GUILLAUMIN Noé

LIRZIN Léo

MAILLARD Enzo

Rendu – étape intermédiaire n°3

1) Résultat sur les cas de tests

a. Mode d'emploi

`MLP_new( int count, … )`

Crée un objet de type MLP et retourne un pointeur vers ce dernier. Cet objet représente un perceptron multicouche ayant `count` couches où `count` >= 2. Le nombre de neurones dans chaque couche est précisé après `count` dans l'ordre.

`MLP_setUsedForClassification( MLP* mlp, bool val )`

Indique à l'objet *`mlp` si le réseau de neurones est utilisé pour la classification ou pas (`val`= true si oui `val`= false sinon.

`MLP_initElements( MLP* mlp, int len )`

Initialise les matrices stockant les entrées et leur sortie attendue dans l'objet *`mlp` pour que ces dernières puissent stocker `len` éléments .

`MLP_addElement( MLP* mlp, int count, … )`

Ajoute une entrée et une sortie attendue utilisés pour l'entraînement à dans l'objet *`mlp`.

`MLP_train( MLP* mlp, int nb_iterations, float alpha, int MSE_interval )`

Entraîne le réseau de neurones contenu dans *`mlp` sur `nb_iterations` itérations et avec un taux de correction `alpha`. Si `MSE_interval` est strictement positif alors la moyenne du MSE calculée pendant l'intervalle courant sera sauvegardé toutes les `MSE_interval` itérations. Si `MSE_interval` est négatif, aucun calcul de MSE n'est fait.

`MLP_quickTrain( MLP* mlp )`

Utilise l'équation normale pour déterminer les poids optimaux pour la régression. Pour que cette méthode fonctionne, le réseau de neurones ne doit avoir que 2 couches.

`MLP_generatePrediction( MLP* mlp, int count, … )`

Génère une prédiction en utilisant le réseau de neurones *`mlp` à partir de l'entrée donné après `count`, le nombre d'élément dans l'entrée.

`MLP_getPrediction( MLP* mlp, int index )`

Récupère l'élément à l'indice `index` de la prédiction générée précédemment dans l'objet *`mlp`.

`MLP_getMSESize( MLP* mlp )`

Donne la taille du tableau dans lequel les valeurs de MSE sont stockées.

`MLP_MSE( MLP* mlp, int index )`

Récupère l'élément à l'indice `index` du tableau stockant les valeurs de MSE.

`MLP_test( MLP* mlp )`

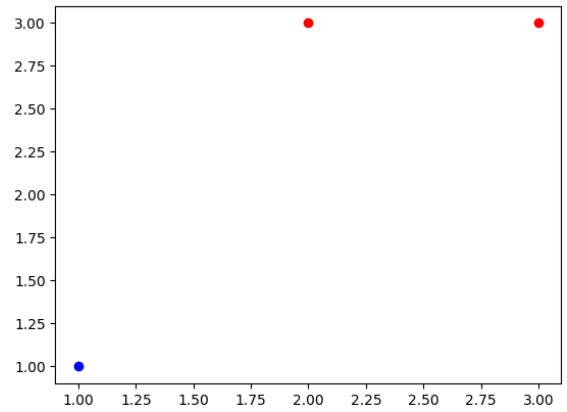
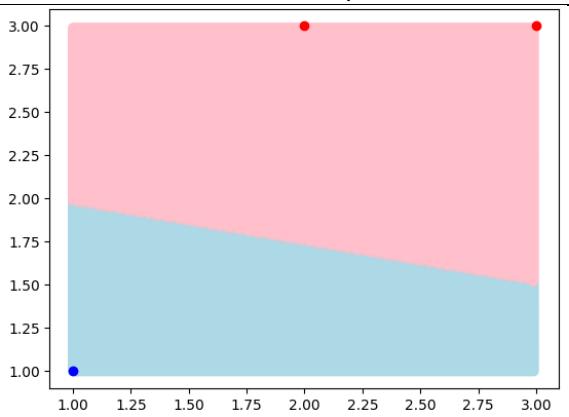
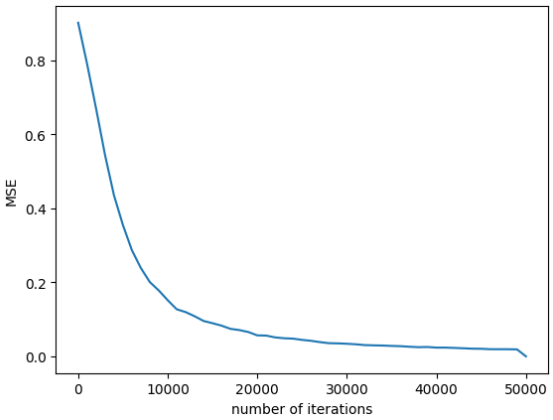
Teste la fiabilité du réseau en générant une prédiction sur chaque entrée stockée et en comparant le signe de chaque élément de la sortie prédite et de la sortie attendue. Retourne en pourcentage le taux de sortie générée correctement prédite.

`MLP_delete( MLP* mlp )`

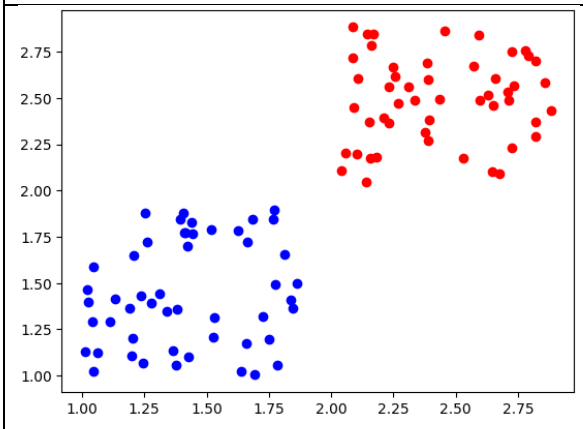
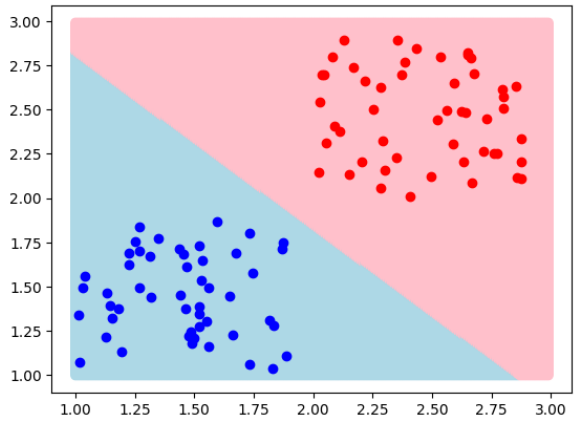
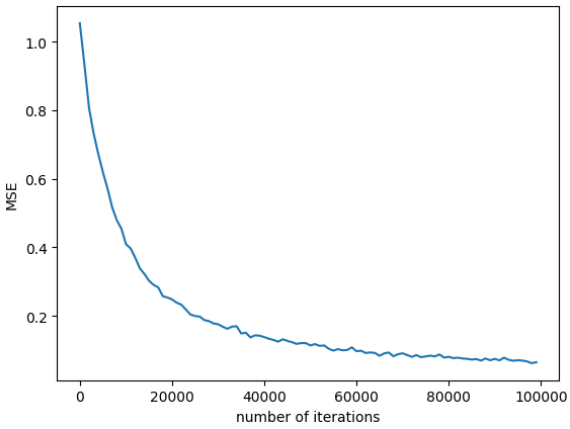
Libère la mémoire qu'occupe l'objet *`mlp`.

b. Classification

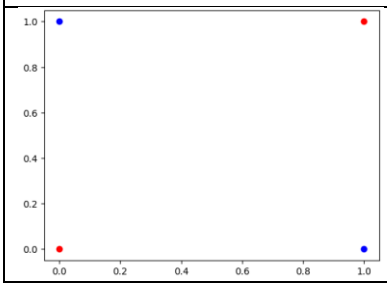
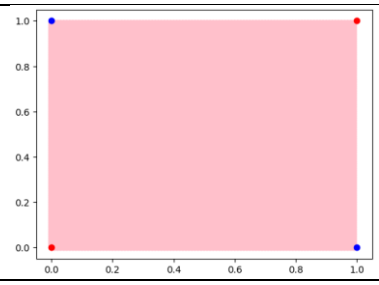
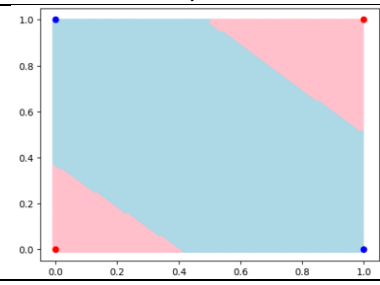
Linear Simple

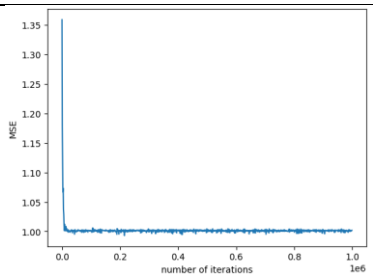
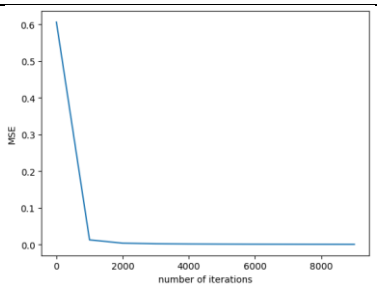
Points du data set	Points du data set classé par un modèle linéaire et un pmc
	  MPL_test : 100% Nombre d'itérations : 50000 Alpha : 0.001

Linear Multiple

Points du data set	Points du data set classé par un modèle linéaire et un pmc
	  MPL_test : 100% Nombre d'itérations : 100000 Alpha : 0.001

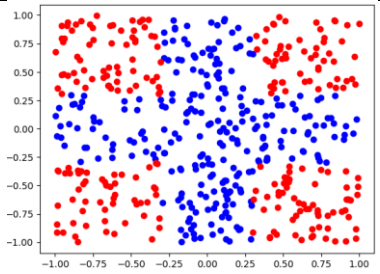
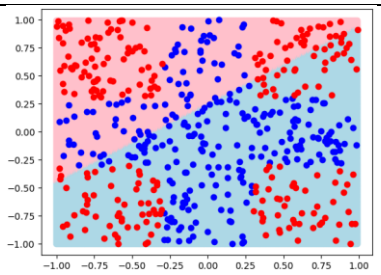
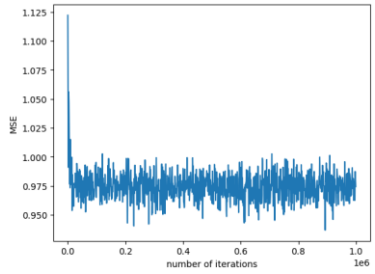
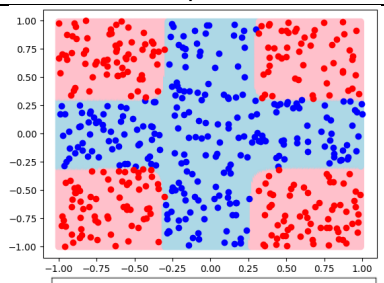
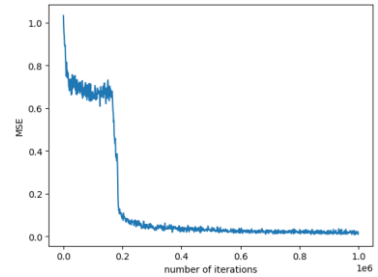
XOR

Points du data set	Points du data set classé par un modèle linéaire	Points du data set classé par un pmc
		

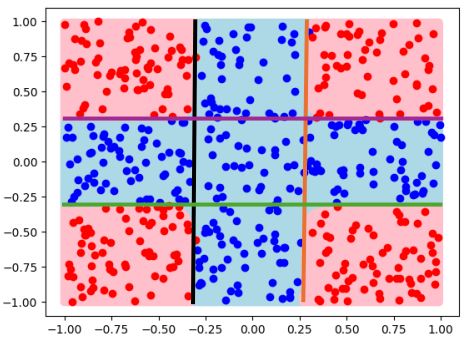
		 MPL_test : 100% Nombre d'itérations : 10000 Alpha : 0.1 Disposition des couches : (2, 2, 1)

Il n'existe pas de droite pouvant diviser correctement les points bleus et les points rouges, c'est pour cela que le modèle linéaire ne peut pas les classifier correctement. Cependant, les points peuvent être classifié correctement en utilisant 2 neurones dans une couche cachées. Cela a pour effet de produire deux droites dans la visualisation.

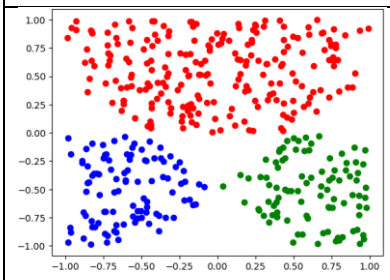
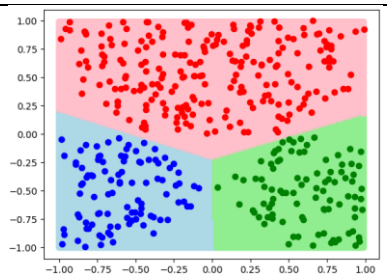
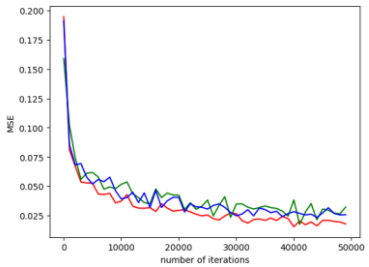
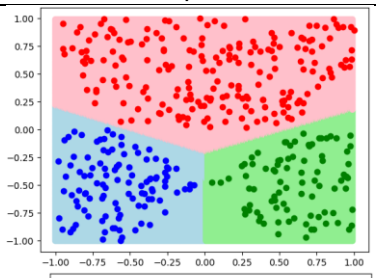
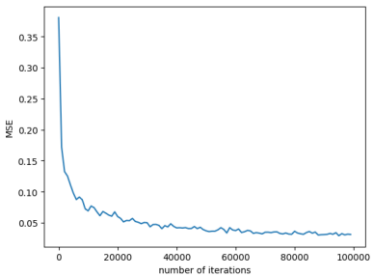
Cross

Points du data set	Points du data set classé par un modèle linéaire	Points du data set classé par un pmc
	  MPL_test : 58.6% Nombre d'itérations : 1000000 Alpha : 0.001	  MPL_test : 99.8% Nombre d'itérations : 1000000 Alpha : 0.01 Disposition des couches : (2, 4, 1)

Il y a 4 neurones dans la couche cachée car il suffit de 4 droites pour diviser l'espace afin de correctement classifier les points d'entrée.

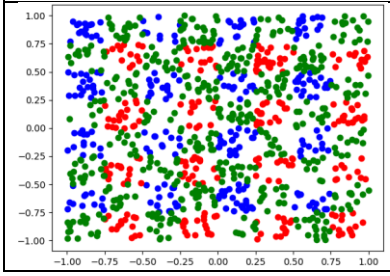
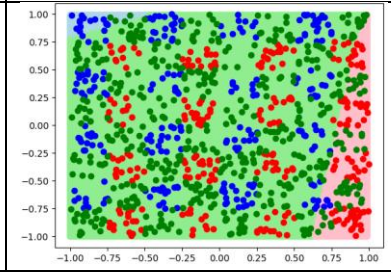
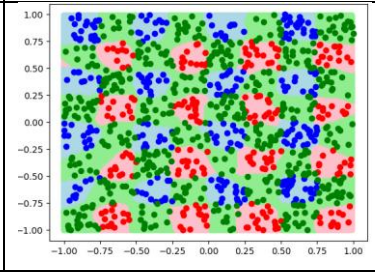


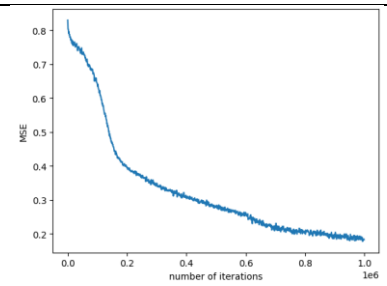
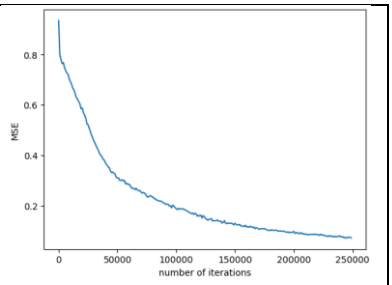
Multi Linear 3 classes

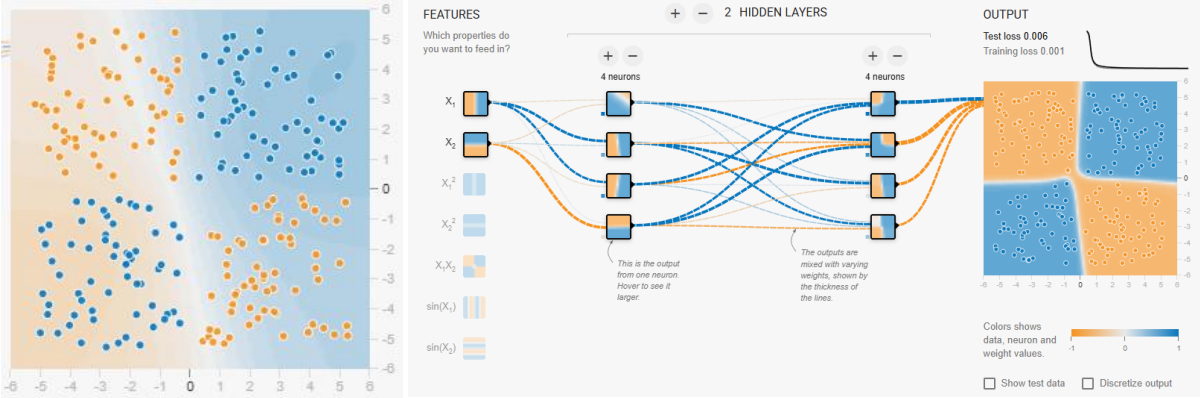
Points du data set	Points du data set classé par trois modèles linéaires	Points du data set classé par un pmc
	  Précision (rouge) : 99.76% Précision (vert) : 99.76% Précision (bleu) : 99.76% Nombre d'itérations : 50000 Alpha : 0.05	  Précision : 99.75% Nombre d'itérations : 100000 Alpha : 0.01 Disposition des couches : (2, 3)

Un perceptron multicouche ayant n neurones dans sa couche de sortie peut être simulé par n modèle linéaire. Pour chaque modèle linéaire, on isole un ensemble de points appartenant à une classe particulière afin qu'il puisse correctement classifier les points de cette classe. Les points de cet ensemble ont une sortie attendue de 1 tandis que les autres points ont une sortie attendue de -1. On interroge ensuite chaque modèle linéaire et la classe retenue est celle liée au modèle linéaire retournant la plus haute valeur. Cette logique se retrouve aussi dans l'interprétation des éléments de la sortie prédite d'un perceptron multicouche. On dédit pour chaque classe un neurone de sortie. Pour chaque point, on définit une sortie 1 sur le neurone dédié à la classe à laquelle appartient le point et -1 pour le reste des autres neurones. Quand on fait une prédiction, on retient la classe liée au neurone ayant la plus haute valeur.

Multi Cross

Points du data set	Points du data set classé par trois modèles linéaires	Points du data set classé par un pmc
		

	 Précision (rouge) : 75.1% Précision (vert) : 51.9% Précision (bleu) : 76.1% Nombre d'itérations : 100000 Alpha : 0.01	 Précision : *94.5%* Nombre d'itérations : *2500000* Alpha : 0.001 Disposition des couches : *(2, 64, 64, 3)*



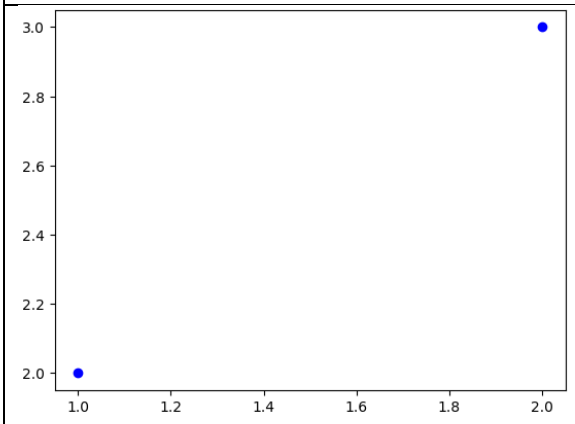
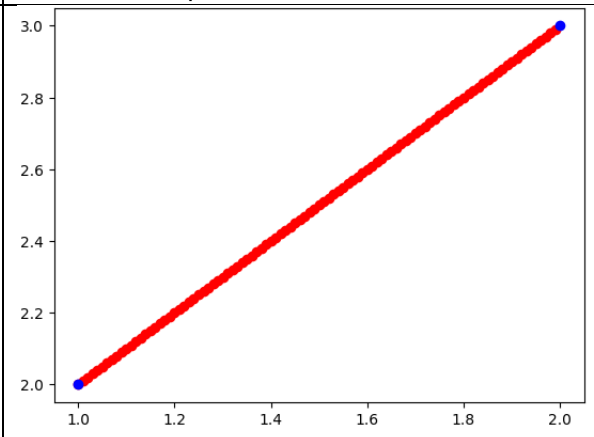
L'intuition d'utiliser deux couches cachées de 64 neurones nous est venu lorsqu'on a manipulé le site https://playground.tensorflow.org/. L'exemple qui nous a intéressé est celui où des points appartenant à deux classes différentes sont réparties dans l'espace en suivant le motif d'un damier, comme le Multi Cross. Deux couches cachées de 4 neurones permettent de classifier des points répartis sur un motif de damier de taille 2 par 2 de manière consistante donc il nous faudrait deux couches cachées de 64 neurones pour correctement classifier de points répartis sur un damier de taille 8 par 8. Cette observation est remise en question par les tests effectués ci-dessous. En effet, la méthode MSE_test retourne le plus haut pourcentage quand les deux couches cachées ont chacune 16 neurones, avec un délai d'entraînement plus court.

Nombre de neurones par couche	Nombre d'itérations	Valeur d'alpha	Temps d'exécution (en secondes)	MSE_test (en pourcentage)
(2, 8, 8, 3)	1000000	0.001	109	61.6
(2, 8, 8, 3)	1000000	0.01	108	60.8
(2, 8, 8, 3)	10000000	0.001	105	56
(2, 16, 4, 3)	10000000	0.001	111	71.8
(2, 16, 4, 3)	5000000	0.01	57	61.8
(2, 16, 4, 3)	10000000	0.01	113	79.1
(2, 16, 16, 3)	5000000	0.01	153	87.8
(2, 16, 16, 3)	5000000	0.001	156	69.3
(2, 16, 16, 3)	10000000	0.001	295	87.7
(2, 16, 16, 3)	25000000	0.001	733	95.1

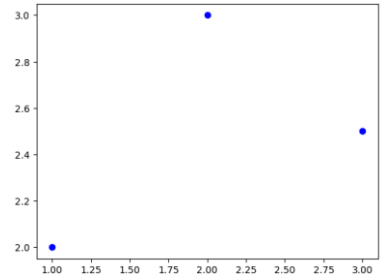
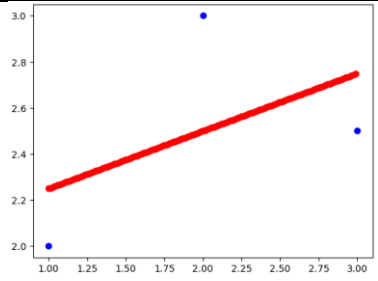
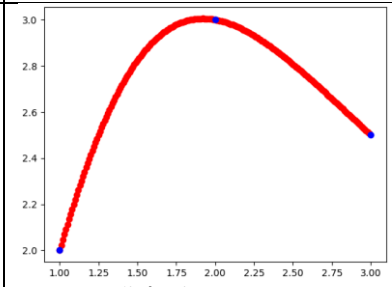
(2, 64, 64, 3)	1000000	0.001	353	90.4
(2, 64, 64, 3)	100000	0.01	42	68.5
(2, 64, 64, 3)	100000	0.001	41	60.8
(2, 64, 64, 3)	100000	0.05	43	48.9
(2, 64, 64, 3)	1000000	0.01	347	90.1
(2, 64, 64, 3)	2500000	0.001	851	94.9
(2, 128, 128, 3)	10000	0.001	27	21
(2, 128, 128, 3)	100000	0.001	141	52
(2, 128, 128, 3)	100000	0.01	140	31
(2, 128, 128, 3)	250000	0.001	331	68.1

c. Régression

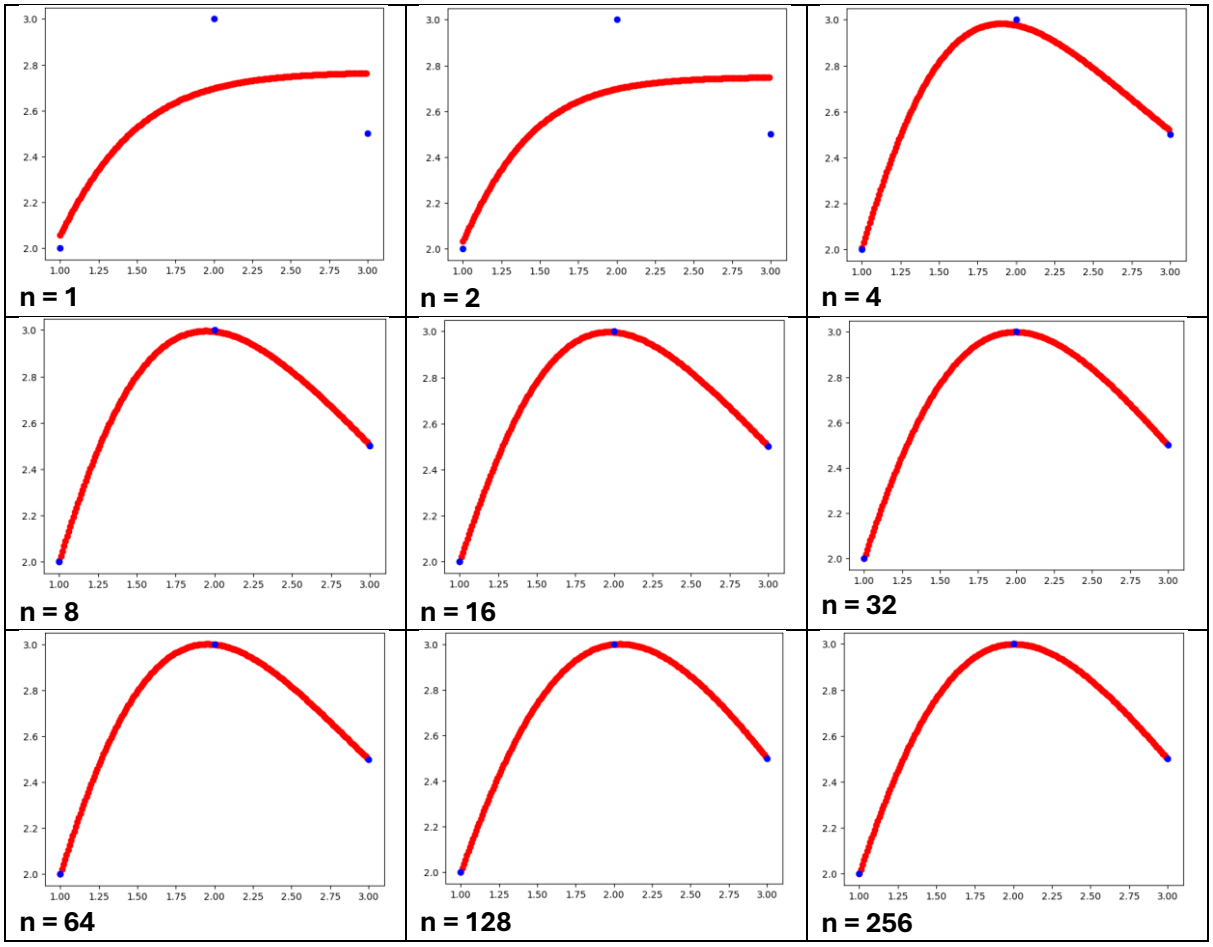
Linear Simple 2D

Points du data set	Points du data set classé par un modèle linéaire et un pmc
	 Entraîné avec l'équation normale

Non Linear Simple 2D

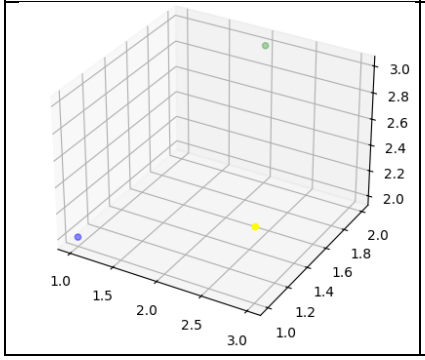
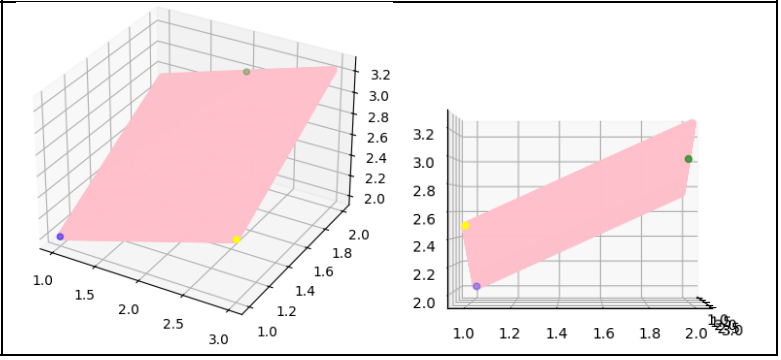
Points du data set	Points du data set classé par un modèle linéaire	Points du data set classé par un pmc
	 Entraîné avec l'équation normale	 Nombre d'itérations : 100000 Alpha : 0.001 Disposition des couches : (2, 16, 1)

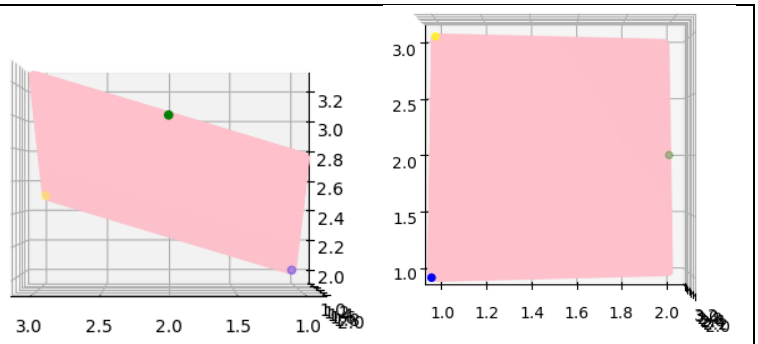
Pour un perceptron ayant un neurone d'entrée, un neurone de sortie et n neurones dans sa seule couche cachée, on obtient le résultat suivant pour 100000 itérations avec un alpha égale à 0.001 :



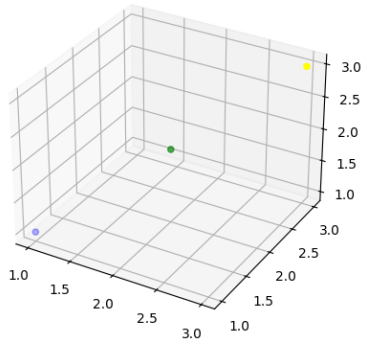
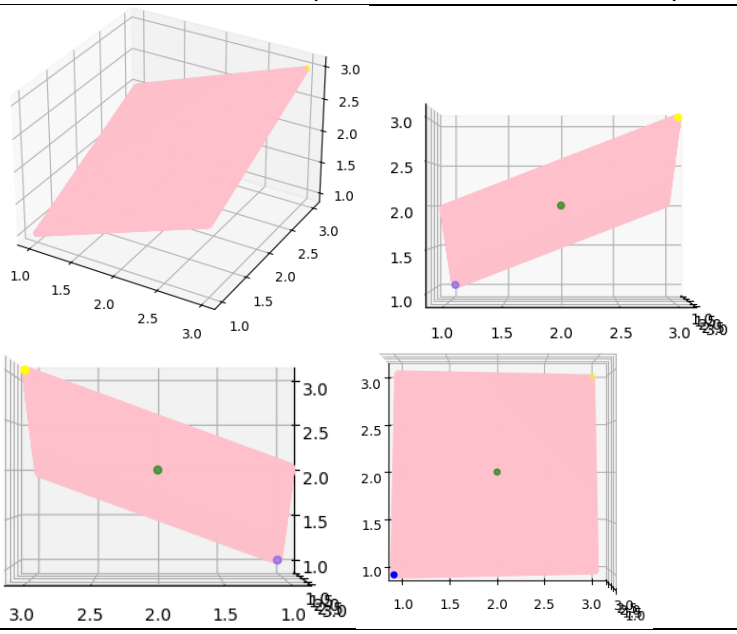
On remarque que la courbe issue des prédictions commence à passer par tous les points lorsque n est entre 5 et 8. On remarque également que plus n est grand, plus le maximum de la courbe se cale sur le deuxième point, et cela de manière de plus en plus consistante.

Linear Simple 3D

Points du data set	Points du data set classé par un modèle linéaire et un pmc
	

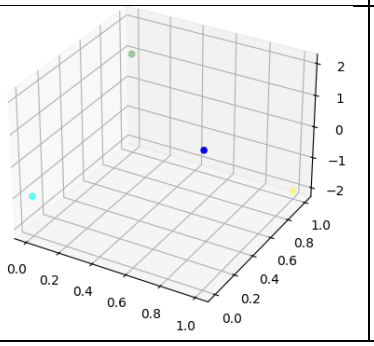
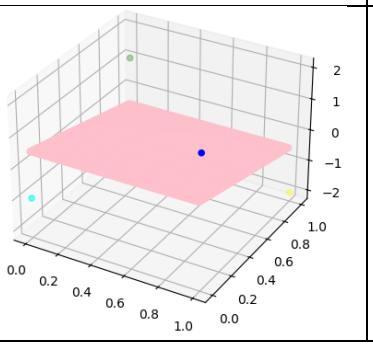
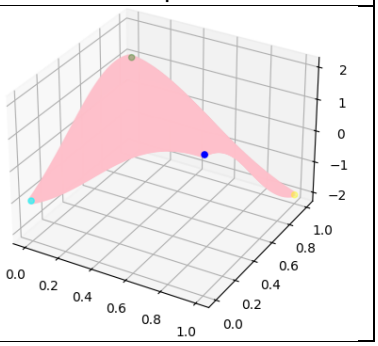
	 Entraîner avec l'équation normale

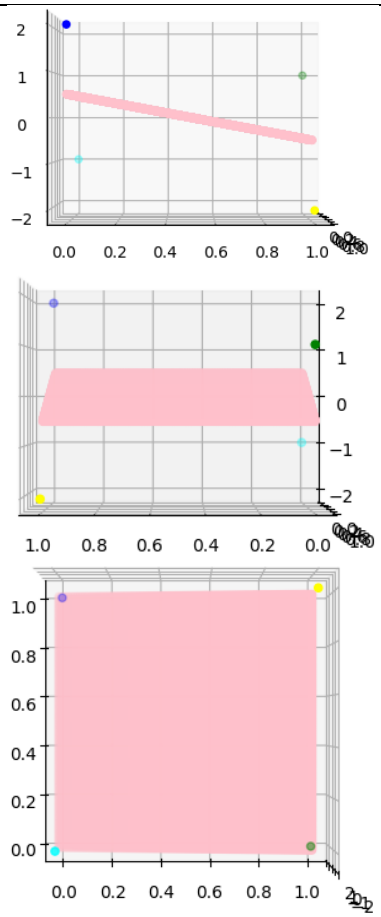
Linear Tricky 3D

Points du data set	Points du data set classé par un modèle linéaire et un pmc
	

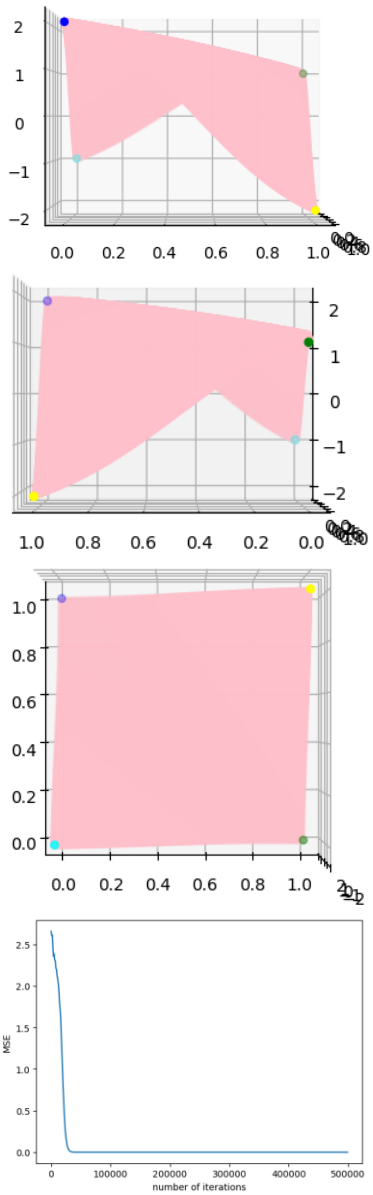
On remarque ici que l'équation normale ne marche pas.

Non Linear Simple 3D

Points du data set	Points du data set classé par un modèle linéaire	Points du data set classé par un pmc
		



Entraîner avec l'équation normale



Nombre d'itérations : 500000
Alpha : 0.001
Disposition des couches :
(2, 2, 1)

Ce test peut être comparé au test XOR, où les points rouges ont une coordonnée en z négative (les points cyan et jaune) tandis que les points bleus ont une coordonnée en z positive (les points bleu et vert). C'est pour cela que le nombre de couche et de neurones par couche est le même que pour le test XOR.

2) Machine Learning pour le jeu vidéo
    a. Rappel de l'objectif

L'objectif de cette partie d'avoir un réseau de neurones capable de jouer à notre jeu type « casse-brique » à partir d'un modèle entrainée de manière supervisée à partir d'un dataset.

L'IA enverra les données du jeu au modèle (position et vitesse de la balle, position du paddle, etc.) et recevra en retour une direction ou aller (gauche, droite, nulle part)

    b. Dataset

Pour créer le dataset, nous avons joué au jeu sur plusieurs parties, et avons récoltés les données suivantes à chaque frames :

- Position X du paddle
- Position X & Y de la balle
- Direction normalisée de la balle (Vecteur 2 dimensions)
- Le nombre de points de vie de chaque *briques (liste fixée au nombre théorique max de briques dans une partie)*
- Les inputs de l'utilisateur
    - Flèche gauche : 0 ou 1
    - Flèche droite : 0 ou 1

Un fichier .csv est créé à chaque parties, ce qui permet après enregistrement de faire un premier tri. (exemple : le joueur a perdu une partie, nous ne souhaitons pas que ces données alimentent le modèle)

Voici une représentation rapide du dataset sur 3 frames *(N = 7*7 = 49)*

inputL	**inputR**	**ballPosX**	**ballPosY**	**BallVelX**	**BallVelY**	**PaddlePosX**	**B0**	**B…**	**BN**
0	1	0,941	0,876	0,707	-0,707	0.771	1	…	2
0	1	0,950	0,868	0,707	-0,707	0.786	1	…	2
0	1	0,953	0,865	0,707	-0,707	0.794	1	…	2

    c. Hypothèses de comportement

Pour ce qui concerne les données sur les briques, nous pensions au début indiquer au modèle leurs positions en plus de leurs points de vie. Nous nous somme cependant rendu compte que la brique à l'index [0] sera toujours en haut à gauche, que celle à l'index [1] sera toujours à sa droite, etc. Nous pensons que le modèle sera assez intelligent pour se dire ou se situe une brique à un index donné.

En essayant de créer le dataset, nous nous sommes rendu compte que l'on essaie surtout de viser dans ces deux situations :

- En début de partie, nous essayons d'envoyer la balle entre deux colonnes, pour que la balle puisse rebondir sur le plus de briques avant de redescendre.
- En fin de partie, lorsque nous essayons de viser une brique en particulier

Nous pensons aussi que l'IA ne pourra pas « perdre ». Toutes les parties que nous enregistrons sont gagnantes, c'est-à-dire que la balle rebondira toujours sur le paddle. Notre hypothèse est que cette ambiguïté est suffisante pour que le modèle comprenne la dimension du paddle.

d. La suite

Ce qui est en train d'être fait :

- Pipeline
    - créer l'algorithme qui à chaque frames va interroger le modèle
    - Créer un « dummy AI » (modèle fictif)
- Finaliser le dataset

Ce qu'il faut faire ensuite :

- Nettoyer le dataset (ajouter du noise ?, supprimer les données redondantes ?)
- Entraîner le modèle
- Ajuster le dataset si besoin et réentraîner le modèle
- Connecter l'IA et le modèle
- Tester, Debugger, Nettoyer le code