INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Projet de Physique P6

STPI/P6/2021 – 22

INTRODUCTION TO MACHINE LEARNING WITH APPLICATION TO AIRFOIL SELF-NOISE PREDICTION

Etudiants :

Tenga CORTAL **Adrien GAULIN**

Hadrien HAVE **Rose PACINI**

Yahui XIA **Tristan LOUKIANENKO**

Enseignant-responsable du projet :

Hassan TOFAILI

Date de remise du rapport : ***12/06/2021***

Référence du projet : ***STPI/P6/2021 – 22***

Intitulé du projet: **Introduction *to machine learning with application to airfoil self-noise prediction***

Type de projet : ***Bibliographie et modélisation***

Objectifs du projet :

Les objectifs de ce projet sont de réussir à appréhender des concepts nouveaux tels que le machine learning, de développer des algorithmes de prédiction en implémentant un réseau de neurones fonctionnel. Enfin, notre but final est de réussir à prédire à l'aide de différentes caractéristiques, le bruit produit par une pale d'avion en deux ou trois dimensions.

Mots-clefs du projet:

Machine Learning
Réseau de neurones
Algorithmes
Prédiction

Si existant, n° cahier de laboratoire associé : /

TABLE DES MATIERES

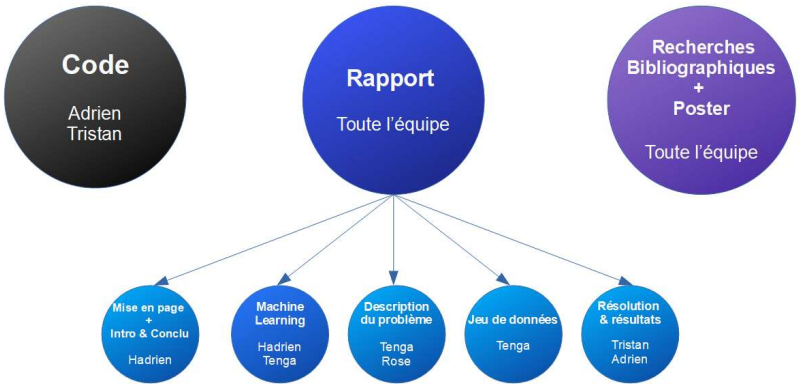
1. INTRODUCTION

De nos jours, l'informatique a pris une place importante dans le monde et surtout dans les sciences. Des branches, telles que le machine learning, se sont développées et permettent aujourd'hui de prédire certains comportements physiques très précisément.
Nous proposons de prédire, grâce au machine learning, le bruit propre d'un profil aérodynamique : l'aile d'avion.

2. METHODOLOGIE / ORGANISATION DU TRAVAIL

Dans le cadre de notre projet, nous sommes passés par différentes étapes très spécifiques pour le mener à bien.
Dans un premier temps, notre encadrant nous a fourni différents documents et manuels d'instruction pour pouvoir installer l'environnement JULIA, un langage récent pouvant être utilisé pour implémenter des algorithmes de machine learning.
Dans un second temps, nous avons travaillé sur le fonctionnement théorique d'un réseau de neurones en se familiarisant avec différents concepts mathématiques. À la suite de ce travail, un rapport médian nous a été demandé afin de regrouper ce que nous avions appris sur les réseaux de neurones.
Dans un troisième temps, nous avons commencé à implémenter un réseau de neurones en JULIA dans le but de résoudre un problème de reconnaissance de chiffres écrits à la main.
Enfin, nous avons implémenté un second réseau de neurones permettant de résoudre la problématique principale de notre projet : prédire le bruit propre d'un profil aérodynamique à l'aide d'un jeu de données contenant différentes caractéristiques telles que la fréquence ou l'angle d'attaque.
Nous pouvions voir en présentiel notre enseignant une semaine sur deux, nous profitions de ces moments pour lui poser les questions relatives à notre projet, tandis que nous profitions des sessions Zoom pour avancer un maximum sur ce dernier.
En dehors des horaires réservés à notre projet, nous communiquions grâce à un groupe créé sur les réseaux sociaux.

Organigramme des tâches réalisées et des étudiants concernés

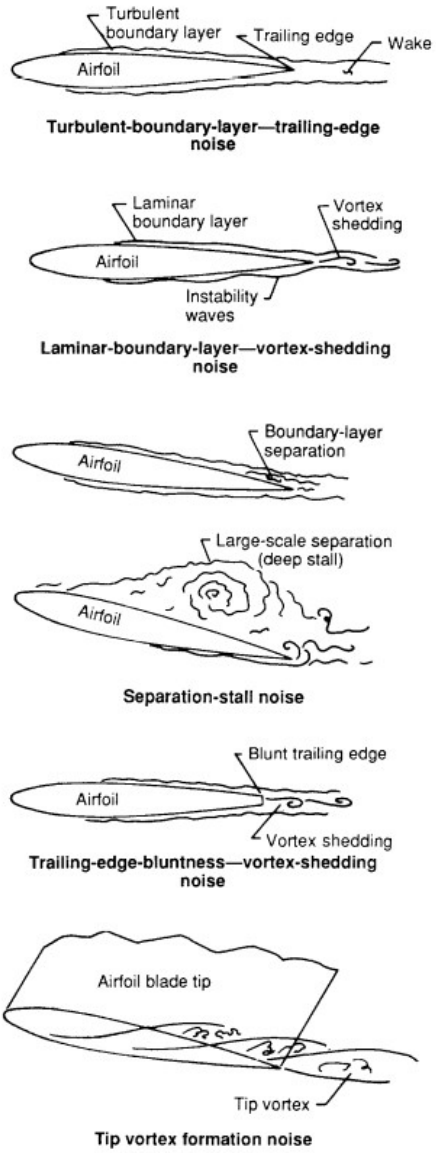


3. TRAVAIL REALISE ET RESULTATS

3.1. Description du problème

3.1.1. Le problème : prédiction du bruit propre d'un profil aérodynamique

Le bruit propre d'un profil aérodynamique est dû à l'interaction entre une pale et la turbulence produite dans sa propre couche limite et près du sillage. C'est le bruit total produit lorsqu'un profil aérodynamique rencontre une entrée lisse et non turbulente.
Sur la figure ci-contre, les conditions d'écoulement subsonique pour cinq mécanismes de bruits propres sont illustrés. À un nombre de Reynolds Rc élevé (selon la longueur de la membrure), les couches limites turbulentes (Turbulent Boundary Layer) se développent sur la majeure partie du profil aérodynamique. Le bruit se produit lorsque cette turbulence passe sur le bord de fuite (Trailling Edge).
À Rc bas, des couches en grande partie laminaires (Laminar Boundary Layer) se développent. Leur instabilité implique un décollement de tourbillon (vortex shedding) et le bruit associé du bord de fuite.
Ensuite, pour des angles d'attaque non nuls, le flux peut se séparer près du bord de fuite du côté de l'aspiration du profil aérodynamique pour produire du bruit en raison de la vorticité turbulente de la charpente. À des angles d'attaque très élevés, l'écoulement séparé près du bord de fuite cède la place à une séparation à grande échelle (deep stall), ce qui fait que le profil aérodynamique émet un bruit à basse fréquence semblable à celui d'un corps non profilé en écoulement.
Une autre source de bruit est le décollement de tourbillon qui se produit dans la petite zone d'écoulement séparée à l'arrière du bord de fuite émoussé. La source restante considérée ici est due à la formation du vortex d'extrémité, qui contient un flux très turbulent et se produit près des extrémités des pales de portance ou des ailes.



Conditions d'écoulement produisant un bruit propre des pales à profil aérodynamique
Source: https://ntrs.nasa.gov/

La question du bruit propre du profil aérodynamique est essentielle aujourd'hui car il peut être nocif pour l'audition et empêche l'utilisation de certaines infrastructures comme les éoliennes ou les rotors par exemple en milieu urbain. Le bruit peut également être dérangeant pour les embarcations sous-marines et de surface nécessitant une certaine furtivité.

On distingue plusieurs types de bruits créés par ce type d'installations. Tout d'abord il y a le bruit tonal qui se caractérise par la répétition de plusieurs sons purs et nettement distincts, celui-ci est assez facile à identifier. Il existe ensuite le bruit à large bande, étant difficile à supprimer car sa bande passante est très large, contrairement au bruit de basse fréquence qui recouvre plutôt des basses fréquences. Enfin, le dernier type de bruit est le bruit impulsif, étant difficile à prédire car il contient des pics impulsifs aléatoires dans le temps.
Plusieurs méthodes ont été mises au point pour prédire le bruit généré, comme le modèle de Brooks. Celui-ci consistait à faire des tests sur un modèle de jet libre auquel on appliquait des légères turbulences. On faisait varier la longueur de la corde, l'angle de la surface d'entraînement, et la forme de la pointe de l'avion. On plaçait ensuite des microphones ou des ordinateurs en fonction du modèle pour enregistrer le bruit. D'autres méthodes ont été utilisées pour prédire le bruit, mais les prédictions ne sont pas efficaces pour des fréquences hautes, alors que les composants de bruits à haute fréquence doivent être correctement pris en compte, car ils causent une gêne à l'oreille humaine et sont fortement réglementés.

3.1.2. Le jeu de données

Notre dataset vient du répertoire open source d'UCI au lien suivant: https://archive.ics.uci.edu/ml/datasets/airfoil+self-noise.
Ces données ont été obtenues à partir d'une série de tests aérodynamiques et acoustiques de sections de pales aérodynamiques bidimensionnelles et tridimensionnelles effectuées dans une soufflerie anéchoïque.
Ce dataset comprend des profils aérodynamiques de différentes tailles à différentes vitesses en soufflerie et à différents angles d'attaque. La portée du profil aérodynamique et la position de l'observateur étaient les mêmes dans toutes les expériences.
Le fichier de données airfoil_self_noise.csv contient 6 variables et 1503 instances.

Nous avons 5 entrées / prédicteurs:

1. Fréquence, en Hertzs.
2. Angle d'attaque, en degrés.
3. Longueur de la membrure, en mètres.
4. Vitesse de l'écoulement libre, en mètres par seconde.
5. Épaisseur de déplacement côté aspiration, en mètres.

Et 1 sortie / cible:

6. Niveau de pression acoustique, en décibels.

Notre objectif était donc de prédire le niveau de pression acoustique en décibels en fonction des autres paramètres.

3.2. Machine Learning

Le premier objectif de notre projet consistait à appréhender et à s'approprier des concepts nouveaux tels que le fonctionnement des « réseaux de neurones » et le « machine learning ».

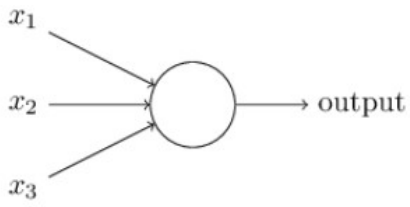
Un réseau de neurones est un paradigme de programmation qui s'inspire de concepts biologiques dont le but est de permettre à un ordinateur d'apprendre à partir d'un jeu de données. Les réseaux de neurones et le deep-learning sont actuellement les méthodes permettant d'obtenir les résultats les plus performants pour le traitement d'images.

Dans cette première partie, nous détaillerons les concepts mathématiques clés pour comprendre le fonctionnement d'un réseau de neurones et comment l'implémenter. Dans un premier temps, nous introduirons les notions de perceptrons et sigmoid neurons puis nous présenterons l'architecture d'un réseau de neurones et illustrerons ce concept avec un exemple concret d'application permettant de reconnaître et classer des chiffres écrits à la main.

3.2.1. Comment fonctionne un perceptron ?

Ce qui est appelé un neurone ou un perceptron est une entité qui contient une certaine valeur calculée à partir de plusieurs entrées.

Un perceptron prend plusieurs valeurs en entrée ou « inputs », $x_1, x_2, \ldots$, puis produit une unique valeur de sortie :



L'ensemble des valeurs d'entrée et de sortie d'un perceptron est binaire, les valeurs sont soit 0, soit 1.

Pour calculer la valeur de sortie, on introduit des poids ou « weights » : $w_1, w_2, \ldots$ exprimant l'importance respective des valeurs d'entrée.

On peut résumer ce concept avec l'expression suivante :

$$output = \begin{cases} 1 \; if \;\; w.x \; + \; b \leq 0 \\ 0 \; if \;\; w.x \; + \; b \geq 0 \end{cases}$$

Où $w = (w_1, w_2, \ldots, w_n)$ et $x = (x_1, x_2, \ldots, x_n)$ et dont les composantes respectives sont les poids et les entrées. Et b est le biais du perceptron qui est une mesure évaluant à quel point il est probable que le perceptron se déclenche. Plus la valeur du biais sera élevée, plus le perceptron sera susceptible de se déclencher.

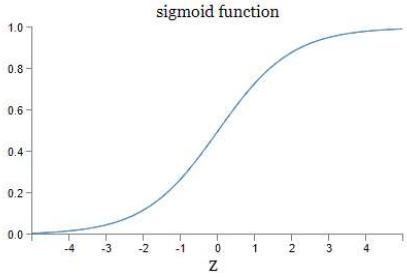
3.2.2. Le fonctionnement des Sigmoid neurons

Ensuite nous allons décrire les sigmoid neurons de la même façon que les perceptrons : tout comme un perceptron, un sigmoid neuron a des valeurs d'entrée $x_1, x_2, \ldots$ Mais au lieu de ne prendre que les valeurs 0 et 1, ces valeurs d'entrée peuvent également prendre n'importe quelle valeur entre 0 et 1. Par exemple, 0,638 est une valeur d'entrée valide pour un sigmoid neuron. Cependant, la valeur de sortie n'est ni 0 ni 1 mais $\sigma(w.x + b)$ où σ est appelé la « sigmoid function » :

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

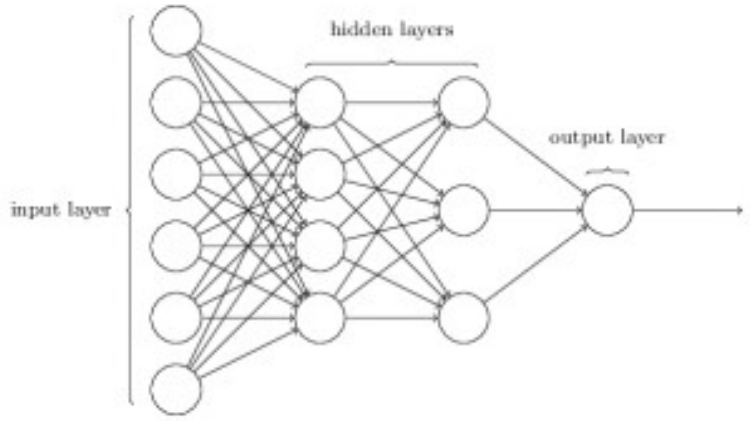
Où $z = w.x + b$ est positif.

Voici l'allure de la courbe de la fonction Sigmoïde:



3.2.3. L'architecture d'un réseau de neurones

Les réseaux de neurones sont formés de couches de neurones. Ces neurones sont les unités centrales de traitement du réseau. Premièrement, nous avons une couche d'entrée ou « input layer ». Les neurones de cette couche représentent les données d'entrée. Il peut s'agir de simple scalaire, de vecteurs multidimensionnels ou de matrices. La couche de sortie ou « output layer » prédit la valeur finale de sortie. Entre ces deux couches, des couches intermédiaires appelées « hidden layers » ou « couches cachées » sont disposées. Ce sont elles qui réalisent la plupart des calculs requis par le réseau de neurones.



Les neurones d'une même couche sont connectés aux neurones de la couche suivante à travers des canaux. Une valeur numérique, le poids (w), est attribué à chacun de ces canaux. Les valeurs d'entrées sont multipliées par les poids correspondants et leur somme est envoyée comme valeur d'entrée pour les neurones de la couche suivante.

Chacun de ces neurones est associé à une valeur numérique appelée biais (b) qui est ensuite ajoutée à la somme reçue en valeur d'entrée, comme pour les perceptrons.

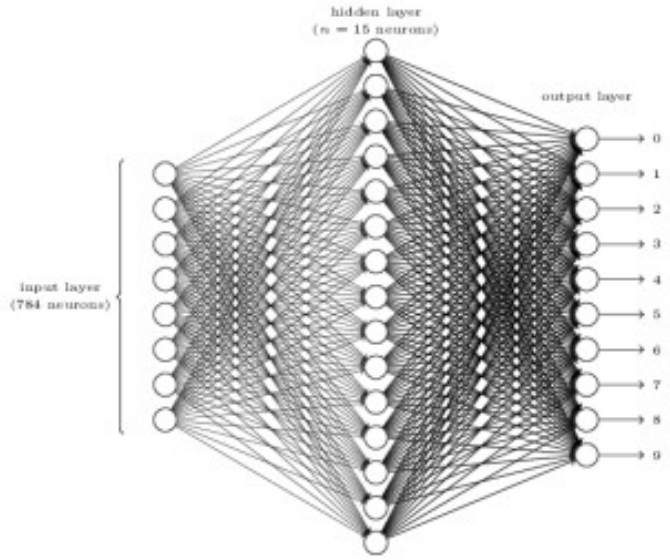
La valeur d'entrée d'un neurone est ensuite transformée par une fonction seuil appelée « activation function », elle retourne une valeur d'activation « a » pour chaque neurone.

Un neurone transmet ses données aux neurones de la couche suivante à travers les canaux. De cette manière, les données sont propagées à travers le réseau. Ce phénomène est appelé « forward propagation ».

Dans la couche de sortie, le neurone avec la valeur la plus haute se déclenche et détermine la valeur de sortie. Les valeurs dans la couche de sortie sont en quelques sortes des probabilités.

3.2.4. Un réseau de neurones simple pour classer des chiffres écrits à la main.

La première tâche pratique de notre projet consistait à implémenter un réseau de neurones permettant de reconnaître et classer des chiffres écrits à la main. Dans la partie suivante, nous détaillerons la construction et le fonctionnement de notre réseau.



Nos données d'apprentissage ou « training data » pour le réseau sont constitués d'un grand nombre d'images de chiffres (entre 0 et 9) écrits à la main puis scannées. Ces images provenant du MNIST data set images possèdent toute la même dimension : 28x28 pixels. Nous avons donc choisi de construire une couche d'entrée de 28x28 =784 neurones.

Les valeurs d'entrées sont des pixels qui prennent comme valeur différentes nuances de gris. 0.0 désignant un pixel blanc et 1.0 un pixel noir. Entre ces deux extremums, les valeurs correspondent à différentes teintes de gris. La couche intermédiaire (hidden layer) contient $n = 15$ neurones. Ce choix est arbitraire et différentes valeurs ont été testées. La couche de sortie du réseau contient 10 neurones et sera donc de la forme $\mathrm{y} = \mathrm{y(x)}$. Si le premier neurone se déclenche, c'est-à-dire si sa valeur de sortie est très proche de 1 ($y(x) = (1,0,0,0,0,0,0,0,0,0)$), alors cela signifiera que le réseau pense que le chiffre est un 0. Si le second neurone se déclenche alors cela indiquera que le réseau pense que le chiffre est un 1. Et ainsi de suite…

Le réseau de neurones implémenté sous Julia est fourni en annexe.

3.2.5. L'apprentissage avec l'algorithme du gradient descent

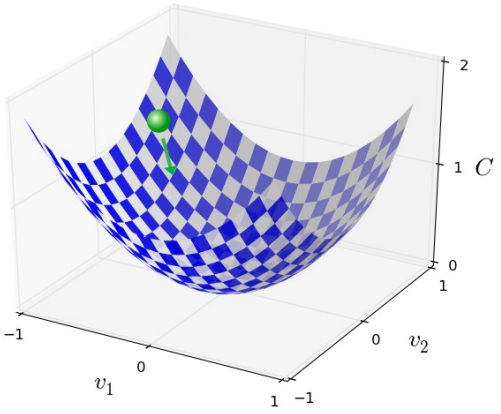
Pour que notre réseau de neurones soit opérationnel, nous avons besoin d'un algorithme qui nous permet de déterminer les poids et les biais afin que notre valeur de sortie puisse nous donner $y(x)$. Pour toutes les données d'apprentissage en entrée, nous définissons une fonction de coût ou « cost function » :

$$C(w,b) = \frac{1}{2n}\sum_{x}^{n}\|y(x) - a(x)\|^2$$

w représente les différents poids du réseau, b les biais, n le nombre total de valeurs d'entrée, $a(x)$ est le vecteur de sortie du réseau lorsque x est une valeur d'entrée et on somme sur toutes les valeurs d'entrée x. $y(x)$ est alors la sortie théorique attendue.

Le but de notre algorithme d'entraînement est de minimiser le coût $C(w,b)$. En effet, on souhaite trouver un ensemble de poids et de biais qui nous permettent d'obtenir un coût le plus faible possible.

En résumé, le fonctionnement du gradient descent algorithm est de calculer de manière répétée le gradient ∇C, et ensuite se déplacer dans la direction qui minimise le gradient. On peut le visualiser de la manière suivante.



Voyons à présent comment appliquer cet algorithme dans un réseau de neurones :

L'idée est d'utiliser le gradient descent algorithm pour trouver les poids w et les biais b qui minimisent la fonction de coût précédente. Précisément, quand $y(x)$ est approximativement égal à la valeur de sortie $a(x)$, pour toutes les valeurs d'entrée. Donc notre algorithme a réalisé un travail efficace s'il peut trouver les poids et les biais tels que $C(w,b) \sim 0$. D'une perspective opposée, l'algorithme n'est pas efficace si $C(w,b)$ est très grand.

Expliquons alors ceci du point de vue mathématique : nous nous déplaçons sur la courbe $C(w,b)$, le prochain point $(w,b)_{i+1}$ que nous voulons atteindre se déduira du point actuel $(w,b)_i$ où nous nous situons, auquel nous soustrayions le gradient du point $(w,b)_i$ que nous multiplions par un certain pas η (à noter qu'un pas très petit rendra l'algorithme plus précis mais plus lent, et inversement pour un pas plus grand). Si nous définissons le point $(w,b)_i$ par le vecteur $\overrightarrow{X_i}$ alors l'équation de la descente de gradient est la suivante :

$$\overrightarrow{X_{i+1}} = \overrightarrow{X_i} - \eta * \nabla_i C(\overrightarrow{X_i})$$

3.2.6. Le fonctionnement de la backpropagation

La backpropagation est une méthode utilisée pour calculer le gradient à un point donné (w,b), ce qui permettra de rendre l'algorithme de gradient descent plus efficace. Détaillons maintenant le fonctionnement de la backpropagation :

Premièrement, les hypothèses que nous devons vérifier à propos de notre fonction C pour appliquer la backpropagation :

- La première hypothèse est que nous devons écrire la fonction de coût comme une moyenne $C = \frac{1}{n}\sum_x C_x$ sur les fonctions de coûts C_x pour chaque valeur d'apprentissage x.

  La backpropagation nous permet de calculer les dérivées partielles $\frac{\partial C_x}{\partial w}$ and $\frac{\partial C_x}{\partial b}$pour chaque valeur d'apprentissage. On retrouve ensuite $\frac{\partial C}{\partial w}$ et $\frac{\partial C}{\partial b}$ en faisant la moyenne des valeurs d'apprentissage.

- La deuxième hypothèse que nous faisons sur le coût est de le réécrire en fonction des valeurs de sortie du réseau de neurones : $C = C(a^l)$, avec a^l contenant les valeurs de sortie de chacun des neurones de la couche l.

Ensuite, nous allons expliquer les objectifs de la backpropagation : elle consiste à comprendre comment la modification des poids et des biais dans un réseau modifie la fonction de coût. En fin de compte, cela signifie calculer les dérivées partielles $\frac{\partial C}{\partial w_j^l}$ et $\frac{\partial C}{\partial b_j^l}$.

On pourra ensuite définir une fonction pour définir l'erreur δ_j^l du neurone j de la couche l .

Puis nous établirons une relation entre δ_j^l et les dérivées partielles $\frac{\partial C}{\partial w_j^l}et\frac{\partial C}{\partial b_j^l}$.

On définit δ_j^l par : $\delta_j^l = \frac{\partial C}{\partial z_j^l}$.

La backpropagation nous donnera un moyen de calculer δ_j^l pour chaque couche, puis de mettre en relation ces erreurs aux quantités d'intérêt réel, $\frac{\partial C}{\partial w_j^l}$ et $\frac{\partial C}{\partial b_j^l}$.

Enfin, la backpropagation est fondée sur quatre équations fondamentales qui, ensemble, nous permettent de calculer à la fois l'erreur δ_j^l et le gradient de la fonction de coût :

- Une équation pour l'erreur dans la couche de sortie, δ^l , dont les composantes sont données par :

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}\sigma'(z_j^l)$$

$$\delta^L = \nabla_a \mathrm{C} \odot \sigma'(l) \qquad (\boldsymbol{BP1})$$

$\odot$ représente le produit Hadamard, les composantes du vecteur $s \odot t$, où s et t sont deux vecteurs de même taille, sont $(s \odot t)_i = s_i * t_i$.

- Une équation pour l'erreur δ^len termes d'erreur pour la couche suivante, δ^{l+1} : En particulier

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad (\boldsymbol{BP2})$$

- En combinant (BP1) et (BP2) on peut calculer l'erreur δ^lpour chaque couche du réseau.
  Une équation pour le taux de variation du coût en fonction des biais du réseau : En particulier

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad (\boldsymbol{BP3})$$

- Une équation pour le taux de variation du coût en fonction des poids dans le réseau : En particulier

$$\frac{\partial C}{\partial w_j^l} = a_j^{l-1}\delta_j^l \qquad (\boldsymbol{BP4})$$

Les preuves de ces quatre équations sont fournies en annexe.

3.3. Résolution du problème et résultats

Après avoir récupéré le dataset “airfoil data set”, la première étape consistait à traiter notre jeu de données afin de pouvoir facilement l’utiliser avec notre réseau de neurones. Pour commencer, nous avons normalisé nos données afin de ramener chacune des observations à notre disposition dans l’intervalle [0, 1]. La technique de normalisation consiste à appliquer l’opération suivante sur toutes les observations de notre jeu de données :

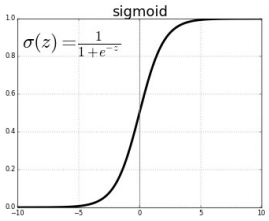
$$x_{normalisé} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Avec x_{min} et x_{max} étant respectivement le minimum et le maximum pour chaque variable.

Ensuite nous devions séparer les 1503 observations (contenant chacune les 6 variables) de notre dataset pour les données de train et de test, nous avons donc créé une fonction qui permet la partition aléatoire du l'entièreté des données en deux sous-jeux que sont donc train et test. Nous avons opté pour une distribution des données de 80% pour le sous-jeu de train (soit 1202 observations) et donc 20% pour le sous-jeu de test (soit 301 observations).

Vient ensuite la construction de notre réseau de neurones. Après avoir testé différents réseau en faisant varier la taille ou les paramètres de celui-ci, le réseau avec lequel nous avons obtenu les résultats les plus probants est un réseau composé de deux couches cachées de 5 et 3 neurones, et comme fonction d’activation la fonction sigmoïde. Comme vu précedemment, la fonction sigmoïde est une fonction définie par :

$$f(x) = \frac{1}{1 + e^{-x}}$$

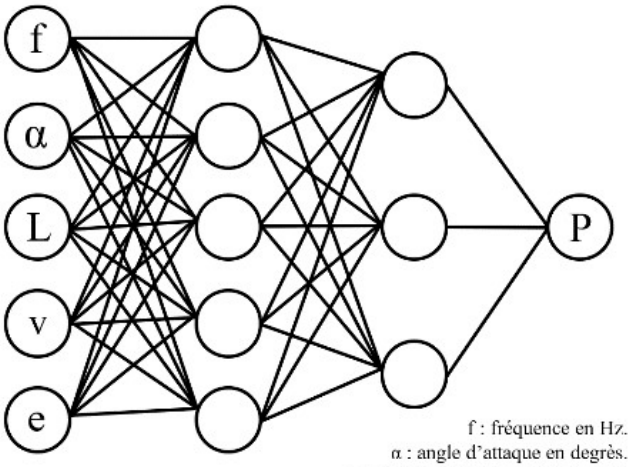


Cette fonction est très souvent utilisée dans l’implémentation des réseaux de neurones car elle facilement dérivable (ou en d’autres termes, sa dérivée est rapidement calculable par une machine) ce qui permet d’accélérer les algorithmes reposant sur le réseau de neurones. Au vu de la valeur renvoyée par cette fonction qui est comprise entre 0 et 1, il est important de normaliser les données avec la formule montrée précédemment afin que toutes les observations du dataset soient comprises entre 0 et 1. Ainsi, on augmente la précision de notre résultat final avec notre réseau de neurones. Pour finir, cette fonction nous a été recommandée par Monsieur TOFAILI afin de construire ce réseau de neurones.

Ensuite, afin de gérer les données que nous donnons à notre réseau de neurones afin de l'entraîner nous avons utilisé une fonction de Julia qui nous permettait de créer des data-batch. (Un data-batch est un échantillon / un lot de données non biaisé du jeu de données initial).

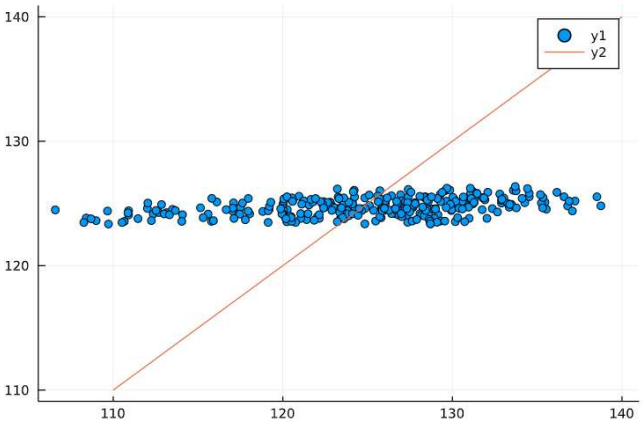
Pour finir sur les paramètres de notre réseau de neurones, la méthode d’optimisation utilisée est la descente de gradient.

Nous pouvons donc représenter notre réseau de neurones de la manière suivante :

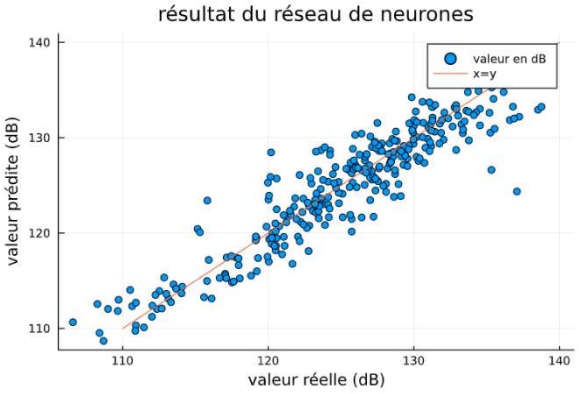


f : fréquence en Hz.
α : angle d'attaque en degrès.
L : longueur de la membrure en m.
v : vitesse de l'écoulement libre en m/s.
e : épaisseur de déplacement, coté aspiration, en m.
P : niveau de pression acoustique en dB.

Nous avons ensuite entraîné notre réseau de neurones, d'abord sur 500 epochs (une epoch signifie entraîner le réseau de neurones en passant une fois sur toutes les données qui lui sont données en train) et avec un pas $\eta = 0.1$ pour notre descente de gradient. Afin de vérifier graphiquement si notre réseau de neurones était suffisamment entraîné, pour chaque cas présent dans notre jeu de données de test, nous avons comparé la prédiction du réseau de neurones avec la vraie valeur sonore, et réalisé un graphique en traçant 301 points (le nombre d'observations dans notre jeu de données de test) ayant chacun pour abscisse la vraie valeur et en ordonnée la valeur prédite. Le graphique ainsi créé est le suivant :



Comme nous pouvons le voir, les valeurs prédites par le réseau de neurones ne sont absolument pas précises (plus les valeurs prédites sont proches de droite rouge d'équation $y = x$, plus la valeur renvoyée par le réseau de neurones est précise). Ce graphique montre que le réseau de neurones n'est pas suffisamment entraîné. Nous avons donc continué à entraîner notre réseau de neurones en effectuant plusieurs tests pour tenter d'obtenir les meilleures valeurs prédites possibles. Finalement après un nombre total de 14 500 epochs(3 000 epochs avec un pas de 0.3, 2 x 500 epochs avec un pas de 0.7 et 1500 epochs avec un pas de 1), le graphique obtenu était le suivant :



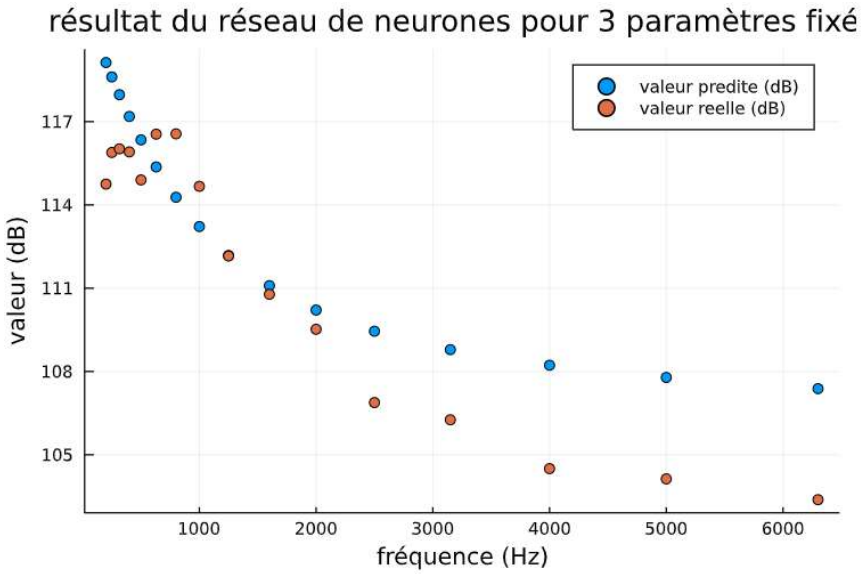
Comme nous pouvons le voir, le résultat semble assez précis, les points sont rapprochés de la droite $y = x$. Cependant, afin d'évaluer rigoureusement la qualité de notre réseau de neurones, nous avons calculé le coefficient de détermination R^2 de notre problème de régression.

Ce coefficient de détermination se calcule avec la formule suivante :

$$R^2 = \sum_{i=1}^{301} \frac{valeur\ prédite - moyenne(valeur\ réelle)}{valeur\ réelle - moyenne(valeur\ réelle)}$$

Et l'on obtient un coefficient $R^2 = 0.90192$. Ce coefficient est proche de 1, la régression que nous avons effectué avec le réseau de neurones est donc bonne. De plus, après avoir effectué d'autres tests avec plus d'epochs, notre valeur de coefficient est restée sensiblement la même. Nous pouvons donc dire que notre valeur de coefficient a fini de converger. C'est également la meilleure valeur que nous ayons obtenue parmi les réseaux de neurones testés.

Nous avons finalement affiché un spectre de fréquences (prédites et réelles) en fonction de la valeur du niveau de pression acoustique:



Nous remarquons que les valeurs prédites par notre réseau de neurones ne suivent pas vraiment la réalité, malgré les différents essais avec d'autres fonctions d'activation (par exemple la fonction ReLU), rien n'était réellement concluant.

4. CONCLUSIONS ET PERSPECTIVES

Ce projet nous a permis de découvrir le machine learning, un champ d'étude de l'intelligence artificielle qui avec une approche mathématique et statistique confère à un ordinateur la capacité “d'apprendre” à partir de données. C'est-à-dire, d'améliorer ses performances pour résoudre une tâche de manière autonome. De plus, nous avons appris à implémenter un algorithme générant un réseau de neurones fonctionnel pouvant prédire le bruit propre (en dB) d'un profil aérodynamique. Malgré un jeu de données assez faible, nous avons réussi à obtenir une précision d'environ 90%. Ce résultat démontre donc l'efficacité de notre réseau de neurones et de cette méthode pour prédire un résultat. Il n'est donc pas surprenant que le machine learning soit de plus en plus employé dans de nombreux secteurs divers et variés tels que les banques ou assurances. Nous sommes très satisfaits d'avoir eu l'occasion de travailler sur ce champ de l'intelligence artificielle en plein essor que nous serons très certainement amenés à rencontrer de nouveau au cours de notre vie professionnelle.

Nous avons également pu acquérir de nombreuses autres compétences au cours de ce projet physique. En effet, nous avons pu apprendre à appréhender des concepts mathématiques avancés en autonomie. L'organisation avec l'écriture d'un rapport contenant chaque notion concernant le fonctionnement d'un réseau de neurones et la communication au sein du groupe nous ont permis de mener ce projet dans de très bonnes conditions, malgré les conditions sanitaires qui ne nous permettaient pas de nous voir régulièrement pour avancer dans ce projet. Nous avons également développé nos compétences dans la recherche bibliographique, comme la vérification des sources et la sélection des informations. Puis nous avons appris à utiliser un nouveau langage de programmation en un temps assez limité. Ce qui nous a permis de travailler sur notre aptitude à s'adapter rapidement à un nouvel outil informatique.

Enfin, toutes les compétences que nous avons apprises tout au long de ce projet sont très appréciées dans le monde du travail et particulièrement pour le métier d'ingénieur. Nous serons donc très certainement amenés à les réutiliser au cours de notre future carrière.

5. BIBLIOGRAPHIE

Neural network 3blue1brown
https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

GradientDescent quick explanation
https://www.youtube.com/watch?v=Gbz8RljxIHo

Julia Cheat sheet
https://juliadocs.github.io/Julia-Cheat-Sheet/fr/

Image processing
https://www.simonwenkel.com/2018/10/15/Introduction-to-Computer-Vision-with-Julia-part1-basic-image-manipulation.html

Michael Nielsen "Neural network and deep learning"
http://neuralnetworksanddeeplearning.com/

Thomas F. Brooks, D. Stuart Pope, Michael A. Marcolini, “Airfoil Self-Noise and Prediction”, Nasa, 1989.

Ashutosh Patria, Yugesh Patnaik, “Random forest and stochastic gradient tree boosting based approach for the prediction of airfoil self-noise”, Procedia Computer Science, 2015.

Dmitrijs Cudihins, “Hands-on, Computer vision with Julia”, Packt, 2018.

Ivo Balbaert, “Julia 1.0 Programming”, Packt, 2018.

6. ANNEXES

6.1. Preuve des équations de la backpropagation

BP proofs

$$BP1 : \delta_j^L = \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L} \times \frac{\partial a_k^L}{\partial z_j^L}$$

However, a_k^L *depends on* z_j^L *only when* $k = j$, *otherwise* $\frac{\partial a_j^L}{\partial z_j^L} = 0$ *when* $k \neq j$

So, we have $\sum_k \frac{\partial C}{\partial a_k^L} \times \frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \times \frac{\partial a_j^L}{\partial z_j^L}$ *but* $a_j^L = \sigma\left(z_j^L\right)$

Thus, $\frac{\partial C}{\partial a_j^L} \times \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \times \frac{\partial \sigma\left(z_j^L\right)}{\partial z_j^L}$

So, $\delta_j^L = \frac{\partial C}{\partial a_j^L} \times \sigma'\left(z_j^L\right)$

$$BP2 : \delta_j^L = \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial z_k^{L+1}} \times \frac{\partial z_k^{L+1}}{\partial z_j^L} = \sum \delta_j^{L+1} \times \frac{\partial z_k^{L+1}}{\partial z_j^L}$$

However, $z_k^{L+1} = \sum_i w_{ki}^{L+1} a_i^L + b_k^{L+1} = \sum w_{ki}^{L+1} \sigma\left(z_j^L\right) + b_k^{L+1}$

So, $\frac{\partial z_k^{L+1}}{\partial z_j^L} = w_{ki}^{L+1} \sigma'\left(z_j^L\right) \rightarrow \delta_j^L = \sum_k \delta_j^{L+1} w_{kj}^{L+1} \sigma'\left(z_j^L\right)$

$$BP3 : \frac{\partial C}{\partial b_j^L} = \frac{\partial C}{\partial z_j^L} \times \frac{\partial z_j^L}{\partial b_j^L}$$

However, $z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$

So, $\frac{\partial z_j^L}{\partial b_j^L} = 1$

Then, $\frac{\partial C}{\partial b_j^L} = \frac{\partial C}{\partial z_j^L} = \delta_j^L$

$$BP4 : \frac{\partial C}{\partial w_{jk}^L} = \frac{\partial C}{\partial z_j^L} \times \frac{\partial z_j^L}{\partial w_{jk}^L}$$

However, $z_j^L = \sum_i w_{ji}^L a_i^{L-1} + b_j^L$

So, $\frac{\partial z_j^L}{\partial w_{jk}^L} = a_k^{L-1}$

Then, $\frac{\partial C}{\partial w_{jk}^L} = \frac{\partial C}{\partial z_j^L} a_k^{L-1} = \delta_j^L a_k^{L-1}$

6.2. Notebook du réseau de neurones pour la reconnaissance de chiffres écrits à la main

In [1]:
```
using DelimitedFiles
using Images
using Flux
using Flux: @epochs, onehot, train!, Data.DataLoader
```

On charge les données et on contruit le réseau de neurones :

In [2]:
```
train = readdlm("mnist_train.csv", ',', skipstart=1)
train_labels = train[:,1]
train_data = train[:,2:end]
model = Chain(Dense(784, 20, σ), Dense(20, 10, σ), Dense(10, 10, σ), softmax)
loss(x, y) = Flux.mse(model(x), y)
parameters = Flux.params(model)
one_hot = Flux.onehotbatch(train_labels, 0:9)
```

Out[2]:
```
10×60000 Flux.OneHotMatrix{Array{Flux.OneHotVector,1}}:
 0  1  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  1  0  0  1  0  1  0  0  0  0     0  0  0  0  0  0  1  0  0  0  0  0
 0  0  0  0  0  1  0  0  0  0  0  0  0     0  0  0  1  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  1  0  0  1  0  1     0  0  0  0  0  0  0  0  1  0  0  0
 0  0  1  0  0  0  0  0  0  1  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 1  0  0  0  0  0  0  0  0  0  0  1  0  …  0  0  0  0  0  1  0  0  0  1  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  1  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     1  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  1  0  0  0  0  0  1  0  0  0  1
 0  0  0  0  1  0  0  0  0  0  0  0  0     0  0  1  0  1  0  0  0  0  0  0  0
```

In [3]:
```
data = DataLoader(train_data', one_hot, batchsize=110)
```

Out[3]:
```
DataLoader{Tuple{LinearAlgebra.Adjoint{Float64,Array{Float64,2}},Flux.OneHotMatrix{Array{Flux.OneHotVector,
1}}}}(([0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0; … ; 0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0], Bool[0 1 … 0 0; 0 0 …
0 0; … ; 0 0 … 0 1; 0 0 … 0 0]), 110, 60000, true, 60000, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10  …  59991, 59992, 5999
3, 59994, 59995, 59996, 59997, 59998, 59999, 60000], false)
```

On entraine le réseau de neurones :

In [4]:
```
η = 1
optimizer = Descent(η)
@epochs 30 train!(loss, parameters, data, optimizer)
```

```
[ Info: Epoch 1
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 2
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 3
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 4
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 5
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 6
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 7
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 8
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 9
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 10
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 11
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 12
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
```

In [5]:
```
test = readdlm("mnist_test.csv", ',', skipstart=1)
test_labels = test[:,1]
test_data = test[:,2:end]
```

Out[5]:
```
10000×784 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 ⋮                        ⋮              ⋱               ⋮
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

On construit une fonction afin de renvoyer le pourcentage de bonne réponses du réseau de neurones

In [6]:
```
preprocess(img) = vec(Float64.(img))
prediction(i) = findmax(model(test_data[i,:]))[2] - 1
prediction_correct(r,n) = ((sum(test_labels[i] == prediction(i) for i in r)) / n) * 100
train_accuracy = prediction_correct(1:5000,5000)
```

Out[6]:
```
83.36
```

On obtient un score de 83,36%. On entraine encore notre réseau de neurones :

In [7]:
```
@epochs 60 train!(loss, parameters, data, optimizer)
```

```
[ Info: Epoch 1
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 2
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 3
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 4
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 5
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 6
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 7
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 8
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 9
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 10
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 11
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
[ Info: Epoch 12
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
```

```
└ @ Main D:\.julia\packages\Flux\goUGu\src\optimise\train.jl:136
```

In [8]:
```
train_accuracy = prediction_correct(1:5000,5000)
```

Out[8]:
```
86.96000000000001
```

On obtient finalement un score de 87,0%, ce qui est un bon score pour ce problème.

6.3. Notebook du réseau de neurones pour la prédiction du bruit propre d'un profil aérodynamique

In [2]:
```
using DelimitedFiles
using Flux
using Flux: @epochs, onehot, train!, Data.DataLoader
using BSON: @save, @load
using DataFrames
using Random: shuffle
using Statistics
using Plots
```

In [3]:
```
#Lecture du dataset fourni sur (https://archive.ics.uci.edu/ml/datasets/airfoil+self-noise#)

data = readdlm("airfoil_self_noise.dat", '\t', '\n');
```

In [4]:
```
#on calcule le maximum et le minimum pour chaque variable

minarray = Array{Float64, 2}(undef, 1, 6)
maxarray = Array{Float64, 2}(undef, 1,  6)
for i = 1:6
    minarray[i] = min(data[:, i]...)
    maxarray[i] = max(data[:, i]...)
end
```

In [32]:
```
#Fonction permettant de diviser le dataset en deux sous-groupes: un d'entrainement et un de test

function partitionTrainTest(data, at = 0.7)
    n = size(data)[1]
    idx = shuffle(1:n)
    train_idx = view(idx, 1:floor(Int, at*n))
    test_idx = view(idx, (floor(Int, at*n)+1):n)
    data[train_idx,:], data[test_idx,:]
end
```

Out[32]:
```
partitionTrainTest (generic function with 2 methods)
```

In [6]:
```
#Division de notre dataset en sous-groupes
train,test = partitionTrainTest(data, 0.8);
```

In [7]:
```
#Verification du nombre de données dans chaque sous-groupe
println("size(test)[1] + size(train)[1] = ", size(test)[1]," + ", size(train)[1], " = ", size(test)[1]+size(tra
```

```
size(test)[1] + size(train)[1] = 301 + 1202 = 1503  and size(data)[1] = 1503
```

In [11]:
```
#on normalise les données de train et de test

p = 6 #nombre de variables
train_norm = Array{Float64, 2}(undef, length(train)÷6, 6)
for i = 1:6
    for j = 1:length(train)÷6
        train_norm[j, i] = (train[j, i] - minarray[i])/(maxarray[i] - minarray[i])
    end
end
test_norm = Array{Float64, 2}(undef, length(test)÷6, 6)
for i = 1:6
    for j = 1:length(test)÷6
        test_norm[j, i] = (test[j, i] - minarray[i])/(maxarray[i] - minarray[i])
    end
end
```

In [12]:
```
#Division au sein du sous groupe d'entrainement en :
# - Un sous groupe contenant les données fournies en entrées (train_data)
# - Un sous groupe contenant les données en sorties (ici le son en dB) (train_labels)

train_labels = train_norm[:,6]
train_data = train_norm[:,1:5]
test_labels = test[: ,6]
test_data = test[:, 1:5]
test_labels_norm = test_norm[:, 6]
test_data_norm = test_norm[:, 1:5];
```

Nous allons maintenant créer le réseau de neurones, avec deux couches cachées de respectivement 5 et 3 neurones, la fonction d'activation sigmoïde, la loss function mse ("mean squared error"), un dataBatch qui contient les données et les labels (permet de facilement entrainer le réseau de neurones), et la descente de Gradient comme optimiseur avec un pas de 0.1 .

In [14]:
```
model = Chain(Dense(5, 5, σ), Dense(5, 3, σ), Dense(3, 1, σ))

#Définition de la loss function et des paramètres du neural network
loss(x, y) = Flux.mse(model(x), y);
parameters = Flux.params(model);
dataBatch = DataLoader(train_data', train_labels', batchsize=110);
optimizer = Descent()
```

Out[14]:
```
Descent(0.1)
```

On commence par entrainer notre réseau de neurones sur 500 epochs (répétition de l'entrainement sur l'ensemble du dataBatch).

In [ ]:
```
@epochs 500 train!(loss, parameters, dataBatch, optimizer)
```

on peut tester le résultat du réseau de neurones sur une seule valeur :

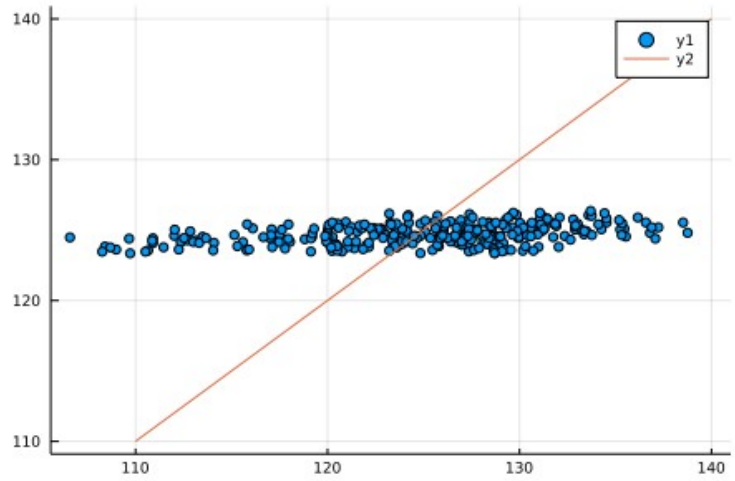
In [16]:
```
#vérification d'une valeur renvoyée par le NN
testindex = 28
println(test_labels_norm[testindex, :])
testvalue = model(test_data_norm[testindex,:])
println(testvalue)
# on dénormalise la valeur :
println("valeur réelle : ", test_labels[testindex, :])
testvalue_denorm = testvalue[1]*(maxarray[6] - minarray[6])+minarray[6]
println("valeur prédite : ", testvalue_denorm)
```

```
[0.6594251070279471]
Float32[0.54981124]
valeur réelle : [128.179]
valeur prédite : 124.05675145351887
```

Maintenant nous allons tracer un graphique qui pour chaque valeur réelle de volume en decibels en abcisse associe la valeur prédite par le réseau de neurones en ordonnées

In [17]:
```
#on plot en abcisse les valeurs réelles et en ordonnée les valeurs prédites
ntest = 301
predictedvalue = Array{Float64, 2}(undef, 301, 1)
predictedvalue_norm = Array{Float64, 2}(undef, 301, 1)
for i = 1:301
    testvalue = model(test_data_norm[i, :])
    predictedvalue_norm[i] = testvalue[1]
    testvalue_denorm = testvalue[1]*(maxarray[6] - minarray[6])+minarray[6]
    predictedvalue[i] = testvalue_denorm
end
x = test_labels;
y = predictedvalue;
plot(x,y, seriestype = :scatter)
plot!([110, 140], [110, 140])
```

Out[17]:



On observe que le réseau de neurones n'est pratiquement pas entrainé, on va donc l'entrainer encore

In [ ]:
```
optimizer = Descent(0.3)
@epochs 3000 train!(loss, parameters, dataBatch, optimizer)
```

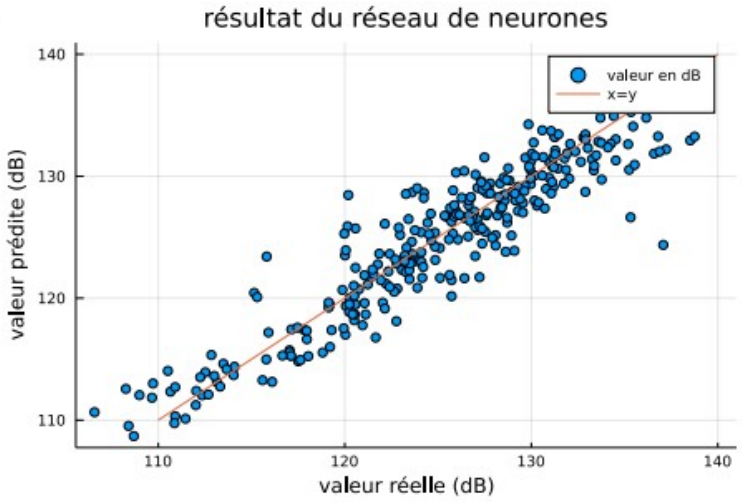
In [ ]:
```
optimizer = Descent(0.7)
@epochs 5000 train!(loss, parameters, dataBatch, optimizer)
```

In [ ]:
```
optimizer = Descent(0.7)
@epochs 5000 train!(loss, parameters, dataBatch, optimizer)
```

In [ ]:
```
optimizer = Descent(1)
@epochs 1500 train!(loss, parameters, dataBatch, optimizer)
```

In [28]:
```
ntest = 301
predictedvalue = Array{Float64, 2}(undef, 301, 1)
predictedvalue_norm = Array{Float64, 2}(undef, 301, 1)
for i = 1:301
    testvalue = model(test_data_norm[i, :])
    predictedvalue_norm[i] = testvalue[1]
    testvalue_denorm = testvalue[1]*(maxarray[6] - minarray[6])+minarray[6]
    predictedvalue[i] = testvalue_denorm
end
x = test_labels;
y = predictedvalue;
plot(x,y, seriestype = :scatter, title = "résultat du réseau de neurones", label = "valeur en dB")
plot!([110, 140], [110, 140], label = "x=y")
xlabel!("valeur réelle (dB)")
ylabel!("valeur prédite (dB)")
```

Out[28]:



au bout de 14500 epochs avec des pas différents pour la descente de gradient, on remarque que le réseau de neurones donne de bien meilleurs résultats et commence a stagner. Calculons maintenant le coefficient de détermination R2 de notre régression

In [31]:
```
function coeffR2(real, prediction)
    num = 0
    denom = 0
    for i = 1:length(real)
        num += (prediction[i] - mean(real))^2
        denom += (real[i] - mean(real))^2
    end
    R2 = num/denom
    R2
end
```

Out[31]: coeffR2 (generic function with 1 method)

In [34]:
```
R2 = coeffR2(test_labels, predictedvalue)
print(R2)
```

```
0.9019287778783459
```

On obtient un coefficient de détermination de 0.9019, ce qui est une bonne valeur pour une régression.

On va maintenant afficher les valeur réelles et prédites mais cette fois çi en fonction d'un seul paramètre qui est la fréquence en Hz (on fixe l'angle d'attaque, la longueur de corde et la vitesse de flux)

In [94]:
```
test_data
index = zeros(0)
for i = 1:1503
    if (data[i, 2] == 12.6) && (data[i, 3] == 0.1524) && (data[i, 4] == 39.6)
        append!(index, i)
    end
end
```

In [100…

```
predite = Array{Float64, 2}(undef, length(index), 1)
reelle = Array{Float64, 2}(undef, length(index), 1)
frequences = Array{Float64, 2}(undef, length(index), 1)
k = 1
for o in index
    i = UInt32(o)
    normed_data = Array{Float64}(undef, 5)
    for j = 1:5
        normed_data[j] = (data[i, j] - minarray[j])/(maxarray[j] - minarray[j])
    end
    predite_norm = model(normed_data)
    predite[k] = predite_norm[1]*(maxarray[6] - minarray[6])+minarray[6]
    reelle[k] = data[i, 6]
    frequences[k] = data[i, 1]
    k += 1
end
```

graphique pour les paramètres suivants : angle d'attaque = 12.6°, longueur de corde = 15.24cm, vélocité du fluide = 39.6 ms

In [101…

```
plot(frequences, predite, seriestype = :scatter, title = "résultat du réseau de neurones pour 3 paramètres fixé
plot!(frequences, reelle, seriestype = :scatter, label = "valeur reelle (dB)")
xlabel!("fréquence (Hz)")
ylabel!("valeur (dB)")
```

Out[101…