 3 Big Data

ALLEXANDRE Matthieu
HOLLANDER Stephane
TAVERNIER Antoine

29/05/2019

Table des matières

Objectifs

Lors de ce rendu, nous devons réaliser plusieurs objectifs. Nous devons implémenter les algorithmes sur le classification/régression linéaire ainsi que le perceptron multi couche pour la classification et la régression. Tous les algorithmes doivent valider un certain nombre de test. De tests simples comme le OR à plus complexe comme le XOR et le test de la croix à 3 classes. De plus nous devions réaliser un début de Front/Backend pour réaliser une interface entre l'utilisateur et notre programme. En bonus nous avons réalisé un projet Unity qui a pour but de représenter, dans un espace 3D, les résultats des cas de tests.

Algorithme Linéaire

Bien que cette partie ait déjà été implémenté pour le rendu précédent, nous avons décidé de la refaire pour simplifier le code. En effet nous utilisions une structure de réseau de neurones pour gérer l'algorithme de Rosenblatt.

Classification Linéaire

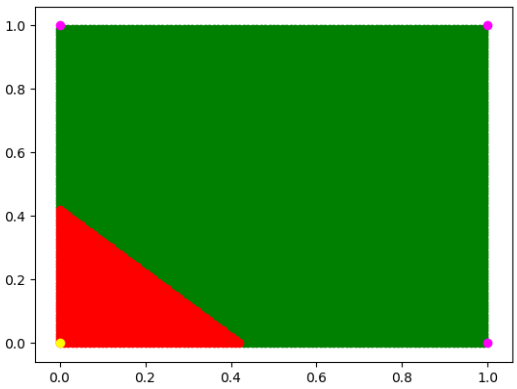
Nous avions une structure trop complexe pour gérer la classification linéaire. Les résultats n'étaient pas mauvais, mais nous voulions le refaire pour que tous les membres du groupe aient bien compris son implémentation.

Nous avons donc recodé cette partie en simplifiant le code pour le rendre plus lisible pour tous. Le principe reste inchangé, nous avons toujours un seul neurone avec n entrées auquel on rajoute un neurone de biais.

Pour chaque époque, nous chargeons les entrées une a une. Nous calculons la sortie du neurone, puis rectifions le poids de chaque entrées grâce à l'implémentation de Rosenblatt. La logique de cette implémentation est basée sur la distance entre la valeur calculée et la valeur attendu. Cette distance donne donc la direction et l'amplitude de la variation de la mise à jour des poids. Au début nous avions on chargement des inputs de façon linéaire, ce qui n'avait pas de gros impact sur les petits dataset, comme les tests case. Mais qui pouvait être un biais pour les datatset bien plus gros. Nous avons donc intégré un système pour choisir aléatoirement une image. Ce n'est toujours pas parfait car l'ordre reste le même selon les Epochs. En effet nous avons stocké les index des inputs dans un vecteur que nous avons mélangé. Pour le prochain rendu nous allons remélangé ce vecteur à chaque Epoch pour avoir une meilleure redistribution.

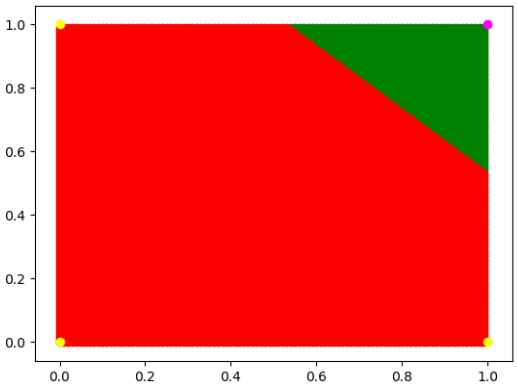
OR

Le cas de test du OR est le cas de test le plus simple. Nous devons séparer linéairement deux classes de points.



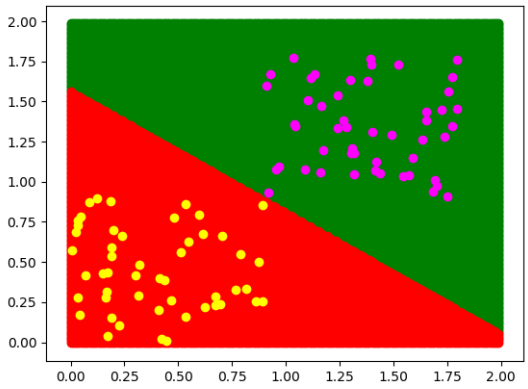
AND

Le cas de test du AND est identique au OR, cela nou permets de confirmer le bon fonctionnement de notre neurone.



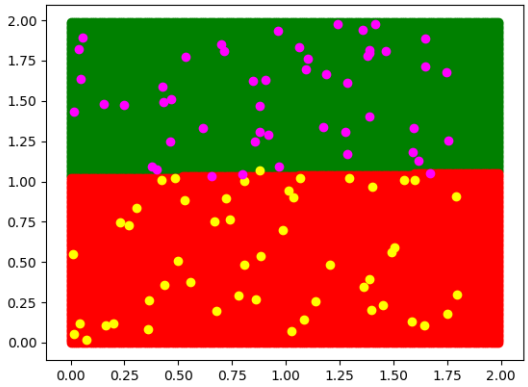
Cas simple

ce cas de test est un test avec plus de valeurs. Le principe est le même que le OR et le AND, il permet de vérifier que le neurone marche bien avec des valeurs aléatoires.



Nous avons donc créé une règle qui génère 100 points, la moitié appartient à une première classe et le reste à une autre classe. Lors de la génération des points, on s'assure qu'ils ne se chevauche pas pour être sûr d'avoir une séparation linéaire.

Dans cet exemple nous avons rajouté du bruit à la génération des points pour observer la réaction de l'apprentissage.



On constaste que quelques points sont du mauvais côté de la séparation. En effet le neurone montre sa limite, il est incapable d'aller chercher ses points, tout simplement car le jeu de données n'est pas séparable linéairement !

Cas simple à 3 classes

L'objectif de ce cas de test est de vérifier que nous sommes capables de séparer linéairement plusieurs classes. Or un neurone n'a qu'une seule sortie, il faut donc entrainer 3 neurones séparés afin produire 3 fonctions affines.

Si nous avons 3 classes, A, B, C. Alors il suffit d'entrainer un premier neurone à séparer la classe A des classes B et C regroupées. On répète ce processus

2 fois pour les classes B et C, puis on obtient 3 classifications linéaire. Il suffit maintenant de faire une prédiction sur les 3 neurones pour récupérer la bonne classe qui lui est associée.

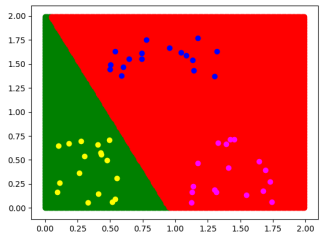


FIGURE 1 – Fit A

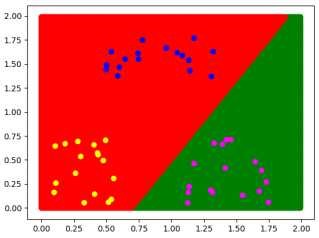


FIGURE 2 – Fit B

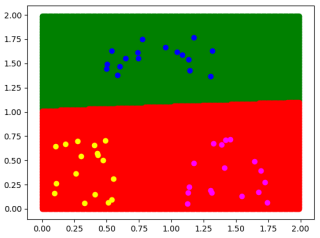
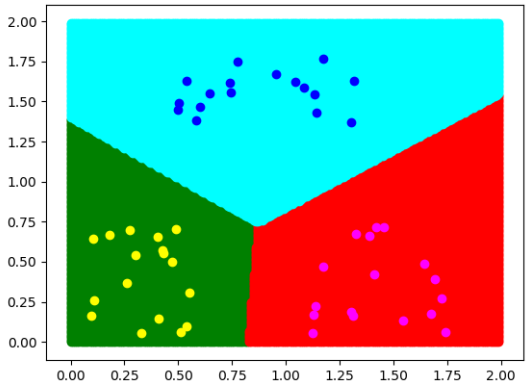


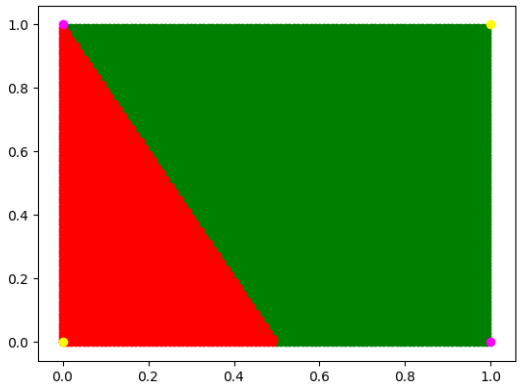
FIGURE 3 – Fit C

Une fois chaque classe classifiée individuellement, il suffit de prendre la prédiction la plus haute pour chacune des classes.



Cas impossible : XOR

Nous sommes dans une classification linéaire, il est impossible pour un neurone seul de réaliser une séparation linéaire du cas du XOR. En effet il nous faut deux droites pour pouvoir séparer les classes. On observe bien ici l'infaisabilité du problème fourni au neurone. Le neurone ne sait que générer une fonction affine et non plusieurs.



On observe encore une fois la limite du neurone seul. Il est impossible de séparer les points $[1, 0]$ $[0, 1]$ dans une première classe et $[1, 1]$ $[1, 1]$ dans une seconde par une droite unique.

Cependant il est quand même possible de résoudre ce cas de test grâce à une transformation polynomiale [1] [2], ce cas ne sera pas démontré dans ce rendu.

Régression Linéaire

La régression linéaire a pour but d'estimer de trouver une valeur plutôt que trouver l'appartenance, ou non, d'une classe. Pour cela nous avons suivis l'implémentation matriciel de la régression linéaire.

On trouve les poids de chaque entrées grâce à cette formule :

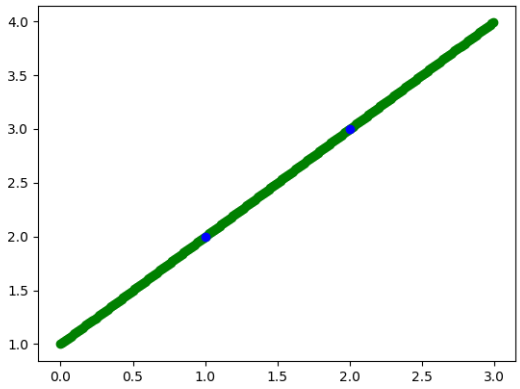
$$W = ((X^T X)^{-1} X^T) Y$$

Pour les calculs matricel, nous avons utilisé la bibliothèque *Eigen*.

Simple 2D test

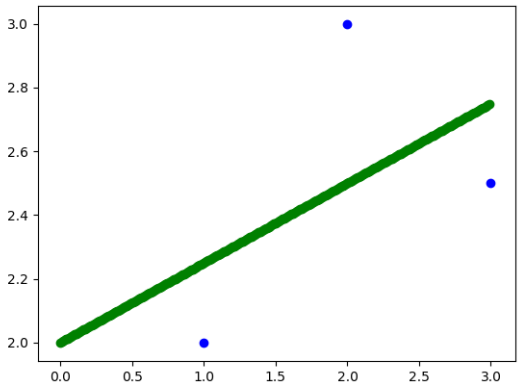
Le premier cas de test est un apprentissage sur 2 points. On est donc sûr d'avoir une unique droite qui passe par ses deux points en résultat.

1. https ://medium.com/@lucaspereira0612/solving-xor-with-a-single-perceptron-34539f395182
2. http ://www.uncini.com/research_activity/pdf/035_ijcnn92.pdf



En effet nous avons bien une droite, de plus cet exemple nous permet de voir le bon fonctionnement du biais, qui nous permets d'avoir une fonction affine ne passant pas uniquement par l'origine.

Le cas suivant est le cas général de la régression linéaire. On imagine par exemple que l'on recherche la taille des pénis des lémuriens en fonctions de leur taille de pattes avant.

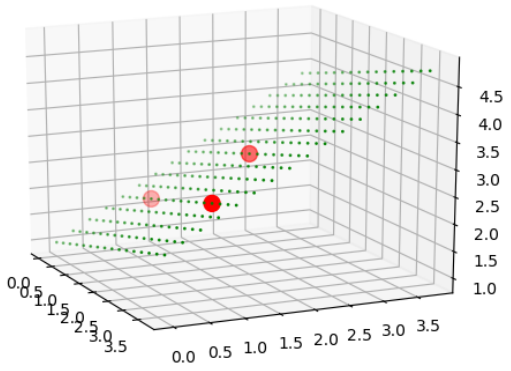


On se retrouve donc avec une estimation de taille de pénis en fonction de leur taille de pattes avant d'après une population de 3 lémuriens. Ce qui n'est pas représentatif d'une population d'une espèce.

Simple 3D test

Le cas 3D nous permets de savoir si notre régression linéaire est capable de traiter plusieures entrées pour un même individu.

On peut reprendre notre exemple précédent avec les lémuriens, en y rajoutant en paramètres la taille de leurs pattes arrière.



On se retrouve donc avec un plan dans l'espace créé par notre 3 dimentions qui sont la tailles des pattes avant, des pattes arrière et de leur pénis.

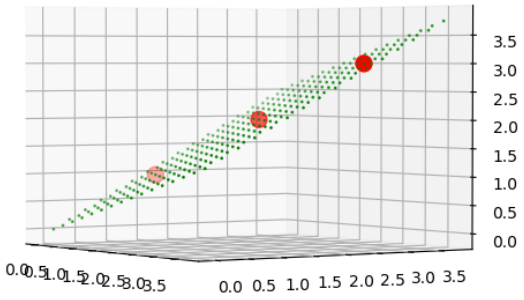
Tricky 3D test

Dans cet exemple, nous sommes dans un cas où toutes nos valeurs son multiples de la première, on se retrouve donc avec une matrice collinéaire. Dans sa représentation graphique, cela veut tout simplement dire que nos données sont toutes alignées sur une droite. Ainsi il y a une infinité de plan qui peut passer par cette droite.

Pour pallier ce problème, nous faisons une très légère modification sur les valeurs d'entrées. Nous ne cherchons même pas à savoir si la matrice d'entrée est colinéaire, nous faisant les modifications dans tous les cas. Cette modification de valeur est très subtile et permets de désaxer très légèrement les points pour pouvoir créer un plan unique entre nos points.

Dans cette exemple, les valeurs d'entrées sont $[1, 1, 2, 2, 3, 3]$, les valeurs attendu de sorti sont $[1, 2, 3]$. On modifie une ou plusieurs valeurs, on se retrouve donc par exemple avec en valeurs d'entrées

$[0.999999999, 1, 2, 2.000000001, 3, 2.999999999]$



Cette manipulation nous permet d'avoir des résultats cohérents avec les valeurs d'entrées d'origines.

Réseau de Neurones

Classification

Cas de tests basiques

Voici les résultats du AND, OR et XOR avec un réseau de neurones simpliste de 2 neurones en couche d'entrée et 1 neurone de sortie.

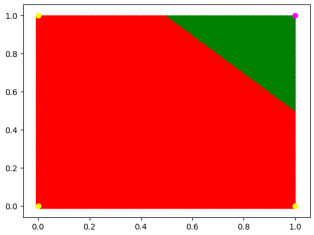


FIGURE 4 – MLP AND

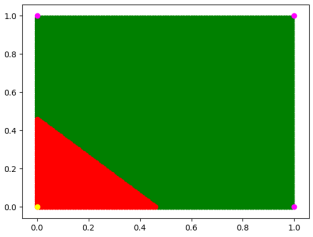


FIGURE 5 – Fit C

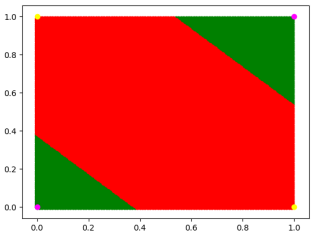


FIGURE 6 – XOR 1

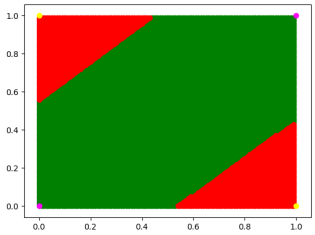
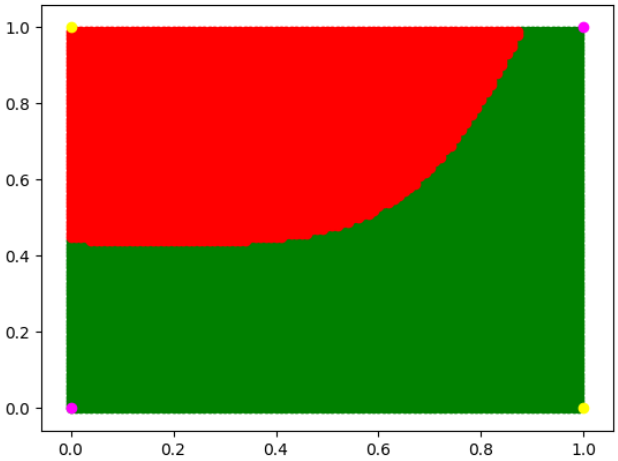


FIGURE 7 – XOR 2

D'après les résultats ci-dessus, tout semble correct, or si on fait plusieurs XOR avec les mêmes hyperparamètres, on peut obtenir des résultats différents.



On suppose que dans le premier cas notre Modèle soit en sous apprentissage, même si on émet des doutes, car le nombres d'epochs et le pas d'apprentissage reste inchangé pour tous les cas du XOR. Il se peut aussi que cela soit normal, dû à la génération aléatoire des poids. En effet nous sommes censés trouver deux droites affines parallèle, mais il se peut que nos poids initiaux créés une tendance qui ne favorise pas la distinction des deux droites affines. On a donc une combinaison de deux droites qui essayent de garder les deux extrémité

Haute-Droite et Basse-Gauche dans le vert, et le reste dans le rouge. D'où le coude formé en rouge qui tente de se rapprocher du point Bas-Droite. On note aussi que lors de ses résultats, on obtient des valeurs de *Loss* très hautes à la normale et sont presque impossible à corriger. On suppose que nous sommes dans un minimum local et que nous n'avons pas encore les outils pour pouvoir sortir de ses cas.

Cross

Le cas du *Cross* à le même comportement que le *XOR*, il marche la majorité du temps, mais de façon aléatoire on se retrouve avec un modèle qui ne marche pas.

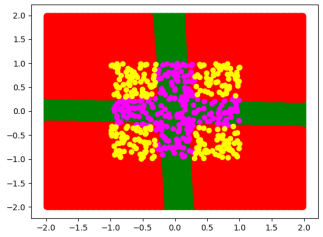


FIGURE 8 – Cross

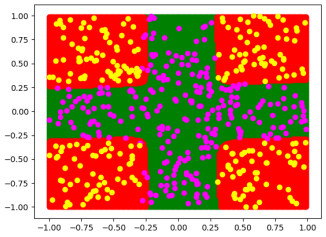


FIGURE 9 – cross

On suppose que ce sont des comportements normaux qui dépendent de la position initiales (les poids) de la recherche de l'optimisation. Il suffirait de jouer avec les hyperparamètres pour sortir de ses cas particuliers pour obtenir le bon résultat. C'est comme si nous étions dans un minimum local, et que la distance entre notre prédiction et nos résultats ont une influence trop restrictive sur l'orientation de la recherche.

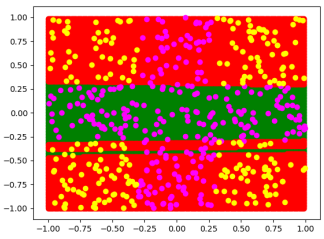


FIGURE 10 – Modèle du Cross

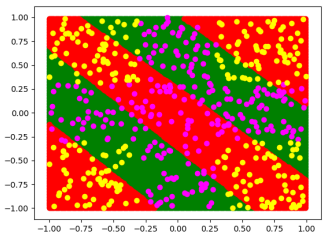


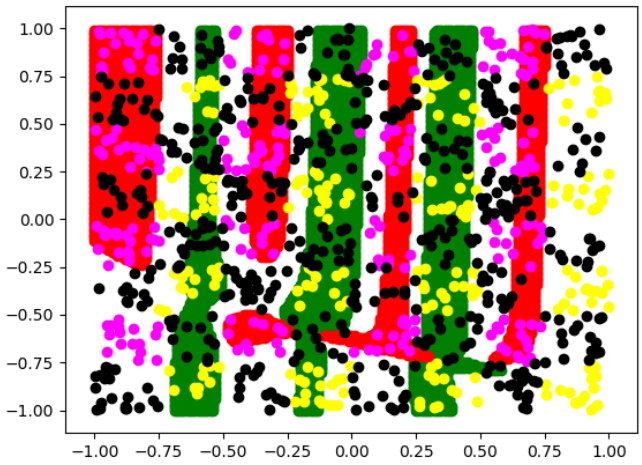
FIGURE 11 – cross

En effet on observe bien que nous possédons les deux features du réseau de neurones $[2, 1]$, pour résoudre ce problème il nous faudrait un outil de température capable d'autoriser de s'éloigner de notre objectif pour mieux s'y rediriger, ou alors relancer l'apprentissage du Modèle jusqu'à trouver une *Loss* correcte.

Multi Cross

Le cas du multi cross est complexe, à l'heure actuelle notre *Loss* n'est pas encore capable de donne un résultat qui reflète un bon apprentissage selon plusieurs classes. En effet nous ne faisons le calcul que d'un neurone de sortie et non de tous les neurones de sortie de la dernière couche (Hormis biais).

Ne connaissant la forme du réseau de neurones capable de bien représenté ce Modèle, nous avons commencé à faire des tests à l'aveugle. Le premier test est basé sur un réseau de neurones $[4, 4, 3]$, on sent que le Modèle n'est pas loin d'accomplir ce qu'on lui donne.



Dans ce cas, on ne sait pas si les prédictions en rouge appartiennent à la classe des points noirs ou des points magentas. De même pour les prédictions en vert pour les points en noirs ou en jaune. Finalement grâce à la dernière prédiction en blanc, on peut supposer que notre réseau de neurones essayait d'assigner les prédictions en rouge pour les points en magentas, le vert aux points jaunes et le blanc aux points noirs. Ici aussi nous pensons que nous nous retrouvons dans des cas de minimum locaux qui nous interdisent de faire des erreurs pour franchir une étape. Il nous manque la possibilité de rater pour mieux réussir.

Voici d'autres tentatives avec des réseaux de neurones plus profond en largeur et/ou en hauteur.

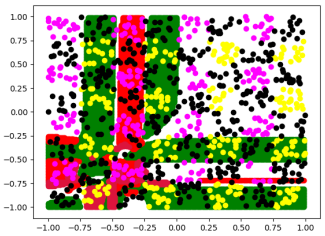


FIGURE 12 – Modèle : [8, 8, 3]

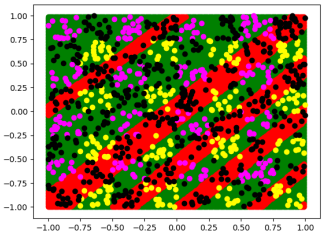


FIGURE 13 – Modèle : [16, 16, 3]

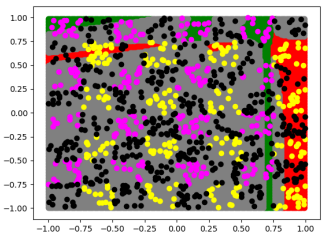


FIGURE 14 – Modèle : [24, 24, 24, 24, 3]

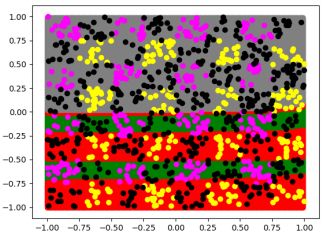
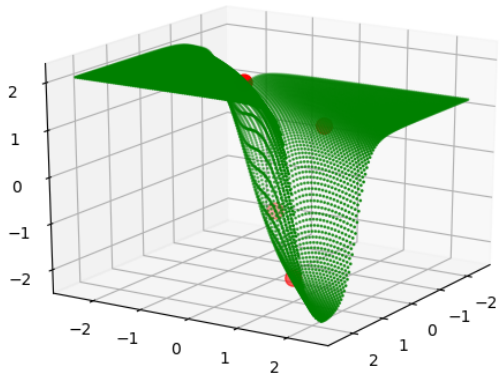


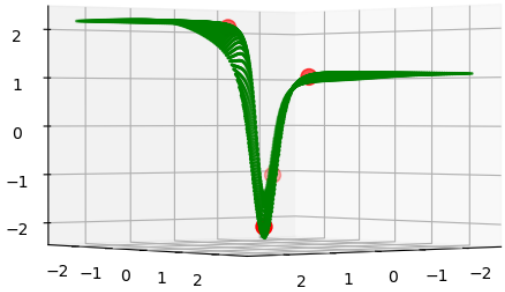
FIGURE 15 – Modèle : [8, 8, 8, 8, 3]

Régression

L'implémentation de la régression a été facile, une fois la classification réussie, nous n'avons qu'à faire de légère modification concernant le calcul du delta de la dernière couche, mais aussi du calcul de la sortie du réseau de neurones qui n'est rien de plus qu'une somme pondérée. Dans la classification c'est une somme pondérée injecté dans une fonction Tangente hyperbolique.

Voici le résultat d'un cas simple d'une régression 3D qui n'est réalisable qu'avec un réseau de neurones. En effet un model linéaire ne serait pas adapté pour résoudre ce problème.





Site Web

Le site Web étant un composant annexe du projet, nous avons de l'utilisé une technologie facile d'utilisation. Nous ne voulions pas apprendre une nouvelle technologie, ce qui serait une perte de temps pour le cœur du projet. Nous avons donc utilisé *Flask* en python pour gérer le Back et Front End.

Son utilisation est très simple, il suffit de créer des routes d'URL dans le fichier python du serveur et d'y accrocher une page *html*.

Nous avons donc une page qui gère le dépôt de fichier, fichier qui est enregistré localement sur le serveur. Par la suite, nous devons appeler notre ordonnanceur en python pour faire une prédiction sur l'image à évaluer.

De plus nous avons réussi à faire une intégration WebGL pour le projet Unity.

Unity : Affichage des cas de test

Ce projet Unity est une initiative pour utiliser un autre outil que Python pour afficher nos résultats des cas de test. Matplotlib est très puissant et facile d'utilisation, c'est pourquoi nous avons fait nos premiers rendus visuels avec celui-ci.

Le projet est donc un environnement 3D avec un repère orthonormée, matérialisé par de longues tiges et un fond.

Le projet lit un fichier CSV contenant les coordonnées de chaque prédiction ainsi que la classe associée. On place par la suite sur le repère tous les points listés dans les CSV avec une couleur par classe.

Nous avons aussi rajouté la possibilité de tirer des boules. Plus tard, nous voulons que l'utilisateur puisse tirer des boules avec des couleurs. Pour ensuite entrainer son propre Modèle avec ses propres points.

Pour l'instant nous ne savons pas encore comment donner la possibilité au joueur de choisir une classe pour changer la couleur des boules qu'il va tirer. De plus, il faut réussir à lancer l'ordonnanceur à partir du projet Unity mais aussi charger un CSV en dehors des assets du projet. En effet pour l'instant, le build WebGL inclut directement les assets dans un fichier commun d'asset. Il faut qu'on trouve une solution pour communiquer avec le serveur WEB afin de lire et d'écrire sur le serveur via le WebGL.

Nous ne savons pas encore si le projet Unity va persister dans le projet final, car il demande beaucoup de recherche sur la technologie, bien plus que sur la recherche de bug du programme en lui-même.

Conclusion

Nous pensons avoir des bases solides pour passer à la prochaine étape, prédire nos classes grâce à notre algorithme de réseau de neurones. Cependant nous avons vu que nous n'étions pas capables de réaliser en permanence de bon résultat. Nous pensons que cela est dû à une correction trop sévère des poids lorsqu'on ne s'écarte rien que d'un petit peu de notre objectif. En effet nous corrigeons les poids en fonction de la distance de nos prédictions et de nos points attendus, or il est impossible de se rapproche de ce point si nous avons un obstacle entre. Nous sommes donc coincés dans ce que l'on pense être un minimum local.

Si les problèmes cités dans ce rapport sont véridicités, alors nous allons devoir réfléchir à contourner ce problème ensemble. Il va falloir enrichir notre data set avec des images possédant du bruit, tel que du texte par-dessus, du flou, des images parasites qui n'appartiennent pas au jeu tel qu'une webcam.