

RAPPORT DE STAGE

LES MÉTHODES D'APPRENTISSAGE EN MACHINE LEARNING ET DEEP LEARNING



Maëlle LE LANNIC
Avril 2024 à Juin 2024

Tuteur de stage – Michel BERTHIER

Enseignant référent – Laurent MASCARILLA

Établissement d'enseignement – La Rochelle Université

Organisme d'accueil – Laboratoire Mathématiques Image et Applications

REMERCIEMENTS

Avant tout développement, je souhaite commencer ce rapport de stage par des remerciements à toutes les personnes qui ont contribué à ce stage au sein de l'Université de La Rochelle.

En premier lieu, je tiens à remercier mon tuteur de stage, Mr Michel BERTHIER, pour son encadrement et son aide tout au long de ce stage. Mr BERTHIER m'a proposé ce sujet de stage qui m'a permis de développer de nouvelles connaissances dans le domaine de l'intelligence artificielle. Il a toujours été disponible pour répondre à mes questions et m'éclairer sur les points où j'ai pu rencontrer des difficultés.

Je tiens également à exprimer ma reconnaissance envers Mr Laurent MASCARILLA, mon professeur à l'Université de La Rochelle en Vision embarquée et intelligence artificielle. Grâce à ses enseignements, j'ai pu acquérir des connaissances nécessaires pour aborder ce stage mais aussi pour la suite de mon parcours académique et professionnel.

Enfin, je remercie l'Université de La Rochelle et le laboratoire MIA pour m'avoir offert cette opportunité de stage ainsi que pour l'accueil tout au long de ces deux mois.

SOMMAIRE

1. Introduction – Résumé général du sujet traité	3
2. Présentation de l'entreprise d'accueil	3
3. Méthodes et techniques mise en œuvre – analyse des résultats	6
1 ^{ère} partie – Apprendre à tirer au canon	6
2 ^{ème} partie – Perceptron multicouche	15
4. Planification et gestions des activités	20
5. Synthèse : conclusion du stage	20
6. Annexe – Références et ressources utilisées	22
1 ^{ère} partie – Sources, références	22
2 ^{ème} partie – Codes	22

1. Introduction – Résumé général du sujet traité

En tant qu'étudiante à l'Université de La Rochelle en Licence 3 Informatique, j'ai effectué du 15 avril au 6 juin 2024 mon stage de fin de licence au sein du laboratoire Mathématiques Image et Applications situé au sein de l'Université. Au cours de ce stage, j'ai pu découvrir des méthodes d'apprentissages.

L'objectif de ce stage était d'illustrer à partir d'un exemple concret les méthodes modernes d'apprentissage. Les tâches principales consistaient à développer des algorithmes d'apprentissage, de régression et de réseaux de neurones en utilisant Python.

Pour réaliser les algorithmes, plusieurs librairies ont été utilisées dont Numpy, PyTorch et TensorFlow. Numpy pour les opérations mathématiques et les manipulations des données, PyTorch et TensorFlow pour le développement et l'entraînement de modèles d'apprentissage profond.

La première partie du stage était axé sur l'entraînement d'un réseau de neurones pour apprendre à tirer au canon en abordant des méthodes dites d'apprentissage. L'objectif étant de déterminer la quantité de poudre à insérer dans le canon, qui permettra de contrôler la vitesse de sortie du boulet et l'angle d'inclinaison du canon au moment du tir.

La seconde partie du stage a eu pour but de programmer un petit perceptron multicouche permettant de faire de la classification à partir d'un jeu de données très simple. Cette fois-ci aucune utilisation de librairie d'apprentissage, le développement se fait uniquement avec Numpy. Les données étant classées en deux classes (bon ou mauvais), le but du réseau est de fournir une frontière de décision entre le bien et le mal afin de pouvoir classer les nouvelles données apparaissant en bonnes ou méchantes.

Le développement et la compréhension du réseau de neurones ont été rendus possibles grâce à l'utilisation de plusieurs concepts mathématiques tels que l'algèbre linéaire, régression affine, gradient, descente de gradient, la pseudo inverse de Moore-Penrose et les équations différentielles. Ces concepts mathématiques sont essentiels dans les algorithmes d'apprentissage, leur compréhension est indispensable pour un développement approfondi des modèles.

2. Présentation de l'entreprise d'accueil

Le MIA est un laboratoire pluridisciplinaire de La Rochelle Université. Initialement appelé Laboratoire de Mathématiques et Applications, le laboratoire est restructuré en 2008 intégrant des chercheurs d'autres laboratoires de l'Université et est devenu le Laboratoire de

Mathématiques Image et Applications (MIA). Le laboratoire a une activité scientifique en mathématiques et informatique dans les domaines de l'analyse d'images et de modélisation.

Écosystème de recherche

Le MIA est membre à l'institut de recherche LUDI (Littoral Urbain Intelligent) de La Rochelle Université dans lequel s'inscrivent certaines de ces thématiques liées à l'environnement.

Exemple – Catherine CHOQUET travaille sur l'objectif que La Rochelle s'est fixé, devenir un territoire 0 carbone d'ici 2030

Le MIA fait également partie de fédérations de recherche

- FREDD (Fédération de Recherche en Environnement et Développement Durable) qui travaille également sur les thématiques environnementales
- MIRES (Mathématiques et interactions, Images et Informations numériques, Réseau et Sécurité) réunissant des acteurs de la région en mathématiques et en sciences du numérique
- MARGAUX, nouvelle structure réunissant les mathématiciens du sud-ouest de la France

Activités, projets

Le laboratoire MIA est reconnu pour ses nombreuses contributions scientifiques et technologiques. Il a un positionnement pointu et assumé de haut niveau autour de la modélisation et de la vision qui sont des domaines dans lesquels il possède une excellente visibilité nationale.

La recherche du MIA en mathématiques appliquée est reconnue et attestée par des publications dans des journaux d'excellent niveau dans le domaine des équations aux dérivées partielles, des systèmes dynamiques ou du traitement d'images avec des articles publiés dans SIAM Journal on Imaging Sciences (SIIMS), JMIV (Journal of Mathematical Imaging and Vision).

La production scientifique dans le domaine de la vision par ordinateur et des méthodes destinées aux données images est également d'excellente qualité avec des publications dans des revues de renom comme IEEE PAMI (Transactions on Pattern Analysis and Machine Intelligence)

Quelques exemples de projets

- ➔ Modélisation mathématique de la perception de la couleur : interdisciplinaire, passionnant et important pour faire avancer l'IA
- ➔ Phase-field models and Lattice Boltzmann models for 3D tumors segmentation in high frequency ultrasound images

- Travaux réalisés avec CREATIS (Centre de Recherche en Acquisition et Traitement de l'Image pour la Santé), un des plus gros laboratoire d'imagerie médicale.
 - Cette thèse s'intéresse au problème de la segmentation 3D de tumeurs de la peau dans des images ultrasons de hautes fréquences, comment estimer au mieux le volume des tumeurs ?
- ➔ Un nouvel ensemble de descripteurs de Fourier Clifford pour les images couleurs : les GCFD3
- ML, traitement d'image et vidéo, partie plutôt informatique,
 - Exemple : algèbre de Clifford => transformation de Fourier sur l'espace RGB (les 3 couleurs en même temps)
- ➔ EDP (modélisation, simulation)
- Exemple : modéliser le déplacement des fronts de l'eau salée et de la surface supérieur de l'aquifère (mouvement des polluants dans les aquifères, par exemple avec LBM)

Section CNU (Comité National des Universités)

L'intérêt du MIA, et une de ses importantes caractéristiques, est de réunir 4 sections CNU. Ils travaillent et publient réellement ensemble.

Chercheurs classés selon leur spécialité

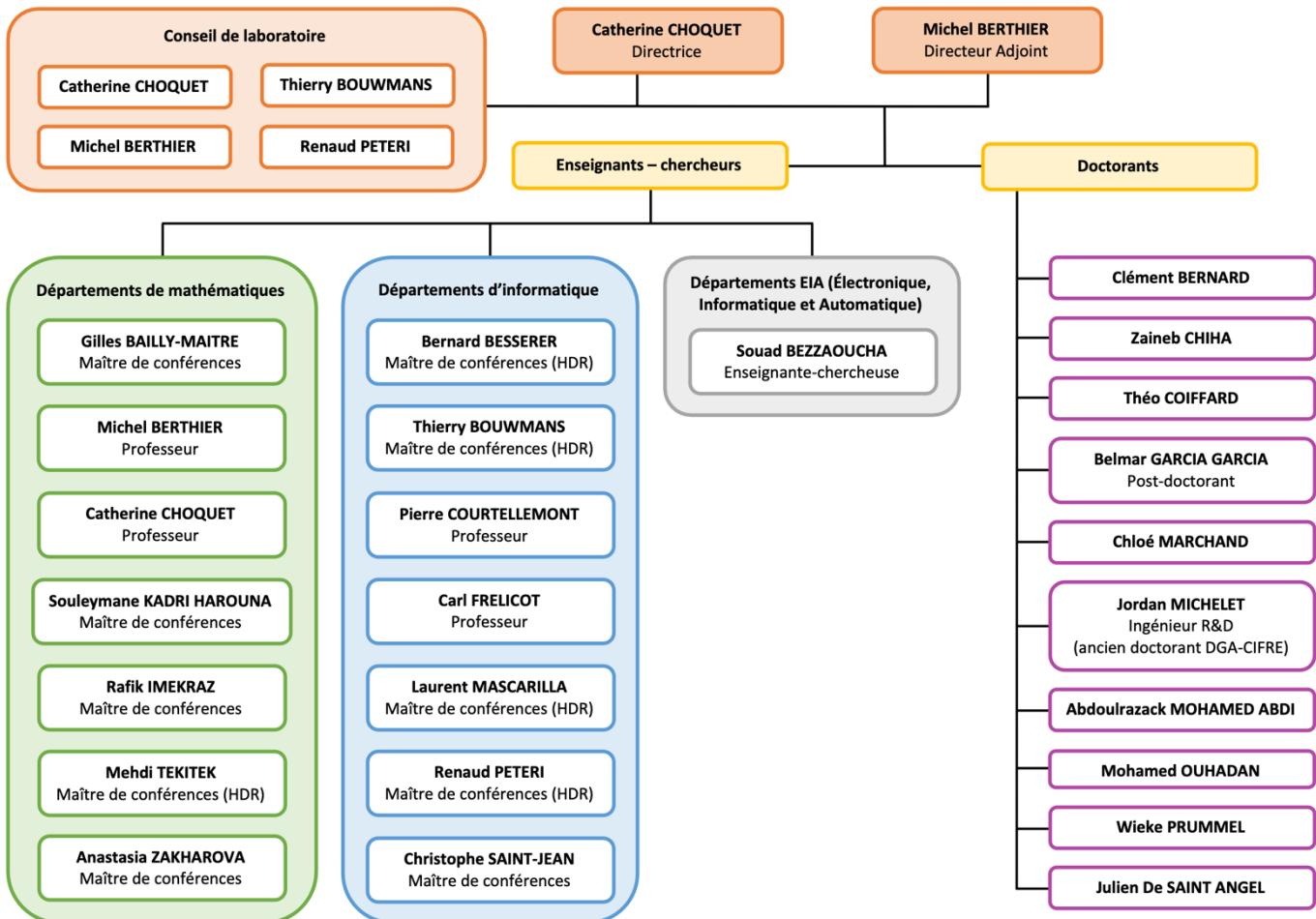
- Michel BERTHIER – section 25 : mathématiques
- Catherine CHOQUET – section 26 : mathématiques appliqués et applications des mathématiques
- Christophe SAINT JEAN – section 27 : informatique
- Renaud PETERI – section 61 : génie informatique, automatique et traitement du signal

Membres du laboratoire MIA

Le laboratoire est constitué d'enseignants-chercheurs, ainsi que de doctorants.

On retrouve des maîtres de conférence (MCF) et des professeurs universitaires (PRU)

- Maître de conférence est le grade obtenu à la suite d'un doctorat (1^{ère} thèse)
- Un professeur universitaire est un ancien MCF qui a réussi un concours spécifique après avoir été habilité à diriger des recherches (HDR), obtenu également par un autre concours.



3. Méthodes et techniques mise en œuvre – analyse des résultats

Comme mentionné dans l'introduction de ce rapport, l'objectif de ce stage était de découvrir les différentes méthodes d'apprentissage. Dans cette partie, je vais détailler les méthodes utilisées et présenter les résultats obtenus.

1^{ère} partie – Apprendre à tirer au canon

Dans cette activité, l'objectif était de faire apprendre à un modèle comment tirer avec un canon. Pour cela, il faut déterminer la quantité de poudre à insérer dans le canon, qui permettra de contrôler la vitesse de sortie du boulet et l'angle d'inclinaison du canon au moment du tir.

Mathématiquement, cela revient à résoudre : $x = T(v, \theta)$

- $x \in \mathbb{R}$, la distance atteinte par le boulet, l'endroit où il touche le sol
- v , la distance de sortie du boulet
- θ , l'angle d'inclinaison du canon au moment du tir

Problématiques – Comment fixer la vitesse v et l'angle θ si je veux atteindre la distance x ?

Comment déterminer la fonction inconnue T ?

La modélisation mathématique

1. Calcul de la fonction T

Le mouvement du boulet de canon est régi par l'équation de Newton : $m\ddot{X}(t) = F(X(t))$

- m , la masse du boulet, $m = 1$
- $X(t) = (x(t), y(t))$, la position du boulet à l'instant t
- $y(t)$, l'altitude du boulet à l'instant t
- $F(X(t))$, la somme des forces s'exerçant sur le boulet à l'instant t

L'équation de Newton est une équation du deuxième ordre, dans notre cas $F(X(t))$ est le vecteur constant de coordonnées $(0, -g)$ où g désigne la constante de gravitation.

On obtient alors le système suivant : $\begin{cases} \ddot{x}(t) = 0 \\ \ddot{y}(t) = -g \end{cases}$

\ddot{x} désigne la dérivée seconde de x et \ddot{y} désigne la dérivée seconde de y , d'où :

$$\begin{aligned} \ddot{x}(t) = 0 & \quad \ddot{y}(t) = -g \\ \Leftrightarrow \dot{x}(t) = a & \quad \Leftrightarrow \dot{y}(t) = -g \times t + a \\ \Leftrightarrow x(t) = a \times t & \quad \Leftrightarrow y(t) = -\frac{1}{2}gt^2 + a \times t \end{aligned}$$

On a donc $y(t) = -\frac{1}{2}gt^2 + at$, avec g la constante de gravitation ($\simeq 9.8$)

$y(t)$ désigne l'altitude du boulet à l'instant t donc a représente la vitesse,

D'où $a = v \times \sin \theta$ car $d = v \times t$

$y(t) = 0$ lorsque le boulet est au sol

$$\begin{aligned} \Leftrightarrow -\frac{1}{2}gt^2 + v \sin \theta t = 0 \\ \Leftrightarrow t(-\frac{1}{2}gt + v \sin \theta) = 0 \\ \Leftrightarrow t = 0 \text{ ou } -\frac{1}{2}gt + v \sin \theta = 0 \end{aligned}$$

$t = 0$: lors du lancement du boulet (encore dans le canon)

$$-\frac{1}{2}gt + v \sin \theta = 0 \Leftrightarrow t = \frac{2v \sin \theta}{g}$$

$x(t)$ désigne la position du boulet à l'instant t (longitude) donc a représente la vitesse horizontale,

D'où $a = v \times \cos \theta$

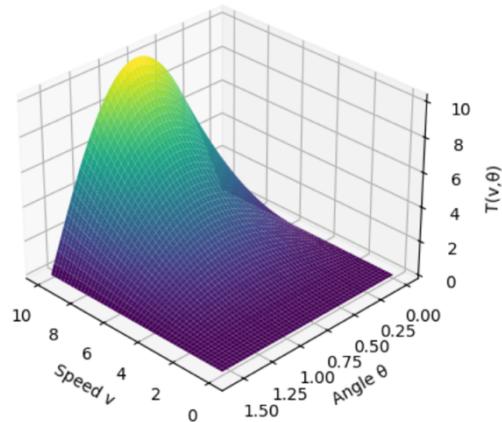
$$x(t) = v \times \cos \theta \times t = v \times \cos \theta \times \frac{2v \sin \theta}{g} = \frac{v^2 \times 2 \times \sin \theta \times \cos \theta}{g} = \frac{v^2 \sin(2\theta)}{g}$$

On en déduit que $T(v, \theta) = x = \frac{v^2 \sin(2\theta)}{g}$

Règles utilisées

- $f(x) = c \Leftrightarrow f'(x) = 0$
- $f(x) = cx \Leftrightarrow f'(x) = c$
- $f(x) = x^n \Leftrightarrow f'(x) = nx^{n-1}$
- $2\sin(x)\cos(x) = \sin(2x)$

Graph of the function T



Afin de visualiser la fonction $T(v, \theta)$, j'ai utilisé 3 méthodes pour simuler les solutions du système qui ont donné le même résultat

- Calcul du résultat avec la fonction T
- Méthode d'Euler
- Bibliothèque Odeint

2^{ème} méthode – la méthode d'Euler consiste à remplacer la dérivée de la fonction par une différence finie et de progresser à travers des pas de discréétisation h pour résoudre une équation différentielle ordinaire de la forme $\frac{dy}{dt} = f(t, y)$ avec une condition initiale de $y(t_0) = y_0$

Il faut choisir un pas de discréétisation h puis calculer les valeurs suivantes de y en utilisant la formule d'Euler : $y_{n+1} = y_n + h \times f(t_n, y_n)$ où y_{n+1} est l'approximation de y à l'instant t_{n+1}

$$t_{n+1} = t_n + h$$

Dans notre problème la condition initiale est $x_0 = y_0 = 0$
c'est-à-dire lorsque le boulet est encore dans le canon.

$$\frac{d}{dt} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ -g \end{pmatrix} = F(x(t), y(t))$$

$$\begin{aligned} \ddot{x}(t) &= 0 & \ddot{y}(t) &= -g \\ \Leftrightarrow \dot{x}(t) &= v \times \cos \theta & \Leftrightarrow \dot{y}(t) &= -g \times t + v \times \sin \theta \end{aligned}$$

$$\text{D'où } z(t+1) = z(t) + h \times z'(t) \Leftrightarrow \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} + h \times \begin{pmatrix} v \times \cos \theta \\ v \times \sin \theta - g \times h \end{pmatrix}$$

3^{ème} méthode – la bibliothèque `odeint` de la bibliothèque `SciPy` permet aussi de résoudre une équation différentielle ordinaire. Il suffit de faire passer la fonction dérivée, les conditions initiales et les points à calculer ainsi que les arguments pour la fonction dérivée s'il y en a.

La méthode d'Euler est simple mais peut manquer de précision par rapport à la bibliothèque Odeint qui convient à une gamme plus large de problèmes offrant une meilleure précision grâce à des algorithmes sophistiqués. Elle est donc à privilégier pour des problèmes nécessitant une résolution numérique de précision au vu de sa robustesse et de sa facilité d'utilisation.

Les données d'apprentissage

Afin d'entraîner le modèle, il faut générer des données. Cependant on ne peut pas tirer au canon. Je les ai donc générées avec la fonction $T(\nu, \theta)$ en ajoutant un peu de bruit. L'apprentissage s'est donc fait à partir d'un jeu de données de 1000 couples $(x, (\nu, \theta))$.

L'une des méthodes les plus simples pour faire des prédictions à partir de données d'apprentissage consiste à utiliser des régressions. Il m'a été demandé de faire des prédictions avec deux types de régression, une en utilisant la pseudo inverse de Moore-Penrose et une autre en utilisant des descentes de gradient.

1. Pseudo inverse de Moore-Penrose

La pseudo-inverse de Moore-Penrose est une généralisation de l'inverse d'une matrice qui peut être appliquée à des matrices carrées ou non (des matrices qui ne sont pas inversibles).

Il m'a été demandé de faire deux démonstrations mathématiques avec des conditions initiales différentes.

$$\Rightarrow x_0 = (A^T A)^{-1} A^T b =: A^+ b$$

Il faut chercher les points critiques de F , c'est-à-dire $d_x F(h)$

$$x_0 = \underset{x \in \mathbb{R}^n}{\operatorname{argmin}} \|Ax - b\|_2^2$$

$$\|Ax - b\|_2^2 = \langle Ax - b, Ax - b \rangle = F(x)$$

$$F(x + h) - F(x) = 2 \langle Ah, Ax - b \rangle$$

À quelle condition $d_x F(h) = 0 \forall h$?

$$F(h) = 2 \langle Ah, Ax - b \rangle = 0 \Leftrightarrow \langle h, A^T(Ax - b) \rangle = 0 \Leftrightarrow A^T(Ax - b) = 0$$

$$\Leftrightarrow x = (A^T A)^{-1} A^T b = A^+ b$$

$$\Rightarrow x_0 = A^T (A A^T)^{-1} b =: A^+ b$$

$$x_0 = \underset{x \in \mathbb{R}^n, Ax=b}{\operatorname{argmin}} \|x\|_2^2 \quad \text{La contrainte } Ax = b \Leftrightarrow Ax - b = 0$$

$$\|x\|_2^2 = \langle x, x \rangle = x^2$$

Il faut utiliser le multiplicateur de Lagrange : λ est un vecteur, il faut donc utiliser sa transposé pour pouvoir le multiplier avec une matrice

$$\mathcal{L}(x, y) = \|x\|_2^2 - \lambda^T (Ax - b) = x^2 - \lambda^T Ax + \lambda^T b$$

$$\frac{\partial \mathcal{L}}{\partial x} = 0 \Leftrightarrow 2x - \lambda^T A \Leftrightarrow x = \frac{\lambda^T A}{2}$$

$$\frac{\partial \mathcal{L}}{\partial y} = 0 \Leftrightarrow Ax - b = 0 \Leftrightarrow A \times \frac{\lambda^T A}{2} = b \Leftrightarrow AA^T \lambda = 2b \Leftrightarrow \lambda = \frac{2b}{AA^T} = (AA^T)^{-1} 2b$$

Ceci est possible car AA^T est inversible

$$x = \frac{\lambda^T A}{2} = \frac{A^T \lambda}{2} = A^T (AA^T)^{-1} b = A^+ b$$

Règles utilisées

- $(A^t B)^t = A^t B \Rightarrow \lambda^t A = (\lambda^t A)^t$
- $(AB)^t = B^t A^t \Rightarrow (\lambda^t A)^t = A^t (\lambda^t)$
- $(A^t)^t = A \Rightarrow (\lambda^t)^t = A^t \lambda$

La pseudo-inverse de Moore-Penrose est utile pour programmer des régressions puisqu'elle permet de résoudre des systèmes d'équation comprenant des matrices qui ne sont pas inversibles.

2. La descente de gradient

Un gradient est un vecteur qui remplace la notion de dérivée pour les fonctions de plusieurs variables. On sait que la dérivée permet de décider si une fonction est croissante ou décroissante. De même, le vecteur gradient indique la direction dans laquelle la fonction croît ou décroît le plus vite.

Le gradient est un vecteur dont les coordonnées sont les dérivées partielles. Il a de nombreuses applications géométriques car il donne l'équation des tangences aux courbes et surfaces de niveau. Surtout, il indique la direction dans laquelle la fonction varie le plus vite.

Le gradient de f en (x_0, y_0) , noté $\text{grad } f(x_0, y_0)$ est le vecteur : $\nabla f(x_0, y_0) = \begin{pmatrix} \frac{\partial f}{\partial x}(x_0, y_0) \\ \frac{\partial f}{\partial y}(x_0, y_0) \end{pmatrix}$

Dans les réseaux de neurones, le gradient joue un rôle dans le processus d'apprentissage et de mise à jour des poids des connexions entre les neurones. La descente de gradient est un processus d'optimisation utilisé pour entraîner les réseaux de neurones. Le but est de minimiser une fonction de coût qui mesure l'erreur entre les prédictions du réseau et les valeurs réelles des données d'entraînement en ajustant les paramètres du modèle.

Pour minimiser l'erreur et déterminer les meilleurs paramètres on peut appliquer la méthode classique, on part d'un point $P_0 = (a_1, \dots, a_n)$ puis on applique la formule de récurrence :

$$P_{k+1} = P_k - \delta \nabla(P_k)$$

3. Prédictions par régression

La plus simple des régressions est la régression affine, dans notre problème cela correspond au système suivant :

$$\begin{pmatrix} v_1 & \theta_1 & 1 \\ v_2 & \theta_2 & 1 \\ \dots & \dots & \dots \\ v_n & \theta_n & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}$$

Ce système permet d'approcher la fonction $(v, \theta) \rightarrow x = T(v, \theta)$ par la fonction :

$$(v, \theta) \rightarrow av + b\theta + c$$

Pour notre fonction $T(v, \theta)$ ce système à ses limites, le système suivant

$$\begin{pmatrix} v_1 & v_1^2 & v_1 \sin(\theta_1) & v_1 \sin(2\theta_1) & v_1^2 \sin(\theta_1) & v_1^2 \sin(2\theta_1) \\ v_2 & v_2^2 & v_2 \sin(\theta_2) & v_2 \sin(2\theta_2) & v_2^2 \sin(\theta_2) & v_2^2 \sin(2\theta_2) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ v_n & v_n^2 & v_n \sin(\theta_n) & v_n \sin(2\theta_n) & v_n^2 \sin(\theta_n) & v_n^2 \sin(2\theta_n) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}$$

est un peu plus complexe et permet d'approcher la fonction $(v, \theta) \rightarrow x = T(v, \theta)$ par la fonction :

$$(v, \theta) \rightarrow a_1v + a_2v^2 + a_3v \sin(\theta) + a_4v \sin(2\theta) + a_5v^2 \sin(\theta) + a_6v^2 \sin(2\theta)$$

1^{ère} méthode – Pseudo-inverse de Moore-Penrose

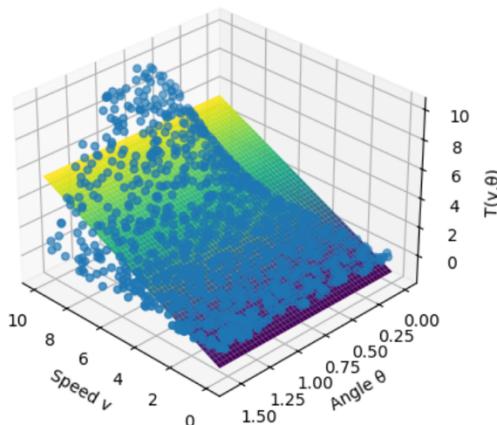
En utilisant la pseudo inverse de Moore-Penrose, on obtient pour la régression linéaire :

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} v_1 & \theta_1 & 1 \\ v_2 & \theta_2 & 1 \\ \dots & \dots & \dots \\ v_n & \theta_n & 1 \end{pmatrix}^+ \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}$$

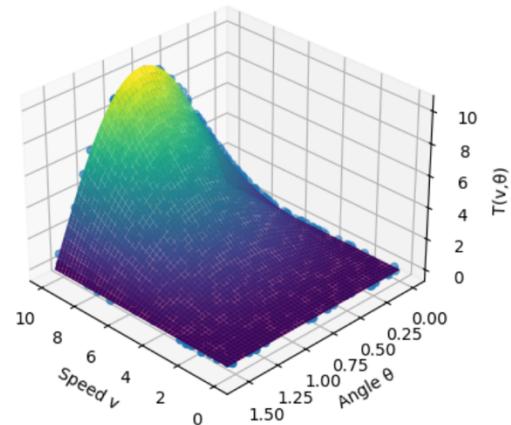
En calculant les coefficients de régression linéaire de cette manière, on peut implémenter la fonction $(v, \theta) \rightarrow av + b\theta + c$ et ainsi obtenir la courbe de régression.

Pour le deuxième système, on procède de la même manière.

Linear regression graph of Learning data with Moore-Penrose



Regression graph of Learning data with Moore-Penrose



2^{ème} méthode – Descente de gradient

Cette fois-ci, l'objectif est de minimiser les fonctions de coût, les loss associés au problème de régression, en utilisant la descente de gradient dans l'espace des paramètres de la régression.

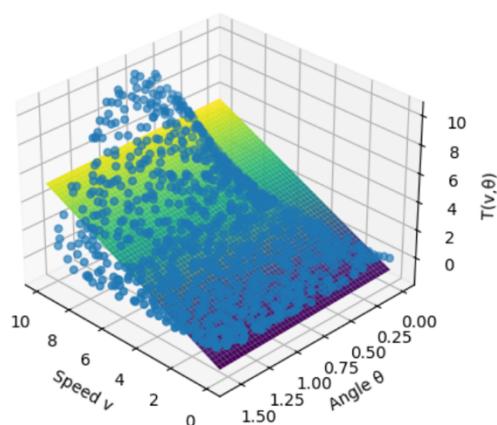
On commence par définir des valeurs initiales aux coefficients de régression a , b et c en les initialisant par exemple à 0. A chaque itération, les gradients de la fonction de coût par rapport aux coefficients sont obtenus par différence automatique avec TensorFlow. Ces gradients sont ensuite utilisés pour mettre à jour les coefficients selon la formule de récurrence.

Chaque itération représente une époque, c'est-à-dire le nombre de fois que l'algorithme parcourt l'ensemble des données d'entraînement pour ajuster les coefficients.

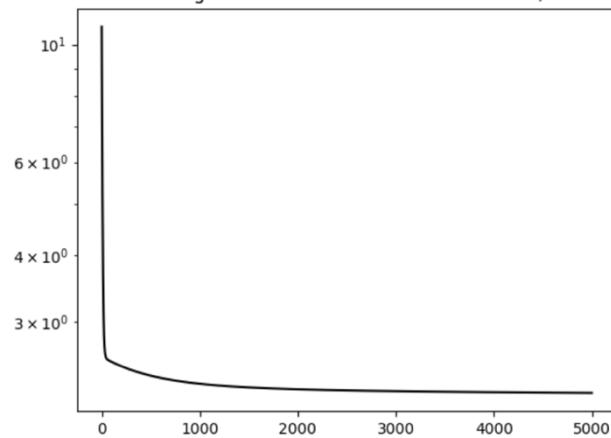
Après l'optimisation, les valeurs ajustées sont utilisées pour prédire les valeurs de sortie.

Pour la régression affine, le loss est donné par $\mathcal{L}(a, b, c) = \sum_{i=1}^{1000} ((a v_i + b \theta_i + c) - x_i)^2$
 x_i représente la valeur des données d'entraînements.

Linear regression graph of Learning data by GD

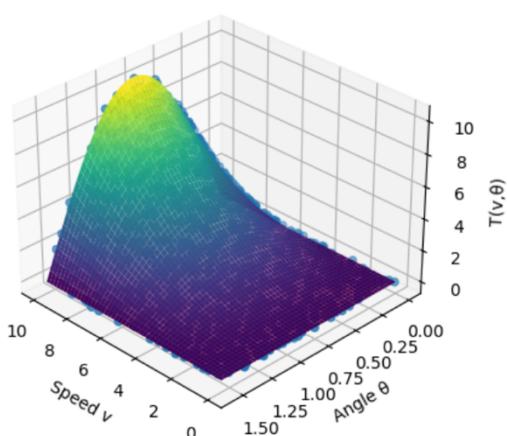


GD for linear regression with Tensorflow : Niter=5000, δ = 0.001

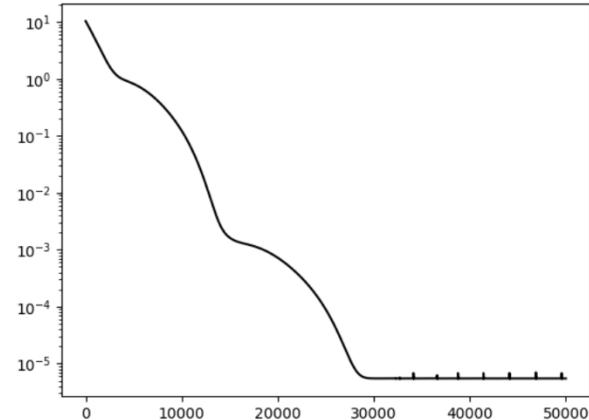


Pour le deuxième système, le loss est donné par : $\mathcal{L}(a_1, a_2, a_3, a_4, a_5, a_6) = \sum_{i=1}^{1000} ((a_1 v_i + a_2 v_i^2 + a_3 v_i \sin(\theta_i) + a_4 v_i \sin(2\theta_i) + a_5 v_i^2 \sin(\theta_i) + a_6 v_i^2 \sin(2\theta_i)) - x_i)^2$

Regression graph of Learning data with DG

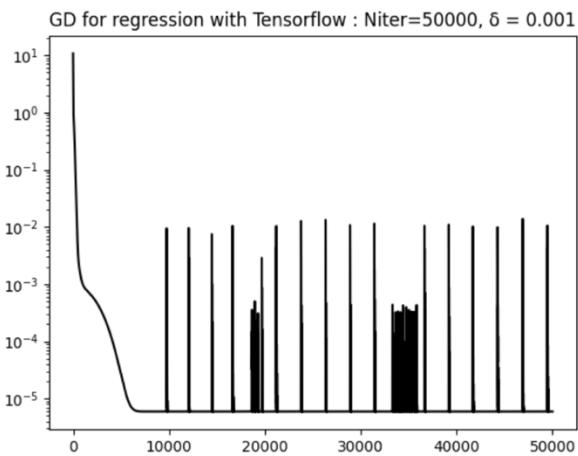


GD for regression with Tensorflow : Niter=50000, initial δ = 0.00001



Pour ce système, on doit utiliser un optimiseur afin de rentre le Learning Rate adaptative. Cela permet d'avoir des pas plus adaptés à chaque paramètre et ainsi éviter des pas trop grands entraînant des valeurs infinies pour les coefficients.

J'ai également diminué le Learning Rate initiale car j'obtenais des oscillations/des pics dans la courbe du loss. Ceci est dû au fait que l'optimiseur fait des pas trop grands dans la direction de la descente du gradient dépassant parfois les minimums locaux ou le minimum global.



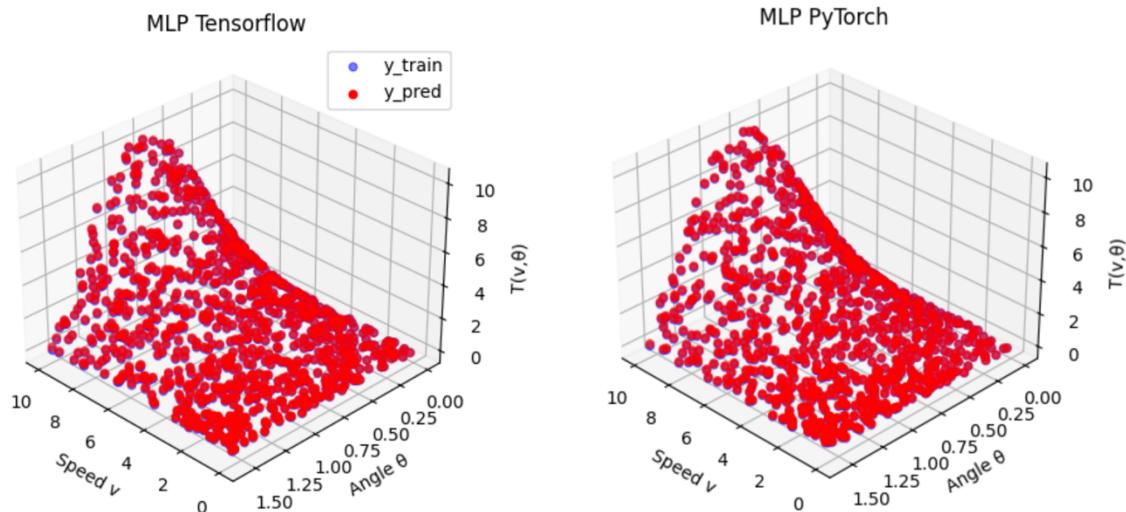
Multi-Layer Perceptron (MLP)

Les modèles d'apprentissage utilisé jusqu'à présent, notamment les régressions, sont les plus simples mais leur mise en œuvre nécessite une certaine expertise puisqu'il faut proposer une fonction à approximer. Les méthodes d'apprentissage modernes permettent d'apprendre des modèles à partir des données d'apprentissage, c'est le cas des réseaux de neurones de type Multi Layer Perceptron (MLP).

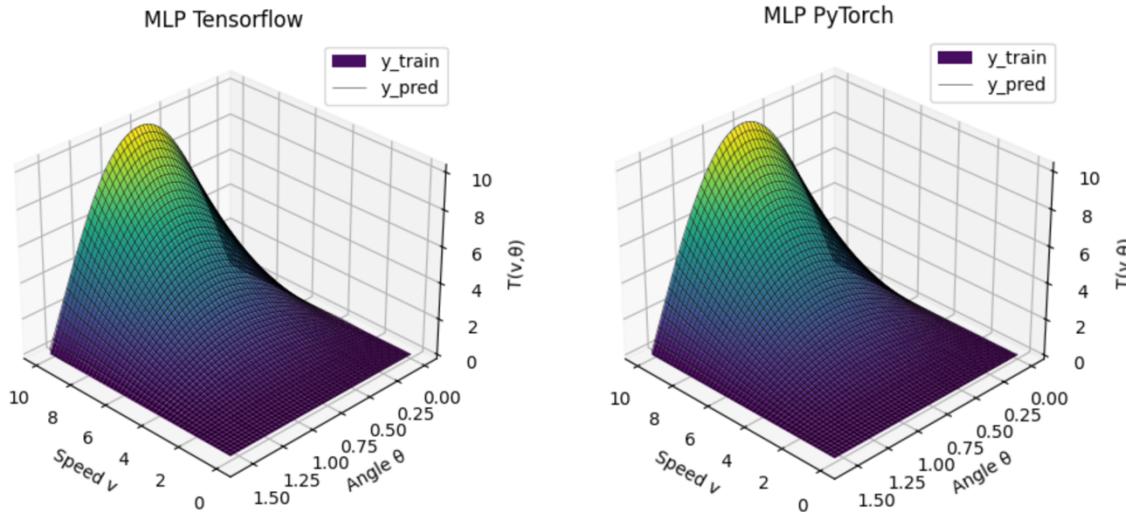
J'ai développé deux programmes python permettant de visualiser les résultats obtenus avec un MLP à partir des données d'apprentissage, une version `Tensorflow` et une version `PyTorch`.

Le MLP comporte trois couches avec une fonction d'activation sigmoïde pour les deux premières et une fonction identité à la sortie de la dernière couche. Pour la fonction de perte, j'ai utilisé la fonction MSE (Mean Squared Error), avec l'optimiseur Adam et un taux d'apprentissage (learning rate) de 0.01. L'apprentissage est réalisé sur 1000 époques (epoch) et des lots de données de taille 100 (batch).

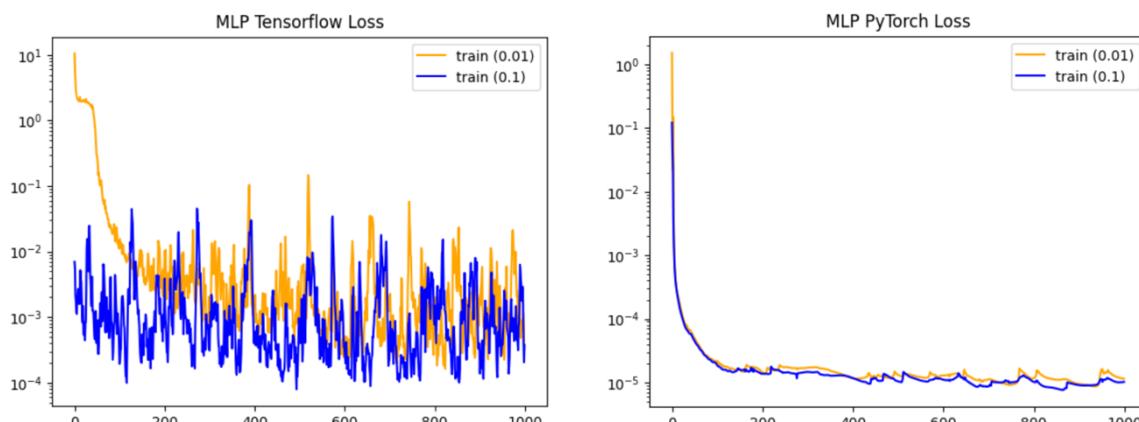
Pour l'apprentissage mlp, j'ai utilisé deux jeux de données ayant un niveau de bruit différent. L'ajout de bruit aux données d'entraînement est une technique permettant d'améliorer la robustesse et la capacité de généralisation des modèles de machines learning. En forçant le modèle à apprendre des caractéristiques plus générales et en le préparant à des données réelles bruitées (capteurs imparfaits, erreur de mesure, etc..), on le rend plus performants et plus fiables. Cela permet d'évaluer la capacité du modèle à maintenir ses performances lorsqu'il est confronté à des variations et des imperfections de données, cela réduit aussi le risque de surapprentissage.



Ces deux graphiques représentent les points correspondant à un couple du premier jeu de données. Les points bleus représentent les valeurs réelles attendues alors que les points rouges représentent les valeurs prédites. On remarque que les points prédits se superposent aux données d'entraînement, ce qui indique une bonne précision dans la prédiction.



Une autre manière de visualiser les résultats est à travers la grille qui représente les prédictions. On constate que cette grille recouvre bien les données d'entraînement ce qui confirme la qualité des prédictions effectuées par le modèle.



Concernant les valeurs de perte (loss), de légères fluctuations peuvent être observé sur la courbe de perte de la version TensorFlow mais compte tenu de l'échelle, elles ne sont pas significatives.

Les résultats obtenus sont les mêmes, on peut cependant noter que la version TensorFlow a une bien meilleure performance par rapport au temps d'exécution qui est d'environ 40 secondes alors que la version PyTorch à besoin d'environ 8 minutes.

2^{ème} partie – Perceptron multicouche

Dans cette activité, qui est une introduction au Deep Learning, l'objectif était de programmer un modèle, un petit perceptron multicouche permettant de faire la classification d'un jeu de données. Le jeu de données utilisé est constitué de 10 points qui sont divisés en deux classes. Le but du réseau est de fournir une frontière de décision entre les deux classes afin de pouvoir classer les nouvelles données apparaissant dans la bonne classe.

Le machine learning est un domaine de l'intelligence artificielle qui consiste à programmer une machine pour que celle-ci apprenne à réaliser des tâches en étudiant des exemples (données) de ces dernières. Pour cela, on programme un algorithme d'optimisation qui va tester différentes valeurs jusqu'à obtenir la combinaison qui minimise la distance entre le modèle et les points. L'algorithme d'optimisation permet de minimiser les erreurs entre le modèle et les données.

Mais qu'est-ce qu'un perceptron ?

Le perceptron entraîne un neurone artificiel sur des données de références pour que celui-ci renforce ses paramètres à chaque fois qu'une entrée est activée en même temps que la sortie y . L'idée est de considérer des neurones fabriqués à partir d'équations d'hyperplans et d'une fonction d'activation.

Pour entraîner un réseau de neurone

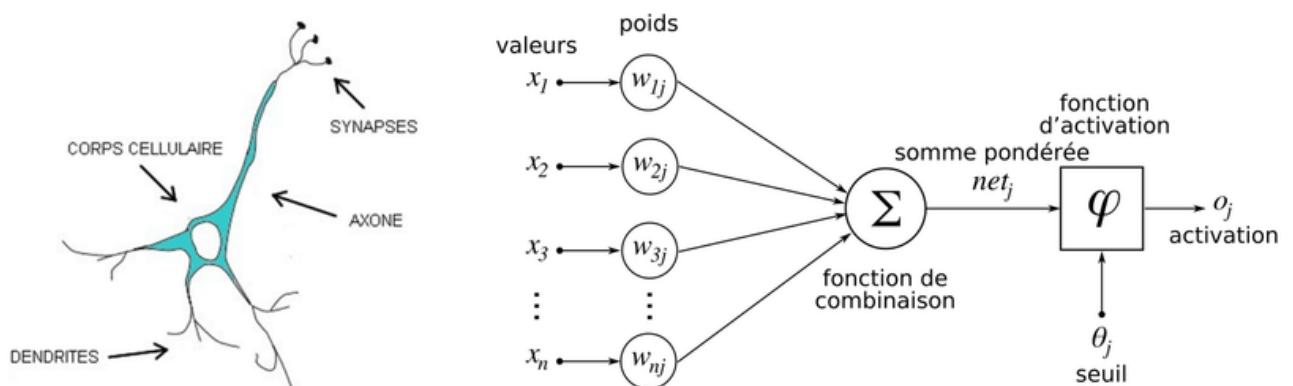
1. Définir une fonction coût permettant d'évaluer les erreurs du modèle
2. Calculer les dérivées partielles permettant de comprendre comment la fonction coût évolue par rapport aux différents paramètres w et b
3. Mettre à jour les paramètres w et b permettant de minimiser la fonction coût grâce à la descente de gradient.

Les neurones sont des cellules excitables connectées les unes aux autres et ayant pour rôle de transmettre des informations au système nerveux. Les dendrites sont les portes d'entrée d'un neurone, à cet endroit au niveau de la synapse le neurone reçoit des signaux lui provenant des neurones qui le précédent. Ces signaux peuvent être de type excitateur ou à l'inverse

inhibiteur. Lorsque la somme de ces signaux dépasse un certain seuil, le neurone s'active et produit un signal électrique. Ce signal circule le long de l'axone en direction des terminaisons pour être envoyé à son tour vers d'autres neurones de notre système nerveux.

On peut faire la comparaison avec un neurone artificiel :

- les dendrites : les entrées du réseau représentent les signaux
- le corps cellulaire : la fonction d'activation, la somme pondérée des signaux passe par la fonction d'activation qui décide d'activer ou non le neurone.
- l'axone : la sortie, le résultat de la fonction d'activation



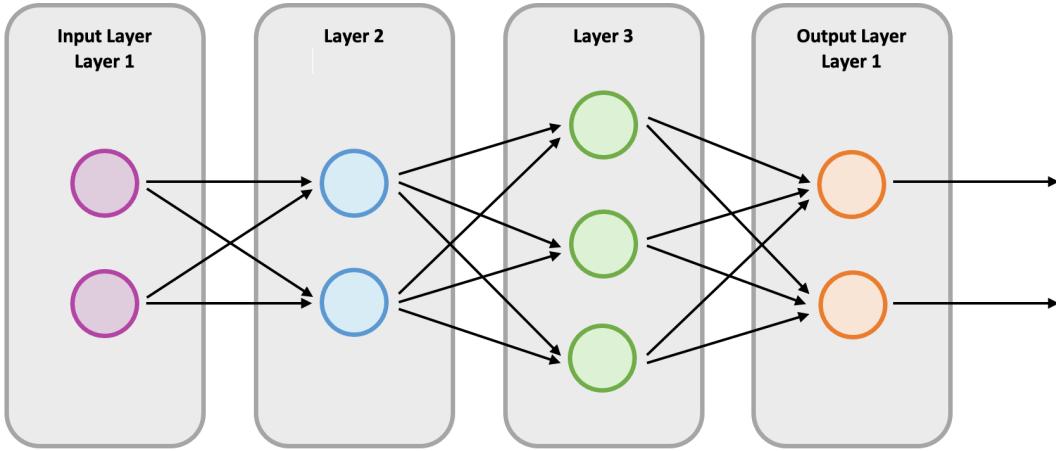
L'idée générale du perceptron est de considérer des neurones fabriqués à partir d'équations d'hyperplans et d'une fonction d'activation. Les hyperplans sont donnés par : $Wx + b = 0$
 x est un vecteur de \mathbb{R}^2 , W est une matrice de taille 2×2 qui contient les poids et b un vecteur de \mathbb{R}^2 contient les biais.

Les poids W et les biais b sont des paramètres que le réseau va se changer de trouver par optimisation.

Lors de l'entrainement d'un réseau de neurones artificiels, 4 étapes se répètent en boucle :

1. Forward propagation – on fait circuler les données de la 1^{ère} jusqu'à la dernière couche afin de produire une sortie y
2. Cost function – on calcule l'erreur entre la sortie y et la sortie de référence que l'on désire avoir, pour cela on utilise une fonction coût
3. Backward propagation – on mesure comment cette fonction coût varie par rapport à chaque couche du modèle en partant de la dernière et en remontant jusqu'à la première
4. La descente de gradient – on corrige chaque paramètre du modèle grâce à l'algorithme de la descente de gradient

Le réseau de neurone que j'ai programmé contient une couche d'entrée, une couche de sortie et deux couches dites cachées.



1. Forward propagation

L'activation du neurone se fait suivant la valeur de $\sigma(Wx + b = 0)$. Dans notre modèle, la fonction d'activation est la sigmoïde donné par $\sigma(x) = \frac{1}{1+e^{-x}}$

Les données d'entrée sont des vecteurs de x de \mathbb{R}^2 , avec comme sortie $a^{[1]} = x \in \mathbb{R}^{n_1}$

La sortie de la deuxième couche est donné par $a^{[2]} = \sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^2$

Avec $W^{[2]}$ est une matrice de taille 2×2 , $b^{[2]}$ est un vecteur de \mathbb{R}^2

La sortie $a^{[3]} = \sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) = \sigma(W^{[3]}a^{[2]} + b^{[3]}) \in \mathbb{R}^3$

Avec $W^{[3]}$ est une matrice de taille 3×2 , $b^{[3]}$ est un vecteur de \mathbb{R}^3

La sortie $a^{[4]} = \sigma(W^{[4]}\sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) + b^{[4]}) = \sigma(W^{[4]}a^{[3]} + b^{[4]}) \in \mathbb{R}^2$

Avec $W^{[4]}$ est une matrice de taille 2×3 , $b^{[4]}$ est un vecteur de \mathbb{R}^2

On considère alors la fonction $F(x) = \sigma(W^{[4]}\sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) + b^{[4]})$

2. Cost function

Comme énoncé dans la partie précédente, en machine learning, une fonction coût (loss function) permet de quantifier les erreurs effectuées par un modèle.

Pour notre modèle, la fonction de coût est donnée par :

$$Cost(W^{[2]}, W^{[3]}, W^{[4]}, b^{[2]}, b^{[3]}, b^{[4]}) = \frac{1}{10} \sum_{i=1}^{10} \frac{1}{2} \|y(x^i) - F(x^i)\|_2^2$$

3. Backward propagation

La Back-propagation consiste à déterminer la sortie du réseau qui varie en fonction des paramètres (w , b) présents dans chaque couche. Pour cela on calcule une chaîne de gradients indiquant comment la sortie varie en fonction de la dernière couche, puis comment l'avant dernière, etc..

Il faut donc déterminer les gradients, pour cela j'ai dû faire des démonstrations :

- On note w_{kj} le paramètre associé au neurone k et provenant de l'entrée j

- On note $[l]$ le numéro de la couche sur laquelle on travaille
- Les entrées de neurone du layer l sont donné par $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$
- Les sorties de neurone du layer l sont donné par $a^{[l]} = \sigma(W^{[l]}a^{[l-1]} + b^{[l]}) = \sigma(z^{[l]}) = \frac{1}{1+e^{-z}}$
- La dérivée de $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- La fonction de coût

$$C = \frac{1}{2} \|y - a^{[L]}\|_2^2 = \frac{1}{2} (y - a^{[L]})^T (y - a^{[L]}) = \frac{1}{2} (y^T y - y^T a^{[L]} - a^{[L]}^T y + a^{[L]}^T a^{[L]})$$

→ $\delta^{[L]} = \sigma'(z^{[L]}) \circ (A^{[L]} - y)$

$$\frac{\partial C}{\partial a^{[L]}} = \frac{1}{2} (-2y + 2a^{[L]}) = a^{[L]} - y \quad \frac{\partial a^{[L]}}{\partial z^{[L]}} = \sigma(z^{[L]}) (1 - \sigma(z^{[L]})) = \sigma'(z^{[L]})$$

$$\text{D'où } \delta^{[L]} = \frac{\partial C}{\partial z^{[L]}} = \frac{\partial C}{\partial a^{[L+1]}} \times \frac{\partial a^{[L+1]}}{\partial z^{[L+1]}} = (a^{[L]} - y) \sigma'(z^{[L]}) = \sigma'(z^{[L]}) \circ (a^{[L]} - y)$$

→ $\delta^{[l]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]}$

$$\delta^{[l]} = \frac{\partial C}{\partial z^{[l]}} = \frac{\partial C}{\partial a^{[l+1]}} \times \frac{\partial a^{[l+1]}}{\partial z^{[l+1]}} \times \frac{\partial z^{[l+1]}}{\partial a^{[l]}} \times \frac{\partial a^{[l]}}{\partial z^{[l]}} = (a^{[l+1]} - y) \times \sigma'(z^{[l+1]}) \times W^{[l+1]} \times \sigma'(z^{[l]})$$

$$\frac{\partial C}{\partial a^{[l+1]}} = a^{[l+1]} - y \quad \frac{\partial a^{[l+1]}}{\partial z^{[l+1]}} = \sigma'(z^{[l+1]})$$

$$\frac{\partial z^{[l+1]}}{\partial a^{[l]}} = W^{[l+1]} \quad \frac{\partial a^{[l]}}{\partial z^{[l]}} = \sigma'(z^{[l]})$$

$$(a^{[l+1]} - y) \times \sigma'(z^{[l+1]}) \times W^{[l+1]} \times \sigma'(z^{[l]}) = \sigma'(z^{[l]}) \times W^{[l+1]} \times \delta^{[l+1]}$$

D'après l'énoncé, $W^{[L]}$ est une matrice de taille (2, 3), $b^{[L]}$ est de taille (2, 1), $a^{[L-1]}$ est de taille (3, 10), donc $z^{[L]} = W^{[L]} \times a^{[L-1]} + b^{[L]}$ est de taille (2, 10)

$a^{[L]}$ et y sont de taille (2, 10) donc $\delta^{[L]} = \sigma'(z^{[L]}) \circ (a^{[L]} - y)$ est de taille (2, 10)

$W^{[L]} \times \delta^{[L]}$ est donc impossible, il faut utiliser la transposé de $W^{[L]}$

$$\text{D'où } \delta^{[l]} = \sigma'(z^{[l]}) \times (W^{[l+1]})^T \times \delta^{[l+1]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]}$$

→ $\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]}$

$$\frac{\partial C}{\partial a_j^{[l]}} = a_j^{[l]} - y \quad \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = \sigma'(z_j^{[l]}) \quad \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1$$

D'où

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{\partial C}{\partial a_j^{[l]}} \times \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \times \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = (a_j^{[l]} - y) \times \sigma'(z_j^{[l]}) \times 1 = \sigma'(z_j^{[l]}) \circ (a_j^{[l]} - y) = \delta_j^{[l]}$$

$$\Rightarrow \frac{\partial c}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]}$$

$$\frac{\partial c}{\partial a_j^{[l]}} = a_j^{[l]} - y \quad \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = \sigma'(z_j^{[l]}) \quad \frac{\partial z_j^{[l]}}{\partial w_{kj}^{[l]}} = a_k^{[l-1]}$$

$$\text{D'où } \frac{\partial c}{\partial w_{kj}^{[l]}} = \frac{\partial c}{\partial a_j^{[l]}} \times \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \times \frac{\partial z_j^{[l]}}{\partial w_{kj}^{[l]}} = (a_j^{[l]} - y) \times \sigma'(z_j^{[l]}) \times a_k^{[l-1]} = \delta_j^{[l]} a_k^{[l-1]}$$

On a donc comme fonction pour les gradients :

$$\delta_b^{[4]} = \frac{\partial c}{\partial b^4} = \sigma'(a^{[4]}) \circ (a^{[4]} - y)$$

$$\delta_w^{[4]} = \frac{\partial c}{\partial w^4} = \delta_b^{[4]} a^{[3]}$$

$$\delta_b^{[3]} = \frac{\partial c}{\partial b^3} = \sigma'(a^{[3]}) \circ (W^{[4]})^T \delta_b^{[4]}$$

$$\delta_w^{[3]} = \frac{\partial c}{\partial w^3} = \delta_b^{[3]} a^{[2]}$$

$$\delta_b^{[2]} = \frac{\partial c}{\partial b^2} = \sigma'(a^{[2]}) \circ (W^{[3]})^T \delta_b^{[3]}$$

$$\delta_w^{[2]} = \frac{\partial c}{\partial w^2} = \delta_b^{[2]} a^{[1]}$$

4. Descent gradient

La descente de gradient consiste à ajuster les paramètres w et b de façon à minimiser les erreurs du modèle, c'est-à-dire à minimiser la fonction coût. Pour ça, il faut déterminer comment est-ce que cette fonction varie en fonction des différents paramètres. C'est pourquoi on calcule le gradient (ou la dérivée) de la fonction coût. En mathématique, la dérivée d'une fonction indique comment cette fonction varie.

Grâce aux gradients, on peut mettre à jour les paramètres (w, b) de chaque couche de telle sorte à ce qu'ils minimisent l'erreur entre la sortie du modèle et la réponse attendue.

On a donc comme fonction pour les gradients :

$$b^{[4]} = b^{[4]} - \alpha \delta_b^{[4]} \quad w^{[4]} = w^{[4]} - \alpha \delta_w^{[4]}$$

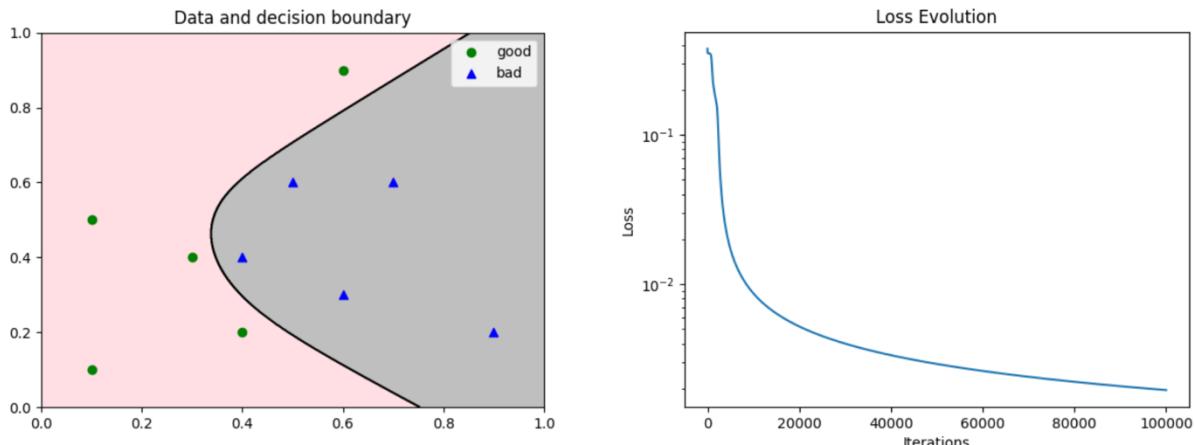
$$b^{[3]} = b^{[3]} - \alpha \delta_b^{[3]} \quad w^{[3]} = w^{[3]} - \alpha \delta_w^{[3]}$$

$$b^{[2]} = b^{[2]} - \alpha \delta_b^{[2]} \quad w^{[2]} = w^{[2]} - \alpha \delta_w^{[2]}$$

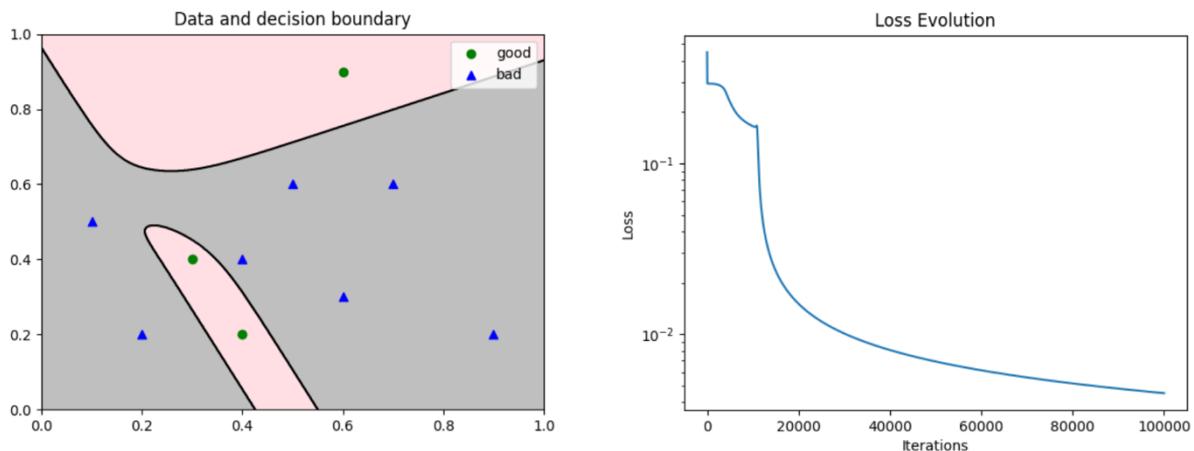
Une fois l'entraînement terminé, il faut prédire la frontière de décision qui est donnée par $F_1(x) > F_2(x)$ avec $F(x) = (F_1(x), F_2(x))$. Pour cela on calcul la fonction F avec les paramètres w et b optimiser, obtenu par l'entraînement.

Le MLP a été testé sur deux jeu de données différents :

1^{er} jeu – 10 points dont 5 points dans la classe bon et 5 points dans la classe mauvais



2^{ème} jeu – 10 points dont 3 points dans la classe bon et 7 points dans la classe mauvais



4. Planification et gestions des activités

Contrairement à un sujet pouvant être proposé par une entreprise qui doit respecter des deadlines, mon sujet était plus axé sur la découverte des méthodes d'apprentissage. En ce qui concerne la gestion du temps j'étais par conséquent assez libre. J'ai pu ainsi accorder plus de temps aux aspects qui me semblait plus complexe tels que les démonstrations mathématiques. Ceci m'a permis de vraiment prendre le temps de vraiment comprendre ces aspects.

5. Synthèse : conclusion du stage

Ce stage dont l'objectif était de découvrir les méthodes d'apprentissage en machine learning et Deep learning, m'a permis de découvrir et d'expérimenter des réseaux de neurones et des techniques de machine learning pour résoudre des problèmes complexes.

Durant ce stage, j'ai eu l'occasion de réaliser des démonstrations sur des concepts mathématiques tels que la descente de gradients et la pseudo-inverse de Moore-Penrose. Ces

démonstrations m'ont permis de mieux comprendre les fondements mathématiques des algorithmes de machine learning et d'optimiser leur implémentation.

J'ai pu découvrir de nouvelles librairies telles que PyTorch et TensorFlow, qui me seront certainement utiles dans la poursuite de mes études ou dans ma carrière professionnelle. En comparant ces deux librairies, on remarque que PyTorch est plus flexible mais TensorFlow est plus rapide. Les résultats obtenus m'ont permis de comprendre le potentiel de ces librairies mais il faudrait sûrement approfondir l'utilisation de PyTorch afin d'obtenir des résultats optimisés notamment pour des traitements de données à grande échelle.

Pour le peu de temps qu'il me reste, je vais m'intéresser au réseau de neurones convolutifs. Arnaud Bodin en parle dans son livre en prenant l'exemple de la base de données MNIST. Je vais également lire les nouveaux chapitres d'Arnaud Bodin qui parle de ChatGPT.

En conclusion, ce stage m'a fourni des bases solides et des outils pertinents pour mes futurs projets académiques et professionnels. Il m'a permis également de confirmer mon ambition de continuer mon parcours dans le domaine de l'intelligence artificielle.

6. Annexe – Références et ressources utilisées

1^{ère} partie – Sources, références

- BODIN Arnaud, RECHER François, « *Deepmath, Mathématiques (simples) des réseaux de neurones (pas trop compliqué)* ». Exo7, (2021)
<http://exo7.emath.fr/cours/livre-deepmath.pdf>
- « *La méthode d'Euler* », FEMTO – Introduction aux méthodes d'analyse numérique, (Nov. 2023)
<https://femto-physique.fr/analyse-numerique/euler.php>
- « *scipy.integrate.odeint* », Scipy documentation
<https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html>
- Manav Narula, « *Calculer la distance Euclidienne en Python* », DelftStack, (30 janvier 2023)
<https://www.delftstack.com/fr/howto/numpy/calculate-euclidean-distance/>
- « *numpy.linalg.pinv* », Numpy documentation
<https://numpy.org/doc/stable/reference/generated/numpy.linalg.pinv.html>
- « Solve an Equation Algebraically » SymPy documentation
<https://docs.sympy.org/latest/guides/solving/solve-equation-algebraically.html>
- « Introduction aux dégradés et à la différentiation automatique », TensorFlow, (19 janvier 2022)
<https://www.tensorflow.org/guide/autodiff?hl=fr>
- GLAZERadr (Adrian), « Multi-Layer-Perception-Pytorch », GitHub, (2023)
<https://github.com/GLAZERadr/Multi-Layer-Perceptron-Pytorch/blob/main/model/MultiLayerPerceptron.ipynb>
- Machine Learnia, « Formation Deep Learning », YouTube, (2021 – 2022)
https://www.youtube.com/playlist?list=PLO_fdBVlfKoanjvTJblbd9V5d9Pzp8Rw

2^{ème} partie – Codes

Apprendre à tirer au canon

La modélisation mathématique

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def plot(x, y, z, title):
    plt.figure()
    ax = plt.axes(projection='3d')
```

```

ax.view_init(30, 135)
ax.plot_surface(x, y, z, cmap='viridis')
ax.set_xlabel('Speed v')
ax.set_ylabel('Angle θ')
ax.set_zlabel('T(v,θ)')
ax.set_title(title)
plt.show()

# Exercise 2 - Function T
def t(v, a, g=9.81):
    return (v * v * np.sin(2 * a)) / g

x_val = np.linspace(0, 10, 1000)
y_val = np.linspace(0, np.pi / 2, 1000) # pi/2 = 90°C radiant
X, Y = np.meshgrid(x_val, y_val)
Z = t(X, Y)
plot(X, Y, Z, 'Graph of the function T')

# Exercise 3 - System solution by Euler
def simulation_euler(v, a, g=9.81, h=0.01):
    x, y = 0, 0
    xd1 = v * np.cos(a)
    yd1 = v * np.sin(a) - g * h
    while y >= 0:
        x += h * xd1
        y += h * yd1
        yd1 += - g * h
    return x

x_val = np.linspace(0, 10, 100)
y_val = np.linspace(0, np.pi / 2, 100)
X, Y = np.meshgrid(x_val, y_val)
Z = np.zeros_like(X)

for i, v in enumerate(x_val):
    for j, a in enumerate(y_val):
        Z[j, i] = simulation_euler(v, a)

plot(X, Y, Z, 'System solution by Euler')

# Exercise 4 - System solution by odeint
def ball(y, time, g):
    return [y[2], y[3], 0, - g] # xd1,yd1

def simulation_odeint(v, a, g=9.81):
    time = np.linspace(0, 10, 1000)
    xd1 = v * np.cos(a)
    yd1 = v * np.sin(a) - g * time[1]
    y0 = [0, 0, xd1, yd1]
    sol = odeint(ball, y0, time, args=(g,))

    x = np.array(sol[:, 0])
    y = np.array(sol[:, 1])

    ind = np.argmax(y < 0) # alt: if y < 0 => cannon-ball on the floor
    return x[ind] # horizontal: distance from the canon

```

```

x_val = np.linspace(0, 10, 100)
y_val = np.linspace(0, np.pi / 2, 100)
X, Y = np.meshgrid(x_val, y_val)
Z = np.zeros_like(X)

for i, v in enumerate(x_val):
    for j, a in enumerate(y_val):
        Z[j, i] = simulation_odeint(v, a)

plot(X, Y, Z, 'System solution by odeint')

```

Les données d'apprentissage

```

import numpy as np
import matplotlib.pyplot as plt
from sympy import *
import tensorflow as tf
from tqdm import tqdm

def plot_scatter(x, y, z, title):
    plt.figure()
    ax = plt.axes(projection='3d')
    ax.view_init(30, 135)
    ax.scatter(x, y, z, s=20)
    ax.set_xlabel('Speed v')
    ax.set_ylabel('Angle θ')
    ax.set_zlabel('T(v,θ)')
    ax.set_title(title)
    plt.show()

def plot_scatter_rg(x, y, z, xr, yr, reg, title):
    plt.figure()
    ax = plt.axes(projection='3d')
    ax.view_init(30, 135)
    ax.scatter(x, y, z)
    ax.plot_surface(xr, yr, reg, cmap='viridis')
    ax.set_xlabel('Speed v')
    ax.set_ylabel('Angle θ')
    ax.set_zlabel('T(v,θ)')
    ax.set_title(title)
    plt.show()

def plot_grad(x, y, dx, dgx, dgy, minx, miny, title):
    plt.figure()
    plt.plot(x, y, color='black')
    plt.scatter(dx, dgy, color='red')
    plt.scatter(minx, miny, color='green')
    plt.title(title)
    plt.show()

    plt.figure()
    plt.plot(dgx, dgy, color='black')
    plt.title(title)
    plt.show()

def t(v, a, g=9.81):
    return (v * v * np.sin(2 * a)) / g

# Exercise - data generation

```

```

def generate_data(nb, bruit=0.1):
    v = np.linspace(0, 10, nb)
    a = np.linspace(0, np.radians(90), nb)
    np.random.shuffle(v)
    z = t(v, a) + bruit * np.sin(a)
    plot_scatter(v, a, z, "Data")
    return np.column_stack((v, a)), z

x_data, y_data = generate_data(1000, 0.01)
print(x_data.shape)
print(y_data.shape)

# Exercise – regression linear with Moore–Penrose

def lin_reg_mp(dx, dy):
    x = np.array([[d[0], d[1], 1] for d in dx])
    mp_inv = np.linalg.pinv(x)
    c = mp_inv @ dy
    v = np.array([d[0] for d in dx])
    at = np.array([d[1] for d in dx])
    vr, atr = np.meshgrid(np.sort(v), np.sort(at))
    reg = c[0] * vr + c[1] * atr + c[2]
    plot_scatter_rg(v, at, dy, vr, atr, reg, 'Linear regression graph of
Learning data with Moore–Penrose')

lin_reg_mp(x_data, y_data)

# Exercise – regression with Moore–Penrose

def reg_mp(dx, dy):
    x = np.array([[d[0],
                  d[0]**2,
                  d[0] * np.sin(d[1]),
                  d[0] * np.sin(2 * d[1]),
                  (d[0]**2) * np.sin(d[1]),
                  (d[0]**2) * np.sin(2 * d[1])]
                  for d in dx])
    mp_inv = np.linalg.pinv(x)
    c = mp_inv @ dy

    xv = np.array([d[0] for d in dx])
    ya = np.array([d[1] for d in dx])
    v, a = np.meshgrid(np.sort(xv), np.sort(ya))
    reg = c[0]*v + c[1]*(v**2) + c[2]*v*np.sin(a) + c[3]*v*np.sin(2*a) +
    c[4]*(v**2)*np.sin(a) + c[5]*(v**2)*np.sin(2*a)
    plot_scatter_rg(xv, ya, dy, v, a, reg, 'Regression graph of Learning
data with Moore–Penrose')

reg_mp(x_data, y_data)

# Exercise – regression linear by descent gradient

def dg_regression_affine(dx, dy, epoch=5000, delta=0.001):
    a = tf.Variable([0.], dtype=tf.float32)
    b = tf.Variable([0.], dtype=tf.float32)
    c = tf.Variable([0.], dtype=tf.float32)
    xi = tf.constant(dy, dtype=tf.float32)

    def loss(a, b, c):
        return tf.reduce_mean(tf.square((a * dx[:, 0] + b * dx[:, 1] + c) -
xi)))

```

```

dgx = []
dgy = []
with tf.device('/cpu:0'):
    for i in range(epoch):
        with tf.GradientTape() as tape:
            lo = loss(a, b, c)
        grad = tape.gradient(lo, [a, b, c])
        a.assign_sub(delta * grad[0])
        b.assign_sub(delta * grad[1])
        c.assign_sub(delta * grad[2])
        dgx.append(i)
        dgy.append(lo.numpy())

v = np.array([d[0] for d in dx])
at = np.array([d[1] for d in dx])
vr, atr = np.meshgrid(np.sort(v), np.sort(at))
reg = a * vr + b * atr + c
plot_scatter_rg(v, at, dy, vr, atr, reg, 'Linear regression graph of
Learning data by GD')

plt.figure()
plt.plot(dgx, dgy, color='black')
plt.yscale('log')
plt.title("GD for linear regression with Tensorflow : Niter=5000, δ =
0.001")
plt.show()

dg_regression_affine(x_data, y_data)

# Exercise – regression by descent gradient
def dg_regression(dx, dy, epoch=50000, delta=0.000005):
    a1 = tf.Variable([0.], dtype=tf.float32)
    a2 = tf.Variable([0.], dtype=tf.float32)
    a3 = tf.Variable([0.], dtype=tf.float32)
    a4 = tf.Variable([0.], dtype=tf.float32)
    a5 = tf.Variable([0.], dtype=tf.float32)
    a6 = tf.Variable([0.], dtype=tf.float32)

    xi = tf.constant(dy, dtype=tf.float32)

    def loss(a1, a2, a3, a4, a5, a6):
        v = dx[:, 0]
        a = dx[:, 1]
        r = a1*v + a2*(v**2) + a3*v*np.sin(a) + a4*v*np.sin(2*a) + a5 *
        (v**2) * np.sin(a) + a6 * (v**2)*np.sin(2*a)
        return tf.reduce_mean(tf.square(r - xi))

    # adaptive learning rate
    optimizer = tf.keras.optimizers.Adam(learning_rate=delta)
    dgx = []
    dgy = []
    with tf.device('/cpu:0'):
        for i in tqdm(range(epoch)):
            with tf.GradientTape() as tape:
                lo = loss(a1, a2, a3, a4, a5, a6)
            grad = tape.gradient(lo, [a1, a2, a3, a4, a5, a6])
            optimizer.apply_gradients(zip(grad, [a1, a2, a3, a4, a5, a6]))
            dgx.append(i)

```

```

        dgy.append(lo.numpy())
    print(a1.numpy(), a2.numpy(), a3.numpy(), a4.numpy(), a5.numpy(),
a6.numpy())

    xv = np.array([d[0] for d in dx])
    ya = np.array([d[1] for d in dx])
    v, a = np.meshgrid(np.sort(xv), np.sort(ya))
    reg = a1*v + a2*(v**2) + a3*v*np.sin(a) + a4*v*np.sin(2*a) + a5*
(v**2)*np.sin(a) + a6*(v**2)*np.sin(2*a)
    plot_scatter_rg(xv, ya, dy, v, a, reg, 'Regression graph of Learning
data with DG')

    plt.figure()
    plt.plot(dgx, dgy, color='black')
    plt.yscale('log')
    plt.title("GD for regression with Tensorflow : Niter=50000, initial δ =
0.000005")
    plt.show()

dg_regression(x_data, y_data)

```

MLP – TensorFlow

```

import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

def t(v, a, g=9.81):
    return (v * v * np.sin(2 * a)) / g

def generate_data(nb, bruit=0.1):
    v = np.linspace(0, 10, nb)
    a = np.linspace(0, np.radians(90), nb)
    np.random.shuffle(v)
    z = t(v, a) + bruit * np.sin(a)
    return v, a, np.column_stack((v, a)), z

v, a, x_train, y_train = generate_data(1000, 0.01)
v2, a2, x_train2, y_train2 = generate_data(1000, 0.1)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(256, input_shape=(2,), activation='sigmoid'))
model.add(tf.keras.layers.Dense(256, activation='sigmoid'))
model.add(tf.keras.layers.Dense(1, activation='linear'))
model.compile(loss='mse',
optimizer=tf.keras.optimizers.Adam(learning_rate=0.01))
print(model.summary())
with tf.device('/GPU:0'):
    print("-----Train data-----")
    train = model.fit(x_train, y_train, epochs=1000, batch_size=100)
    print("\n-----Train 2 data-----")
    test = model.fit(x_train2, y_train2, epochs=1000, batch_size=100)

plt.plot(train.history['loss'], color='orange')
plt.plot(test.history['loss'], color='blue')
plt.legend(['train (0.01)', 'train (0.1)'])
plt.yscale('log')
plt.title("MLP Tensorflow Loss")
plt.show()

```

```

y_pred = model.predict(x_train)
plt.figure()
ax = plt.axes(projection='3d')
ax.view_init(30, 135)
ax.scatter(v, a, y_train, color='blue', alpha=0.5)
ax.scatter(v, a, y_pred, color='red')
ax.legend(['y_train', 'y_pred'])
ax.set_xlabel('Speed v')
ax.set_ylabel('Angle θ')
ax.set_zlabel('T(v,θ)')
ax.set_title("MLP Tensorflow")
plt.show()

V, A = np.meshgrid(np.sort(v), a)
z_grid = t(V, A) + 0.01 * np.sin(A)
X = np.column_stack((V.flatten(), A.flatten()))

pred = model.predict(X)
y_pred = pred.reshape(V.shape)

plt.figure()
ax = plt.axes(projection='3d')
ax.view_init(30, 135)
ax.plot_surface(V, A, z_grid, cmap='viridis')
ax.plot_wireframe(V, A, y_pred, color='black', linewidth=0.5, alpha=0.7)
ax.set_xlabel('Speed v')
ax.set_ylabel('Angle θ')
ax.set_zlabel('T(v,θ)')
ax.set_title("MLP Tensorflow")
ax.legend(['y_train', 'y_pred'])
plt.show()

```

MLP – PyTorch

```

import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from tqdm import tqdm

def t(v, a, g=9.81):
    return (v * v * np.sin(2 * a)) / g

def generate_data(nb, bruit=0.1):
    v = np.linspace(0, 10, nb)
    a = np.linspace(0, np.radians(90), nb)
    np.random.shuffle(v)
    z = t(v, a) + bruit * np.sin(a)
    return v, a, np.column_stack((v, a)), z.reshape(-1, 1)

v, a, x_data_train, y_data_train = generate_data(1000, 0.01)
x_train = torch.tensor(x_data_train, dtype=torch.float32)
y_train = torch.tensor(y_data_train, dtype=torch.float32)

v2, a2, x_data_train2, y_data_train2 = generate_data(1000, 0.1)
x_train2 = torch.tensor(x_data_train2, dtype=torch.float32)
y_train2 = torch.tensor(y_data_train2, dtype=torch.float32)

device = torch.device('cpu')
class NeuralNetwork(nn.Module):

```

```

def __init__(self):
    super().__init__()
    self.c1 = nn.Linear(2, 256)
    self.c2 = nn.Linear(256, 256)
    self.c3 = nn.Linear(256, 1)
    self.activation = nn.Sigmoid()

def forward(self, x):
    x = self.activation(self.c1(x))
    x = self.activation(self.c2(x))
    x = self.c3(x)
    return x

model = NeuralNetwork().to(device)
print(model)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

def train(x_data, y_data, model, criterion, optimizer):
    model.train()
    for batch in range(100):
        pred = model(x_data.to(device))
        loss = criterion(pred, y_data.to(device))

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

def test(x_data, y_data, model, criterion):
    model.eval()
    with torch.no_grad():
        pred = model(x_data.to(device))
        loss = criterion(pred, y_data.to(device)).item()
    return loss

loss = []
loss2 = []

for epochs in tqdm(range(1000)):
    current_loss = 0.0
    train(x_train, y_train, model, criterion, optimizer)
    loss.append(test(x_train, y_train, model, criterion))
    train(x_train2, y_train2, model, criterion, optimizer)
    loss2.append(test(x_train2, y_train2, model, criterion))
print("Training Done!")

x = np.arange(1000)
plt.figure()
plt.plot(x, loss, color='orange')
plt.plot(x, loss2, color='blue')
plt.legend(['train (0.01)', 'train (0.1)'])
plt.yscale('log')
plt.title("MLP PyTorch Loss")
plt.show()

model.eval()
with torch.no_grad():
    y_pred = model(x_train.to(device))

```

```

plt.figure()
ax = plt.axes(projection='3d')
ax.view_init(30, 135)
ax.scatter(v, a, y_train, color='blue', alpha=0.5)
ax.scatter(v, a, y_pred, color='red')
ax.legend(['y_train', 'y_pred'])
ax.set_xlabel('Speed v')
ax.set_ylabel('Angle θ')
ax.set_zlabel('T(v,θ)')
ax.set_title("MLP PyTorch")
plt.show()

V, A = np.meshgrid(np.sort(v), a)
z_grid = t(V, A) + 0.01 * np.sin(A)
Xx = np.column_stack((V.flatten(), A.flatten()))
X = torch.tensor(Xx, dtype=torch.float32)

model.eval()
with torch.no_grad():
    pred = model(X.to(device))
y_pred = pred.reshape(V.shape)

plt.figure()
ax = plt.axes(projection='3d')
ax.view_init(30, 135)
ax.plot_surface(V, A, z_grid, cmap='viridis')
ax.plot_wireframe(V, A, y_pred, color='black', linewidth=0.5, alpha=0.7)
ax.set_xlabel('Speed v')
ax.set_ylabel('Angle θ')
ax.set_zlabel('T(v,θ)')
ax.set_title("MLP PyTorch")
ax.legend(['y_train', 'y_pred'])
plt.show()

```

Perceptron multicouche avec Numpy

La modélisation

```

import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

# data 1
good = np.array([[0.1, 0.1], [0.1, 0.5], [0.3, 0.4], [0.4, 0.2], [0.6,
0.9]])
bad = np.array([[0.4, 0.4], [0.5, 0.6], [0.6, 0.3], [0.7, 0.6], [0.9,
0.2]])

# data 2
#good = np.array([[0.3, 0.4], [0.4, 0.2], [0.6, 0.9]])
#bad = np.array([[0.1, 0.5], [0.2, 0.2], [0.4, 0.4], [0.5, 0.6], [0.6,
0.3], [0.7, 0.6], [0.9, 0.2]])

plt.figure()
plt.scatter(good[:, 0], good[:, 1], c='green', marker='o')
plt.scatter(bad[:, 0], bad[:, 1], c='blue', marker='^')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.title("data")
plt.show()

```

```

data = np.vstack((good, bad))
# data 1
y = np.array([[1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [0, 1], [0, 1], [0,
1],[0, 1], [0, 1]])
# data 2
#y = np.array([[1, 0], [1, 0], [1, 0], [0, 1], [0, 1], [0, 1], [0, 1], [0,
1],[0, 1], [0, 1]])

def initialisation():
    w2 = np.random.randn(2, 2)
    b2 = np.random.randn(2, 1)
    w3 = np.random.randn(3, 2)
    b3 = np.random.randn(3, 1)
    w4 = np.random.randn(2, 3)
    b4 = np.random.randn(2, 1)
    return w2, b2, w3, b3, w4, b4

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def forward(x, w2, b2, w3, b3, w4, b4):
    a2 = sigmoid(w2 @ x + b2)
    a3 = sigmoid(w3 @ a2 + b3)
    a4 = sigmoid(w4 @ a3 + b4)
    return a2, a3, a4

def loss(w2, b2, w3, b3, w4, b4, x, y):
    f = sigmoid(w4 @ sigmoid(w3 @ sigmoid(w2 @ x + b2) + b3) + b4)
    return 1/10 * np.sum(1/2 * np.linalg.norm(y - f, ord=2))

def d(a):
    return a * (1 - a)

def backward(x, y, a2, a3, a4, w3, w4):
    db4 = d(a4) * (a4 - y)
    dw4 = db4 @ a3.T
    db3 = d(a3) * w4.T @ db4
    dw3 = db3 @ a2.T
    db2 = d(a2) * w3.T @ db3
    dw2 = db2 @ x.T
    return dw2, dw3, dw4, db2, db3, db4

def update(dw2, dw3, dw4, db2, db3, db4, w2, b2, w3, b3, w4, b4, lr=0.1):
    w4 = w4 - lr * dw4
    b4 = b4 - lr * db4
    w3 = w3 - lr * dw3
    b3 = b3 - lr * db3
    w2 = w2 - lr * dw2
    b2 = b2 - lr * db2
    return w2, b2, w3, b3, w4, b4

def predict(x, w2, b2, w3, b3, w4, b4):
    _, _, a4 = forward(x, w2, b2, w3, b3, w4, b4)
    return a4

def train(x, y, lr=0.1, epochs=100000):
    w2, b2, w3, b3, w4, b4 = initialisation()
    a2 = np.random.randn(2, 1)

```

```

a3 = np.random.randn(3, 1)
a4 = np.random.randn(2, 1)
train_loss = []
for _ in tqdm(range(epochs)):
    for m in range(0, 10):
        xx = np.array([[x[m][0]], [x[m][1]]])
        yy = np.array([[y[m][0]], [y[m][1]]])
        a2, a3, a4 = forward(xx, w2, b2, w3, b3, w4, b4)
        dw2, dw3, dw4, db2, db3, db4 = backward(xx, yy, a2, a3, a4, w3,
w4)
        w2, b2, w3, b3, w4, b4 = update(dw2, dw3, dw4, db2, db3, db4,
w2, b2, w3, b3, w4, b4, lr)
        epoch_loss = 0
        for m in range(0, 10):
            xx = np.array([[x[m][0]], [x[m][1]]])
            yy = np.array([[y[m][0]], [y[m][1]]])
            epoch_loss += loss(w2, b2, w3, b3, w4, b4, xx, yy)
    train_loss.append(epoch_loss)

plt.figure()
plt.plot(train_loss, label='training loss')
plt.yscale('log')
plt.title("Loss Evolution")
plt.ylabel('Loss')
plt.xlabel('Iterations')
plt.show()
return w2, b2, w3, b3, w4, b4, a2, a3, a4

w2, b2, w3, b3, w4, b4, a2, a3, a4 = train(data, y)

x = np.linspace(0, 1, 1000)
y = np.linspace(0, 1, 1000)
X, Y = np.meshgrid(x, y)
Z = np.zeros(X.shape)
for i in tqdm(range(X.shape[0])):
    for j in range(X.shape[1]):
        p = np.array([[X[i, j]], [Y[i, j]]])
        f = predict(p, w2, b2, w3, b3, w4, b4)
        Z[i, j] = 1 if f[0][0] > f[1][0] else 0

plt.figure()
plt.contourf(X, Y, Z, levels=[-0.5, 0.5, 1.5], colors=['grey', 'pink'],
alpha=0.5)
plt.contour(X, Y, Z, levels=[0.5], colors='black')
plt.scatter(good[:, 0], good[:, 1], c='green', marker='o')
plt.scatter(bad[:, 0], bad[:, 1], c='blue', marker='^')
plt.legend(['good', 'bad'])
plt.title('Data and decision boundary')
plt.show()

```